Adding Hygiene to Gambit Scheme

Université de Montréal

July 4, 2023

Université de Montréal
Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Ce mémoire intitulé

Adding Hygiene to Gambit Scheme

Présenté par

Antoine Doucet

A été évalué par un jury composé des personnes suivantes

Stefan Monnier
Président-rapporteur

Marc Feeley
Directeur de recherche

Sébastien Roy
Membre du jury

Résumé

Le langage de programmation Scheme est reconnu pour son puissant système de macro-transformations. La représentation du code source d'un programme, sous forme de données manipulables par le langage, permet aux programmeurs de modifier directement l'arbre de syntaxe abstraite sous-jacent. Les macro-transformations utilisent une syntaxe similaire aux procédures régulières mais, elles définissent plutôt des procédures à exécuter lors de la phase de compilation. Ces procédures retournent une représentation sous forme d'arbre de syntaxe abstraite qui devra être substitué à l'emplacement de l'appel du transformateur. Les procédures exécutées durant la phase de compilation profitent de la même puissance que celles exécutées durant de la phase d'évaluation. Avec ce genre de système de macro-transformations, un programmeur peut créer des règles de syntaxe spécialisées sans aucun coût additionnel en performance: ces extensions syntactiques permettent l'abstraction de code sans les coûts d'exécution habituels reliés à la création d'une fermeture sur le tas.

Cette représentation pour le code source de Scheme provient directement du langage de programmation Lisp. Le code source est représenté sous forme de listes manipulables de symboles, ou bien de listes contenants d'autres listes: une structure appelée S-expression. Cependant, avec cette approche simpliste, des conflits de noms peuvent apparaître. En effet, l'association référée par un certain identifiant est déterminée exclusivement par le contexte lexical de celui-ci. En déplaçant un identifiant dans l'arbre de syntaxe abstraite, il est possible que cet identifiant se retrouve dans un contexte lexical contenant une certaine association pour un identifiant du même nom. Dans de tels cas, l'identifiant déplacé pourrait ne plus référer à l'association attendue, puisque cette seconde association pourrait avoir prévalence sur la première. L'assurance de transparence référentielle est alors perdue. En conséquence, le choix de nom pour les identifiants vient maintenant influencer directement le comportement du programme, générant des erreurs difficiles à comprendre. Les conflits de noms peuvent être corrigés manuellement dans le code en utilisant, par exemple, des noms d'identifiants uniques. La préservation automatique de la transparence référentielle se nomme hygiène, une notion qui a été beaucoup étudiée dans le contexte des langages de la famille Lisp.

La dernière version du *Scheme revised report*, utilisée comme spécification pour le langage, étend ce dernier avec un support pour les macro-transformations hygiéniques. Jusqu'à maintenant, l'implémentation *Gambit* de Scheme ne fournissait pas de tel système à sa base. Comme contribution, nous avons ré-implémenter le système de macro de Gambit pour supporter les macro-transformations hygiéniques au plus bas niveau de l'implémentation. L'algorithme choisi se base sur l'algorithme *set of scopes* implémenté dans le langage Racket et créé par Matthew Flatt. Le lan-

gage Racket s'est grandement inspiré du langage Scheme mais, diverge sur plusieurs fonctionnalités importantes. L'une de ces différences est le puissant système de macro-transformation sur lequel Racket base la majorité de ses primitives. Dans ce contexte, l'algorithme a donc été testé de façon robuste.

Dans cette thèse, nous donnerons un aperçu du langage Scheme et de sa syntaxe. Nous énoncerons le problème d'hygiène et décrirons différentes stratégies utilisées pour le résoudre. Nous justifierons par la suite notre choix d'algorithme et fourniront une définition formelle. Finalement, nous présenterons une analyse de la validité et de la performance du compilateur en comparant la version originale de Gambit avec notre version supportant l'hygiène.

Mots clefs: Compilation, Langage, Méta-programmation, Macro, Hygiène, Scheme, Syntaxe

Abstract

The Scheme programming language is known for its powerful macro system. With Scheme source code represented as actual Scheme data, macro transformations allow the programmer, using that data, to act directly on the underlying abstract syntax tree. Macro transformations use a similar syntax to regular procedures but, they define procedures meant to be executed at compile time. Those procedures return an abstract syntax tree representation to be substituted at the transformer's call location. Procedures executed at compile-time use the same language power as run-time procedures. With the macro system, the programmer can create specialized syntax rules without additional performance costs. This also allows for code abstractions without the expected run-time cost of closure creations.

Scheme's representation of source code using values inherits that virtue from the Lisp programming language. Source code is represented as a list of symbols, or lists of other lists: a structure coined S-expressions. However, with this simplistic approach, accidental name clashes can occur. The binding to which an identifier refers to is determined by the lexical context of that identifier. By moving an identifier around in the abstract syntax tree, it can be caught within the lexical context of another binding definition with the same name. This can cause unexpected behavior for programmers as the choice of names can create substantial changes in the program. Accidental name clashes can be manually fixed in the code, using name obfuscation, for instance. However, the programmer becomes responsible for the program's safety. The automatic preservation of referential transparency is called hygiene and was thoroughly studied in the context of lambda calculus and Lisp-like languages.

The latest Scheme revised report, used as a specification for the language, extend the language with hygienic macro transformations. Up to this point, the Gambit Scheme implementation wasn't providing a built-in hygienic macro system. As a contribution, we re-implemented Gambit's macro system to support hygienic transformations at its core. The algorithm we chose is based on the set of scopes algorithm, implemented in the Racket language by Matthew Flatt [Fla16]. The Racket language is heavily based on Scheme but, diverges on some core features. One key aspect of the Racket language is its extensive hygienic syntactic macro system, on which most core features are built on: the algorithm was robustly tested in that context.

In this thesis, we will give an overview of the Scheme language and its syntax. We will state the hygiene problem and describe different strategies used to enforce hygiene automatically. Our algorithmic choice is then justified and formalized. Finally,

we present the original Gambit macro system and explain the changes required. We also provide a validity and performance analysis, comparing the original Gambit implementation to our new system.

Keywords: Compilation, Language, Metaprogramming, Macro, Hygiene, Scheme, Syntax

Contents

Remerciements

Je tiens à remercier Marc Feeley qui m'a permis de m'impliquer dès le début de mon parcours et m'a soutenu tout au long de celui-ci. Je remercie également Stefan Monnier et tous les membres du laboratoire de langage pour toutes ces discussions et présentations motivantes. Je remercie chaleureusement ma famille et leur support inestimable, du primaire à l'université et, en particulier, à mon père qui aurait assurément été fier de cet accomplissement. Finalement, je remercie et félicite mes amis d'avoir pu m'encourager jusqu'au bout !

Chapter 1

Introduction

Gambit is an implementation of the Scheme programming language whose development spans over three decades. The system has been used for programming language research, education, and industrial needs. Scheme is often associated with advanced language constructs such as higher-order functions, continuations, and macros. Our work focuses on the macro facility provided by Gambit. Macro transformations allow the language users to implement their own syntactic extensions. This can be used, for instance, to create a domain-specific language on top of the language's core syntax. Historically, Gambit provides `define-macro` as the main form used to create macro transformers. The form is directly inspired by the `defmacro` form in Lisp and gives great power to the programmer.

However, this form must be used carefully as it can lead to a variety of issues related to naming. For this reason, the `define-macro` form is called unhygienic. The two latest specification report for the Scheme language requires a hygienic macro system that avoids these issues [FM+07] [SCG13]. The work reported in this thesis has implemented a new macro system that conforms to the new standard. To introduce the problem, we begin with a brief explanation of general compilation steps, before looking into some of the Gambit specificity.

Compilation and Evaluation Pipeline

The compilation of a program is a translation process. Programmers write their code in a particular *source language*, the compiler transforms its textual representation into a specific *target language*, with the same semantics. The compiler is said to be an *implementation* of the source language that is to be translated. Compilers are written using some *host language* and, most of the time, offer features useful for debugging and optimization. As compilers simply transform the source code, we can use multiple compilers in sequence in what is called a compilation pipeline. To be able to run the initial program, the target language of the last compiler used in the sequence should target the final machine's particular processor.

Generally speaking, the compilation of a program uses the same sequence of steps. At first, the compiler must *read* the source code, building a manipulable representation within the host language. This step, named *parsing*, starts with the creation of a concrete syntax tree containing the different textual *tokens* appearing

in the source code. The program's syntax is verified, according to some context-free grammar, without any information about the language's semantics. Afterwards, the program's concrete syntax tree is transformed into an abstract syntax tree. This abstract representation encodes the program's structure in a way that is well suited for analysis, grouping and keeping the important information only.

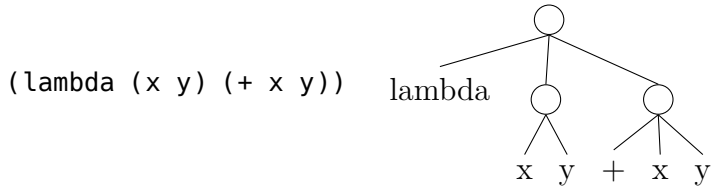


Figure 1.1: A simple Scheme program and it's AST

Once the final syntax tree is constructed, the subsequent compilation phase transforms this syntactic description into the target language, while preserving semantics. To do so, most high-level programming languages first convert the tree to an intermediate language representation (IR). Those intermediate languages are designed to abstract the machine to a level closer to the source language. Usually, an intermediate language targets some virtual machine that is simpler than a physical machine, and designed to allow for low-level optimizations. Some virtual machines are developed to be used with a variety of source languages, such as LLVM, and are designed to act as a high-level assembly language. Others were designed with a particular source language in mind. For instance, Java programs are converted to an intermediate representation to be run on the JVM, a Java-specific virtual machine. The use of an intermediate language allows the compiler to target different languages without the need to change the first few stages of the compilation pipeline depending on each target. Virtual machines can also abstract differences between compilation and interpretation.

A good number of compilers use a preprocessor program in their pipeline to transform the program's structure before further analysis. Such an *expander* implements syntactic extensions to the language. Those extensions can be simple textual substitutions that can be implemented within the parsing phase, or more complex transformations, such as acting on the abstract syntax tree representation. In the C programming language, programmers can define macro transformations as textual substitutions, which are performed by a preprocessor prior to the usual parsing of the program. The following C program defines and uses a macro `ABS` to print the absolute value of a number.

```
#define ABS(n) (negative(n) ? -(n) : (n))

int negative(int n) { return (n < 0); }

void some_procedure(int a) {
  printf("%d",
         ABS(a));
}
```

```
int negative(int n) { return (n < 0); }

void some_procedure(int a) {
  printf("%d",
         (negative(a) ? -(a) : (a)));
}
```

Figure 1.2: A simple C macro and the expanded code

On the other hand, the Lisp programming language allows transformations on the program's abstract syntax tree directly and this is known as a syntactic preprocessor.

```
(defmacro abs (n)
  (list 'if (list 'negative n)
              (list '- n)
              n))

(defun negative (n) (< n 0))

(defun some-procedure (a)
  (print (abs a)))
```

```
(defun negative (n) (< n 0))

(defun some-procedure (a)
  (print (if (negative a)
             (- a)
             a)))
```

Figure 1.3: A simple Lisp macro and the expanded code

Those language extensions, when provided by the programmer, are referred to as macro transformers or macros for short and their use in code is called meta programming. Macro systems were mainly popularized by the Lisp programming language as it was the first broadly used high-level language to include syntactic pre processing. With a Lisp system, programmers have access to the whole language to perform syntactic transformations: extensions can be expressed directly within the source language itself, with the same syntax. The use of macros differs from normal code execution and this semantic break often leads to problems when used casually as will be explained later.

The Scheme Pipeline

We briefly describe the different stages usually involved in the compilation and evaluation of a program, for the Scheme programming language. As is the case for most languages, differences between interpretation and compilation pipelines lie in the final stage only, as the parsing and expansion phase are common to both.

Parsing As a first step, the parser takes a source code representation as plain text and constructs a data structure, representing the program's abstract syntax tree called *S-expression*. Scheme provides support for textual substitutions during the parsing phase, known as reader-macros, but do not allow the programmers to create their own transformers. The Scheme programming language uses an LL(1) grammar which can be handled using a simple recursive descent parser without backtracking: the parsing of such a grammar requires a single *look-ahead* character to construct the S-expression, thus proceeding in linear time. The operation is equivalent to the Scheme built-in procedure `read`. If we abstract the file reading steps, we can think of the `read` procedure as having the type **String** $\rightarrow$ **Sexp**, where **Sexp** is the S-expression type. The "typing" notation used for the pipeline's description is used to give a simple intuition for the interpretation of each phase's output's.

Expansion The expansion phase, transforms the S-expression that is constructed by the parsing phase. The expansion is done recursively: after each transformer application step, the compiler will recursively expand the resulting S-expression, until no expansion is possible in the final expression. The syntactic preprocessor allows transformers to use any core construct of the language, giving the full language's power to the programmer at *compile-time*. We denote this operation with the procedure `expand` : **Sexp** $\rightarrow$ **Sexp**. Note that this procedure is not generally exposed to the programmer.

Intermediate Representation (IR) In the case of a program's compilation, once the S-expression is fully expanded, it is transformed into an intermediate representation which can, in turn, be transformed again depending on the target machine, that can be a real or virtual one.

Interpretation It is also possible to use an interpreter that executes the program directly from the fully expanded S-expression. The execution is the *run-time* phase. The predefined `eval` procedure embodies the interpretation of S-expressions. It can be used on its own at both run-time or compile-time, as it is modelled in the host and target languages. We can think of the procedure as having type **Sexp** $\rightarrow$ **Data**, where the **Data** type stands for any Scheme value (the result of the program).

Gambit Scheme Interpretation and Compilation

The Gambit system includes the Gambit Scheme Interpreter (GSI) program for interpretation, as well as the Gambit Scheme Compiler (GSC) for compilation. As Gambit allows different target languages (C, JavaScript, Python) a unique intermediate representation is used between the S-expression representation of expanded code and the full compilation. This intermediate representation targets Gambit's specific virtual machine, the Gambit Virtual Machine (GVM). We now show a simplified view of the compilation and interpretation steps in Gambit.

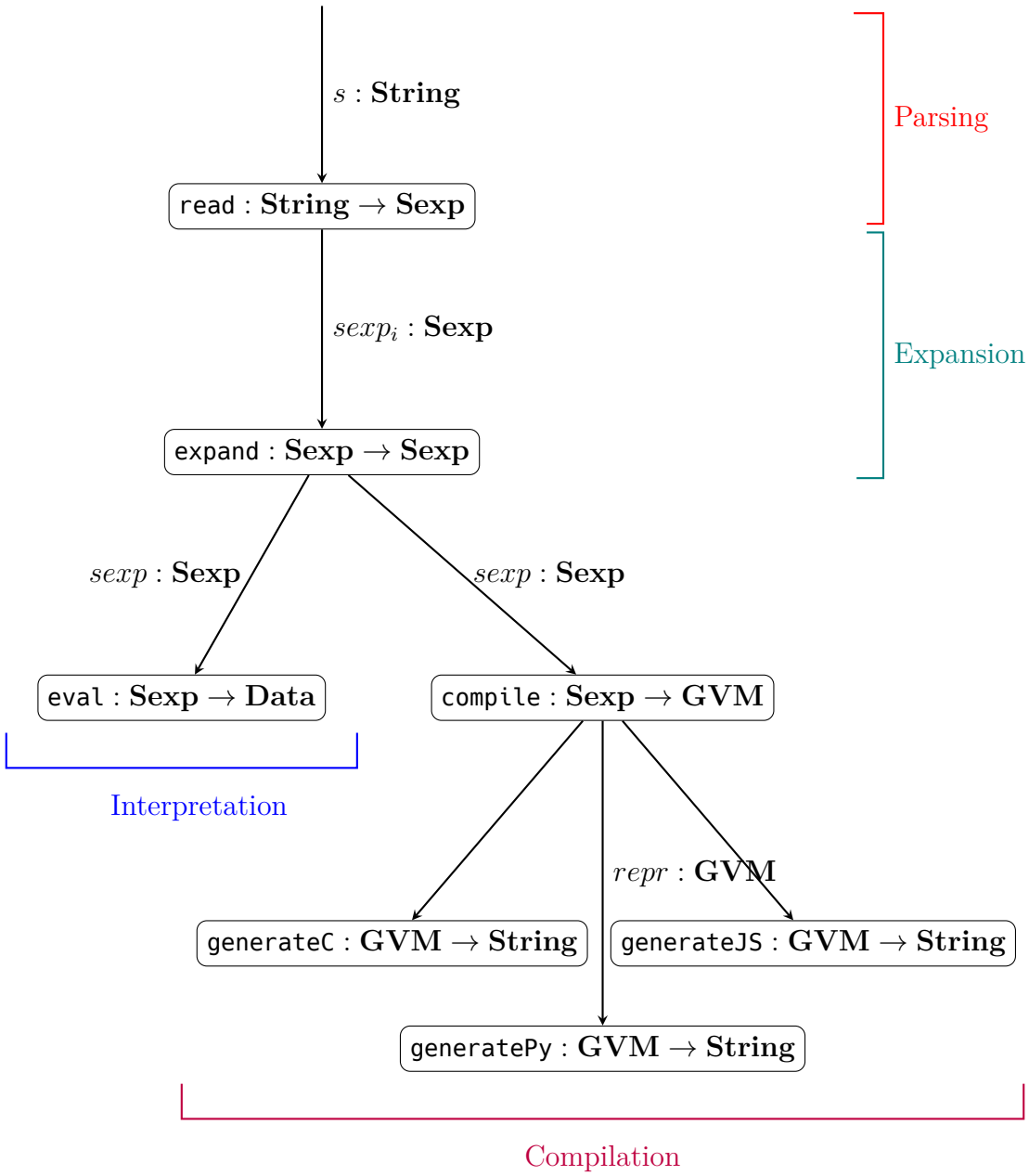


Figure 1.4: Gambit's Compilation Pipeline

Gambit's macro expansion uses the same syntactic pre processing technique as with the Lisp language by performing substitutions on the abstract syntax tree. Our work is focused on improving the expansion stage.

The Hygiene Problem

As macro transformers perform substitutions, in the textual code or its AST, *name clashes* can occur. Recall our simple C macro definition given earlier. Identifiers inserted by substitution will always take the lexical context of their final position in the code. Using the same `ABS` macro in two places that are in lexical scope of different definitions of `negative` would cause the macro's code to behave differently than intended.

```
#define ABS(n) (negative(n) ? -(n) : (n))

int negative(int n) { return (n < 0); }

void some_procedure(int a) {
    int positive = ABS(a);
    int negative = -positive;

    printf("%d %d %d",
           positive,
           negative,
           ABS(a));
}
```

```
int negative(int n) { return (n < 0); }

void some_procedure(int a) {
    int positive = (negative(a) ? (-a) : (a)));
    int negative = -positive;

    printf("%d %d %d",
           positive,
           negative,
           (negative(a) ? (-a) : (a)));
}
```

Figure 1.5: A simple case of name clash with a C macro

The hygiene problem is widely documented in the context of the lambda calculus, Scheme and other Lisp flavours. A hygienic macro system makes sure that no accidental name clashes can occur. The `define-macro` form is unhygienic as no safety net is provided to protect the programmer against accidental name clashes. As a temporary measure, Gambit does include forms meant to be used within a hygienic macro system such as `define-syntax` and `syntax-case`, without providing the hygiene guarantees.

This thesis discusses the implementation of a hygienic macro system for the Gambit system. We will begin with a formal definition of Scheme's syntax in order to explain the hygiene problem thoroughly. We then present some algorithms theorised and provide justifications for our choice. We then focus on the algorithm by presenting its formalization. Subsequently, we describe Gambit's original system, and its particular features, before an explanation of the modifications required to implement hygiene. Finally, we provide some validity and performance analysis of the new system.

Chapter 2

Scheme and Macro Transformers

This chapter summarizes the syntax of the Scheme programming language, as well as its powerful syntactic extension abilities. The language's initial macro system, as most of its features, was heavily inspired by work on the Lisp programming language and its dialects.

Scheme Syntax

The Scheme programming language's syntax is extremely simple, using parentheses as the main delimiter. The program's structure can be easily described using a tree-like data structure coined S-expressions that exhibits a direct isomorphism with the program's abstract syntax tree (AST) [KFFD86]. A parenthesized group of tokens represents branching in the tree while any individual token is a leaf.

Language Construct's Overview

Scheme is a general purpose programming language supporting the functional programming paradigm. The language is dynamically typed and includes several built-in types. It implements numerical values with the integer[1], float and complex types, as well as the values `#f` and `#t` for the booleans *true* and *false*. Scheme also supports different containers, such as strings, vectors, lists, etc. We list below some of Scheme's values and their corresponding built-in types.

Scheme's values	built-in type(s)
`0`, `1.2`, ...	*fixnum*, *bignum*, float, ...
`#t`, `#f`	boolean
`"a-string"`	string
`'a-symbol`	symbol
`'(0 1 2)`, `'("a" #t 2)`	list
`+`, `(lambda (x) x)`	procedure
...	...

[1] Scheme's integer types includes different variant

The programmer can create new procedural abstractions using the `lambda` form, which is inspired by the lambda calculus. For instance, `(lambda (x) x)` defines the identity procedure: a procedure of one argument `x` returning its value. We can apply a procedure to some arguments using the prefix notation. For instance `(+ 1 2)` calls the `+` procedure with `1` and `2` as actual parameters. To declare new local bindings, we use the `let` special form: the form `(let ((x 0) (y 1)) (+ x y))` binds two new variables, `x` and `y` to their respective value, `0` and `1`, within the subexpression `(+ x y)`. The simplest use of `let` can be seen as an applied procedure. Indeed, the previous `let` form is equivalent to the following applied procedure of two arguments: `((lambda (x y) (+ x y)) 0 1)`. To define a top-level variable binding, the `define` form can be used as such: `(define x 42)`. The branching form `if` allows for conditional branching. After evaluating the first expression, any value which is not `#f` is interpreted as true. The `(if #t` $expr_1$ $expr_2$`)` form evaluates $expr_1$ while `(if #f` $expr_1$ $expr_2$`)` evaluate $expr_2$. As Scheme requires tail-call optimization, the language makes heavy use of recursion, and does not provide the looping construct usually available in more mainstream languages such as `for`, `do`, or `while`. The language allows mutation of variables using the `set!` built-in. As a final note, the type notation we will use for this thesis doesn't match the built-in types of the language but is used for the static analysis of the code.

S-expressions

S-expressions were initially designed as a representation of Lisp code. The recursive structure in itself is simple but powerful: it can express any binary tree and, by extension, any tree-like structure. Their uses have been extended to other problems as well. In Scheme, parenthesized groups of S-expressions represent S-expressions themselves. We note that all Scheme programs are valid S-expression but not all S-expressions represent valid Scheme programs.

