Computation At Compile-Time

and the Implementation of Libbug

Bill Six

For Mom and Dad. Thanks for everything.

Preface

This is a book about compiler design for people who have no interest in studying compiler design. ...Umm, then who wants to read this book? Let me try this again... This book is the study of source code which is discarded by the compiler, having no representation in the generated machine code. ...Ummm, still not right... This book is about viewing a compiler not only as a means of translating source code into machine code, but also viewing it as an interpreter capable of any general purpose computation. ...Closer, but who cares?... I think I got it now. This is a book about "Testing at Compile-Time"!

What do I mean by that? Let's say you're looking at source code with which you are unfamiliar, such as the following:

```
{define permutations
 [|l|
   (if (null? l)
       ['()]
       [{let permutations ((l l))
          (if (null? (cdr l))
              [(list l)]
              [(flatmap [|x|
                         (map [|y| (cons x y)]
                              (permutations (remove x l)))]
                        l)])}])]
```

What does the code do? How did the author intend for it to be used? In trying to answer those questions, fans of statically-typed programming languages might lament the lack of types, as types help them to reason about programs and help them to deduce where to look to find more information. In trying to answer those questions, fans of dynamically-typed languages might argue "Look at the tests!", as tests ensure the code functions in a user-specified way and they serve as a form of documentation. But where are those tests? Probably in some other file whose filesystem path is similar to the current file's path (e.g., src/com/BigCorp/HugeProject/Foo.java is tested by test/com/BigCorp/HugeProject/FooTest.java). Then you'd have to find the file, open the file, look through it while ignoring tests which are for other methods. Frankly, it's too much work and it interrupts the flow of coding, at least for me.

But how else would a programmer organize tests? Well in this book, which is the implementation of a library called "libbug", tests are specified as part of the procedure's definition and they are executed at compile-time. Should any test fail the compiler will exit in error, like a type error in a statically-typed language. Furthermore, the book you are currently reading is embedded into the source code of libbug; it is generated only upon successful compilation of libbug and it couldn't exist if a single test failed.

So where are these tests then? The very alert reader may have noticed that the opening '{' in the definition of "permutations" was not closed. That is because the definition of "permutations" is completed by specifying tests to be run at compile-time.

```
(equal? (permutations '())
        '())
(equal? (permutations '(1))
        '((1)))
(equal? (permutations '(1 2))
        '((1 2)
          (2 1)))
(equal? (permutations '(1 2 3))
        '((1 2 3)
          (1 3 2)
          (2 1 3)
          (2 3 1)
          (3 1 2)
          (3 2 1)))}
```

Why does this matter? Towards answering the questions “what does the code do?” and “how did the author intend for it to be used?”, there is neither searching through files nor guessing how the code was originally intended to be used. The fact that the tests are collocated with the procedure definition means that the reader can read the tests without switching between files, perhaps before reading the procedure’s definition. And the reader may not even read the procedure at all if the tests gave him enough information to use it successfully. Should the reader want to understand the procedure, he can mentally apply the procedure to the tests to understand it.

Wait a second. If those tests are defined in the source code itself, won’t they be in the executable? And won’t they run every time I execute the program? That would be unacceptable as it would both increase the size of the binary and slow down the program at start up. Fortunately the answer to both questions is no, because in chapter 8 I show how to specify that certain code should be interpreted by the compiler instead of being compiled. Lisp implementations such as Gambit are particularly well suited for this style of programming because unevaluated Lisp code is specified using a data structure of Lisp; because the compiler is an interpreter capable of being augmented with the same code which it is compiling. Upon finishing compilation, the compiler has *become* the very program it is compiling.

Contents

CHAPTER 1

Introduction

Libbug is Bill's Utilities for Gambit Scheme: a "standard library" of procedures which augments Scheme's small set of built-in procedures. Libbug provides procedures for list processing, streams, control structures, general-purpose evaluation at compile-time, and a compile-time test framework written in only 9 lines of code! Programs written using libbug optionally may be programmed in a relatively unobstructive "literate programming"[1] style, so that a program can be read linearly in a book form.

1.1 Prerequisites

The reader is assumed to be somewhat familiar with Scheme, with Common Lisp-style macros, and with recursive design. If the book proves too difficult for you, read "Simply Scheme" [Harvey01][2] or "The Little Schemer" [Friedman96]. Since libbug uses Gambit Scheme's Common Lisp-style macros, the author recommends reading "On Lisp" [Graham94][3]. The other books listed in the bibliography, all of which inspired ideas for this book, are all recommended reading but are not necessary to understand the content of this book.

1.2 Parts

This book is a "literate program". New procedures defined are dependent upon either standard Gambit Scheme procedures or procedures which have already been defined earlier in the book. In writing the book, however, it became quite apparent that the foundation upon which libbug is constructed is by far the most difficult material. Reading it in a completely linear format would cause the reader to get lost in the "how", before understanding "why".

As such, the ordering of the book was rearranged in an effort to keep the reader engaged and curious. The book begins with "Part 1, The Implementation of Libbug" and ends with "Part 2, Foundations Of Libbug" The Gambit compiler, however, compiles Part 2 first, then Part 1.

1.3 Conventions

Code which is part of libbug will be outlined and will have line numbers on the left.

[1] http://lmgtfy.com/?q=literate+programming
[2] available on-line for no cost
[3] available on-line for no cost

```
;; This is part of libbug.
```

Example code which is not part of libbug will not be outlined nor will it have line numbers.

```
(+ 1 ("This is NOT part of libbug"))
```

Some examples within this book show sessions of the use of the "bug-gsi" Read-Evaluate-Print-Loop (REPL). Such examples will look like the following:

```
> (+ 1 2)
3
```

The line on which the user entered text begins with a ">". The result of evaluating that line appears on the subsequent line. In this case, 1 added to 2 evaluates to 3.

1.3.1 Syntactic Conventions

In libbug, the notation

```
(fun arg1 arg2)
```

means evaluate "fun", "arg1" and "arg2" in any order, then apply "fun" to "arg1" and "arg2"; standard Scheme semantics for invoking a procedure. But since macros are not normal procedures and do not necessarily respect those semantics, in libbug, the notation

```
{fun1 arg1 arg2}
```

is used to denote to the reader that the standard evaluation rules do not apply. For instance, in

```
{define x 5}
```

{} are used because "x" may be a new variable. As such, "x" cannot currently evaluate to anything.

Not all macro applications use {}. If the macro respects Scheme's standard order of evaluation, macro application will use standard Scheme notation:

```
((compose [|x| (* x 2)]) 5)
```

1.4 Getting the Source Code

The Scheme source code is located at http://github.com/billsix/bug[4]. The Scheme files produce the libbug library, as well as this book. Currently the code works on various distributions of Linux, on FreeBSD, and on Mac OS X. The build currently does not work on Windows.

You will need a C compiler such as GCC, Autoconf, Automake, and Gambit Scheme[5] version 4.8 or newer.

To compile the book and library, execute the following on the command line:

[4]This book was generated from git commit 2dc565f8b90feb22eb9209668283ac1bd4c4a6b8

[5]http://gambitscheme.org

```
$ ./autogen.sh
$ ./configure --prefix=$BUG_HOME --enable-pdf
$ make
$ make install
```

- The argument to “prefix” tells Autoconf the location into which libbug should be installed when “make install” is executed. “$BUG_HOME” is an environment variable that I have not defined, so the reader should substitute “$BUG_HOME” with an actual filesystem path.
- “–enable-pdf” means to build this book as a PDF. To disable the creation of the PDF, substitute “–enable-pdf=no”.

1.5 Comparison of Compile-Time Computations in Other Languages

What exactly is computation at compile-time? An introduction to the topic is provided in Appendix A, demonstrated in languages of more widespread use (C and C++), along with a comparison of their expressive power.

Part I

The Implementation of Libbug

CHAPTER 2

Introductory Procedures

This chapter begins the definition of libbug's standard library of Scheme procedures and macros[1], along with tests which are run as part of the compilation process. If any test fails, the compiler will exit in error, much like a type error in a statically-typed language.

To gain such functionality libbug cannot be defined using Gambit Scheme's "##define", "##define-macro", and "##define-structure", since they only define variables and procedures for use at run-time[2]. Instead, definitions within libbug use "libbug-private#define", "libbug-private#define-macro", and "libbug-private##define-structure"[3], which are implemented in Chapter 8. How they are implemented is not relevant yet, since the use of these procedure-defining procedures will be explained incrementally.

```
1 (include "bug-language.scm")
2 {##namespace ("libbug-private#" define define-macro define-structure)}
3 {##namespace ("bug#" if)}
```

- On line 1, the code which makes computation at compile-time possible is imported. The contents of that file are in Chapter 8.
- On line 2, Gambit's "##namespace" procedure is invoked, to tell the compiler that all subsequent uses of "define", "define-macro", and "define-structure" shall use libbug's version of those procedures instead of Gambit's.
- On line 3, all subsequent uses of "if" shall use libbug's version.

[1]The code within chapters 2 through 7 inclusive is found in "src/main.bug.scm".