Definition 1 (**S-expression**). *An S-expression* $s : \textbf{\textit{Sexp}}\ a$ *is a grouping of atomic tokens, that could be formalized by the following inductively defined union type:*

$$\begin{aligned} \forall a.\ \textbf{\textit{Sexp}}\ a ::= \ & \textbf{\textit{List}}\ (\textbf{\textit{Sexp}}\ a) \\ \mid \ & a \end{aligned}$$

In most Lisp-like languages, a list of n elements is represented as a parenthesized group of tokens or other lists, denoted `(`e_1 ... e_n`)`.

We consider Scheme's S-expression to be a more precise type, namely **Sexp Atom**, where the **Atom** type contains every non-parenthesized terminal of the Scheme syntax. We shall shorten the Scheme S-expression type **Sexp Atom** by **Sexp** when obvious from the context.

Scheme uses a prefix notation for applications. An S-expression containing a parenthesized group of expressions can represent a function application, macro transformer application, or core form use, as determined by the first sub-expression of

the S-expression. For instance, `(cons 0 1)` is a functional application of the procedure `cons` with the expressions `0` and `1` as actual parameters. On the other hand, the S-expression `(if #t 0 1)` is a core form use, simply because the keyword `if` is recognized as such. In the same way, if `cons` was bound to a macro transformer instead, `(cons 0 1)` would represent a macro application.

Textual Substitution

In addition to the application's notation described above, the language supports textual substitutions, with the use of *reader-macros*. Those macros act on the textual representation of the code, during the parsing phase.

Definition 2 (Reader-macro)**.** *Reader macros are textual substitutions applied systematically during the reading phase. The Scheme programming language does not allow users to interact directly with this compilation phase's expansion. For instance, the reader macro* `'`*, used as* `'x` *is expanded at read-time into* `(quote x)`*, while* `,x` *is expanded into* `(unquote x)`*.*

However, the programmer cannot extend the set of transformations offered.

Dynamic Construction of S-expressions

One, if not the most, notable feature of the Scheme language is its ability to construct some source code, programmatically, within the language itself: S-expressions can be constructed using Lists and Symbols, two primitive data types of Scheme.

For completeness, we include a formal definition of the pair data structure, on which lists are based on.

Definition 3 (**Scheme's Pair Datatype**)**.** *The* ***Pair*** *primitive data type is a heterogeneous pair. The data type has the following constructor with two fields:*

$$\texttt{cons} \ : \ \forall t_a, t_b.\ (t_a \to t_b \to \textbf{\textit{Pair}}\ t_a\ t_b)$$

Let $a : A$ *and* $b : B$*, we can create a pair* `(cons a b)` : ***Pair*** A B *represented as* `(a . b)`*, or* `(a x ...)` *if* `b` *is the pair* `(x ...)`*, or* `(a)` *if* `b` *is the empty list.*

Introducing the special object `()`, (pronounced *null*), used as the end-of-list object, singly linked lists can be implemented directly with pairs. Each pair then contains an element in its first field and a reference to the next pair (or *null*) in its second.

Definition 4 (**Scheme's List Datatype**)**.** *A Scheme list is a singly linked list of pairs of heterogeneous types.*
Let $a_1 : A_1$*, ...,* $a_n : A_n$ *be Scheme values, then we can construct a list* `(list` a_1 `...` a_n`)` : ***Pair*** A_1 (***Pair*** ...)*, represented by* `(`a_1 `. ... (`a_n `. ()))` *or shortly* `(`a_1 `...` a_n`)`*.*

We denote homogeneous lists of type a with the type ***List a****, such that* `()` $: \forall \boldsymbol{a}.$ ***List a****. A list lst is constructed as follows:*

$$\begin{aligned} lst,\ lst_{rst} &: \textbf{\textit{List a}} \\ e &: \boldsymbol{a} \\ lst &::= \texttt{(cons } e\ \texttt{lst}_{rst}\texttt{)} \\ &\mid\ \texttt{'()} \end{aligned}$$

Definition 5 (**Scheme's symbol primitive type**)**.** *The* ***Symbol*** *primitive type holds the source-code representation of an atomic token as a chain of characters memory. In this thesis, symbols are represented with the character* `'` *prepended to any non-parenthesized source-code token's textual representation to clearly destinguish symbols from variables. The special form* `quote` *is used to create a new symbolic representation.*

Specials forms, such as `quote`, are a special case of *macro-transformer*: they manipulate the AST directly. Let's consider the code `(quote` s`)` where s : **Sexp**. If s is a non-parenthesized piece of data, `(quote` s`)` is the symbolic representation of that token: the sequence of characters as appearing in the code's textual format. For instance, `(quote a-variable)` encode the character chain "a-variable" in a structure similar to the primitive Scheme's type **String**. Otherwise, if s is a parenthesized group of expressions, *quoting* s results in a linked list containing the source-code representation of each element of the group. It follows that the `quote` special form can be used to create lists as well. The following lists an approximation for the special form's rules, where $\text{exp}_a \leftrightsquigarrow \text{exp}_b$ means that exp_a and exp_b are `equal?`, but not necessarily `eq?` [2]. The following figure shows the equivialence between the quote reader-macro and its read-macro's expansion, from the language user's point of view. From the compiler point of view however, they are exactly the same.

$$\begin{aligned} \texttt{(quote (}E_1\ \ldots\ E_n\texttt{))} &\leftrightsquigarrow \texttt{(list (quote } E_1\texttt{) } \ldots\ \texttt{(quote } E_n\texttt{))} \\ \texttt{(quote 0)} &\leftrightsquigarrow \texttt{'0} \\ \texttt{(quote sym)} &\leftrightsquigarrow \texttt{'sym} \end{aligned}$$

Figure 2.1: `quote` special form rules

The language includes a *reader-macro* `'` as syntactic sugar for the special form `quote`. For instance `(quote a-variable)` is equivalent to the form `'a-variable`. Note that the previous list simply shows correspondances between language expressions and their values as shown by the interpreter to the language's user: We always consider `(quote 0)` to be the same as `'0`, as they are only a *reader-macro* appart from

[2]The `eq?` procedure compares two Scheme objects according to their location in memory. As symbols are *interned* in a symbol table, two symbols are `eq?` if the characters in their representations are the same. On the other hand, the list constructor creates a new object dynamically. However, the `equal?` procedure compares two objects based on their structure at run-time.

eachother. We represent Scheme data holding S-expressions by prepending the `'` character to its source-code representation. For instance, `'3`, `'var`, `'()` and `'(1 . 2)` all represent S-expressions. The character `'` is then used both as syntactic sugar for the `quote` special form as well as a character prepending textual token in order to represent symbol and pair data types.
In addition, the special form `quasiquote` can be used to ease S-expression construction. The form is similar to the `quote` special form but, allows the use of the `unquote` special form. *Unquoting* a token simply *cancel out* the effect of a `quasiquote` operator. As with `quote` and its reader-macro `'` the reader-macros `` ` `` and `,` are used, respectively, for the `quasiquote` and `unquote` special forms.

$$
\begin{aligned}
\texttt{(quasiquote sym)} &\leftrightsquigarrow \texttt{'sym} \\
\texttt{(quasiquote 0)} &\leftrightsquigarrow \texttt{'0} \\
\texttt{(quasiquote (unquote } E\texttt{))} &\leftrightsquigarrow E \\
\texttt{(quasiquote (}E_1\ \ldots\ E_n\texttt{))} &\leftrightsquigarrow \texttt{(list (quasiquote } E_1\texttt{) ... (quasiquote } E_n\texttt{))}
\end{aligned}
$$

Figure 2.2: `quasiquotequote` and `unquote` special form rules

Finally, the reader macro `,@`, said *unquote-splice*, splices a list of S-expression, inside a quasiquote use. For instance, if `arg1` is a variable containing the list `'(`a_1 `...` a_n`)`, then `` `(proc ,@`$arg1$`) `` yield the S-expression `'(proc` a_1 `...` a_n`)`.

Summarizing the different reader-macros and special forms used:

$$
\begin{aligned}
\texttt{'sym} &\leftrightsquigarrow \texttt{(quote sym)} \\
&\leftrightsquigarrow \texttt{'sym} \\
\texttt{`sym} &\leftrightsquigarrow \texttt{(quasiquote sym)} \\
&\leftrightsquigarrow \texttt{'sym} \\
\texttt{`,sym} &\leftrightsquigarrow \texttt{(quasiquote (unquote sym))} \\
&\leftrightsquigarrow \texttt{sym} \\
\texttt{'(sym}_1\ \texttt{sym}_2\texttt{)} &\leftrightsquigarrow \texttt{(list 'sym}_1\ \texttt{'sym}_2\texttt{)} \\
&\leftrightsquigarrow \texttt{'(sym}_1\ \texttt{sym}_2\texttt{)} \\
\texttt{`(sym}_1\ \texttt{sym}_2\texttt{)} &\leftrightsquigarrow \texttt{(list `sym}_1\ \texttt{`sym}_2\texttt{)} \\
&\leftrightsquigarrow \texttt{'(sym}_1\ \texttt{sym}_2\texttt{)} \\
\texttt{`(sym ,@a-list)} &\leftrightsquigarrow \texttt{'(sym a-list}_1\ \texttt{a-list}_2\ \texttt{...}\ \texttt{a-list}_n\texttt{)}
\end{aligned}
$$

Figure 2.3: reader-macro and special form for S-expression creation

Abstract Syntax Trees

In most programming languages, the parsing of a program creates a concrete representation of the program's tokens (concrete syntax tree or parse tree). The parse tree conforms to the context-free grammar of the language. For instance, the parse tree of a `C` program would include an individual token for each bracket encountered. Such information can be compacted, with further analysis, in an abstract representation of the code: a minimal representation of the code's meaning. With Scheme's syntax, a single pass on the source code is needed to create the abstract representation of the code. Indeed, as the S-expression grammar is LL(1), a single *look-ahead* character is needed to parse parenthesized groups of tokens. As groups of atoms are already in their minimal representation as an S-expression, the AST can be constructed directly.

Let's consider the following S-expression and its corresponding parse tree:

```
(* (+ a b) 3)
```

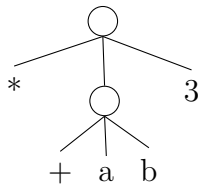


Figure 2.4: S-expression source-code representation and its AST representation

As a more complex example, we consider the following piece of code and its AST:

```
(let ((a 0))
   (let ((b 1))
     (+ a b)))
```

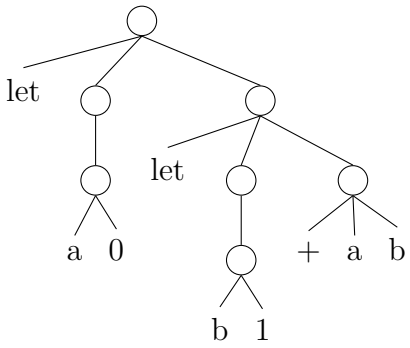


Figure 2.5: A Scheme's S-expression and its AST representation

The parsing phase of a Scheme program is a simple transformation from a textual representation of the S-expression to a valid and manipulable representation of that S-expression using primitive Scheme data types. The compilation phase then receives raw Scheme data to work with.

The following code constructs the previous S-expression as raw Scheme data:

```
(list 'let
      (list (list 'a '0))         '(let ((a 0))
      (list 'let                     (let ((b 1))
            (list (list 'b '1))        (+ a b)))
            (list '+ 'a 'b)))
```

Figure 2.6: Manually constructed S-expression representation and its representation as given by the interpreter

The procedural representation of S-expressions using raw Scheme value is another way to visualize the same AST. This feature is directly linked to the macro system: transformations used during the expansion phase are nothing more than usual Scheme procedure manipulating S-expressions. Similarly, the core forms of the language work directly on the AST.

Core forms

A core form is a parenthesized group of S-expressions starting with some special keyword. They represent the built-in constructs of the language and have their own unique semantics when recognized by the compiler. When compiling a core form, the compiler uses the AST representation of that core form use to generate the compiled code.
We briefly introduce some important core forms of the language, and explain their run-time semantics.

`begin` Core Form

The `begin` core form is used as a sequencing operator.

BEGIN :: `(begin` $expr_1$ $expr_2$ `...)`
$expr_1$, $expr_2$, ... : **Sexp**

Figure 2.7: `begin` core form

Each expression is sequentially evaluated, and the last value obtained is returned.

`if` Core Form

The `if` core form is the primitive branching operator.

IF :: `(if` $expr_{\texttt{cond}}$ $expr_{\texttt{true}}$ $expr_{\texttt{false}}$`)`

$expr_{\text{cond}},\ expr_{\text{true}},\ expr_{\text{false}}$: **Sexp**

Figure 2.8: `if` core form

After an evaluation of the first expression $expr_{\text{cond}}$, if the result does not evaluate to `#f` (read *false*), then the $expr_{\text{true}}$ expression is evaluated. Otherwise, the expression $expr_{\text{false}}$ is evaluated.

`lambda` Core Form

The `lambda` core form is used to create new procedures, stored as closures in memory.

LAMBDA :: `(lambda (`id_1 ...`)` $body_1$ $body_2$ ...`)`
| `(lambda (`id_1 ... `.` $rest$`)` $body_1$ $body_2$ ...`)`
| `(lambda` $rest$ $body_1$ $body_2$ ...`)`

$id_1,\ ...,\ rest$: **Symbol**

$body_1,\ body_2,\ ...$: **Sexp**

Figure 2.9: `lambda` core form

The form `(lambda ()` $body_1$ $body_2$ ...`)` creates a procedure without parameters.

The form `(lambda (`id_1 ... id_n`)` $body_1$ $body_2$ ...`)` creates a procedure of $n \geq 1$ arguments.

The form `(lambda (`id_1 ... id_n `.` $rest$`)` $body_1$ $body_2$ ...`)` creates a procedure of *at least* n arguments. The extra parameters are then bound as a list to the $rest$ symbol.

Finally, the form `(lambda` $rest$ $body_1$ $body_2$ ...`)` creates a procedure with a variable number of arguments, which are bound as a list to the $rest$ symbol.

As Scheme supports higher-order procedures, closures can be used as actual data by themselves and therefore, can be used as an argument to other procedures.

Procedure Application Core Form

We can consider procedure application as a core form itself, even if no special keyword is reserved. Any S-expression starting with an S-expression that is neither

a reserved keyword nor an identifier bound to a macro transformer is a procedure application.

APPLICATION :: `(`$expr_1$ $expr_2$ `...)`
$expr_1,\ expr_2,\ ...$: **Sexp**

Figure 2.10: application core form

`let` Core Form

The `let` core form allows for local binding definitions.

LET :: `(let ((`id_1 $value_1$`) ...)` $body_1$ $body_2$ `...)`
id_1 : **Symbol**
$value_1$: **Sexp**
$body_1,\ body_2,\ ...$: **Sexp**

Figure 2.11: `let` core form

Every symbol id_i is bound to the value given by the evaluation of $value_i$, in the lexical context used for the body's evaluation. Multiple variants of the form exist, such as `let*` which binds the identifiers sequentially, and the `letrec` form that allows for recursive binding definitions and mutually recursive procedure definitions.

Some Scheme implementations implement the `let` special form in terms of `lambda` and application. Indeed, the form's semantics is equivalent to an applied `lambda` expression.

`(let ((`id_1 $value_1$`) ...)` $body_1$ $body_2$ `...)`
$\Longleftrightarrow$
`((lambda (`id_1 `...)` $body_1$ $body_2$ `...)` $value_1$ `...)`

Figure 2.12: `let` form and `lambda` equivalence

`define` Core Form

The `define` core form is used for global binding definition.

DEFINE :: `(define` id_0 $expr$`)`
| `(define (`id_0 id_1 ...`)` $body_1$ $body_2$ ...`)`
| `(define (`id_0 id_1 ... `.` $rest$`)` $body_1$ $body_2$ ...`)`

id_0, id_1, ..., $rest$: **Symbol**

$body_1$, $body_2$, ... : **Sexp**

Figure 2.13: `define` core form

The form `(define` id $expr$`)` evaluate $expr$ and binds the identifier id to that value. The S-expression `(define (f x y) (+ x y))` can be used as a syntactic sugar for `(define f (lambda (x y) (+ x y)))`, thus creating a new named procedure.

`(define (`id_0 id_1 `... .` $rest$`)` $body_1$ $body_2$ ...`)`
$\Longrightarrow$ `(define` id_0 `(lambda (`id_1 `... .` $rest$`)` $body_1$ $body_2$ ...`))`
`(define (`id_0 id_1 `...)` $body_1$ $body_2$ ...`)`
$\Longrightarrow$ `(define` id_0 `(lambda (`id_1 ...`)` $body_1$ $body_2$ ...`))`

Figure 2.14: `define` core form's syntactic sugar

When the form is used in a local context, it acts as a local binding definition.

`define-macro` Core Form

The `define-macro` core form is used to implement syntactic extensions to the language. They are discussed in the next section.

DEFINE-MACRO :: `(define-macro (`id_0 id_1 ...`)` $body_1$ $body_2$ ...`)`
| `(define-macro (`id_0 id_1 ... `.` $rest$`)` $body_1$ $body_2$ ...`)`

id_0, id_1, ..., $rest$: **Symbol**

$body_1$, $body_2$, ... : **Sexp**

Figure 2.15: `define-macro` core form

Macro Application Core Form

We can consider macro applications as a core form, even if no special keyword is reserved. Any parenthesized expression starting with an identifier bound to a macro

transformer has the same semantics.

$$\begin{aligned} \text{MACRO-APPLICATION} :: & \texttt{(}id\ expr_1\ \texttt{...)} \\ id : & \textbf{Symbol} \\ expr_1,\ expr_2,\ \ldots : & \textbf{Sexp} \end{aligned}$$

Figure 2.16: macro application core form

The application of a macro transformer to some expressions executes the transformer at compile time, substituting the core form use by the transformation's result.

Syntactic extensions will be discussed in detail, in the following section.

Macro Transformers

Macro transformers are Scheme procedures taking S-expressions as arguments and returning a new one. As the main distinctive feature, those transformations are executed at *compile-time*.

Definition 6 (Macro Transformers)**.** *A macro transformer* τ : $\boldsymbol{Sexp} \rightarrow \boldsymbol{Sexp}$ *is a procedure, executed at compile-time, which transforms the program's AST into a new one. A macro transformer defines a syntactic extension to the core language.*

Recall that code executed at compile-time is handled by the preprocessor during the expansion phase of the compilation or evaluation pipelines. Run time, on the other hand, is the execution of the expanded program.

Defining Macro Transformer

To define new macro transformers, different forms have been introduced to the language's syntax. The `define-macro` form was directly inspired by the `defmacro` special form, available in Lisp dialects. The form was included in some Scheme implementation such as MIT Scheme and Gambit Scheme. As a simple example, we consider the following transformer's definition:

```
(define-macro (a-new-macro arg1 arg2)
  `(let ((a ,arg1))
     (let ((b ,arg2))
       (+ a b))))
```

Figure 2.17: macro definition using the `define-macro` core form

The previous definition bounds the identifier `a-new-macro` to the following procedure, of type **Sexp** → **Sexp** → **Sexp**.

```
(lambda (arg1 arg2)
  `(let ((a ,arg1))
     (let ((b ,arg2))
       (+ a b))))
```

Figure 2.18: The bound transformer from the previous definition

As a way to introduce hygiene to the language specification, the `define-syntax` form, alongside the `syntax-rules` special form for which it was conceived, was added to the fifth Scheme revised report (R5RS) [ADH+98]. The form is available in the Gambit ecosystem but, without support for hygiene. That alternative form-defining transformer takes the whole S-expression as a single parameter of type **Sexp**. The following definition is equivalent to the previous one, if it was called with two arguments [3].

```
(define-syntax a-new-macro-syntax
  (lambda (sexp)
    (let ((id    (list-ref sexp 0))
          (arg0 (list-ref sexp 1))
          (arg1 (list-ref sexp 2)))
      `(let ((a ,arg0))
         (let ((b ,arg1))
           (+ a b))))))
```

Figure 2.19: Almost equivalent macro definition using Gambit unhygienic `define-syntax` core form

Macro application semantics differs from regular procedure application by the compiler's handling of actual parameters. The following subsection summarizes the two different calling mechanisms.

Handling Formal Parameters

Procedure Application

For usual procedure applications, Scheme uses the call-by-value mechanism: when applying a procedure to some expressions $expr_1$, $expr_2$, ... $expr_n$, each of the actual parameter $expr_i$, must be firstly evaluated. The computed values are then stored in memory, typically on the stack, before proceeding with the procedure's body's evaluation. For instance, in the procedure application `(f (+ 1 2) a)` the subexpressions `(+ 1 2)` and `a` must be evaluated before proceeding with the procedure call of `f`. Within the procedure's body, any mutation to one of the formal parameters takes effect on the cells of those formal parameters.

[3] As the forms are unhygienic, name capture can occur in different ways for both forms. This will be discussed later on.

Macro Transformer Expansion

Macro transformers, on the other hand, use a mechanism reminescent of *call-by-name*. As macro transformer procedures are executed at compile time, they cannot have access to the final value the variable will hold at execution time, as it would, obviously, require a complete evaluation of some code that was designed to be run in the final compiled program. Instead, the symbolic representation of each argument is used as arguments to the procedure.

Consider the following program and its AST:

```
(let ()
  (define-macro (m a b)
    (list ('+ b a)))
  (let ((var 1))
    (m var 2)))
```

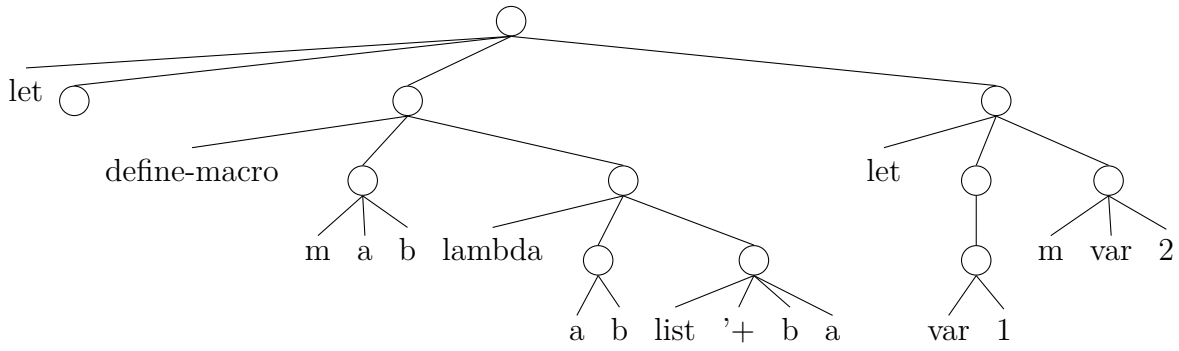


Figure 2.20: A program and its AST representation

The final AST, after macro expansion, will look as such:

```
(let ()
  (let ((var 1))
    (list 2 var)))
```

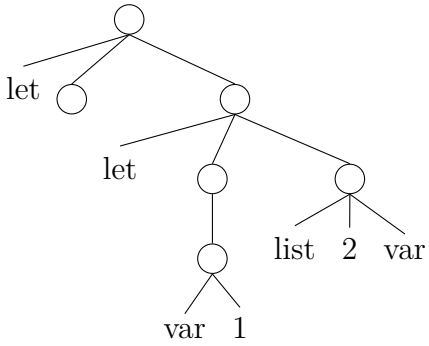


Figure 2.21: expanded AST

From the point of view of the programmer, the semantics of application changes for macro-transformers. In contrast to the *call-by-value* convention, the formal parameters' value is not computed: the system simply moves the source-code pieces around

freely. As a result, macro calls are S-expression substitutions in the abstract syntax tree.