[2]well... that statement is not true for "##define-macro", but it makes for a simpler explanation upon first reading

[3]Per convention within libbug, procedures namespaced to "libbug-private" are not compiled into the library or other output files; such procedures are meant for private use within the implementation of libbug.

2.1 noop

The first definition is “noop”, a procedure which takes no arguments and which evaluates to the symbol ’noop.

```
1 {define noop
2   ['noop]}
```

- On line 1, the libbug-private#define macro[4] is invoked.
- On line 1, the variable name “noop”.
- On line 2, the lambda literal to be stored into the variable. Libbug includes a Scheme preprocessor “bug-gscpp”, which expands lambda literals into lambdas. In this case

  ```
  ['noop]
  ```

  is expanded into

  ```
  (lambda () 'noop)
  ```

```
1 {unit-test
2   (equal? (noop) 'noop)}
```

- On line 1, an expression which evaluates to a boolean is defined. This is a test which will be evaluated at compile-time. Should the test fail, the build process will fail and neither the shared library nor the document which you are currently reading will be created. Tests are not present in the created library.

“noop” does not look useful at first glance, but it is frequently used when a procedure is required but the resulting value of it is not. For instance, “noop” is used as a default “exception-handler” for many procedures within libbug.

[4]defined in section 8.9

2.2 identity

“identity” is a procedure of one argument which evaluates to its argument. [Church51, p. 2]

```
1 {define identity
2   [|x| x]}
```

- On line 2, “bug-gscpp” expands

  ```
  [|x| x]
  ```

  to

  ```
  (lambda (x) x)
  ```

  This expansion works with multiple arguments, as long as they are between the “|”s [5].

libbug-private#define can take more than one test as parameters.

```
{unit-test
 (equal? "foo" (identity "foo"))
 (equal? identity (identity identity))}
```

[5]Since “bug-gscpp” uses “|”s for lambda literals, Scheme’s block comments are not allowed in libbug programs

2.3 all?

Like regular Scheme's "and", but takes a list instead of a variable number of arguments, and all elements of the list are evaluated before "all?" is applied.

```
1 {define all?
2   [|l|
3    (if (null? l)
4        [#t]
5        [(if (not (car l))
6             [#f]
7             [(all? (cdr l))])])]}
```

- On line 3, "if", which is currently namespaced to "bug#if"[6], takes lambda expressions for the two parameters. Libbug pretends that #t and #f are "Church Booleans" [Pierce02, p. 58], and that "bug#if" is just syntactic sugar:

  ```
  {define #t [|t f| (t)]}
  {define #f [|t f| (f)]}
  {define bug#if [|b t f| (b t f)]}
  ```

  As such, "bug#if" would not be a special form, and is more consistent with the rest of libbug.

```
{unit-test
 (all? '())
 (all? '(1))
 (all? '(#t))
 (all? '(#t #t))
 (not (all? '(#f)))
 (not (all? '(#t #t #t #f)))}
```

Tests in libbug are defined for two purposes. Firstly, to ensure that expected behavior of a procedure does not change when that procedure's internal definition has changed. Secondly, as a form of documentation of the procedure. Libbug is unique[7] in that the tests are collocated with the procedure definitions. The reader is encouraged to read the tests for a procedure before reading the implementation; since in many cases, the tests are designed specifically to guide the reader through the implementation.

[6]defined in section 8.7
[7]as far as the author knows

2.4 satisfies?

When writing multiple tests, why explicitly invoke the procedure repeatedly with varying inputs and outputs, as was done for "all?"? Instead, provide the procedure and a list of input/output pairs.

```
{define satisfies?
  [|f list-of-pairs|
   (all? (map [|pair| (equal? (f (car pair))
                              (cadr pair))]
              list-of-pairs))]}
```

[8]

```
{unit-test
 (satisfies?
  [|x| (+ x 1)]
  '(
    (0 1)
    (1 2)
    (2 3)
    ))
 (satisfies?
  all?
  '(
    (() #t)
     ((1) #t)
     ((#t) #t)
     ((#t #t) #t)
     ((#f) #f)
     ((#t #t #t #f) #f)))
 }
```

For the remaining procedures, if the tests do an adequate job of explaining the code, there will be no written documentation.

[8]Within libbug, a parameter named "f" usually means the parameter is a procedure.

2.5 while

Programmers who are new to the Scheme language may be surprised that the language provides no built-in syntax for looping, such as “for” or “while”. A better question though, is why don’t other languages provide primitives from which you can create those looping constructs yourself? “Take the red pill.”

```
{define while
  [|pred? body|
   {let while ((val 'noop))
     (if (pred?)
         [(while (body))]
         [val])}]}
```

[9]

```
{unit-test
 {let ((a 0))
   {and (equal? (while [(< a 5)]
                       [{set! a (+ a 1)}])
                #!void)
        (equal? a 5)}}
 {let ((a 0))
   {and (equal? (while [(< a 5)]
                       [{set! a (+ a 1)}
                        'foo])
                'foo)
        (equal? a 5)}}}
```

[9]Within libbug, a parameter named “pred?” or “p?” usually means the parameter is a predicate, meaning a procedure which returns true or false.

2.6 numeric-if

A conditional expression for numbers, based on their sign. “numeric-if” uses Gambit’s keyword syntax. “ifPositive”, “ifZero”, and “ifNegative” are an optionals argument, each with their default value as the value in the “noop” variable.

```
{define numeric-if
  [|n #!key (ifPositive noop) (ifZero noop) (ifNegative noop)|
   (if (> n 0)
       [(ifPositive)]
       [(if (= n 0)
            [(ifZero)]
            [(ifNegative)])])]}
```

[Graham94, p. 150, called “nif”]

Keyword arguments are optionally defined, and use the following syntax.

```
{unit-test
 (satisfies?
  [|n|
   (numeric-if n
               ifPositive: ['pos]
               ifZero: ['zero]
               ifNegative: ['neg])]
  '(
    (5 pos)
    (0 zero)
    (-5 neg)
    ))}
```

2.7 atom?

```
{define atom?
  [|x|
   {or (number? x)
       (symbol? x)}]}
```

[10]

```
{unit-test
 (satisfies?
  atom?
  '(
    (1 #t)
    (1/3 #t)
    (a #t)
    ((make-vector 3) #f)
    (() #f)
    ((a) #f)
    ))
 }
```

[10]Within libbug, a parameter named "x" usually means the parameter can be of any type.

2.8 complement

```
{define complement
  [|f|
   [|#!rest args|
     (not (apply f args))]]}
```

[Graham94, p. 63]

```
{unit-test
 (satisfies?
  pair?
  '(
     (1 #f)
     ((1 2) #t)
     ))
 (satisfies?
  (complement pair?)
  '(
     (1 #t)
     ((1 2) #f)
     ))
 }
```

CHAPTER 3

Lists

3.1 copy

Creates a shallow copy of the list.

```
{define copy
  [|l|
   (map identity l)]}
```

[1]

```
{unit-test
 {let ((a '(1 2 3 4 5)))
   {and (equal? a (copy a))
        (not (eq? a (copy a)))}}
 }
```

For a thorough description of “equal?” vs “eq?”, see [Dybvig03, p. 122-129].

[1]Within libbug, a parameter named “l” usually means the parameter is is a list.

3.2 proper?

Tests that the last element of the list is the sentinel value "'()". Will not terminate on a circular list.

```
{define proper?
  [|l|
   (if (null? l)
       [#t]
       [(if (pair? l)
            [(proper? (cdr l))]
            [#f])])]}
```

```
{unit-test
 (satisfies?
  proper?
  '(
    (() #t)
    ((4) #t)
    ((1 2) #t)
    (4 #f)
    ((1 2 . 5) #f)
    ))}
```

3.3 first

```
{define first
  [|l #!key (onNull noop)|
   (if (null? l)
       [(onNull)]
       [(car l)])]}
```

[Harvey01, p. 59]

```
{unit-test
 (satisfies?
  first
  '(
    (() noop)
    ((1 2 3) 1)
    ))
 (satisfies?
  [|l| (first l onNull: [5])]
  '(
    (() 5)
    ((1 2 3) 1)
    ))}
```

3.4 but-first

```
{define but-first
  [|l #!key (onNull noop)|
   (if (null? l)
       [(onNull)]
       [(cdr l)])]}
```

[Harvey01, p. 59]

```
{unit-test
 (satisfies?
  but-first
  '(
    (() noop)
    ((1 2 3) (2 3))
    ))
 (satisfies?
  [|l| (but-first l onNull: [5])]
  '(
    (() 5)
    ((1 2 3) (2 3))
    ))}
```

3.5 last

```
{define last
  [|l #!key (onNull noop)|
   (if (null? l)
       [(onNull)]
       [{let last ((l l))
          (if (null? (cdr l))
              [(car l)]
              [(last (cdr l))])}])]}
```

[Harvey01, p. 59]

```
{unit-test
 (satisfies?
  last
  '(
    (() noop)
    ((1) 1)
    ((2 1) 1)
    ))
 (satisfies?
  [|l| (last l onNull: [5])]
  '(
    (() 5)
    ((2 1) 1)
    ))}
```