In this chapter, we presented the specific syntax of to the Scheme Language. After a summary of the language's common constructs and features, we formalized the source code representation and source code transformations, by introducing the concepts of S-expressions and macro transformations. We focused on the relation between Scheme's source code representation and its abstract syntax trees and discussed the calling convention used for procedures and macro transformers. The next section summarizes the hygiene problem, which is the focus of our contribution.

Chapter 3

Hygiene

Allowing direct transformations of the AST gives a lot of power to the programmer. However, when a variable occurs freely in the transformer's procedure, or actual parameters, name shadowing can occur in the fully expanded code, as the identifier takes the lexical context of its final position in the abstract syntax tree. Errors resulting from accidental name capture cannot be caught, as the compiler is oblivious to the lexical context and, cannot discern deliberate name capture use cases. Errors are either hidden or obfuscated: the programmer must then take care of the whole program's safety with each and every macro transformer definition and use. They must build their own safety net.

Hygienic macro systems ensure that no *accidental* name capture can occur. In many cases, however, deliberate name capture can be a powerful tool: most systems then allow explicit name capture as a manual process. It has been shown that, in practice, most transformers do not use deliberate name capture [KFFD86]. Macro systems enforcing hygiene change the default behaviour of the compiler, making name capture an explicit process instead of an implicit one.

The Lisp programming language was one of the first popular languages with heavy emphasis on its macro system, and hygiene has been studied thoroughly in that context [KFFD86] [BR88] [Dyb92].

Hygienic macro systems exist in multiple language implementations, such as Rust, Racket, and some Lisp implementations. The Scheme programming language's specification includes support for hygienic macros since it's fourth revised report. Most Scheme implementation, in practice, do not implement the full specification's language, as they often specialize in specific applications. The Gambit implementation did not offer such a feature, as an efficient hygienic system would have required a rework of the compiler's core.

Macro-expansion and silent errors

The section begins by showing some common problems arising during the process of macro transformer creation, bringing insight on the name capture problem and its prevalence.

Multiple Evaluation

As a common use of syntactic extensions, it is possible to save the creation cost of a new closure on the heap, by the use of macro transformers instead of procedures. As transformations occur at compile-time, the S-expressions given as parameters aren't evaluated: only their representation is available. Then, to copy the same S-expression at different places in the AST could cause this specific piece of code to be executed multiple times during the run-time phase. As canonical solution, one can create a new local variable, binding the S-expression's value at execution time. The new identifier can then be used as a placeholder for the S-expression.

Let's consider the following piece of code.

```
(pretty-print
  (double (ackermann 4 1)))
```

If `double` is defined as a usual procedure, it could have been defined by the following top-level definition:

```
(define (double arg1)
  (+ arg1 arg1))
```

To create a macro transformer which does the same job, one might try the naive following definition:

```
(define-macro (double arg1)
  `(+ ,arg1 ,arg1))
```

However, this definition for `double` causes some unexpected behaviour: the call to `(ackermann 4 1)` is evaluated twice in the final program. Indeed, at compile time, the procedure `(lambda (arg1) `(+ ,arg1 ,arg1))` is called with the S-expression `'(ackermann 4 1)` as an argument, which yields the S-expression `'(+ (ackermann 4 1) (ackermann 4 1))` as a result. We can inspect the resulting AST for some additional insight on the problem.

```
(pretty-print
  (double (ackermann 4 1)))
```

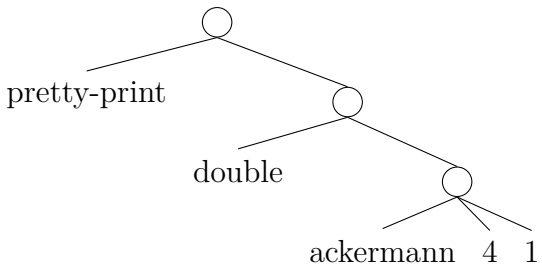


Figure 3.1: Original S-expression and AST

```
(pretty-print
  (+ (ackermann 4 1)
     (ackermann 4 1)))
```

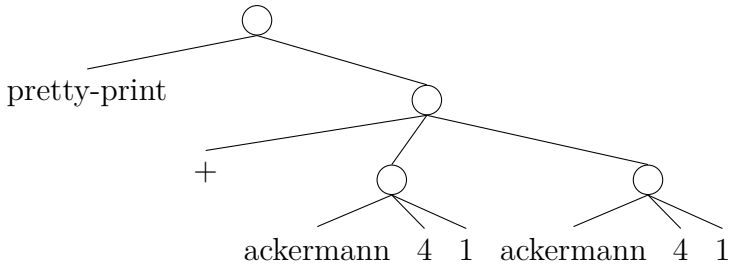


Figure 3.2: expanded S-expression and AST

We then notice the problem easily: The S-expression passed as a parameter appears twice in the expanded code. To create a macro which evaluates `(ackernann 4 1)` exactly once, we can introduce a new identifier to bind its evaluation's result.

```
(define-macro (double arg1)
  `(let ((result ,arg1))
     (+ result result)))
```

At execution time, the result of `(ackermann 4 1)` is stored in the local variable `result`. The value can then be used for the addition without the need to compute it again. After expansion, the resulting AST is as such:

```
(pretty-print
  (let ((result (ackermann 4 1)))
    (+ result result)))
```

The introduction of new identifiers during macro expansion is a common technique to avoid multiple executions of a piece of code. Binding introduction by macro substitution can be useful for many other features as well. As a second example, we consider the anaphoric `if` form, which binds the condition's value to a new identifier `it`, available in both branches.

```
(define-macro (aif condition true false)
  `(let ((it ,condition))
     (if it
         ,true
         ,false)))
```

Figure 3.3: anaphoric `if` form

The special form allows for each branch to refer to the condition's value using the identifier `it`. As an example, we can consider the following use:

```
(define (is-even? n)
  (if (= (modulo n 2) 0)
      n    ;; will be treated as True by if
      #f))

(aif (is-even? (ackermann 4 1))
     (+ it 1)
     it)
```

In this case, the variable `it`, occurring freely in `(+ it 1)`, is meant to be used as a placeholder for the identifier defined within the let form of the S-expression returned by the transformer.

As a last example, we consider a `define-structure` form which expands to multiple new declarations:

```
(define-macro (define-structure name . fields)

  (define (append-symbol . syms)
    (string->symbol (apply string-append
                           (map symbol->string syms))))

  (define (create-constructor name fields)
    `(define (,(append-symbol 'make- name) ,@fields)
       (vector ,@fields)))

  (define (create-getters name fields)

    (define (create-getter name field i)
      `(define (,(append-symbol name '- field) obj)
         (vector-ref obj ,i)))

    (map (lambda (field i) (create-getter name field i))
         fields
         (iota (length fields))))

  `(begin
    ,(create-constructor name fields)
    ,@(create-getters name fields)))
```

Figure 3.4: `define-structure` example macro

```
(define-structure position x y)
(define (distance-origin pos)
  (sqrt
    (+ (expt (position-x pos) 2)
       (expt (position-y pos) 2))))
```

```
(begin
  (define (make-position x y)
    (vector x y))

  (define (position-x obj)
    (vector-ref obj 0))

  (define (position-y obj)
    (vector-ref obj 1)))

(define (distance-origin pos)
  (sqrt
    (+ (expt (position-x pos) 2)
       (expt (position-y pos) 2))))
```

Figure 3.5: A simple program using `define-structure` and its expansion

We now show how the insertion of new identifiers, by substitution in the AST, can result in a problem known as accidental name capture.

Name Capture

Upon the substitution of an identifier in the abstract syntax tree, it may get captured by the local lexical context of its final position, thus, changing the variable it refers to. It follows that we cannot infer, at first glance, the lexical context of an S-expression used as an argument for a transformer without further analysis. The problem is equivalent to the substitution problem in lambda calculus and was thoroughly studied in that context [KFFD86].

We firstly formalize the problem before a brief summary of different solutions theorized.

Definition 7 (name capture)**.** *Name capture occurs when a binding is inserted in an S-expression where the same name is used for another binding, thus shadowing its original definition.*

Let's consider a procedure `inc-proc`, which increments its argument by one, as well as a macro-transformer `inc-macro` with the same purpose.

```
(define (inc-proc arg1)
  (let ((const 1))
    (+ const arg1)))

(define-macro (inc-macro arg1)
  `(let ((const 1))
     (+ const ,arg1)))
```

Figure 3.6: Two seemingly equivalent ways to represent the same computation

However, even if those two approaches *seem* equivalent, the following fragment of code evaluates to *false* (`#f`), disproving the equivalence.

```
(let ((const 42))
  (= (inc-proc const)
     (macro-proc const)))
```

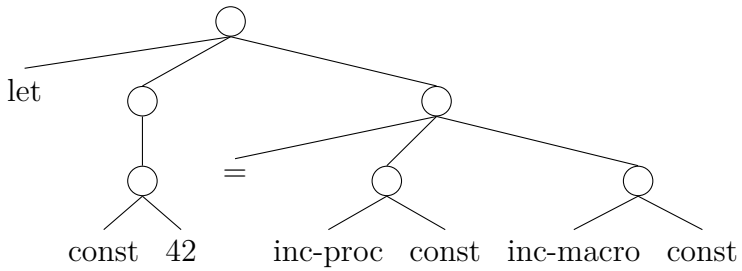


Figure 3.7: Test for computation equivalence

Inspecting the code's expansion, the problem becomes clearer:

```
(let ((const 42))
  (= (inc-proc const)
     (let ((const 1))
       (+ const const))))
```

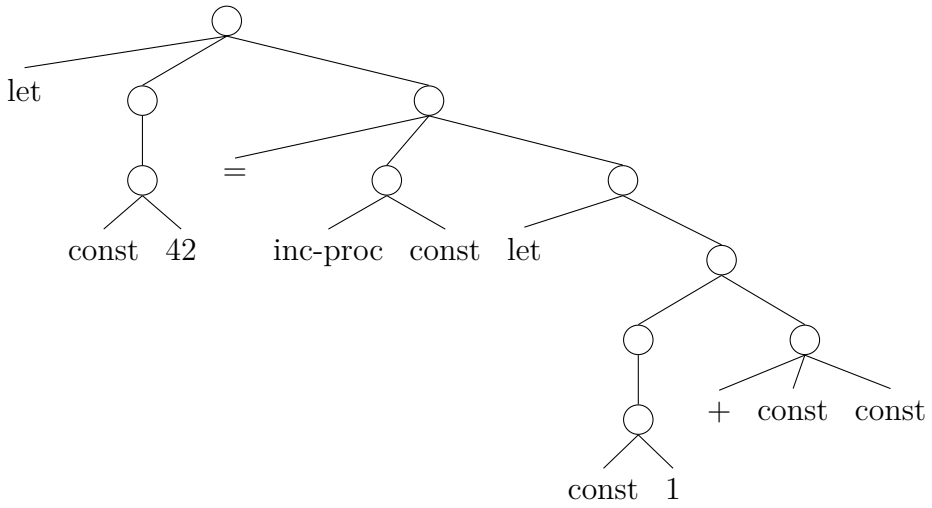


Figure 3.8: Test for computation equivalence (expanded)

We notice that the transformation inserted the `const` symbol in the original AST. However, the identifier is now bound to the value `1`, as it appears in the lexical context of the inner local `let` form introduced by the transformation: the inserted identifier has been, effectively, *shadowed* by the introduced identifier.

This problematic instance can be manually fixed with the use of a unique symbol as name for the introduced identifier. Scheme offers the `gensym` : **Symbol** → **Symbol** procedure for this purpose. This procedure allows for the dynamic creation of *uninterned* symbols, which, in opposition with interned symbols, are guaranteed to be unique [1].

Definition 8 (uninterned symbol)**.** *Uninterned symbols are created dynamically to guarantee their uniqueness throughout the program. The procedure* `gensym` : ***Symbol*** → ***Symbol*** *creates a fresh symbol, named using, as a hint, the symbol passed as an argument. Uninterned symbols are noted* `"#:foo42"`*: the symbol given as a hint, followed by a unique integer and prefixed by* `"#:"`*.*

We can then redefine our previous transformer to make use of this functionality.

```
(define-macro (inc-macro arg1)
  (let ((const-id (gensym 'const)))
    `(let ((,const-id 1))
       (+ ,const-id ,arg1))))
```

[1] Two interned symbols with the same structure (name) refer to the same location in memory, they are `eq?`, as well of `equal?`. However, two uninterned symbols never share the same location nor the same structure. They cannot be `eq?` nor `equal?`, thus always differentiable.

With the guarantee that the newly introduced identifier uses a different symbol than every other defined identifier, no accidental shadowing can occur.

```
(let ((const 42))
  (= (inc-proc const)
     (let ((#:const529 1))
       (+ #:const529 const))))
```

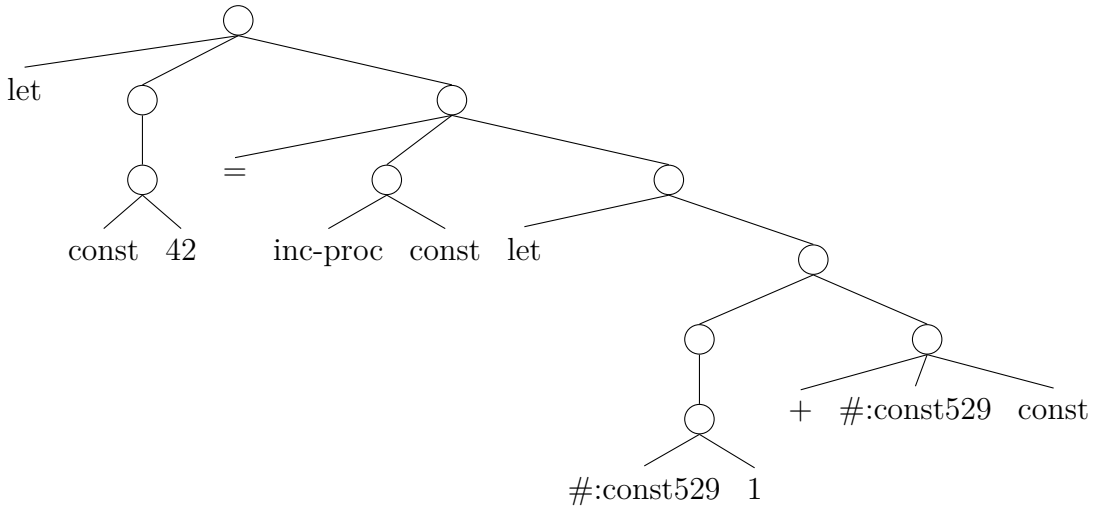


Figure 3.9: Test for computation equivalence using `gensym` (expanded)

However, name capture can happen even if no binding definition appears in the transformer itself. For instance, a macro-transformer might expect the `if` identifier to be bound to the conditional core form but, the identifier might have been redefined depending on the location of the macro-procedure call in the AST.

For instance, the following macro might cause problems:

```
(define-macro (a-macro a b)
  `(+ ,a ,b))
```

In the following code, even if both macro calls execute the same expansion, with the same S-expressions as parameters, we can expect different results, as the lexical context of the second call is different.

```
(begin
  (pretty-print (a-macro 0 1))     ; -> print 0
  (let ((+ (lambda (a b) "plus")))
    (pretty-print (a-macro 0 1)))) ; -> print "plus"
```

Using a procedure instead would have captured the initial `+` definition within the created closure, yielding the exact same result both times. We can conclude that any identifier appearing freely in either the transformer body, or inside any actual parameter can cause accidental name capture.

Breaking Hygiene

Referential transparency disallows any kind of binding introduction. However, name capture can be a useful tool. Considering the mentioned anaphoric `if` form as an example, we want every occurrence of the `it` identifier, occurring within the last two forms argument, to refer to the binding introduced by the expansion. When hygiene was firstly theorised, some critisism was made about its necessity. For instance, Doug Hoyte, well known in the community for his work on Common Lisp, qualified the safety measure to be a "beginner's safety guard-rail that serve only to reduce what one can do with [macros]." [Hoy] [Hoy08].

Modern hygienic macro systems, in most cases, offer some mechanism to break the hygiene condition manually, for such a purpose. The system thus simply changes the default behaviour of the compiler regarding name capture: it must be made explicit in the code while providing automatic safety for macros that do not require it. We consider that a good solution for the hygiene problem would allow deliberate name capture in a consistent and practical manner.

The previous chapter formalized the hygiene problem, mainly in the context of the Scheme language. After some examples highlighting problems that can occur while using macro, we focus on the problem known as accidental name capture. We then present a popular solution used to manually avoid accidental name capture, based on name obfuscation with the use of uninterned symbols. We finally expressed the need for a way to break hygiene manually, in some contexts. The following chapter summarizes some of the approaches theorised to preserve hygiene, specifically for the Lisp and Scheme languages.

Chapter 4

Automatic Hygiene

The process of macro transformer writing requires the programmer to build their own safety net, or else, to be particularly careful of each use. As a consequence, the code can become easily obfuscated. Solutions to the hygiene problem provide automatic mechanisms to ensure that no accidental capture can occur. As a result, the default paradigm of the compiler regarding name capture is inverted: it must now be made explicit in the code.

This chapter describes some of the procedural approaches that have been suggested ensuring hygiene, for Lisps and Scheme dialects.

KFFD Algorithm

The KFFD [KFFD86] algorithm was one of the first automatic and hygienic macro systems theorized for the Lisp programming language. We summarize the algorithm and some terminology used by the authors.

Considering a macro-transformer `m`, the form `(m` $\texttt{expr}_1$ `...` $\texttt{expr}_n$`)` represents a *syntactic extension* to the language as the special form `m` is not part of the core language's syntax.

Definition 9. *syntactic extension*
Let `m` *be a macro transformer. The form*
`(m` $\texttt{expr}_1$ `...` $\texttt{expr}_n$`)`
is a syntactic extension. The S-expressions $expr_1$, ..., $expr_n$ *are called the syntactic scope of the extension.*

Definition 10. *occurrence*
An S-expression s is said to occur in a syntax tree if it is a tree that is not nested within the syntactic scope of a syntactic extension. Let m_1 *and* m_2*, two macro transformers, then the syntactic extension* `(`m_1 `0 1)` *occurs within the S-expression* `(lambda (x) (`m_1 `0 1))`*, but, it does not occur within* `(`m_2 `0 (lambda (x) (`m_1 `0 1)))`*.*

The notion of occurrence emphasizes the dual nature of every expression: they may be considered as elements of the core language or as a syntactic extension, depending on the context. As macro-transformers move AST nodes around, we can only be sure of the interpretation of a particular expression if it does not appear in any syntactic extension. In the above example, m_2 may extract parts of its arguments to construct the expansion arbitrarily.

Definition 11. *transcription*
A transcription is the result of applying a macro transformer to an occurrence of a syntactic extension.

They then define the hygiene condition for macro-transformers, verbatim as :

Definition 12 (Hygiene Condition for Macro Expansion)**.** *Generated identifiers that become binding instances in the fully expanded program must only bind variables that are generated at the same transcription step (HC/ME).*

The choice of identifier's name for a procedure doesn't matter in the program. This concept is known as α-equivalence. We call α-conversion, the process of substituting a binding's name, as well as every identifier that refers to it. However, one cannot simply use α-conversion at every transformation step. Indeed, it has been shown to be impossible: it is not immediately obvious which identifiers are to be free or bound after a specific transcription step. In any case, name capture can be desirable in some contexts and an identifier used as such must not be renamed independently.

The solution proposed is then to keep track of each identifier's origin. To attach some lexical information to identifiers, *time-stamps* are used to reason about their transcription phases. Time-stamps are implemented using simple non-negative integers. As downsides to the algorithm, bindings that the programmer wants to expose as public must be manually noted as such.

Syntactic Closure

Syntactic closures were introduced by A.Bawden and J.Rees in 1988 as a mechanism to control hygiene condition in a more idiomatic manner [BR88].

Recall that the compiler is oblivious to the lexical context of identifiers at compile time. The strategy proposed uses an analog to the usual closures returned by lambda abstraction: a binding environment coupled with an expression to work with. As closures are a solution to the scoping problem at run-time, syntactic closures aim to solve the same problem at compile time.

The new construct is meant to be used wherever S-expressions can be used. Instead of providing meaning to an expanded expression, depending on its final position, syntactic closures carry their own context.
In the following example, `obj-exp` and `list-var` are bound to syntactic closures which expand, respectively, to `(cadr exp)` and `(caddr exp)`, within the syntactic

context provided as parameters to the expander (`syntactic-env`). The whole transformer returns a new syntactic closure using the global environment `scheme-syntactic-environment`, where only the core forms are defined.

```
(lambda (syntactic-env exp)
  (let ((obj-exp (make-syntactic-closure
                  syntactic-env
                  '()
                  (cadr exp))))
    (let ((list-var (make-syntactic-closure
                     syntactic-env
                     '()
                     (caddr exp))))
      (make-syntactic-closure
        scheme-syntactic-environment
        '()
        `(set! ,list-var (cons ,obj-exp ,list-var))))))
```

Figure 4.1: An expander for a `push` macro using syntactic closures

In whichever context the expander is used, the programmer does not need to worry about accidental name capture for the `set!` identifier, for instance, as the expression `` `(set! ,list-var (cons ,obj-exp ,list-var)) `` is always expanded in the context for which `set!` is bound to the built-in `set!` (the global expansion context `scheme-syntactic-environment`).

Using syntactic closures, deliberate name capture is direct: a simple list holds the name of every variable to be captured. We consider the following expander, where any occurrence of the variable name `throw` within `body-exp` is meant to be captured by the procedure used by the `call-with-current-continuation` call.

```
(lambda (syntactic-env exp)
  (let ((body-exp (make-syntactic-closure
                   syntactic-env
                   '(throw)
                   (cadr exp))))
    (make-syntactic-closure
      scheme-syntactic-environment
      '()
      `(call-with-current-continuation
        (lambda (throw) ,body-exp)))))
```

Figure 4.2: A transformer for a `catch` macro using syntactic closure and deliberate name capture

Pattern matching based hygiene

The `syntax-rules` form was incorporated to the language's specification as a high-level pattern matching utility, ensuring hygiene automatically. The form allows for the deconstruction and reconstruction of source code. Hygiene is ensured by the use of pattern variables, using a specific environment for pattern substitutions.