3.6 but-last

```
{define but-last
  [|l #!key (onNull noop)|
   (if (null? l)
       [(onNull)]
       [{let but-last ((l l))
          (if (null? (cdr l))
              ['()]
              [(cons (car l)
                     (but-last (cdr l)))])}])]}
```

[Harvey01, p. 59]

```
{unit-test
 (satisfies?
  but-last
  '(
    (() noop)
    ((1) ())
    ((2 1) (2))
    ((3 2 1) (3 2))
    ))
 (satisfies?
  [|l| (but-last l onNull: [5])]
  '(
    (() 5)
    ((3 2 1) (3 2))
    ))
 }
```

3.7 filter

```
{define filter
  [|p? l|
   {let filter ((l l))
     (if (null? l)
         ['()]
         [{let ((first (car l)))
            (if (p? first)
                [(cons first (filter (cdr l)))]
                [(filter (cdr l))])}])}]}
```

[Harvey01, p. 331][2]. [Abelson96, p. 115].

```
{unit-test
 (satisfies?
  [|l| (filter [|x| (not (= 4 x))]
               l)]
  '(
    (() ())
    ((4) ())
    ((1 4) (1))
    ((4 1 4) (1))
    ((2 4 1 4) (2 1))
    ))}
```

[2]Simply Scheme has an excellent discussion on section on Higher-Order Functions and their combinations [Harvey01, p. 103-125]

3.8 remove

```
{define remove
  [|x l|
   (filter [|y| (not (equal? x y))]
           l)]}
```

```
{unit-test
 (satisfies?
  [|l| (remove 5 l)]
  '(
    ((1 5 2 5 3 5 4 5 5) (1 2 3 4))
    ))}
```

3.9 fold-left

Reduce the list to a scalar by applying the reducing procedure repeatedly, starting from the “left” side of the list

```
{define fold-left
  [|f acc l|
   {let fold-left ((acc acc) (l l))
     (if (null? l)
         [acc]
         [(fold-left (f acc
                        (car l))
                     (cdr l))])}]}
```

3

[Abelson96, p. 121]

```
{unit-test
 (satisfies?
  [|l| (fold-left + 5 l)]
  '(
    (() 5)
    ((1) 6)
    ((1 2) 8)
    ((1 2 3 4 5 6) 26)
    ))
```

Understanding the first test may give the reader false confidence in understanding “fold-left”. To understand how “fold-left” really works, pay close attention to how it works with non-commutative procedures, such as “-”.

```
 (satisfies?
  [|l| (fold-left - 5 l)]
  '(
    (() 5)
    ((1) 4)
    ((1 2) 2)
    ((1 2 3 4 5 6) -16)))}
```

[3] Within libbug, a parameter named “acc” usually means the parameter is is an accumulated value

3.10 fold-right

Reduces the list to a scalar by applying the reducing procedure repeatedly, starting from the “right” side of the list

```
{define fold-right
  [|f acc l|
   {let fold-right ((l l))
     (if (null? l)
         [acc]
         [(f (car l)
             (fold-right (cdr l)))])}]}
```

[Abelson96, p. 116 (named “accumulate”)]

```
{unit-test
 (satisfies?
  [|l| (fold-right + 5 l)]
  '(
    (() 5)
    ((1) 6)
    ((1 2) 8)
    ((1 2 3 4 5 6) 26)
    ))
 (satisfies?
  [|l| (fold-right - 5 l)]
  '(
    (() 5)
    ((1) -4)
    ((1 2) 4)
    ((1 2 3 4 5 6) 2)))
 }
```

3.11 scan-left

Like fold-left, but every intermediate value of fold-left's accumulator is put onto the resulting list

```
{define scan-left
  [|f acc l|
   {let ((acc-list (list acc)))
     {let scan-left ((acc acc)
                     (last-cell acc-list)
                     (l l))
       (if (null? l)
           [acc-list]
           [{let ((newacc (f acc
                             (car l))))
              (scan-left newacc
                         {begin
                           {set-cdr! last-cell (list newacc)}
                           (cdr last-cell)}
                         (cdr l))}])}}]}
```

```
{unit-test
 (satisfies?
  [|l| (scan-left * 1 l)]
  '(
    (() (1))
    ((2) (1 2))
    ((2 3) (1 2 6))
    ((2 3 4) (1 2 6 24))
    ((2 3 4 5 ) (1 2 6 24 120))
    ))
 }
```

3.12 append!

Like Scheme's "append", but recycles the last cons cell, so it's faster but it mutates the input.

```
{define append!
  [|#!rest ls|
   {##define append!
     [|l1 l2|
      (if (null? l1)
          [l2]
          [{let ((head l1))
             {let append! ((l1 l1))
               (if (null? (cdr l1))
                   [{set-cdr! l1 l2}]
                   [(append! (cdr l1))])}
             head}])]}
   (fold-right append! '() ls)]}
```

```
{unit-test
 (equal? (append! '()
                  '(5))
         '(5))
 (equal? (append! '(1 2 3)
                  '(5))
         '(1 2 3 5))
 {let ((a '(1 2 3)))
   (append! a '(5))
   (not (equal? '(1 2 3) a))}
 {let ((a '(1 2 3))
       (b '(4 5 6)))
   (append! a b '(7))
   (equal? a '(1 2 3 4 5 6 7))}
 }
```

3.13 flatmap

```
{define flatmap
  [|f l|
   (fold-left append! '() (map f l))]}
```

[Abelson96, p. 123]

```
{unit-test
 (satisfies?
  [|l| (flatmap [|x| (list x
                           (+ x 1)
                           (+ x 2))]
                l)]
  '(
    ((10 20) (10 11 12 20 21 22))
    ))
 }
```

Mutating cons cells which were created in this procedure still respects referential-transparency from the caller's point of view.

3.14 take

```
{define take
  [|n l|
   (if {or (null? l)
           (= n 0)}
       ['()]
       [(cons (car l)
              (take (- n 1)
                    (cdr l)))])]}
```

```
{unit-test
 (satisfies?
  [|n| (take n '(a b))]
  '(
    (0 ())
    (1 (a))
    (2 (a b))
    (3 (a b))
    ))}
```

3.15 take-while

```
{define take-while
  [|p? l|
   {let take-while ((l l))
     (if {or (null? l)
             ((complement p?) (car l))}
         ['()]
         [(cons (car l)
                (take-while (cdr l)))])}]}
```

```
{unit-test
 (satisfies?
  [|x| (take-while [|y| (not (equal? x y))]
                   '(a b c))]
  '(
    (a ())
    (b (a))
    (c (a b))
    (d (a b c))
    ))}
```

3.16 drop

```
{define drop
  [|n l|
   (if {or (null? l)
           (= n 0)}
       [l]
       [(drop (- n 1)
              (cdr l))])]}
```

```
{unit-test
 (satisfies?
  [|n| (drop n '(a b))]
  '(
    (0 (a b))
    (1 (b))
    (2 ())
    (3 ())
    ))}
```

3.17 drop-while

```
{define drop-while
  [|p? l|
   {let drop-while ((l l))
     (if {or (null? l)
             ((complement p?) (car l))}
         [l]
         [(drop-while (cdr l))])}]}
```

```
{unit-test
 (satisfies?
  [|x| (drop-while [|y| (not (equal? x y))]
                   '(a b c))]
  '(
    (a (a b c))
    (b (b c))
    (c (c))
    (d ())
    (e ())
    ))}
```

3.18 enumerate-interval

```
{define enumerate-interval
  [|low high #!key (step 1)|
   (if (> low high)
       ['()]
       [(cons low
              (enumerate-interval (+ low step)
                                  high
                                  step: step))])]}
```

```
{unit-test
 (equal? (enumerate-interval 1 10)
         '(1 2 3 4 5 6 7 8 9 10))
 (equal? (enumerate-interval 1 10 step: 2)
         '(1 3 5 7 9))}
```

3.19 any?

```
{define any?
  [|l|
   (if (null? l)
       [#f]
       [(if (car l)
            [#t]
            [(any? (cdr l))])])]}
```

```
{unit-test
 (satisfies?
  any?
  '(
    (() #f)
    ((1) #t)
    ((#t) #t)
    ((#t #t) #t)
    ((#f) #f)
    ((#t #t #t #f) #t)))
 }
```

3.20 zip

```
{define zip
  [|#!rest lsts|
   (if (any? (map null? lsts))
       ['()]
       [(cons (apply list (map car lsts))
              (apply zip (map cdr lsts)))])]}
```

```
{unit-test
 (equal? (zip '() '())
         '())
 (equal? (zip '(1) '(4))
         '((1 4)))
 (equal? (zip '(1 2) '(4 5))
         '((1 4)
           (2 5)))
 (equal? (zip '(1 2 3) '(4 5 6))
         '((1 4)
           (2 5)
           (3 6)))
 (equal? (zip '(1) '())
         '())
 (equal? (zip '() '(1))
         '())
```

```
 (equal? (zip '() '() '())
         '())
 (equal? (zip '(1 2 3)
              '(4 5 6)
              '(7 8 9))
         '((1 4 7)
           (2 5 8)
           (3 6 9)))
```

```
(equal? (zip '() '() '() '())
        '())
(equal? (zip '(1 2 3)
             '(4 5 6)
             '(7 8 9)
             '(10 11 12))
        '((1 4 7 10)
          (2 5 8 11)
          (3 6 9 12)))
}
```