We can consider the following use of `syntax-rules` as an example:

```
(define-syntax let*
  (syntax-rules ()
    ((_ () expr1 expr2 ...)
     (let () expr1 expr2 ...))
    ((_ ((id1 value1) (id2 value2) ...) expr1 expr2 ...)
     (let ((id1 value1))
       (let* ((id2 value2) ...) expr1 expr2 ...)))))
```

```
(define-syntax simple-cond
  (syntax-rules (else)
    ((_ (else expr1 expr2 ...) rst1 rst2 ...)
     (error "simple-cond: else keyword can only
             be used in the last clause"))
    ((_ (condition expr1 expr2 ...) rst1 rst2 ...)
     (if condition
         (begin
           expr1
           expr2 ...)
         (simple-cond rst1 rst2 ...)))
    ((_ (else expr1 expr2)
     (begin expr1 expr2)))
    ((_ (condition expr1 expr2))
     (simple-cond (condition expr1 expr2) (else (void))))))
```

Later on, the `syntax-case` special form was introduced, alongside more primitive functionalities to manipulate and create *syntax* [Dyb92]. The pattern-matching language is the same as the `syntax-rules` specification. The notion of *syntax* and *syntax-object* are at the core of the chosen macro system and will be formalized thoroughly. In modern Scheme compilers, the `syntax` and `syntax-case` forms can be implemented on top of a lower-level system. Some other systems implement an hygienic layer on top of their system's core or, implement hygiene directly within the `syntax-case`'s form, with some possible implementation of *pattern variables* [CT10].

Our algorithmic choice manipulates syntax directly and constitutes the low-level core of the macro systems. More complex constructs such as those pattern matching utilities are then built on top of it.

This chapter briefly described different strategies theorized to ensure hygiene, notably the KFFD algorithm and syntactic closures. We followed with a description of some pattern-matching-based hygiene strategy. Those high-level constructs can be implemented themselves using a lower-level macro system, such as the two we presented, or provide hygiene directly with a more complex implementation of *pattern-variables*. Most of the modern Scheme systems with support for hygiene implements the `syntax` and `syntax-case` special forms but, their underlying details differ. The following chapter explains in detail the algorithm we chose. The macro system we implemented uses lower-level primitives to manipulate `syntax`. Higher-level constructs, including `syntax-case` and `syntax-rules`, were built on top of it.

Chapter 5

Set of Scopes

The Set of Scopes algorithm was introduced in Racket in 2016 [Fla16] to provide primitive forms for creation and manipulation of *syntax*. The algorithm achieves referential transparency by tracking the lexical context of every identifier occuring in the code. To do so, a set of *scopes* is attached to every identifier, giving information about which binding the identifier refers to. This extended symbolic representation is named *syntax objects* and will be formalized later on. Upon the introduction of a new binding, a new *scope* object is created to represent the binding site. When *resolving* a particular identifier, it's set of scopes is compared, using simple set comparisons, to determine to which binding it refers. With *syntax*, the compiler has total knowledge of each identifier's original lexical context, even when moved around by expansions. Furthermore, breaking hygiene is possible by manual mutations of the identifier's scope set.

This chapter explains in detail the algorithm's core, as implemented and theorized for the Racket system. We begin with an introduction to the scoping mechanism and a formalization of the data type used.

An Introduction

The use of plain S-expressions as the representation for source code exhibit a two-dimensional nature. An expression can be characterized by its position within another form, combined with that subexpression depth in the tree [HW08]. However, when moved around by macro expansions, only the final position of an S-expression remains after the transformation. To memorize the lexical context of every expression, we then introduce the notion of *scopes*. For every binding definition, a new scope is created, and propagated through the subexpressions that are in lexical scope of our new definition. To get a visual insight, we can assign a *color* for each *scope* object in our code representation. For the following example, we use a color notation which is greatly inspired by Matthew Flatt's multiple conferences.

```
(let ((x 0))
  (let ((y 1))
    (+ x y)))
```

To ensure hygiene in the system, only the lexical information regarding identifiers

is relevant. By stacking the colors on top of each other, we can represent visually each identifier with its *set of scopes*. Here, the color red is the scope of the outer `let` and the color blue is the scope of the inner `let`. By default, every identifier are in the *core scope* : here, identifiers that are not colored are in the *core scope* only.

```
(let ((x 0))
  (let ((y 1))
    (+ x y)))
```

With each `let` form, we introduce a fresh scope and extend the set of scopes of every identifier appearing in the form's body with it.

In addition, when expanding macro application calls, we also create a fresh scope. As with binding definition, the scope is propagated to every identifier occurring in the expansion's result. Let's consider a macro transformer `(inc x)` which expands into `(let ((y 1)) (+ y x))`. The following code, using the color notation, shows the set of scopes created by the set-of-scopes algorithm after the processing of the two first `let` forms's binding creation:

```
(let ((x 0))
  (let ((y 1))
    (+ x (inc y))))
```

In the next expansion step, we expand the inner `let` form's body and expand the `inc` macro use. For the expansion of the macro transformer, after application of the transformer, we extend every identifier introduced with a unique scope. Here we use the color green as the scope of the identifiers in the `inc` macro transformer's expansion. Note that the macro call's argument `y` is simply moved around.

```
(let ((x 0))
  (let ((y 1))
    (+ x (let ((y 1))
           (+ y y)))))
```

Finally, after expansion of the macro application, we expand its result, expanding our inserted `let` form. We use the yellow color for the newly introduced `let`.

```
(let ((x 0))
  (let ((y 1))
    (+ x (let ((y 1))
           (+ y y)))))
```

Identifiers are compared for equality using their name and set of scopes. We can find the binding of a given identifier by comparing its set of scopes with every other identifier's set within the whole program. During expansion, the algorithm must match each identifier used as reference to the correct binding definition. Considering each identifier appearing in **B** that have the same name as an identifier `id`, the candidate identifier with the biggest set of scopes that is subset of `id`'s set of scopes is the corresponding binding for the reference `id`. The following exemple is used to explain the algorithm's correctness.

Considering the `a` identifier in the following code, which is the result of the expansion process:

```
(let ((a 0))
  (let ((a (+ a 1)))
    (+ a (let ((a 1))
           (+ a a)))))
```

To *resolve* the identifier `a` to its correct binding, we perform set of scopes comparisons with every identifier recorded with the same name. The correct binding for the reference will be the identifier recorded with the biggest set of scopes that is a subset of `a` 's set. In our example, the environment contains three bindings with the same name `a`.

First of all, we know that `a` is not the referenced binding as its set of scopes is not a subset of . Indeed, as `a` 's set of scope include the green binding, we know that any reference to that identifier must have been introduced by the macro expansion associated to that scope as well.

On the other hand, both `a` and `a` are candidate binding for the reference `a` . To find the correct one, we must pick the candidate with the biggest set of scopes. Let's ignore macro transformer for a moment and consider the two bindings `a` and `a` . We can conclude that `a` appears in a `let` form introducing the red color, while `a` appears in a let form introducing the blue color, as well as the let form introducing the red color. As a result, the let form introducing the blue color must be inside the other let form, as it cannot be the inverse. As the two bindings share the same name, `a` is meant to shadow `a` . In our exemple, we can thus conclude that `a` is the binding referenced by `a` .

The algorithm will be stated in detail after an introduction of the required data types.

Source Code Representation and Syntax Objects

We now formalize the notion of scope, set of scopes, as well as syntax objects.

Definition 13 (scope)**.** *A scope* σ_i : ***Scope*** *is a unique comparable object, used to keep track of binding origin. The special scope* s : ***Scope*** *is the core scope.*

Definition 14 (set-of-scopes)**.** *A set of scope* Σ : ***Set Scope*** *is an unordered collection of scopes defined as*

$$\begin{aligned} &\sigma_i : \textbf{\textit{Scope}} \\ &\Sigma : \textbf{\textit{Set Scope}} \\ &\Sigma ::= \{\} \\ &\quad \mid \{\sigma_i\} \cup \Sigma \end{aligned}$$

The symbolic notation's extension, which adds scope information on an identifier, is named syntax-object and is defined as such :

Definition 15 (syntax-object)**.** *a syntax object* x^{Σ} : ***Syntax-object*** *associate a symbol* x : ***Symbol*** *with a set of scope* Σ : ***Set Scope*** *with* ***Syntax-object*** *as a shorthand for the* ***Symbol*** $\times$ ***Set Scope*** *type.*

To simplify type definition, we consider every Scheme object a : **Atom** as syntax objects, where scoping information is irrelevant. We then represent the program's source code using the *syntax* data type.

Definition 16 (syntax)**.** *A syntax* s : ***Syntax*** *is defined by the following inductively defined union type :*

$$\begin{aligned} \textbf{\textit{Syntax}} ::= &\ \textbf{\textit{List Syntax}} \\ \mid &\ \textbf{\textit{Syntax-object}} \end{aligned}$$

We notice the correspondence with our S-expression definition: **Syntax** is equivalent to the **Sexp Syntax-object** type. Then, we can represent extended source code as expressions of type **Sexp Syntax-object**, referred to as *syntax*, or *syntax-objects* (when obvious from the context).

Syntax's Representation Conversion

The procedure `datum->syntax` : **Sexp** $\rightarrow$ **Syntax** is used to create syntax from an S-expression, by extending every object identifier with an empty set-of-scopes. It's functional inverse, `syntax->datum` : **Syntax** $\rightarrow$ **Sexp**, remove scope information from every symbol, yielding a usual S-expression back.

To discern core forms from the user's definitions, we include the special scope s : **Scope**, representing the core scope. The implementation then includes the `datum->core-syntax` : **Sexp** $\rightarrow$ **Syntax** procedure, extending every identifier in a given S-expression with a set of scope containing s.

The Algorithm

The algorithm works in three phases, where each of those requires a single pass on the tree. The first phase transforms plain S-expressions into their extended syntactic representation, using the `datum->syntax` procedure. The second phase register encountered binding definitions while propagating the scopes created to the inner expressions. This phase takes care of macro-expansions while ensuring referential transparency. In the third and final phase, the compiler resolves identifiers by providing unique names depending on the lexical context inferred. In this final pass, lexical information is dropped to allow the conventional pipeline to resume.

This section formalizes the algorithm's second phase, starting with a definition of most of the helper function in use and the structure required. We put focus on the algorithm's core including some details on the Racket's implementation specifics. In the next chapter, we follow with our own implementation, explaining changes made to the Gambit system.

Scope Propagation Θ^+

Recall that an identifier $x_i^{\Sigma_{x_i}}$: **Syntax-object** associates a symbol x_i to the set of scopes Σ_{x_i}. For every new binding site, we create a unique σ_i : **Scope** and extend the set of scopes of every identifier which are in lexical scope of that binding. The scope propagation procedure recursively extends the set of scopes of every identifier appearing in an expression and its subexpressions.

Considering the following expression :

```
(let ((a 0))
  (let ((b 1))
    (+ a b)))
```

A new scope σ_a : **Scope** is created for the binding site of `a` and a new scope σ_b : **Scope** for `b`. Then, every identifier appearing in the first let-form's body (including `let`, `+`, `a` and `b`) must be extended with σ_a, while every identifier within the second one, must also be extended with σ_b. We also extend each newly defined identifier with the scope representing their own binding site.

$$\texttt{(}\mathbf{let}^{\Sigma_{let_a}}\ \texttt{((}\mathbf{a}^{\Sigma_{a_1}\cup\{\sigma_a\}}\ \texttt{0))}\ \texttt{(}\mathbf{let}^{\Sigma_{let_b}\cup\{\sigma_a\}}\ \texttt{((}\mathbf{b}^{\Sigma_{b_1}\cup\{\sigma_a,\sigma_b\}}\ \texttt{1))}\ \texttt{(}+^{\Sigma_{+}\cup\{\sigma_a,\sigma_b\}}\ \mathbf{a}^{\Sigma_{a_2}\cup\{\sigma_a,\sigma_b\}}\ \mathbf{b}^{\Sigma_{b_2}\cup\{\sigma_a,\sigma_b\}}\texttt{)))}$$

We define `add-scope` : **Syntax** $\rightarrow$ **Scope** $\rightarrow$ **Syntax**, represented by $\Theta^+[\![E]\!]_\sigma$, as the transformation that extends every set of scopes of every identifier occurring in the expression E with the scope σ. We define the similar procedure `flip-scope` : **Syntax** $\rightarrow$ **Scope** $\rightarrow$ **Syntax**, represented by $\Theta^{\oplus}[\![E]\!]_\sigma$. which extends the set of scope of every identifier in E which isn't already scoped by σ and remove the scope otherwise. For every new transformer, a new scope is created. When the transformer is applied, the bindings created within the transformer must keep the extra binding,

while identifiers introduced from other transformer, or passed as arguments doesn't.

$\Theta^{+}[\![x^{\Sigma}]\!]_{\sigma}$

$$\frac{}{\Theta^{+}[\![x^{\Sigma}]\!]_{\sigma} = x^{\Sigma \cup \{\sigma\}}}$$

$$\frac{}{\Theta^{+}[\![(E_1 \;\; E2 \;\; ...)]\!]_{\sigma} = (\Theta^{+}[\![E_1]\!]_{\sigma} \;\; \Theta^{+}[\![E_2]\!]_{\sigma} \;\; ...)}$$

$\Theta^{\oplus}[\![x^{\Sigma}]\!]_{\sigma}$

$$\frac{\sigma \notin \Sigma}{\Theta^{\oplus}[\![x^{\Sigma}]\!]_{\sigma} = x^{\Sigma \cup \{\sigma\}}} \qquad \frac{\sigma \in \Sigma}{\Theta^{\oplus}[\![x^{\Sigma}]\!]_{\sigma} = x^{\Sigma \setminus \{\sigma\}}}$$

$$\frac{}{\Theta^{\oplus}[\![(E_1 \;\; E2 \;\; ...)]\!]_{\sigma} = (\Theta^{\oplus}[\![E_1]\!]_{\sigma} \;\; \Theta^{\oplus}[\![E_2]\!]_{\sigma} \;\; ...)}$$

Figure 5.1: Scope propagation

Expansion environment

The expansion environment uses two separate tables to keep track of every identifier defined, as well as the lexical context at each expansion step. The *global binding table* registers associations between new identifier definitions and their initial binding sites. In addition, the *compile time environment* is propagated throughout the expansion, keeping track of variable and macro-transformer definitions according to the current lexical context.

Combining the scoping mechanism with this environment representation is sufficient to ensure referential transparency throughout the expansion phase.

Global Bindings Table

Upon expansion of a new variable or macro definition, a representation for the new binding site is created. The table $\mathbf{B} : (\mathbf{Set\ Binding})$ (using $\mathbf{B}$ as a shorthand for the $\mathbf{B}$ table's type) record every binding $\beta = \texttt{binding}(x^{\Sigma}, \underline{x})$, with $\beta : \mathbf{Binding}$, which associate the symbol $x : \mathbf{Symbol}$ and the set of scopes $\Sigma : \mathbf{Set\ Scope}$ to the unique symbol representing the binding site $\underline{x} : \mathbf{Symbol}$. As the table is global, all the information is kept at all times throughout the last two phases of the algorithm: this allows for precise information from the compiler even if some macro application results in invalid code.

Definition 17 (binding)**.** *A binding* β : ***Binding*** *associates an identifier* x^{Σ} : ***Syntax-object*** *to a unique representation of the binding location (binding key)* $\underline{x}$:

Symbol. *If the binding site is at top level, then* $\underline{x} = x$*, otherwise,* $\underline{x}$ *is an uninterned fresh symbol, as given by the* `gensym` *procedure.*

Definition 18 (Global Binding Table)**.** *The global binding table* $\mathbf{B}$ *is the set of every binding* β *created during the expansion phase.*

$$\begin{aligned}\beta_i &::= \texttt{binding}(x_i^{\Sigma x_i},\ \underline{x_i})\\ \mathbf{B} &::= \{\}\\ &\quad \mid \mathbf{B} \cup \{\beta_i\}\end{aligned}$$

We say that the expression $op[\![E_1]\!]$ yields D_1, noted $op[\![E_1]\!] \implies D_1$, if the computation of $op[\![E_1]\!]$ return the data D_1 as a result.

We define the β^+ procedure, which records a binding in the global binding table, as follows:

$\beta^+[\![E]\!]^{\mathbf{B}}$

$$\frac{\begin{array}{c}\underline{x} = \texttt{gensym}()\\ \beta = \texttt{binding}(x^{\Sigma},\ \underline{x}),\\ \mathbf{B}' = \mathbf{B} \cup \beta\end{array}}{\beta^+[\![x^{\Sigma}]\!]^{\mathbf{B}} \implies (\mathbf{B}',\ \underline{x})}$$

Figure 5.2: binding recording in global binding table

To find the original binding site of an identifier occurrence, its set of scopes is compared to every other identifier for which there exists an association in $\mathbf{B}$.

Let's consider the identifier x^{Σ_x}.
We define Θ as the set of every set of scopes Σ_i associated to an identifier $y_i^{\Sigma_i}$, with $x = y_i$, appearing in a binding of $\mathbf{B}$:

$$\Theta[\![x]\!]^{\mathbf{B}} = \{\Sigma_i \mid \exists \Sigma_i, \underline{y_i},\ \texttt{binding}(x^{\Sigma_i},\ \underline{y_i}) \in \mathbf{B}\}$$

We then define $\Theta_{\subset}$ as the set of every element of Θ that are subsets to Σ_x:

$$\Theta_{\subset}[\![x^{\Sigma_x}]\!]^{\mathbf{B}} = \{\Sigma_i \mid \Sigma_i \subset \Sigma_x \wedge \Sigma_i \in \Theta[\![x]\!]^{\mathbf{B}}\}$$

If $|\Theta_{\subset}| = 0$ then we cannot resolve.
Otherwise, let $\Sigma_{\subset} \in \Theta_{\subset}$, be the greatest set of scope with

$$\forall \Sigma_j \in \Theta_{\subset}, |\Sigma_{\subset}| < |\Sigma_j|$$

We then consider $\underline{x}_{\subset}$ with $\texttt{binding}(x^{\Sigma_{\subset}},\ \underline{x}_{\subset}) \in \mathbf{B}$. If $\exists \Sigma_i \in \Theta_{\subset}$ with $\sigma_{\subset} \not\subset \sigma_i$ then the use of x_x^{Σ} is ambiguous: we cannot resolve. The ambiguity can be caused by broken macros.

Definition 19 (`resolve`)**.** *The procedure* `resolve` : $\mathbf{B} \rightarrow$ ***Syntax-object*** $\rightarrow$ ***Maybe Symbol*** *takes an identifier in argument and compares its set of scopes with the set of every identifier appearing in an entry of* $\mathbf{B}$*.*

$$\frac{\begin{array}{c}\forall \Sigma_i \in \Theta_{\subset}[\![x^{\Sigma_x}]\!]^{\mathbf{B}}, |\Sigma_{\subset}| \geq |\Sigma_i| \\ \forall \Sigma_i \in \Theta[\![x]\!]^{\mathbf{B}}, \Sigma_{\subset} \subset \Sigma_i \\ \texttt{binding}(x^{\Sigma_{\subset}},\ \underline{x}_{\subset}) \in \mathbf{B}\end{array}}{\textsc{resolve}[\![x^{\Sigma_x}]\!]^{\mathbf{B}} \implies \underline{x}_{\subset}}$$

Figure 5.3: binding resolve

Compile Time Environment

When expanding an identifier, the system must ensure that its use is correct, according to the lexical context of its surrounding expression (or top-level). The compile-time environment represents the lexical context and is propagated according to the usual language's scheme.

Definition 20 (compile-time binding)**.** *A compile-time binding* β : ***CTE-Binding*** *represents an occurrence of an identifier in the lexical context. The binding records the occurrence of a binding site* $\underline{x}$ *as a variable or macro transformer. If the compile-time binding represents a macro transformer, the evaluated procedure* $\overline{\tau}$ *is then associated as a plain Scheme procedure.*

$$\begin{aligned}\gamma_i ::= &\{\texttt{cte-binding}(\underline{x_i},\ \texttt{var})\} \\ \mid\ &\{\texttt{cte-binding}(\underline{x_i},\ \texttt{macro}(\overline{\tau_{x_i}}))\}\end{aligned}$$

Definition 21 (compile time environment)**.** *The compile time environment* Γ : (***Set CTE-Binding***) *keep tracks of the lexical context, throughout the expansion phase.*

$$\begin{aligned}\Gamma ::= &\{\} \\ \mid\ &\Gamma \cup \{\gamma_i\}\end{aligned}$$

As previously with the $\mathbf{B}$ table, we use Γ as shorthand for the Γ table's type. We also define γ^+, the procedure that record a binding site, as either representing a variable or transformer definition, in the compile-time environment.

$\gamma^+[\![\underline{x}]\!]^\Gamma$

$$\frac{\gamma = \texttt{cte-binding}(\underline{x},\ \texttt{var}),\ \Gamma' = \Gamma \cup \gamma}{\gamma^+[\![\underline{x}]\!]^\Gamma_{\texttt{var}} \implies \Gamma'}$$

$$\frac{\gamma = \texttt{cte-binding}(\underline{x},\ \texttt{macro}(\overline{\tau})),\ \Gamma' = \Gamma \cup \gamma}{\gamma^+[\![\underline{x}]\!]^\Gamma_{\texttt{macro}(\overline{\tau})} \implies \Gamma'}$$

Figure 5.4: binding recording in compile-time environment

Full Environment

With the combination of those two environments, we can define the operations to record and lookup an identifier's occurrence. We firstly define the γ^+ transformation which records a variable or macro-transformer identifier in the full environment $(\mathbf{B},\ \Gamma)$.