3.21 permutations

```
{define permutations
  [|l|
   (if (null? l)
       ['()]
       [{let permutations ((l l))
          (if (null? (cdr l))
              [(list l)]
              [(flatmap [|x| (map [|y| (cons x y)]
                                  (permutations (remove x l)))]
                        l)])}])]}
```

```
{unit-test
 (satisfies?
  permutations
  '(
    (() ())
    ((1) ((1)))
    ((1 2) ((1 2)
            (2 1)))
    ((1 2 3) ((1 2 3)
              (1 3 2)
              (2 1 3)
              (2 3 1)
              (3 1 2)
              (3 2 1)))
    ))}
```

Inspired by [Abelson96, p. 124], although I think they have a slight mistake in their code. Given their definition (permutations '()) evaluates to '(()), instead of '().

See also [Knuth97, p. 45]

3.22 ref-of

The inverse of list-ref.

```
{define ref-of
  [|l x #!key (onMissing noop)|
   (if (null? l)
       [(onMissing)]
       [{let ref-of ((l l)
                     (index 0))
          (if (equal? (car l) x)
              [index]
              [(if (null? (cdr l))
                   [(onMissing)]
                   [(ref-of (cdr l) (+ index 1))])])}])]}
```

```
{unit-test
 (satisfies?
  [|x| (ref-of '(a b c d e f g) x)]
  '(
    (z noop)
    (a 0)
    (b 1)
    (g 6)
    ))
```

```
 (satisfies?
  [|x| (ref-of '(a b c d e f g)
               x
               onMissing: ['missing])]
  '(
    (z missing)
    (a 0)
    ))
```

```
{let ((l '(a b c d e f g)))
  (satisfies?
   [|x| (list-ref l (ref-of l x))]
   '(
     (a a)
     (b b)
     (g g)
     ))}
 }
```

3.23 partition

Partitions the input list into two lists, with the criterion being whether or not the application of the procedure “p?” to each element of the input list evaluated to true or false.

```
{define partition
  [|l p?|
   {let partition ((l l)
                   (trueList '())
                   (falseList '()))
     (if (null? l)
         [(list trueList falseList)]
         [(if (p? (car l))
              [(partition (cdr l)
                          (cons (car l) trueList)
                          falseList)]
              [(partition (cdr l)
                          trueList
                          (cons (car l) falseList))])])}]}
```

```
{unit-test
 (satisfies?
  [|l| (partition l [|x| (<= x 3)])]
  '(
    (() (()
         ()))
    ((3 2 5 4 1) ((1 2 3)
                  (4 5)))
    ))}
```

In section 7.3, “destructuring-bind” allows for a more convenient syntax when using “partition”.

```
> (destructuring-bind (trueList falseList)
                      (partition '(3 2 5 4 1)
                                 [|x| (<= x 3)])
                      trueList)
(1 2 3)
> (destructuring-bind (trueList falseList)
                      (partition '(3 2 5 4 1)
                                 [|x| (<= x 3)])
                      falseList)
(4 5)
```

3.24 sort

```
{define sort
  [|l comparison?|
   {let sort ((l l))
     (if (null? l)
         ['()]
         [{let* ((current-node (car l))
                 (p (partition (cdr l)
                               [|x| (comparison?
                                     x
                                     current-node)]))
                 (less-than (car p))
                 (greater-than (cadr p)))
            (append! (sort less-than)
                     (cons current-node
                           (sort greater-than)))}])}]}
```

```
{unit-test
 (satisfies?
  [|l| (sort l <)]
  '(
    (() ())
    ((1 3 2 5 4 0) (0 1 2 3 4 5))
    ))}
```

3.25 reverse!