$\textsc{record}[\![x^\Sigma]\!]^{(\mathbf{B},\ \Gamma)}$

$$\frac{\begin{array}{rcl}\beta^+[\![x^\Sigma]\!]^{\mathbf{B}} & \implies & (\mathbf{B}',\ \underline{x}),\\ \gamma^+[\![\underline{x}]\!]^\Gamma_{\texttt{var}} & \implies & \Gamma'\end{array}}{\textsc{record}[\![x^\Sigma]\!]^{(\mathbf{B},\ \Gamma)}_{\texttt{var}} \implies (\mathbf{B}',\ \Gamma')}$$

$$\frac{\begin{array}{rcl}\beta^+[\![x^\Sigma]\!]^{\mathbf{B}} & \implies & (\mathbf{B}',\ \underline{x}),\\ \gamma^+[\![\underline{x}]\!]^\Gamma_{\texttt{macro}(\overline{\tau})} & \implies & \Gamma'\end{array}}{\textsc{record}[\![x^\Sigma]\!]^{(\mathbf{B},\ \Gamma)}_{\texttt{macro}(\overline{\tau})} \implies (\mathbf{B}',\ \Gamma')}$$

Figure 5.5: full binding recording

Finally, we define the LOOKUP procedure, which determines if an identifier should represent a variable or macro-transformer, according to the lexical context we tracked.

$$\frac{\textsc{resolve}[\![x^{\Sigma_x}]\!]^{\mathbf{B}} \implies \underline{x} \quad \texttt{binding}(\underline{x},\ b) \in \Gamma}{\textsc{lookup}[\![x^{\Sigma_x}]\!]^{(\mathbf{B},\ \Gamma)} \implies b}$$

Figure 5.6: identifier lookup

Expansion ϕ

Once the S-expression has been converted to its syntax representation, the expansion procedure `expand` : $\mathbf{B} \to \Gamma \to$ **Syntax** $\to$ **Pair B Syntax**, represented as $\phi[\![E]\!]^{(\mathbf{B}\ \Gamma)}$, transforms some syntax while recording bindings along the way. The procedure takes the full environment as a parameter and yields both the updated $\mathbf{B}$ environment and the final expanded syntax.

Every core form of the algorithm is recognized according to the leading identifier of some parenthesized group of syntax. For special form uses, the leading identifier must resolve to the top-level definition's binding.

Sequencing

Every Scheme program is wrapped in an implicit `begin` core form. The form is used for operation sequencing. For our formalization, we propagate the $\mathbf{B}$ table through each expression's expansion in a sequence, to express that changes to the $\mathbf{B}$ table persist through each individual expansion steps, both in a local context or at top-level.

$$\frac{\begin{array}{c}\phi[\![E_1]\!]^{(\mathbf{B},\ \Gamma)} \implies (B_1,\ E'_1) \\ \phi[\![E_2]\!]^{(\mathbf{B}_1,\ \Gamma)} \implies (B_2\ E'_2) \\ \ldots \\ \phi[\![E_n]\!]^{(\mathbf{B}_{n-1},\ \Gamma)} \implies (B_n,\ E'_n)\end{array}}{\phi[\![\texttt{(begin}\ E_1\ E_2\ \ldots\ E_n\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \implies (\mathbf{B}_n,\ \texttt{(begin}\ E'_1\ E'_2\ \ldots\ E'_n\texttt{)})}$$

Figure 5.7: Sequencing

Lambda Abstraction

The lambda special form creates a new closure while introducing formal parameters as local bindings in the local context of the body expression. As for every new variable definition, a new scope σ : **Scope** is created to represent the lexical scope of the new definition. The scope is propagated, using the Θ^+ transformation, to both the specific identifier and the form's body.

$$\frac{\begin{array}{c}\textsc{resolve}[\![\texttt{lambda}^{\Sigma_{\texttt{lambda}}}]\!]^{(\mathbf{B},\ \Gamma)} = \texttt{lambda} \\ E = \texttt{(begin}^{\{s\}}\ E_1\ \ldots\ E_n\texttt{)} \\ \sigma = \texttt{genscope}() \\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_{x_1}}]\!]_\sigma]\!]^{(\mathbf{B},\ \Gamma)}_{\texttt{var}} = (\mathbf{B}_1,\ \Gamma_1) \\ (\ldots) = (\ldots) \\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_{x_n}}]\!]_\sigma]\!]^{(\mathbf{B}_{n-2},\ \Gamma_{n-2})}_{\texttt{var}} = (\mathbf{B}_n,\ \Gamma_n) \\ \phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}_n,\ \Gamma_n)} \implies (\mathbf{B}',\ E')\end{array}}{\begin{array}{c}\phi[\![\texttt{(lambda}^{\Sigma_{\texttt{lambda}}}\ \texttt{(}x^{\Sigma_1}\ \ldots\ x^{\Sigma_n}\texttt{)}\ E_1\ E_2\ \ldots\ E_n\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \\ \implies \\ (\mathbf{B}',\ \texttt{(lambda}^{\Sigma_{\texttt{lambda}}}\ \texttt{(}\Theta^+[\![x^{\Sigma_1}]\!]_\sigma\ \ldots\ \Theta^+[\![x^{\Sigma_n}]\!]_\sigma\texttt{)}\ E'\texttt{))}\end{array}}$$

Figure 5.8: lambda abstraction

Procedure Application

A list of expressions which does not start with a variable associated to any macro transformer or core form present in the current environment, represents a usual procedural application. The expander goes through each subexpression of the list, propagating the resulting **B** environment at each step, similarly to the the `begin` special form.

$$\frac{\begin{array}{rcl}\forall\overline{\tau}.\ \textsc{lookup}[\![E_1]\!]^{(\mathbf{B},\ \Gamma)} & \neq \texttt{macro}(\overline{\tau}) & \\ \phi[\![E_1]\!]^{(\mathbf{B},\ \Gamma)} & \implies & (\mathbf{B}_1,\ E_1') \\ \phi[\![E_2]\!]^{(\mathbf{B}_1,\ \Gamma)} & \implies & (\mathbf{B}_2,\ E_2') \\ \ldots & \implies & \ldots \\ \phi[\![E_n]\!]^{(\mathbf{B}_{n-2},\ \Gamma)} & \implies & (\mathbf{B}_n,\ E_n')\end{array}}{\phi[\![\texttt{(}E_1,\ E_2,\ \ldots,\ E_n\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \implies (\mathbf{B}_n,\ \texttt{(}E_1',\ E_2',\ \ldots,\ E_n'\texttt{)})}$$

Figure 5.9: Application Form

Local definition

The `let` special form records a new binding in the environment used for the form's body expansion. It is semantically equivalent to an applied abstraction.

$$\texttt{((lambda (}x\texttt{)}\ E\texttt{)}\ E_x\texttt{)} \iff \texttt{(let ((}x\ E_x\texttt{))}\ E\texttt{)}$$

$$\frac{\begin{array}{c}\texttt{(let}^{\Sigma_{\text{let}}}\ \texttt{((}x_1^{\Sigma_1}\ E_1\texttt{)}\ \ldots\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ E_1\ \ldots\ E_n\texttt{)} \\ \Longrightarrow \\ \texttt{(let}^{\Sigma_{\text{let}}}\ \texttt{((}x_1^{\Sigma_1}\ E_1\texttt{)}\ \ldots\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ \texttt{(begin}^{\{s\}}\ E_1\ \ldots E_n\texttt{))}\end{array}}{\begin{array}{c}\text{RESOLVE}[\![\texttt{let}^{\Sigma_{\text{let}}}]\!]^{\mathbf{B}} \Longrightarrow \texttt{let} \\ \sigma = \texttt{genscope}() \\ \phi[\![E_x]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow (\mathbf{B}',\ E_x') \\ \text{RECORD}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]_{\texttt{var}}^{(\mathbf{B}',\ \Gamma)} \Longrightarrow (\mathbf{B}'',\ \Gamma') \\ \phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}'',\ \Gamma')} \Longrightarrow (\mathbf{B}''',\ E') \\ \hline \phi[\![\texttt{(let}^{\Sigma_{\text{let}}}\ \texttt{((}x^{\Sigma_x}\ E_x\texttt{))}\ E\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow (\mathbf{B}''',\ \texttt{(let}^{\Sigma_{\text{let}}}\ \texttt{((}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E_x'\texttt{))}\ E'\texttt{))}\end{array}}$$

Figure 5.10: local definition

Local Macro Definition

Local macro definitions are similar to the let special form, as the right-hand side of each definition represent transformation that can be used during the expansion, it must be fully evaluated. The evaluation of the right-hand side of a local macro definition goes through the last three algorithm phases, before evaluating the resulting expression. We name this process `eval*`.

$$\frac{\begin{array}{c}\phi[\![\mathbf{B},\ \Gamma,\ (\texttt{let-syntax}^{\Sigma_{\texttt{let-syntax}}}\ ((x_1^{\Sigma_{x_1}}\ E_{x_1})\ \ldots\ (x_n^{\Sigma_{x_n}}\ E_{x_n}))\ E_1\ E_2\ \ldots)]\!]^{(\mathbf{B},\ \Gamma)} \\ \Longrightarrow \\ \phi[\![(\texttt{let-syntax}^{\Sigma_{\texttt{let-syntax}}}((x_1^{\Sigma_{x_1}}\ E_{x_1}))\\ (\texttt{let-syntax}^{\Sigma_{\texttt{let-syntax}}}((x_2^{\Sigma_{x_2}}\ E_{x_2}))\\ \ldots\\ (\texttt{let-syntax}^{\Sigma_{\texttt{let-syntax}}}\ ((x_n^{\Sigma_{x_n}}\ E_{x_n}))\\ (\texttt{begin}^{\{s\}}\ E_1\ E_2\ \ldots))\ldots))]\!]^{(\mathbf{B},\ \Gamma)} \\ \\ \textsc{resolve}[\![\texttt{let-syntax}^{\Sigma_{\texttt{let-syntax}}}]\!]^{\mathbf{B}} \Longrightarrow \texttt{let-syntax} \\ \sigma = \texttt{genscope}() \\ \phi[\![E_x]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow (\mathbf{B}',\ E_x') \\ \texttt{eval*}[\![E_x']\!]^{(\mathbf{B},\ \Gamma)} = (\mathbf{B}',\ \overline{\tau}) \\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B}',\ \Gamma)}_{\texttt{macro}(\overline{\tau})} \Longrightarrow (\mathbf{B}'',\ \Gamma') \\ \phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}'',\ \Gamma')} \Longrightarrow (\mathbf{B}''',\ E') \end{array}}{\phi[\![(\texttt{let-syntax}^{\Sigma_{\texttt{let-syntax}}}\ ((x^{\Sigma_x}\ E_x))\ E)]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow (\mathbf{B}''',\ E')}$$

Figure 5.11: local macro definition

Macro Application

A macro application form is a list starting with any identifier associated to a macro-transformer. To distinguish between expression resulting from a macro-application and those who were originally present in the AST, we create a new scope.

$$\frac{\begin{array}{c}\textsc{lookup}[\![M]\!]^{(\mathbf{B},\ \Gamma)} = \texttt{macro}(\overline{\tau}) \\ \sigma = \texttt{genscope}() \\ \texttt{eval*}[\![(\overline{\tau}\ (M\ E_1\ \ldots\ E_n))]\!]^{(\mathbf{B},\ \Gamma)} = M' \end{array}}{\phi[\![(M,\ E_1,\ \ldots,\ E_n)]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow \phi[\![\Theta^{\oplus}[\![M']\!]_\sigma]\!]^{(\mathbf{B},\ \Gamma)}}$$

Figure 5.12: Macro Expansion

The `define` and `define-syntax` are analogous to their local counterpart, again with small modification to the way bindings are recorded. Those forms are omitted from the semantics description.

Quote and Quote-syntax

The `quote-syntax` special form acts as the quote special form, but for syntax objects instead of plain symbols. Both quote and quote-syntax are implemented the same way during this phase : the form does not change. However, during the last expansion phase, `quote-syntax` keeps the syntax object unchanged.

$$\frac{}{\texttt{(quote } E\texttt{)} \Longrightarrow \texttt{(quote } E\texttt{)}}$$

$$\frac{}{\texttt{(quote-syntax } E\texttt{)} \Longrightarrow \texttt{(quote-syntax } E\texttt{)}}$$

Figure 5.13: `quote` and `quote-syntax` special form

Other Special Forms

Scheme's implementations often include some variant special forms for local definition. The special forms `let*`, `letrec` and `letrec*` differ in the order of evaluation and lexical scope of their binding definitions. The `let*` special form evaluates each binding sequentially while propagating the new environment at each step. The `letrec` special form binding allow recursive definition for the defined bindings, as well as mutually recursive bindings. At last, the `letrec*` combine the two previous variants.

$$\frac{}{\phi[\![\mathbf{B},\ \Gamma,\ \texttt{(let*}^{\Sigma_{\texttt{let*}}}\ \texttt{((}x_1^{\Sigma_{x_1}}\ E_{x_1}\texttt{)}\ \ldots\ \texttt{(}x_n^{\Sigma_{x_n}}\ E_{x_n}\texttt{))}\ E_1\ E_2\ \ldots\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow \phi[\![\texttt{(let}^{\Sigma_{\texttt{let*}}}\texttt{((}x_1^{\Sigma_{x_1}}\ E_{x_1}\texttt{))}\ \texttt{(let}^{\Sigma_{\texttt{let*}}}\texttt{((}x_2^{\Sigma_{x_2}}\ E_{x_2}\texttt{))}\ \ldots\ \texttt{(let}^{\Sigma_{\texttt{let*}}}\ \texttt{((}x_n^{\Sigma_{x_n}}\ E_{x_n}\texttt{))}\ \texttt{(begin}^{\{s\}}\ E_1\ E_2\ \ldots\texttt{))}\ldots\texttt{))}\ldots\texttt{))}]\!]^{(\mathbf{B},\ \Gamma)}}$$

Figure 5.14: `let*` form definition

$$\frac{\begin{array}{c}\texttt{(letrec}^{\Sigma_{\texttt{letrec}}}\ \texttt{((}x_1^{\Sigma_1}\ E_1\texttt{)}\ \ldots\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ E_1\ \ldots E_n\texttt{)} \\ \Longrightarrow \\ \texttt{(letrec}^{\Sigma_{\texttt{letrec}}}\ \texttt{((}x_1^{\Sigma_1}\ E_1\texttt{)}\ \ldots\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ \texttt{(begin}^{\{s\}}\ E_1\ \ldots E_n\texttt{))}\end{array}}{\begin{array}{c}\textsc{resolve}[\![\texttt{letrec}^{\Sigma_{\texttt{letrec}}}]\!]^{\mathbf{B}} \Longrightarrow \texttt{letrec} \\ \sigma = \texttt{genscope}() \\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B},\ \Gamma)}_{\mathsf{var}} \Longrightarrow (\mathbf{B}_1,\ \Gamma_1) \\ \forall i \in [2,n],\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B}_{i-2},\ \Gamma_{i-2})}_{\mathsf{var}} \Longrightarrow (\mathbf{B}_i,\ \Gamma_i) \\ \phi[\![E_1]\!]^{(\mathbf{B}_n,\ \Gamma_n)} \Longrightarrow (\mathbf{B}'_1,\ E'_x) \\ \forall i \in [2,n],\ \phi[\![E_i]\!]^{(\mathbf{B}'_{i-2},\ \Gamma_n)} \Longrightarrow (\mathbf{B}'_i,\ E'_i) \\ \phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}'_n,\ \Gamma_n)} \Longrightarrow (\mathbf{B}'',\ E')\end{array}}$$

$$\frac{}{\begin{array}{c}\phi[\![\texttt{(letrec}^{\Sigma_{\texttt{letrec}}}\ \texttt{((}x^{\Sigma_1}\ E_1\texttt{)}\ \ldots\ \texttt{(}x^{\Sigma_n}\ E_n\texttt{))}\ E\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \\ \Longrightarrow \\ (\mathbf{B}'',\ \texttt{(letrec}^{\Sigma_{\texttt{letrec}}}\ \texttt{((}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{)}\ \ldots\ \texttt{(}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{))}\ E'\texttt{))}\end{array}}$$

Figure 5.15: `letrec` form definition

$$\frac{}{\begin{array}{c}\texttt{(letrec*}^{\Sigma_{\texttt{letrec*}}}\ \texttt{((}x_1^{\Sigma_1}\ E_1\texttt{)}\ \texttt{...}\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ E_1\ ...E_n\texttt{)} \\ \Longrightarrow \\ \texttt{(letrec*}^{\Sigma_{\texttt{letrec*}}}\ \texttt{((}x_1^{\Sigma_1}\ E_1\texttt{)}\ \texttt{...}\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ \texttt{(begin}^{\{s\}}\ E_1\ ...E_n\texttt{))}\end{array}}$$

$$\begin{array}{c}\textsc{resolve}[\![\texttt{letrec*}^{\Sigma_{\text{letrec*}}}]\!]^{\mathbf{B}} \Longrightarrow \texttt{letrec*} \\ \sigma = \texttt{genscope}()\end{array}$$

$$\begin{array}{c}\textsc{record}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]_{\texttt{var}}^{(\mathbf{B},\ \Gamma)} \Longrightarrow (\mathbf{B}_1,\ \Gamma_1) \\ \phi[\![E_1]\!]^{(\mathbf{B}_1,\ \Gamma_1)} \Longrightarrow (\mathbf{B}'_1,\ E'_x)\end{array}$$

$$\begin{array}{c}\forall i \in [2,n],\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]_{\texttt{var}}^{(\mathbf{B}'_{i-2},\ \Gamma_{i-2})} \Longrightarrow (\mathbf{B}_i,\ \Gamma_i) \\ \forall i \in [2,n],\ \phi[\![E_i]\!]^{(\mathbf{B}_i,\ \Gamma_i)} \Longrightarrow (\mathbf{B}'_i,\ E'_i)\end{array}$$

$$\frac{\phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}'_n,\ \Gamma_n)} \Longrightarrow (\mathbf{B}'',\ E')}{\begin{array}{c}\phi[\![\texttt{(letrec*}^{\Sigma_{\texttt{letrec*}}}\ \texttt{((}x^{\Sigma_1}\ E_1\texttt{)}\ ...\ \texttt{(}x^{\Sigma_n}\ E_n\texttt{))}\ E\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \\ \Longrightarrow \\ (\mathbf{B}'',\ \texttt{(letrec*}^{\Sigma_{\texttt{letrec*}}}\ \texttt{((}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{)}\ ...\ \texttt{(}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{))}\ E'\texttt{))}\end{array}}$$

Figure 5.16: `letrec*` form definition

$$\frac{\begin{array}{c}\texttt{(letrec*-syntax}^{\Sigma_{\texttt{letrec*-syntax}}}\ \texttt{((}x_1^{\Sigma_1}\ E_1\texttt{)}\ \texttt{...}\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ E_1\ ...E_n\texttt{)} \\ \Longrightarrow \\ \texttt{(letrec*-syntax}^{\Sigma_{\texttt{letrec*-syntax}}}\ \texttt{((}x_1^{\Sigma_1}\ E_1\texttt{)}\ \texttt{...}\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ \texttt{(begin}^{\{s\}}\ E_1\ ...E_n\texttt{))}\end{array}}{\begin{array}{c}\text{RESOLVE}[\![\texttt{letrec*-syntax}^{\Sigma_{\texttt{letrec*-syntax}}}]\!]^{\mathbf{B}} \Longrightarrow \texttt{letrec*-syntax} \\ \sigma = \texttt{genscope}() \\ \\ \text{RECORD}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B},\ \Gamma)}_{\texttt{macro(unbound)}} \Longrightarrow (\mathbf{B}_1,\ \Gamma_1) \\ \text{EVAL}[\![E_1]\!]^{\mathbf{B}_1,\ \Gamma_1} \Longrightarrow \overline{\tau}_1 \\ \text{RECORD}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B}_1,\ \Gamma_1)}_{\texttt{macro}(\overline{\tau}_1)} \Longrightarrow (\mathbf{B}'_1,\ \Gamma'_1) \\ \\ \forall i \in [2,n],\ \text{RECORD}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B}'_{i-2},\ \Gamma'_{i-2})}_{\texttt{macro(unbound)}} \Longrightarrow (\mathbf{B}_i,\ \Gamma_i) \\ \text{EVAL}[\![E_i]\!]^{\mathbf{B}_i,\ \Gamma_i} \Longrightarrow \overline{\tau}_i \\ \text{RECORD}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B}_i,\ \Gamma_i)}_{\texttt{macro}(\overline{\tau})} \Longrightarrow (\mathbf{B}'_i,\ \Gamma'_i) \\ \\ \phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}'_n,\ \Gamma'_n)} \Longrightarrow (\mathbf{B}'',\ E')\end{array}}{\begin{array}{c}\phi[\![\texttt{(letrec*-syntax}^{\Sigma_{\texttt{letrec*-syntax}}}\ \texttt{((}x^{\Sigma_1}\ E_1\texttt{)}\ \texttt{...}\ \texttt{(}x^{\Sigma_n}\ E_n\texttt{))}\ E\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \\ \Longrightarrow \\ (\mathbf{B}'',\ \texttt{(letrec*-syntax}^{\Sigma_{\texttt{letrec*-syntax}}}\ \texttt{((}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{)}\ \texttt{...}\ \texttt{(}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{))}\ E'\texttt{))}\end{array}}$$

Figure 5.17: `letrec*-syntax` form definition

Top Level Definition

Most Scheme implementations includes the `define` and the `define-syntax` special forms which are analogous to their local counterpart. The use of a `define` or `define-syntax` form in a local context is equivalent to the uses of the `let` or the `let-syntax` form respectively.
Those forms won't be formalized as it complexifies the notation used.

Expansion's Last Phase

The last phase of the expansion consists of a single pass on the program's expression, where each identifier $x_i^{\Sigma_{x_i}}$ encountered is *resolved* to its corresponding binding key $\underline{x_i}$. The binding key is then used in the expanded code, to refers to the correct binding, avoiding name clashes. Within this phase, unbound identifiers are made obvious to the compiler and they can be treated accordingly.

In this chapter, we formalized most of the set of scopes algorithm, following the Racket's specification. We focused on the algorithm's core without Racket's speci-

ficities such as its module system. In the following, we begin with an overview of the original Gambit's macro system. We then formalize the data structures used and present the algorithm's implementation we conceived.

Chapter 6

Gambit's set of Scopes

Gambit's implementation of the Scheme language conforms to the R5RS revised report, with supports for some of the functionalities appearing in the following reports. Since its sixth revised report, the language's specification includes differents popular hygienic transformations, namely the `define-syntax`, `syntax-rules` and `syntax-case` forms. These forms were made available in the Gambit system, without offering hygiene.

To implement the set of scope algorithm on top of Gambit's original system, the language's compilation pipeline was changed, as well as both the compilation and expansion environment. Our choice of a hygienic macro system implementation was motivated by the debugging utility and extensibility that the set of scopes algorithm offers. As the system records every binding, the compiler can give complete error messages, even in case of undefined variables or broken macros. The modularity of the environment representation also allows for particularly good visualization of the lexical context, and combines well with Gambit's original environments.

This chapter explains the modifications required to the Gambit Scheme implementation to implement hygiene. We then summarize the process of rewriting some of the Gambit special form, such as `syntax-case`, with our new macro system. We begin with a definition of the Gambit source code representation, which extends the simple S-expression representation with extra information useful for debugging.

Source Code Representation

The Gambit implementation of the Scheme language extends every object's usual S-expression representation with the source file's name and textual position (character/line). We represent this location information by the **Location** type.

Definition 22 (Source Object)**.** *Gambit's source-objects extend the S-expression representation to include source file's location information :*

$$\begin{aligned} \textbf{\textit{Source}} ::= & (\textbf{\textit{List Source}}) \times \textbf{\textit{Location}} \\ | \quad & \textbf{\textit{Atom}} \times \textbf{\textit{Location}} \end{aligned}$$

We extended the source-object datatype with an extra field containing a, potentially uninitialized, reference to a set of scope object. We then define Gambit's syntax-objects as any source object with an initialized set of scope object. This make the conversion between the two structures more efficient. To transform a source-object into a syntax-object (as with `datum->syntax`), we simply add the reference to a new (or existing) set of scopes to the source-object's last field by mutation.

Definition 23 (Gambit's Syntax-object)**.** *Gambit's syntax-objects extend the representation of symbol containing source objects with an additional field containing a set of scope reference.*

$$\textbf{\textit{Syntax-object}}_G ::= \textbf{\textit{Symbol}} \times \textbf{\textit{Location}} \times \textbf{\textit{Set Scpope}}$$

Definition 24 (Gambit's Syntax)**.** *Gambit's syntax extends the general source object representation to include a set of scopes on source containg a symbol as atom.*

$$\begin{aligned} \textbf{\textit{Syntax}}_G = {} & (\textbf{\textit{List Syntax}}_G) \times \textbf{\textit{Location}} \\ | \ & \textbf{\textit{Syntax-object}}_G \\ | \ & \textbf{\textit{Source}} \end{aligned}$$

We shall omit the $_G$ subscripts for the remaining of the chapter.

As the algorithm must know the constructs exposed to the programmer before the implementation of syntax objects in our code, we can include source objects directly as syntax object by implementing the following strategy. By extending the Gambit syntax object definition to include an optional reference to a set-of-scope for every atomic source object, source objects can be replaced altogether with our new syntax representation of code. [1]. The extra field on every source-object gives a slight memory overhead for users who wish to disable the hygiene system altogether.

Set of Scopes Implementation

Gambit use two compile-time environments. The `interaction-cte` environment is used for the `eval` procedure, during the *interpretation* of code with GSI. The second environment is used for the *compilation* of the code with GSC. The two environments interact, during compilation, to share common macro transformers, as the `interaction-cte` is defined earlier in the pipeline and as the interpreter allows calls to the `compile-file` procedure[2].

[1] As simple optimisation, we can sometimes mutate a syntax-object in place, instead of performing a full copy as required with `datum->syntax`.

[2] The `compile-file` procedure takes a file path as argument and invokes the compilation pipeline on that file. To do so, a compilation compile-time environment representation must be created which includes the interaction-cte's local definitions.

To implement our macro system on top of the Gambit's current implementation, we changed both environments to match, more closely, the structure needed for the set of scopes algorithm, while preserving a consistent representation for both environments.

The following section explains changes made to the `interaction-cte`.

The Interaction Compile Time Environment

The original environment was based on a circular linked list of elements representing the different types of bindings and was closely tuned to the REPL feature of the language.

cte : CTE
cte ::= `top-cte`(*previous-cte*)
| `cte-frame`(*previous-cte*, *names*, *up*, *over*)
| `cte-macro`(*previous-cte*, *name*, *transformer*)
| `cte-namespace`(*previous-cte*, *name*, *aliases*)
| `cte-decl`(*previous-cte*, *name*, *value*)

`top-cte` : **CTE** → **CTE**
`cte-frame` : **CTE** → (**List Symbol**) → **Fixnum** → **Fixnum** → **CTE**
`cte-macro` : **CTE** → **Symbol** → (**Syntax** → **Syntax**) → **CTE**
`cte-namespace` : **CTE** → **Symbol** → **List** (**Pair Symbol Symbol**) → **CTE**
`cte-decl` : **CTE** → **Symbol** → **Symbol** → **CTE**

Figure 6.1: Gambit's Original `interaction-cte`

With this environment representation, for every new binding, macro binding, declarations or namespace defined, the expander inserts a new element in front of the linked list. We briefly describe the original binding's representation.

The `cte-frame`(names, up, over, next-cte) represents variables defined in the same `let` or `lambda` form. The `up` and `over` fields are integers related to the Debrujin encoding of variables, which allows variable lookup by their definition's order, instead of symbol comparison.

The `cte-macro`(name, transformer, next-cte) object contains the evaluated macro transformer associated to the macro's name.

The `cte-namespace`(name, aliases, next-cte) associates the namespace's name with a list of name substitutions for that namespace.

The `cte-decl`(name, value, next-cte) gives a value to the named declaration. Declarations are, in most cases, used to enable/disable optional extensions of the

compiler.

As the structure is cyclic, the `top-cte` node has the role of a sentinel node, representing the start and end of the list: if the current cte is the `top-cte` object, it indicates that we are at top-level.

The structure has many advantages such as a small memory cost and efficient (constant time) single-stepping and backtracing. While looking up an identifier in the environment, the compiler processes by linear search in the list, in $O(n)$ relative to the number of bindings, macro transformers, namespaces and declarations. When a namespace binding is encountered during the linear lookup, the compiler compares the variable's name to namespace bindings' aliases. As a result, namespaces name substitution can be inferred without additional cost.

To ensure hygiene, we recall that the compiler must record every variable that can be used, including primitives and core forms. In the original environment, during the compiler's compilation, primitives and core form exposed to the programmer are pre-inserted into the `interaction-cte` environment as default content.

The New Representation

To replace the expansion phase of the original Gambit implementation completely, we modified both the interaction and compilation environments. We focus our attention, once again, on the interaction `cte` as the compilation environment is a simplified version of it. As a first implementation step, we kept the Gambit's original expansion phase in the pipeline: after our new expansion phase, the original expansion resume. This allows for incremental integration of the debugging utility, before we can remove the original expansion phase altogether. This also ensured that no feature support was lost, at every implementation step.

We begin with a description of our modification, followed by an overview of some of the Gambit's features and their integration in our new system.

Algorithm's Data Structures

We briefly describe our implementation choice for the data structures required by the set of scopes algorithm, and explain the changes made to the environments.

Global Binding Table B

The global binding table implementation requires a structure supporting fast extension by mutation. For the use of the `resolve` procedure, it also requires direct access to each association, as well as a way to loop through every association, one by one, without specific order. For an efficient implementation, we expect the mutations of the table to run in constant time, and call to the `resolve` procedure to run in linear time ($O(n)$, with n, the number of associations recorded in $\mathbf{B}$). We thus implement the $\mathbf{B}$ table by hashing, using the `table` hash-table implementation

present in the Gambit system. As Gambit's hash tables store data in a continuous chunk of memory, we can easily access every association linearly. Gambit proposes the `table->list` to efficiently convert hash tables into linked lists.

Compile Time Environment Γ

To implement the compile time environment Γ, the structure used must support not only constant time access but also constant time non-mutable extensions. The Gambit implementation allows, specifically with its REPL utility, to define or redefine top-level transformers, from any local environment. It follows that the structure must support constant time mutation as well.

We modified the Gambit implementation of the `table` object to support non-mutable operations on the same data structure. To do so, the underlying structure was changed toward a hashed array mapped trie (HAMT). This data structures giving amortized performance in the same order as conventional hash tables. However, the structure is *persistent*, [3] allowing efficient data sharing.

For the table's implementation, we use our new `table` structure. When extending the top-level environment, we can mutate the top-cte's cte directly. On the other hand, we can add local bindings by non-mutable extension, sharing most of the table's data with its parent `cte`, before propagating the context into the local definition expression's inner body.

Integration of the B and Γ Tables

To incorporate the algorithm's environment, the local Γ table and the global **B** table must be kept accessible from every `cte` objects. However, the Γ and **B** tables are not sufficient to conserve efficient access to the last inserted `cte-frame` element, as HAMT are unordered collections. For an efficient implementation of the REPL's backtracing feature, we must keep, a reference to the *parent* `cte` object. Similarly, to conserve the original namespace and declaration implementations, reference to the parent `cte` is required. It follows that we had to conserve the original `cte`'s linked list structure but, also allow access to both tables at all times.

The local Γ table must be updated with every `cte-frame` or `cte-macro` creation. We can extend every `cte` objects with a field containing the current Γ table but, as only `cte-frame` and `cte-macro` object can update the table, we extended those specific objects only. The `cte-macro` object can be simplified but, we ensured retro-compatibility with the original debugging features. As the HAMT structure is persistent, every extension of the Γ table shares most of its data with the previous `cte`.

We store the **B** table's reference in the `top-cte` object. To allow fast access to the table from any `cte` object, we added a reference to *some* of them. Note that we store a reference to the `cte-top` object, instead of a reference to the **B** table, as

[3] Persistent data structures always keep a copy of their current state when modified. In our particular case, the structure is conserved during insertion.

access to the `cte-top` object is required by some of the operations of the original implementation. Furthermore, this allows us to mutate the `cte-top`'s Γ table from any `cte` object.

Instead of extending every object with the two references, we extend the `cte-top`, `cte-frame` and `cte-macro` only. If the current `cte` is a `cte-namespace` or `cte-declaration`, we can access the Γ and $\mathbf{B}$ table by going through the linked list until a `cte-frame`, `cte-macro` or `cte-top` object is encountered.

Interaction Compile Time Environment

We then define the new `interaction-cte`:

$$
\begin{aligned}
&\text{cte} : \text{CTE}\\
&\text{cte} ::= \texttt{top-cte}(\textit{previous-cte},\ \mathbf{B},\ \Gamma)\\
&\quad \mid \texttt{cte-frame}(\textit{previous-cte},\ \Gamma,\ \textit{cte-top},\ \textit{names},\ \textit{up},\ \textit{over})\\
&\quad \mid \texttt{cte-macro}(\textit{previous-cte},\ \Gamma,\ \textit{cte-top},\ \textit{name},\ \textit{transformer})\\
&\quad \mid \texttt{cte-core-macro}(\textit{previous-cte},\ \Gamma,\ \textit{cte-top},\ \textit{name},\ \textit{transformer})\\
&\quad \mid \texttt{cte-namespace}(\textit{previous-cte},\ \textit{name},\ \textit{aliases})\\
&\quad \mid \texttt{cte-decl}(\textit{previous-cte},\ \textit{name},\ \textit{value})
\end{aligned}
$$

$$
\begin{aligned}
\texttt{top-cte} &: \mathbf{CTE} \to \Gamma \to \mathbf{B} \to \mathbf{CTE}\\
\texttt{cte-frame} &: \mathbf{CTE} \to \Gamma \to \mathbf{CTE} \to (\mathbf{List\ Symbol}) \to \mathbf{Fixnum} \to \mathbf{Fixnum} \to \mathbf{CTE}\\
\texttt{cte-macro} &: \mathbf{CTE} \to \Gamma \to \mathbf{CTE} \to \mathbf{Symbol} \to (\mathbf{Syntax} \to \mathbf{Syntax}) \to \mathbf{CTE}\\
\texttt{cte-core-macro} &: \mathbf{CTE} \to \Gamma \to \mathbf{CTE} \to \mathbf{Symbol} \to (\mathbf{Syntax} \to \mathbf{Syntax}) \to \mathbf{CTE}\\
\texttt{cte-namespace} &: \mathbf{CTE} \to \mathbf{Symbol} \to \mathbf{List}\ (\mathbf{Pair\ Symbol\ Symbol}) \to \mathbf{CTE}\\
\texttt{cte-decl} &: \mathbf{CTE} \to \mathbf{Symbol} \to \mathbf{Symbol} \to \mathbf{CTE}
\end{aligned}
$$

Figure 6.2: Gambit's New `interaction-cte`

Beside the incorporation of the two tables, we also added a new `cte` node: the `cte-core-macro` object is used to store core forms transformers, with the same structure as regular user defined transformers. Recall that macro application's semantic expands the transformer application's result after the evaluation of the transformer call. Core forms, on the other hand, have specific semantics that must be processed differently. The core forms of our expansion correspond to the core form formalised in the previous chapter, in addition to some other forms that require special processing, such as `quasiquote`, that can be optimized with direct access to the environment.

Interface and Compilation Environment

As previously mentioned, the compilation environment is, roughly, a simplified version of the interaction `cte`. We conceived an interface for the procedures needed for

the hygienic expansion, thus abstracting the environment's specific implementation by using a different mapping for both interpretation and compilation.

Gambit's Features

Our new structures are consistent and retro-compatible with the original expansion phase of the compiler. We discuss briefly some of the Gambit's features and explain how we preserved them in our new macro system.

REPL Utility

The backtracing feature of the REPL was tightly linked to the `interaction-cte`'s representation. By conserving the linear structure of the `cte`, the ordering of each definition is kept intact and allows for the debugging feature to be integrated without much change. When backtracing, we can access a previous Γ table by following the parent `cte` reference of each node. As the **B** global binding table is implemented with an hash table, we can remove the association in constant time. When using the REPL, Gambit allows for top-level definition from any local context. To preserve the feature, we can introduce bindings in the top-level Γ table of the `cte-top` object by mutation. For identifiers lookup, we can search in both the local Γ table and the `cte-top`'s Γ table.

The original expansion uses specific procedures to control single-stepping in the code: the number of single step each core form uses must be specified within each form. We integrated that feature in the last phase of the set of scopes algorithm.

Debugging Features

While debugging, using the REPL or with print-debugging, the compiler can present a particular piece of source code in a familiar and comprehensive manner, using the *pretty-printer*. For instance, programmers can print the code contained in some closures stored in memory, by using the `pp` procedure: `(pp (lambda (x) x))`. For this case, instead of printing the closure's address directly, Gambit gives a well-indented representation of the code: `(lambda (x) x)`. The representation corresponds to the fully expanded code contained in the closure.

However, using the pretty-printer to show source code containing macro expansion do not produce any clues about which expansions were used. Considering a transformer `a-macro` defined with `(define-macro (a-macro) '0)`, the call `(pp (lambda (x) (a-macro)))` will output `(lambda (x) 0)`. As a programmer can structure their code with macro calls, the pretty-printed code can become as unrecognizable and obfuscated.

To avoid some of these issues, the Gambit's pretty-printer can output core form uses in their pre-expanded form. For known core transformers, such as `cond` and `quasiquote`, the expansion is predictable and cannot be redefined. Usually those core forms are widely used in code base and reduce the code size drastically. For instance, we can consider the `cond` core form which expands into an `if` cascade.

```
(cond
  ((test1?) body1)
  ((test2?) body2)
  ((test3?)
     => some-procedure)
  (else
    body-else))
```

```
(if test1?
    body1
    (if test2?
        body2
        (let ((result (test3?)))
          (if result
              (some-proc result)
              body-else))))
```

Figure 6.3: simplified `cond` special form expansion

The predictability of those expansions helps Gambit determine if some expanded code is the result of one of the core form's transformation. The system proceed by recognition of pattern in the expanded form, which is called *decompilation*. Decompilation is an heuristic process as the different uses of the same core form can produce the same expansion. For instance, a call to `(pretty-print (lambda (x) `(0 1 ,x)))` output `(lambda (x) `(,0 ,1 ,x))`. Indeed, both `(lambda (x) `(,0 ,1 ,x))` and `(lambda (x) `(0 1 ,x))` expand to the same code.

To support the feature in our new system, we had to modify the decompilation process, as some of those form's expansion changed. We also include those core transformers to the core forms of our algorithm.

Module Integration

The Gambit module system is an implementation of the R7RS specification [FH20] [SCG13]. Combining the namespace feature of the language with the special `include` procedure, users can import named procedures and transformers in a single abstracted special form `import`. The `include` special form `read`s a source-code file and inserts its AST in place.

Modules can be compiled independently and be statically/dynamically linked to other programs. The pre-compiled modules must be able to share their global variables and global macro definitions with the compiler during the compilation of a program that use that module.

With our hygienic system, we want to track every identifier appearing in module's initial code as well. Furthermore, the Γ table must get recovered to include the macro definitions exposed by the module. We thus include a serialised version of the `cte-top`'s Γ and $\mathbf{B}$ tables. When linking a module statically, the compiler combines the serialised table, with the current expansion's `cte`. Multiple use of the same module can be optimised to skip that step.

Representing scope object as Gambit object is not sufficient for serialisation and static linking: the object's addresses used to differentiate scopes within a module could clash with the adresses used by the compiler while compiling program using that module. We then replace our scope object implementation with pairs containing an integer, and a unique module tag. The tag can be, for instance, the MD5 hash of the file. We extend our environment with a counter that can be used for

every scope creation.

The current chapter summarised the set of scope algorithm implementation, defining the core of our new macro system. We also presented some of the interesting features of the Gambit language that we had to conserve within the new system. The following chapter presents the tests made to validate our algorithm implementation and also includes a performance analysis.

Chapter 7

Macro System's Validity and Performance Analysis

This chapter summarizes how we ensured the new system's validity and conformity to the R6RS specifications [FM+07] and include some performance analysis.

As a first step to prove the system's validity, we valided the algorithm's core implementation as specified by Racket's specification. As a second step, we validated the macro system's higher-level pattern matching forms with the R6RS specification and some other conforming Scheme implementations.

Algorithm Validity

Our new system is based on Racket's set of scopes implementation. To validate the core of our new algorithm, we selected some of the tests that are used by Racket to validate its own algorithm. The Racket language differs from other Scheme dialects, notably, by its tower of evaluation and non-standard module implementation. We then chose tests used to validate the primitive procedure used in the algorithm as well as simple core forms. However, Gambit's use of source-objects instead of regular datum leads to some noticeable differences in the primitives' behaviour. As we are allowing for backward-compatibility with the original system, we tested for those specific cases as well during the development. We finally extended those tests with some of our own.

Macro System Validity

The low-level constructs defined by our algorithm constitute the core of our macro system. Those primitives allow for more complex extensions, including pattern matching tools such as `syntax-case`. Those pattern-matching forms, also including `with-syntax` and `syntax-rules`, are included in the R6RS report. In practice, in common code bases, most transformers use `syntax-rules` based pattern matching utilities for most, or the entirety of their code.

`syntax-case` Specification

The article presented earlier proposes, in addition to the complete form specification, an exhaustive list of `syntax-case` form's use-cases [Dyb92]. Those examples validate the different mechanisms used to perform hygienic expansion, including ways to break hygiene. As the specification was included, with little modifications, in the R6RS revised report, we used those examples to ensure our own system conforms to it.

The article includes, among others, the canonical hygiene example of a short-circuiting `or` macro.

```
(define-syntax or
  (lambda (x)
    (syntax-case x ()
      ((_) (syntax #f))
      ((_ e) (syntax e))
      ((_ e1 e2 e3 ...)
       (syntax (let ((t e1)) (if t t (or e2 e3 ...))))))))

(let ((t "okay"))
  (or #f t))
```

Figure 7.1: `or` macro using `syntax-case` and a use case requiring hygiene

The authors also implement common alternative pattern matching tools in terms of `syntax-case`. Our implementation of the form `with-syntax` follows the article's implementation, ignoring error checking.

```
(define-syntax with-syntax
  (lambda (x)
    (syntax-case x ()
      ((_ ((p e0) ...) e1 e2 ...)
       (syntax (syntax-case (list e0 ...) ()
                 ((p ...) (begin e1 e2 ...))))))))
```

Figure 7.2: `with-syntax` special form implementation in term of `syntax-case`

The `syntax-rules` implementation, is also defined.

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((_ (k ...) ((keyword . pattern) template) ...)
       (with-syntax (((dummy ...)
                      (generate-temporaries
                        (syntax (keyword ...)))))
         (syntax
          (lambda (x)
            (syntax-case x (k ...)
              ((dummy . pattern) (syntax template))
              ...)))))))
```

Figure 7.3: `syntax-rules` special form implementation in term of `syntax-case`

In addition, the article proposes some other constructs used as low-level facilities for macro expansion. For instance, the predicates `free-identifier?`, `bound-identifier?` as well as the `generate-temporaries` procedure. Those procedures were added to Gambit and match this specification. In the end, we were able to implement the entirety of the examples listed in the paper, exploring different edge cases.

Other Scheme Implementations

To test those higher-level pattern-matching constructs, we can choose validity tests from different Scheme implementations without worrying about their underlying implementation details and low-level form. In addition to Racket's test suite, used to validate the algorithm's core, we chose some of the R6RS compliant Scheme implementations. The Chibi Scheme implementation [chi] conforms, for most of its features, to the R7RS specification and includes the `syntax-case` and `syntax-rules` special forms [SCG13]. It includes a test suite conforming to the R6RS specification.

In addition to the Scheme revised reports, the implementation of some popular modules are selected, by a committee of developers, researchers, and users, under the label of *Scheme Request For Implementation (SRFI)*. As the `syntax-case` form is now part of the language core specification, some of those SRFI are implemented with it. This allows us to test our implementation as Gambit conforms to the R7RS module system and includes many of such modules.

We begin the analysis with a quantitative comparison of performance between the new Gambit macro system and its predecessor. This section gives an overview of our results for time performance for both the Gambit interpreter (GSI) and the Gambit compiler (GSC), during the expansion phase, at compile time.

As a first implementation step, the new system keeps the original macro system's expansion phase intact, to enable backward-compatibility. To produce more efficient code, we can combine the last phase of the set of scopes algorithm with the original

system's expansion phase. To succeed in this step, the single-stepping features offered in Gambit can be combined with the last set of scopes algorithm phase, which `resolve` the identifiers of the expanded code. The optimization would save a single pass over the code. For an S-expression s, a pass on the code $p(s)$ has a time complexity such as:

$$a : \textbf{Atom},\ p(a) = O(1)$$
$$\forall s_i : \textbf{Sexp},\ p((\ s_1\ \ s_2\ \ ...\ \ s_n\ )) = \sum_{i=1}^{n} p(s_i) + O(1)$$

Currently, some of the special transformers, such as `define-macro`, `with-syntax` and `syntax-rules` are implemented using the `syntax-case` form to generate the appropriate transformer. The Gambit implementation used to compile our new system implements `syntax-case` non-hygienically, with different data structures. The full bootstrap of the compiler must be done to generate the correct transformers at compile time during the compiler's compilation.

Choice of Tests and Methodology

As a first test phase, we computed the time required for both the compiler and interpreter to execute the `add-scope`, `expand` and `compile` procedure of the algorithm, mimicking a simple expansion, for programs containing from 500 to 4000 nested local binding creations. We also tested the same program but using bindings for local macro transformations instead and varied the program shape.

In the second test phase, we compared the new macro system with the original Gambit implementation, computing the compilation/interpretation time for simple programs. We tested the interpreter over a selection of core form uses. For each of those core forms, we tested a succession of thirty calls, where each form uses the last binding defined if it applies.

As a third test phase, we tested some of the core forms used for macro transformations, as well as predefined macro transformations such as `syntax-rules` and `define-macro`. We also compared the original unhygienic `define-syntax` form with our new hygienic variant. As our new system provides an additional security feature for transformer uses, we can expect those tests to run somewhat slower than it would with the original system.

For both the second and third test phases, for each given test file, we compared the original system with the new one for both compilation and interpretation. To produce a good estimate, each test file was interpreted and compiled fifteen times with each of the systems. We then obtained four test categories, for both test groups and both interpretation and compilation. The results of each category were then compiled to find the average and standard deviation of each test file over the group of fifteen interpretations or compilations.

Subsequently, we computed the average and standard deviation of the tests within each category. To do so, we repeated the whole process to create a sample pool of fifteen results for each category. We then tallied the average of each test file result's average over that sample pool.

Tests Results

The following figures summarize our test results.

Firstly, we show in figure 7.4 the interpretation time for a simple program relative to the number of local bindings. We computed the interpretation time by computing the *real time* required to execute the `add-scope`, `expand` and `compile` procedure, by creating the syntax-object dynamically and then using the `real-time` Gambit's procedure.

The programs tested contains 500 to 4000 local bindings and consist of nested `let` where each of the `let`s bind a variable to an atomic value.

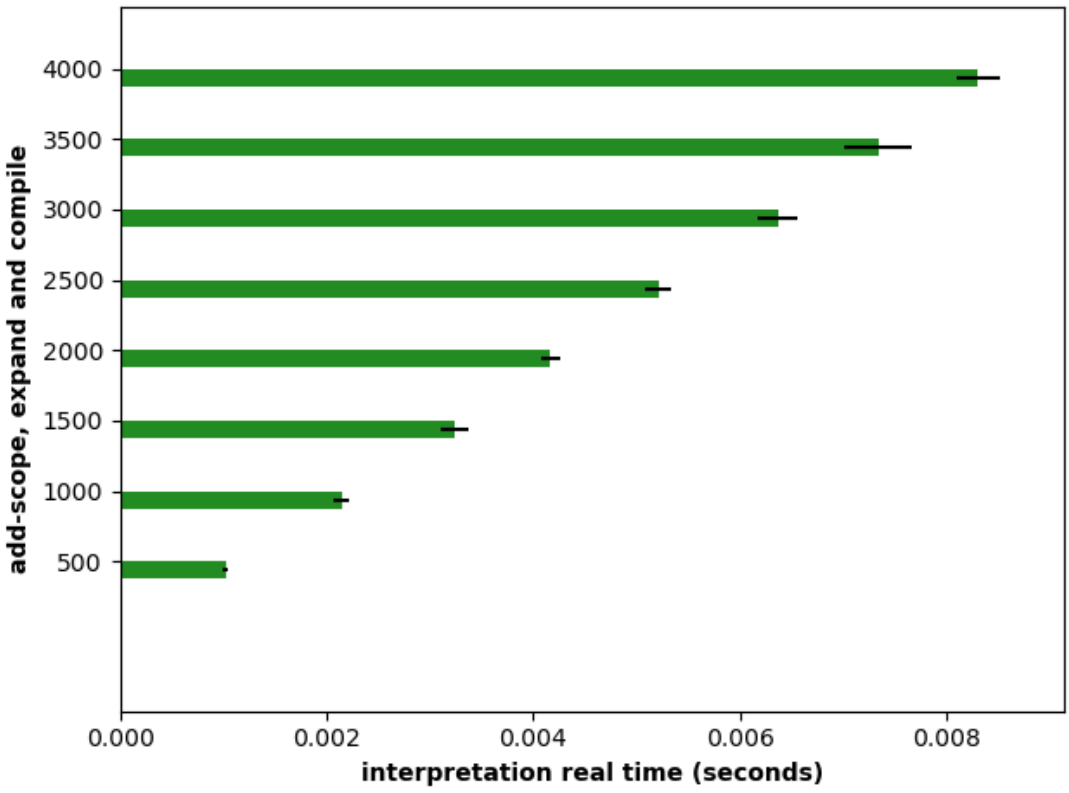


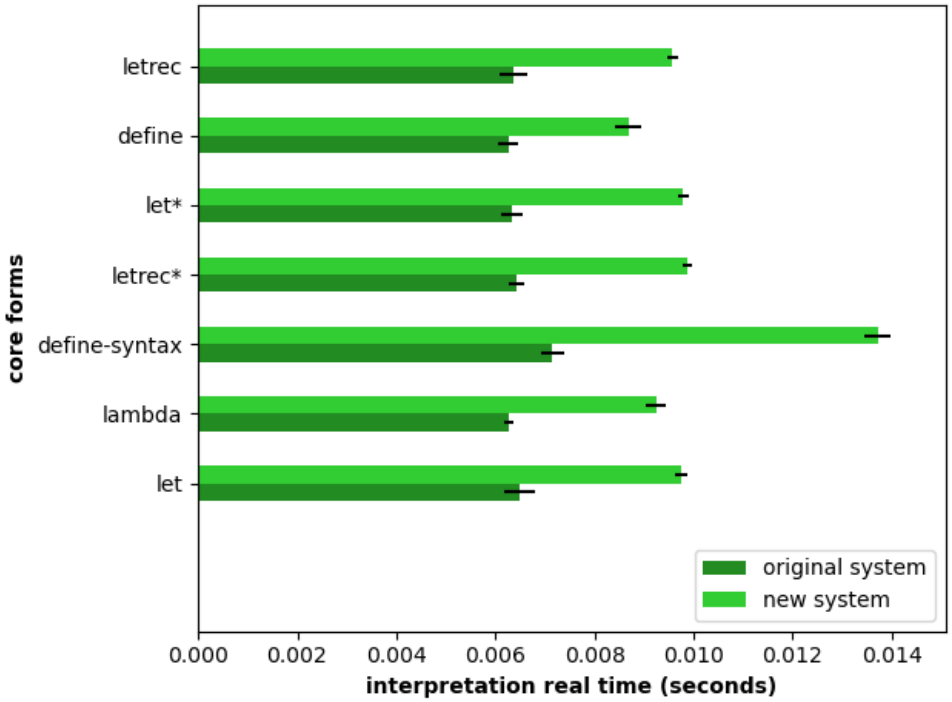
Figure 7.4: interpretation time for programs depending on the number of bindings

We notice that the interpretation time seems linear for this program, as a function of the number of local bindings. Then, we varied the shape of the program to change the ratio of bindings currently in scope for each of the variable references, with similar performance results. For simple transformations, using local macro

transformer bindings, the compiler performs approximately the same as using regular variable binding. This validates the asymptotic complexity of our algorithm for cases which don't involve complex macro transformations.

For the next figure, the interpretation and compilation time compute the time taken to invoke a new process and go through the test file. For compilation, we compute the time needed to compile the Scheme code to C source code only.

The first figure shows performance results for different primitive core forms for interpretation with GSI.



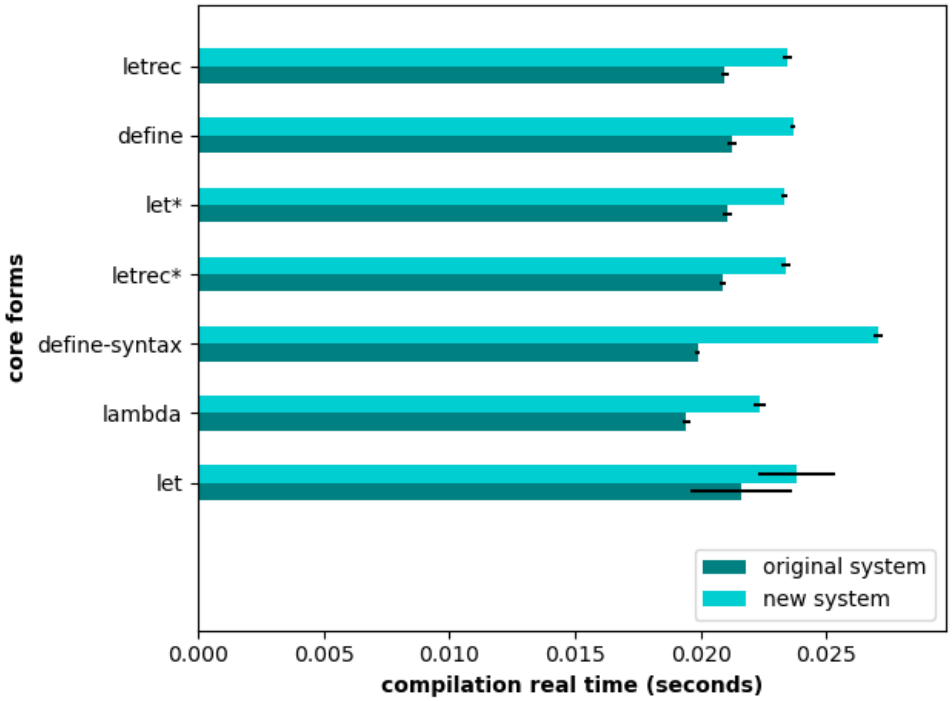
Core forms	Original System		New System	
	average (ms)	σ (ms)	average (ms)	σ (ms)
`lambda`	6.83	0.41	9.19	0.36
`let`	6.93	0.43	9.59	0.20
`let*`	6.64	0.22	9.84	0.44
`letrec`	6.69	0.19	10.03	0.57
`letrec*`	6.84	0.42	9.81	0.17
`define`	6.74	0.31	8.98	0.47
`define-syntax`	7.51	0.21	13.58	0.25

Figure 7.5: interpretation time for different primitive core forms

For these examples, the new system is between 1.2x and 1.9x slower than the original system. We notice an overhead for the `define-syntax` core form. We notice that the `define-syntax` core form has the biggest overhead compared to the original system. This was expected as hygiene support changes the transformer's evaluation phase the most. The other core forms, involving no transformer, compare approxi-

mately the same. This figure combines the results for the system core form as well as the `define-macro` and `define-syntax` forms[1].

The figure 7.6 shows the results of the core-form's tests using the Gambit compiler GSC. Recall that the tests compute the time required to compile the Scheme program into C source code.



Core forms	Original System		New System	
	average (ms)	σ (ms)	average (ms)	σ (ms)
`lambda`	17.46	0.19	20.75	0.22
`let`	19.65	1.90	21.64	0.29
`letrec`	18.93	0.19	21.81	0.30
`let*`	19.24	0.27	21.50	0.12
`letrec*`	18.97	0.22	21.58	0.10
`define`	19.18	0.18	21.91	0.18
`define-syntax`	18.02	0.27	24.99	0.22

Figure 7.6: compilation time for different primitive core form

With the compiler, the comparison between the two systems is best reflected as the execution time of each program isn't included in the total time and as some of the core forms are not fully optimized. We notice that the new system is between 1.1x and 1.4x slower than the original one.

[1]Recall that the `define-macro` form was a core form in the original Gambit system while the `define-syntax` is a core form within the new system.

We notice that the performance difference between the two systems is worst for the `define-syntax` form. We then compared the compilation time for programs made of 50 nested `let-syntax` forms. Each of those programs differs from the next by an increase in the number of binding created and used in each of the program's created transformers.

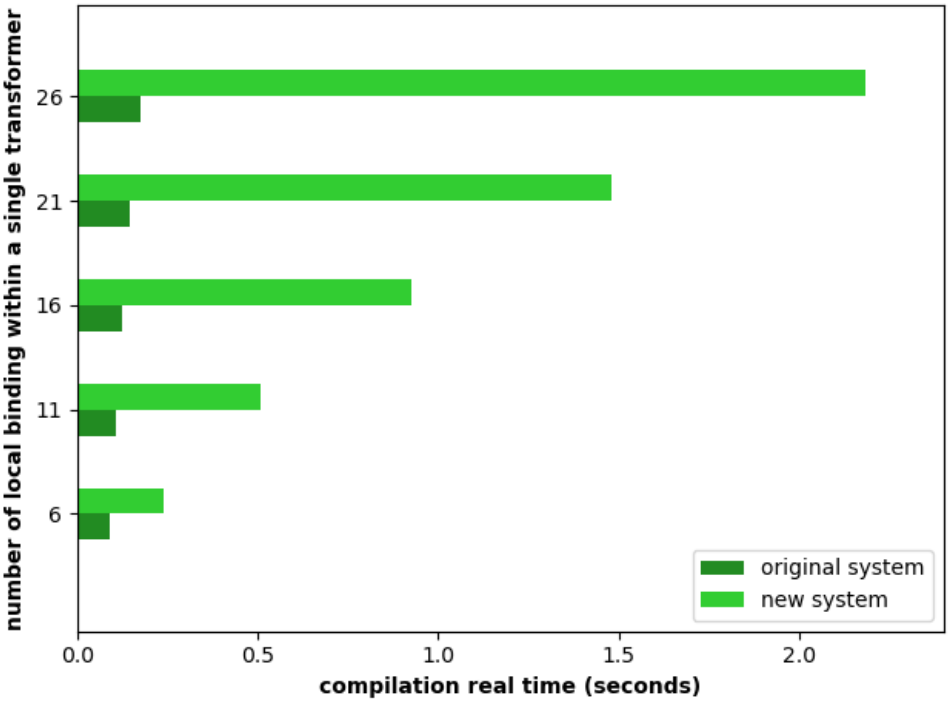


Figure 7.7: compilation time as a function of the number of local bindings within a single transformer

We can notice important performance issues for the new system. However, when comparing the difference for the compilation time of the first two tests, including respectively six and eleven local bindings within a single transformer, with the two last tests, the difference for both the new and original Gambit increased by about 3 times. This leads us to believe that the asymptotic behavior for the two systems' time complexity is very similar.

In conclusion, the new system adds a modest overhead for macro expansion that we believe is acceptable for typical programmers. Firstly, as the new system allows for backward-compatibility, the `eval` procedure uses the old macro expander before completing the regular evaluation. Secondly, the new system in itself requires an additional pass on the fully expanded abstract syntax tree during the *compilation* phase of the set of scopes algorithm, which also includes multiple calls to the `resolve` procedure.

The figure 7.8 shows the results for other commonly used forms defining syntactic extensions in Scheme. In the original Gambit's macro system, the `define-macro` form was a core form while `define-syntax` was built on top of it. In our new system, it is the opposite as `define-macro` is defined in terms of syntax-case: The

`syntax-rules` and `define-macro` forms are implemented as core form based on the `syntax-case` form. These forms require the use of `syntax-case` during the macro-expansion of the compiler compilation and thus, use the unhygienic `syntax-case` included in the original Gambit system.

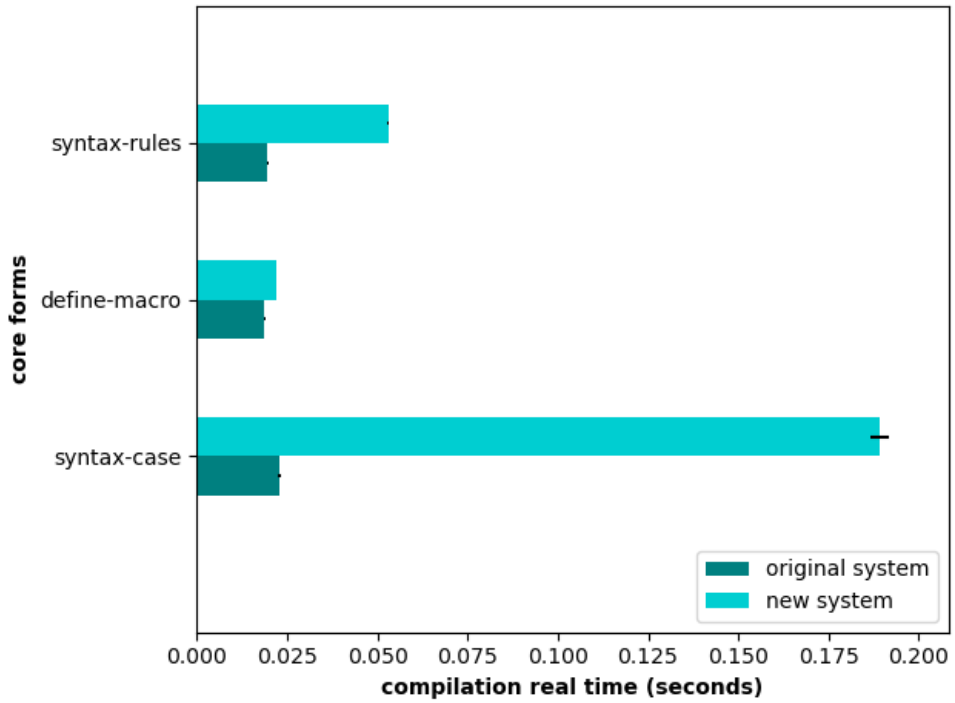


Figure 7.8: compilation time for different forms defining syntactic extensions

The `syntax-case` algorithm is particularly slower than its original counterpart. Indeed, the algorithm implemented is sub-optimal, as only its validity is required for bootstrapping the compiler, as a first step.

This chapter presented the validity tests used to validate our algorithm, as well as performance tests used to compare our new implementation to the original Gambit system. We conclude that our implementation is valid but, less efficient in some key aspects. Regarding our core system, more optimizations are needed to rewrite some of Gambit's functionalities to work with the hygiene system. Following these modifications, the original Gambit's expansion phase can then be merged into our system to achieve backward-compatibility at lower cost.

Chapter 8

Conclusion

Our Contribution

Our contribution is focused on the choice and implementation of the set of scope algorithm in the context of the Gambit system. The new macro system brings the Gambit language closer to the R7RS specification. Many SRFI, and common modules, use macro heavily. When using code provided by another programmer, hygiene guarantee become even more relevant as they might have written the code within a hygienic system. The solution proposed for the hygiene problem helps Gambit achieve both its research and production goals by allowing the safe use of a wider range of existing utility and tools. Furthermore, the new system can be used to introduce programmers to hygiene and macro transformers in general, achieving Gambit's third goal as an educational tool.

In this thesis, we began with a brief presentation of the Scheme language, followed by an introduction to the hygiene problem. We then summarized some of the solutions used in the past and for other Scheme and Lisp implementations. After motivating our choice, we explained Racket's set of scopes algorithm in detail. We then focused on the Gambit system and explained the changes we made to support the algorithm. We finished by supporting the validity of our implementation and provided a short analysis of the performance changes in the system.

Similar Works

Not every Scheme implementation supports hygiene and the strategies used differ widely. As a notable example, we can consider the Chibi compiler, a minimal implementation of Scheme conforming, in most parts, to the R7RS standard. This Scheme implementation support hygiene using a selection of different algorithms, including the syntactic closure algorithm and a variant of the set of scopes algorithm. Other popular Scheme implementations such as the Chez Scheme and Chicken compilers also support hygiene. Both the Chicken Scheme and MIT/GNU Scheme implementations support hygiene using the explicit renaming algorithm [chi21] [H+22].

Toward Future Contributions

Recently, Racket's core was reimplemented using Chez Scheme, a popular implementation of the Scheme language. Both Gambit and Racket aimed toward the same

goals, with a purpose in research, production, and education. The emplementation of the Racket language on top of the Gambit system would allow Racket and its specific tools, to share features and interact with Gambit. As an interesting feature, the Gambit system supports compilation toward *JavaScript*.

A reimplementation of Racket with Gambit would allow support of the Racket language by the online tool easily. As Racket's macro system is at the core of the language, a safe implementation of macro transformation is required for its implementation. Our contribution could be the first step toward this reimplementation.

Bibliography

[ADH+98] H. Abelson, R. K. Dybvig, C.T. Haynes, et al. Revised[5] report on the algorithmic language scheme. *ACM SIGPLAN Lisp Pointers*, 1998.

[BR88] A. Bawden and J. Rees. Syntactic closures. *MIT*, 1988.

[chi] chibi-scheme. `http://synthcode.com/wiki/chibi-scheme`.

[chi21] Module (chicken syntax), 2021. `http://wiki.call-cc.org/man/5/Module%20(chicken%20syntax)#explicit-renaming-macros`.

[CT10] P. Costanza and T.D'Hondt. Embedding hygiene-compatible macros in an unhygienic macro system. 2010.

[Dyb92] R. K. Dybvig. Writing hygienic macros in scheme with syntax-case. *Indiana University*, 1992.

[FH20] M. Feeley and F. Hamel. An r7rs compatible module system for termite scheme. *ELS'20*, 2020.

[Fla16] M. Flatt. Binding as sets of scopes. 2016.

[FM+07] R. B. Findler, J. Matthews, et al. Revised[6] report on the algorithmic language scheme. 2007.

[H+22] C. Hanson et al. Mit/gnu scheme reference manual (release 11.2). 2022.

[Hoy] D. Hoyte. Macro basics. `https://letoverlambda.com/index.cl/guest/chap3.html`.

[Hoy08] D. Hoyte. *Let Over Lambda*. Lulu.com, 2008.

[HW08] D. Herman and M. Wand. *Northeastern University*, 2008.

[KFFD86] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. *Indiana University*, 1986.

[SCG13] A. Shinn, J. Cowan, and A. A. Gleckler. Revised[6] report on the algorithmic language scheme. 2013.

Appendix A

Appendix

A.1 Algorithm formalisation

$$\sigma_i : \textbf{Scope} \qquad x_i : \textbf{Symbol}$$
$$\Sigma_{x_i} : \textbf{Set Scope} \qquad \underline{x_i} : \textbf{Uninterned-Symbol}$$
$$\Sigma_{x_i} ::= \{\} \qquad \overline{\tau_i} : \textbf{Closure}$$
$$\mid \{\sigma_i\} \cup \Sigma_{x_i} \qquad x_i^{\Sigma_{x_i}} : \textbf{Identifier}$$

$$\textbf{Env} ::= \mathbf{B} \times \Gamma$$

$$\beta_i : \textbf{Binding} \qquad \gamma_i : \textbf{CTE-Binding}$$
$$\beta_i ::= \texttt{binding}(x^{\Sigma_{x_i}},\ \underline{x_i}) \qquad \gamma_i ::= \texttt{cte-binding}(\underline{x_i},\ \texttt{var})$$
$$\mid \quad \texttt{cte-binding}(\underline{x_i},\ \texttt{macro}(\overline{\tau_i}))$$
$$\mathbf{B} ::= \{\} \qquad \Gamma ::= \{\}$$
$$\mid \{\beta_i\} \cup \mathbf{B} \qquad \mid \{\gamma_i\} \cup \Gamma$$

$\Theta^{+}[\![x^{\Sigma}]\!]_{\sigma}$

$$\frac{}{\Theta^{+}[\![x^{\Sigma_0}]\!]_{\sigma} = x^{\Sigma_0 \cup \{\sigma_i\}}}$$

$$\frac{}{\Theta^{+}[\![(E_0\ \ E1\ \ ...)]\!]_{\sigma} = (\Theta^{+}[\![E_0]\!]_{\sigma}\ \ \Theta^{+}[\![E_1]\!]_{\sigma}\ \ ...)}$$

$\Theta^{\oplus}[\![x^{\Sigma}]\!]_{\sigma}$

$$\frac{\sigma \notin \Sigma}{\Theta^{\oplus}[\![x^{\Sigma}]\!]_{\sigma} = x^{\Sigma \cup \{\sigma_i\}}} \qquad \frac{\sigma \in \Sigma}{\Theta^{\oplus}[\![x^{\Sigma}]\!]_{\sigma} = x^{\Sigma \setminus \{\sigma_i\}}}$$

$$\frac{}{\Theta^{\oplus}[\![(E_0\ \ E1\ \ ...)]\!]_{\sigma} = (\Theta^{\oplus}[\![E_0]\!]_{\sigma}\ \ \Theta^{\oplus}[\![E_1]\!]_{\sigma}\ \ ...)}$$

Figure A.1: Scope propagation

$\beta^{+}[\![E]\!]^{\mathbf{B}}$

$$\frac{\begin{array}{c}\underline{x} = \texttt{gensym}() \\ \beta = \texttt{binding}(x^{\Sigma},\ \underline{x}), \\ \mathbf{B}' = \mathbf{B} \cup \beta\end{array}}{\beta^{+}[\![x^{\Sigma}]\!]^{\mathbf{B}} \implies (\mathbf{B}',\ \underline{x})}$$

Figure A.2: binding recording in global binding table

$\gamma^{+}[\![\underline{x}]\!]^{\Gamma}$

$$\frac{\gamma = \texttt{cte-binding}(\underline{x},\ \texttt{var}),\ \Gamma' = \Gamma \cup \gamma}{\gamma^{+}[\![\underline{x}]\!]^{\Gamma}_{\texttt{var}} \implies \Gamma'}$$

$$\frac{\gamma = \texttt{cte-binding}(\underline{x},\ \texttt{macro}(\overline{\tau})),\ \Gamma' = \Gamma \cup \gamma}{\gamma^{+}[\![\underline{x}]\!]^{\Gamma}_{\texttt{macro}(\overline{\tau})} \implies \Gamma'}$$

Figure A.3: binding recording in compile-time environment

$\textsc{record}[\![x^{\Sigma}]\!]^{(\mathbf{B},\ \Gamma)}$

$$\frac{\begin{array}{rcl}\beta^{+}[\![x^{\Sigma}]\!]^{\mathbf{B}} & \Longrightarrow & (\mathbf{B}',\ \underline{x}),\\ \gamma^{+}[\![\underline{x}]\!]^{\Gamma}_{\mathtt{var}} & \Longrightarrow & \Gamma'\end{array}}{\textsc{record}[\![x^{\Sigma}]\!]^{(\mathbf{B},\ \Gamma)}_{\mathtt{var}} \Longrightarrow (\mathbf{B}',\ \Gamma')}$$

$$\frac{\begin{array}{rcl}\beta^{+}[\![x^{\Sigma}]\!]^{\mathbf{B}} & \Longrightarrow & (\mathbf{B}',\ \underline{x}),\\ \gamma^{+}[\![\underline{x}]\!]^{\Gamma}_{\mathtt{macro}(\overline{\tau})} & \Longrightarrow & \Gamma'\end{array}}{\textsc{record}[\![x^{\Sigma}]\!]^{(\mathbf{B},\ \Gamma)}_{\mathtt{macro}(\overline{\tau})} \Longrightarrow (\mathbf{B}',\ \Gamma')}$$

Figure A.4: full binding recording

$$\frac{\begin{array}{c}\forall \Sigma_i \in \Theta_{\subset}[\![x^{\Sigma_x}]\!]^{\mathbf{B}}, |\Sigma_{\subset}| \geq |\Sigma_i|\\ \forall \Sigma_i \in \Theta[\![x]\!]^{\mathbf{B}}, \Sigma_{\subset} \subset \Sigma_i\\ \mathtt{binding}(x^{\Sigma_{\subset}},\ \underline{x}_{\subset}) \in \mathbf{B}\end{array}}{\textsc{resolve}[\![x^{\Sigma_x}]\!]^{\mathbf{B}} \Longrightarrow \underline{x}_{\subset}}$$

Figure A.5: binding resolve

$$\frac{\begin{array}{c}\textsc{resolve}[\![x^{\Sigma_x}]\!]^{\mathbf{B}} \Longrightarrow \underline{x}\\ \mathtt{binding}(\underline{x},\ b) \in \Gamma\end{array}}{\textsc{lookup}[\![x^{\Sigma_x}]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow b}$$

Figure A.6: identifier lookup

$$\frac{\begin{array}{c}\phi[\![E_1]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow (B_1,\ E'_1)\\ \phi[\![E_2]\!]^{(\mathbf{B}_1,\ \Gamma)} \Longrightarrow (B_2\ E'_2)\\ \ldots\\ \phi[\![E_n]\!]^{(\mathbf{B}_{n-1},\ \Gamma)} \Longrightarrow (B_n,\ E'_n)\end{array}}{\phi[\![(\mathtt{begin}\ E_1\ E_2\ \ldots\ E_n)]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow (\mathbf{B}_n,\ (\mathtt{begin}\ E'_1\ E'_2\ \ldots\ E'_n))}$$

Figure A.7: Sequencing

$$\frac{\begin{array}{c}\textsc{resolve}[\![\texttt{lambda}^{\Sigma_l}]\!]^{(\mathbf{B},\ \Gamma)} = \texttt{lambda} \\ E = \texttt{(begin}^{\{s\}}\ E_0\ \ldots\ E_n\texttt{)} \\ \sigma = \texttt{genscope}() \\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_{x_0}}]\!]_\sigma]\!]^{(\mathbf{B},\ \Gamma)}_{\texttt{var}} = (\mathbf{B}_0,\ \Gamma_0) \\ (\ldots) = (\ldots) \\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_{x_n}}]\!]_\sigma]\!]^{(\mathbf{B}_{n-1},\ \Gamma_{n-1})}_{\texttt{var}} = (\mathbf{B}_n,\ \Gamma_n) \\ \phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}_n,\ \Gamma_n)} \implies (\mathbf{B}',\ E')\end{array}}{\begin{array}{c}\phi[\![\texttt{(lambda}^{\Sigma_l}\ \texttt{(}x^{\Sigma_0}\ \texttt{...}\ x^{\Sigma_n}\texttt{)}\ E_0\ E_1\ \texttt{...}\ E_n\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \\ \implies \\ (\mathbf{B}',\ \texttt{(lambda}^{\Sigma_l}\ \texttt{(}\Theta^+[\![x^{\Sigma_0}]\!]_\sigma\ \ldots\ \Theta^+[\![x^{\Sigma_n}]\!]_\sigma\texttt{)}\ E'\texttt{)})\end{array}}$$

Figure A.8: lambda abstraction

$$\frac{\begin{array}{rl}\forall\overline{\tau}.\ \textsc{lookup}[\![E_0]\!]^{(\mathbf{B},\ \Gamma)} & \neq \texttt{macro}(\overline{\tau}) \\ \phi[\![E_0]\!]^{(\mathbf{B},\ \Gamma)} & \implies (\mathbf{B}_0,\ E'_0) \\ \phi[\![E_1]\!]^{(\mathbf{B}_0,\ \Gamma)} & \implies (\mathbf{B}_1,\ E'_1) \\ \ldots & \implies \ldots \\ \phi[\![E_n]\!]^{(\mathbf{B}_{n-1},\ \Gamma)} & \implies (\mathbf{B}_n,\ E'_n)\end{array}}{\phi[\![\texttt{(}E_0,\ E_1,\ \ldots,\ E_n\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \implies (\mathbf{B}_n,\ \texttt{(}E'_0,\ E'_1,\ \ldots,\ E'_n\texttt{)})}$$

Figure A.9: Application Form

$$\frac{\begin{array}{c}\texttt{(let}^{\Sigma_l}\ \texttt{((}x_0^{\Sigma_0}\ E_0\texttt{)}\ \ldots\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ E_0\ \ldots\ E_n\texttt{)} \\ \Longrightarrow \\ \texttt{(let}^{\Sigma_l}\ \texttt{((}x_0^{\Sigma_0}\ E_0\texttt{)}\ \ldots\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ \texttt{(begin}^{\{s\}}\ E_0\ \ldots E_n\texttt{))}\end{array}}{\begin{array}{c}\textsc{resolve}[\![\texttt{let}^{\Sigma_l}]\!]^{\mathbf{B}} \Longrightarrow \texttt{let} \\ \sigma = \texttt{genscope}() \\ \phi[\![E_x]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow (\mathbf{B}',\ E_x') \\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B}',\ \Gamma)}_{\texttt{var}} \Longrightarrow (\mathbf{B}'',\ \Gamma') \\ \phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}'',\ \Gamma')} \Longrightarrow (\mathbf{B}''',\ E') \\ \hline \phi[\![\texttt{(let}^{\Sigma_l}\ \texttt{((}x^{\Sigma_x}\ E_x\texttt{))}\ E\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow (\mathbf{B}''',\ \texttt{(let}^{\Sigma_l}\ \texttt{((}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E_x'\texttt{))}\ E'\texttt{))}\end{array}}$$

Figure A.10: local definition

$$\frac{\begin{array}{c}\phi[\![\mathbf{B},\ \Gamma,\ \texttt{(let-syntax}^{\Sigma_l}\ \texttt{((}x_0^{\Sigma_{x_0}}\ E_{x_0}\texttt{)}\ \ldots\ \texttt{(}x_n^{\Sigma_{x_n}}\ E_{x_n}\texttt{))}\ E_0\ E_1\ \ldots\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \\ \Longrightarrow \\ \phi[\![\texttt{(let-syntax}^{\Sigma_l}\texttt{((}x_0^{\Sigma_{x_0}}\ E_{x_0}\texttt{))} \\ \quad \texttt{(let-syntax}^{\Sigma_l}\texttt{((}x_1^{\Sigma_{x_1}}\ E_{x_1}\texttt{))} \\ \quad\quad \ldots \\ \quad\quad \texttt{(let-syntax}^{\Sigma_l}\ \texttt{((}x_n^{\Sigma_{x_n}}\ E_{x_n}\texttt{))} \\ \quad\quad\quad \texttt{(begin}^{\{s\}}\ E_0\ E_1\ \ldots\texttt{))}\ldots\texttt{))}]\!]^{(\mathbf{B},\ \Gamma)}\end{array}}{\begin{array}{c}\textsc{resolve}[\![\texttt{let-syntax}^{\Sigma_l}]\!]^{\mathbf{B}} \Longrightarrow \texttt{let-syntax} \\ \sigma = \texttt{genscope}() \\ \phi[\![E_x]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow (\mathbf{B}',\ E_x') \\ \texttt{eval*}[\![E_x']\!]^{(\mathbf{B},\ \Gamma)} = (\mathbf{B}',\ \overline{\tau}) \\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B}',\ \Gamma)}_{\texttt{macro}(\overline{\tau})} \Longrightarrow (\mathbf{B}'',\ \Gamma') \\ \phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}'',\ \Gamma')} \Longrightarrow (\mathbf{B}''',\ E') \\ \hline \phi[\![\texttt{(let-syntax}^{\Sigma_l}\ \texttt{((}x^{\Sigma_x}\ E_x\texttt{))}\ E\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \Longrightarrow (\mathbf{B}''',\ E')\end{array}}$$

Figure A.11: local macro definition

$$\frac{\begin{array}{c}\textsc{lookup}[\![M]\!]^{(\mathbf{B},\ \Gamma)} = \texttt{macro}(\overline{\tau}) \\ \sigma = \texttt{genscope}() \\ \texttt{eval*}[\![(\overline{\tau}\ \texttt{(}M\ \ E_0\ \ ...\ \ E_n\texttt{))}]\!]^{(\mathbf{B},\ \Gamma)} = M'\end{array}}{\phi[\![\texttt{(}M,\ \ E_0,\ \ ...,\ \ E_n\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \implies \phi[\![\Theta^{\oplus}[\![M']\!]_{\sigma}]\!]^{(\mathbf{B},\ \Gamma)}}$$

Figure A.12: Macro Expansion

$$\frac{}{\texttt{(quote}\ E\texttt{)} \implies \texttt{(quote}\ E\texttt{)}}$$

$$\frac{}{\texttt{(quote-syntax}\ E\texttt{)} \implies \texttt{(quote-syntax}\ E\texttt{)}}$$

Figure A.13: `quote` and `quote-syntax` special form

$$\frac{}{\begin{array}{c}\phi[\![\mathbf{B},\ \Gamma,\ \texttt{(let}^{*\Sigma_l}\ \texttt{((}x_0^{\Sigma_{x_0}}\ \ E_{x_0}\texttt{)}\ ...\ \texttt{(}x_n^{\Sigma_{x_n}}\ \ E_{x_n}\texttt{))}\ E_0\ \ E_1\ \ ...\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \\ \implies \\ \phi[\![\texttt{(let}^{\Sigma_l}\texttt{((}x_0^{\Sigma_{x_0}}\ \ E_{x_0}\texttt{))} \\ \quad \texttt{(let}^{\Sigma_l}\texttt{((}x_1^{\Sigma_{x_1}}\ \ E_{x_1}\texttt{))} \\ \quad\quad ... \\ \quad\quad \texttt{(let}^{\Sigma_l}\ \texttt{((}x_n^{\Sigma_{x_n}}\ \ E_{x_n}\texttt{))} \\ \quad\quad\ \texttt{(begin}^{\{s\}}\ \ E_0\ \ E_1\ \ ...\texttt{))...))}]\!]^{(\mathbf{B},\ \Gamma)}\end{array}}$$

Figure A.14: `let*` form definition

$$\frac{}{\begin{array}{c}\texttt{(letrec}^{\Sigma_l}\ \texttt{((}x_0^{\Sigma_0}\ E_0\texttt{)}\ \texttt{...}\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ E_0\ ...E_n\texttt{)} \\ \Longrightarrow \\ \texttt{(letrec}^{\Sigma_l}\ \texttt{((}x_0^{\Sigma_0}\ E_0\texttt{)}\ \texttt{...}\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ \texttt{(begin}^{\{s\}}\ E_0\ ...E_n\texttt{))}\end{array}}$$

$$\frac{\begin{array}{c}\textsc{resolve}[\![\texttt{letrec}^{\Sigma_l}]\!]^{\mathbf{B}} \Longrightarrow \texttt{letrec} \\ \sigma = \texttt{genscope}() \\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B},\ \Gamma)}_{\mathsf{var}} \Longrightarrow (\mathbf{B}_0,\ \Gamma_0) \\ \forall i \in [1,n],\ \textsc{record}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B}_{i-1},\ \Gamma_{i-1})}_{\mathsf{var}} \Longrightarrow (\mathbf{B}_i,\ \Gamma_i) \\ \phi[\![E_0]\!]^{(\mathbf{B}_n,\ \Gamma_n)} \Longrightarrow (\mathbf{B}'_0,\ E'_x) \\ \forall i \in [1,n],\ \phi[\![E_i]\!]^{(\mathbf{B}'_{i-1},\ \Gamma_n)} \Longrightarrow (\mathbf{B}'_i,\ E'_i) \\ \phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}'_n,\ \Gamma_n)} \Longrightarrow (\mathbf{B}'',\ E')\end{array}}{\begin{array}{c}\phi[\![\texttt{(letrec}^{\Sigma_l}\ \texttt{((}x^{\Sigma_0}\ E_0\texttt{)}\ ...\ \texttt{(}x^{\Sigma_n}\ E_n\texttt{))}\ E\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \\ \Longrightarrow \\ (\mathbf{B}'',\ \texttt{(letrec}^{\Sigma_l}\ \texttt{((}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{)}\ ...\ \texttt{(}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{))}\ E'\texttt{)})\end{array}}$$

Figure A.15: `letrec` form definition

$$\frac{\begin{array}{c}\texttt{(letrec*}^{\Sigma_l}\ \texttt{((}x_0^{\Sigma_0}\ E_0\texttt{)}\ \texttt{...}\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ E_0\ ...E_n\texttt{)} \\ \Longrightarrow \\ \texttt{(letrec*}^{\Sigma_l}\ \texttt{((}x_0^{\Sigma_0}\ E_0\texttt{)}\ \texttt{...}\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ \texttt{(begin}^{\{s\}}\ E_0\ ...E_n\texttt{))}\end{array}}{\begin{array}{c}\text{RESOLVE}[\![\texttt{letrec*}^{\Sigma_l}]\!]^{\mathbf{B}} \Longrightarrow \texttt{letrec*} \\ \sigma = \texttt{genscope}() \\ \\ \text{RECORD}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]_{\texttt{var}}^{(\mathbf{B},\ \Gamma)} \Longrightarrow (\mathbf{B}_0,\ \Gamma_0) \\ \phi[\![E_0]\!]^{(\mathbf{B}_0,\ \Gamma_0)} \Longrightarrow (\mathbf{B}'_0,\ E'_x) \\ \\ \forall i \in [1,n],\ \text{RECORD}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]_{\texttt{var}}^{(\mathbf{B}'_{i-1},\ \Gamma_{i-1})} \Longrightarrow (\mathbf{B}_i,\ \Gamma_i) \\ \forall i \in [1,n],\ \phi[\![E_i]\!]^{(\mathbf{B}_i,\ \Gamma_i)} \Longrightarrow (\mathbf{B}'_i,\ E'_i) \\ \\ \phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}'_n,\ \Gamma_n)} \Longrightarrow (\mathbf{B}'',\ E')\end{array}}{\begin{array}{c}\phi[\![\texttt{(letrec*}^{\Sigma_l}\ \texttt{((}x^{\Sigma_0}\ E_0\texttt{)}\ ...\ \texttt{(}x^{\Sigma_n}\ E_n\texttt{))}\ E\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \\ \Longrightarrow \\ (\mathbf{B}'',\ \texttt{(letrec*}^{\Sigma_l}\ \texttt{((}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{)}\ ...\ \texttt{(}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{))}\ E'\texttt{))}\end{array}}$$

Figure A.16: `letrec*` form definition

$$\frac{}{\begin{array}{c}\texttt{(letrec*-syntax}^{\Sigma_l}\ \texttt{((}x_0^{\Sigma_0}\ E_0\texttt{)}\ \texttt{...}\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ E_0\ ...E_n\texttt{)} \\ \Longrightarrow \\ \texttt{(letrec*-syntax}^{\Sigma_l}\ \texttt{((}x_0^{\Sigma_0}\ E_0\texttt{)}\ \texttt{...}\ \texttt{(}x_n^{\Sigma_n}\ E_n\texttt{))}\ \texttt{(begin}^{\{s\}}\ E_0\ ...E_n\texttt{))}\end{array}}$$

$$\text{RESOLVE}[\![\texttt{letrec*-syntax}^{\Sigma_l}]\!]^{\mathbf{B}} \Longrightarrow \texttt{letrec*-syntax}$$
$$\sigma = \texttt{genscope}()$$

$$\text{RECORD}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B},\ \Gamma)}_{\texttt{macro(unbound)}} \Longrightarrow (\mathbf{B}_0,\ \Gamma_0)$$
$$\text{EVAL}[\![E_0]\!]^{\mathbf{B}_0,\ \Gamma_0} \Longrightarrow \overline{\tau}_0$$
$$\text{RECORD}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B}_0,\ \Gamma_0)}_{\texttt{macro}(\overline{\tau}_0)} \Longrightarrow (\mathbf{B}'_0,\ \Gamma'_0)$$

$$\forall i \in [1, n],\ \text{RECORD}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B}'_{i-1},\ \Gamma'_{i-1})}_{\texttt{macro(unbound)}} \Longrightarrow (\mathbf{B}_i,\ \Gamma_i)$$
$$\text{EVAL}[\![E_i]\!]^{\mathbf{B}_i,\ \Gamma_i} \Longrightarrow \overline{\tau}_i$$
$$\text{RECORD}[\![\Theta^+[\![x^{\Sigma_x}]\!]_\sigma]\!]^{(\mathbf{B}_i,\ \Gamma_i)}_{\texttt{macro}(\overline{\tau})} \Longrightarrow (\mathbf{B}'_i,\ \Gamma'_i)$$

$$\phi[\![\Theta^+[\![E]\!]_\sigma]\!]^{(\mathbf{B}'_n,\ \Gamma'_n)} \Longrightarrow (\mathbf{B}'',\ E')$$

$$\frac{}{\begin{array}{c}\phi[\![\texttt{(letrec*-syntax}^{\Sigma_l}\ \texttt{((}x^{\Sigma_0}\ E_0\texttt{)}\ \texttt{...}\ \texttt{(}x^{\Sigma_n}\ E_n\texttt{))}\ E\texttt{)}]\!]^{(\mathbf{B},\ \Gamma)} \\ \Longrightarrow \\ (\mathbf{B}'',\ \texttt{(letrec*-syntax}^{\Sigma_l}\ \texttt{((}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{)}\ \texttt{...}\ \texttt{(}\Theta^+[\![x^{\Sigma_x}]\!]_\sigma\ E'_x\texttt{))}\ E'\texttt{))}\end{array}}$$

Figure A.17: `letrec*-syntax` form definition

A.2 Gambit's original `interaction-cte`

cte : CTE
cte ::= `top-cte`(*previous-cte*)
| `cte-frame`(*previous-cte*, *names*, *up*, *over*)
| `cte-macro`(*previous-cte*, *name*, *transformer*)
| `cte-namespace`(*previous-cte*, *name*, *aliases*)
| `cte-decl`(*previous-cte*, *name*, *value*)

`top-cte` : **CTE** → **CTE**
`cte-frame` : **CTE** → (**List Symbol**) → **Fixnum** → **Fixnum** → **CTE**
`cte-macro` : **CTE** → **Symbol** → (**Syntax** → **Syntax**) → **CTE**
`cte-namespace` : **CTE** → **Symbol** → **List** (**Pair Symbol Symbol**) → **CTE**
`cte-decl` : **CTE** → **Symbol** → **Symbol** → **CTE**

Figure A.18: Gambit's Original `interaction-cte`

A.3 Gambit's new `interaction-cte`

cte : CTE
cte ::= `top-cte`(*previous-cte*, **B**, Γ)
| `cte-frame`(*previous-cte*, Γ, *cte-top*, *names*, *up*, *over*)
| `cte-macro`(*previous-cte*, Γ, *cte-top*, *name*, *transformer*)
| `cte-core-macro`(*previous-cte*, Γ, *cte-top*, *name*, *transformer*)
| `cte-namespace`(*previous-cte*, *name*, *aliases*)
| `cte-decl`(*previous-cte*, *name*, *value*)

`top-cte` : **CTE** → Γ → **B** → **CTE**
`cte-frame` : **CTE** → Γ → **CTE** → (**List Symbol**) → **Fixnum** → **Fixnum** → **CTE**
`cte-macro` : **CTE** → Γ → **CTE** → **Symbol** → (**Syntax** → **Syntax**) → **CTE**
`cte-core-macro` : **CTE** → Γ → **CTE** → **Symbol** → (**Syntax** → **Syntax**) → **CTE**
`cte-namespace` : **CTE** → **Symbol** → **List** (**Pair Symbol Symbol**) → **CTE**
`cte-decl` : **CTE** → **Symbol** → **Symbol** → **CTE**

Figure A.19: Gambit's New `interaction-cte`