Reverses the list quickly by reusing cons cells

```
{define reverse!
  [|l|
   (if (null? l)
       ['()]
       [{let reverse! ((cons-cell l) (reversed-list '()))
          (if (null? (cdr cons-cell))
              [{set-cdr! cons-cell reversed-list}
               cons-cell]
              [{let ((rest (cdr cons-cell)))
                 {set-cdr! cons-cell reversed-list}
                 (reverse! rest cons-cell)}])}])]}
```

```
{unit-test
 (satisfies?
  reverse!
  '(
    (() ())
    ((1) (1))
    ((2 1) (1 2))
    ((3 2 1) (1 2 3))
    ))}
```

CHAPTER 4

Lifting

4.1 string-liftList

Strings are sequences of characters, just as lists are sequences of arbitrary Scheme objects. “string-liftList” takes a one-argument list processing procedure, and evaluates to an equivalent procedure for strings.

```
{define string-liftList
  [|f|
   [|#!rest s|
    (list->string
     (apply f
            (map string->list s)))]]}
```

4.2 string-reverse

```
{define string-reverse
  (string-liftList reverse!)}
```

```
{unit-test
 (satisfies?
  string-reverse
  '(
    ("" "")
    ("foo" "oof")
    ("bar" "rab")
    ))
 }
```

4.3 string-take

```
{define string-take
  [|n s|
   ((string-liftList [|l| (take n l)])
    s)]}
```

```
{unit-test
 (satisfies?
  [|s| (string-take 2 s)]
  '(
    ("" "")
    ("foo" "fo")
    ))
 }
```

4.4 string-drop

```
{define string-drop
  [|n s|
   ((string-liftList [|l| (drop n l)])
    s)]}
```

```
{unit-test
 (satisfies?
  [|s| (string-drop 2 s)]
  '(
    ("" "")
    ("foo" "o")
    ("foobar" "obar")
    ))
 }
```

4.5 character-liftInteger

```
{define character-liftInteger
  [|f|
   [|#!rest n|
    (integer->char
     (apply f
            (map char->integer n)))]]}
```

```
{unit-test
 (satisfies?
  (character-liftInteger [|c| (+ c 1)])
   '(
     (#\a #\b)
     (#\b #\c)
     (#\c #\d)
     ))}
```

4.6 string-map

```
{define string-map
  [|f s|
   ((string-liftList [|l| (map f l)])
    s)]}
```

The “Caesar Cipher”. [Stallings03, p. 30].

```
{unit-test
 (satisfies?
  [|s| (string-map
        [|c|
         ((character-liftInteger [|base-char c|
                                  (+ base-char
                                     (modulo (+ (- c base-char)
                                                3)
                                             26))])
          #\a
          c)]
        s)]

  '(
    ("" "")
    ("abc" "def")
    ("nop" "qrs")
    ("xyz" "abc")
    ))
 }
```

4.7 symbol-liftList

Symbols are sequences of characters, just as lists are sequences of arbitrary Scheme objects. “symbol-liftList” takes a one-argument list processing procedure, and evaluates to an equivalent procedure for symbols.

```
{define symbol-liftList
  [|f|
   [|#!rest s|
    (string->symbol
     (apply (string-liftList f)
            (map symbol->string s)))]]}
```

```
{unit-test
 (satisfies?
  (symbol-liftList reverse)
  '(
    (foo oof)
    (bar rab)
    ))
 (equal? ((symbol-liftList append!) 'foo 'bar)
         'foobar)
 }
```

Streams

Streams are sequential collections like lists, but the “cdr” of each pair must be a zero-argument lambda value. That lambda is automatically evaluated when “(stream-cdr s)” is evaluated. For more information, consult “The Structure and Interpretation of Computer Programs”[1].

5.1 Stream structure

“bug#define-structure”[2] takes as parameters the name of the datatype, and a variable number of fields.

```
{define-structure stream
  a
  d}
```

“bug#define-structure” will create a constructor procedure named “make-stream”, accessor procedures “stream-a”, “stream-d”, and updating procedures “stream-a-set!” and “stream-d-set!”. For streams, none of these generated procedures are intended to be evaluated directly by the programmer. Instead, the following are to be used.

5.2 stream-car

Get the first element of the stream.

```
{define stream-car
  stream-a}
```

[Abelson96, p. 321].

5.3 stream-cdr

Forces the evaluation of the next element of the stream.

[1] although, they define “stream-cons” as syntax instead of passing a lambda to the second argument
[2] defined in section 8.12

```
{define stream-cdr
  [|s|
   {force (stream-d s)}]}
```

[Abelson96, p. 321].

5.4 stream-cons

Like “cons”, creates a pair. The second argument must be a zero-argument lambda value.

```
{define-macro stream-cons
  [|a d|
   (if {and (list? d)
            (equal? 'lambda (car d))
            (not (null? (cdr d)))
            (equal? '() (cadr d))}
       [`(make-stream ,a {delay ,(caddr d)})]
       [(error "bug#stream-cons requires a zero-argument \
                lambda in it's second arg")])]}
```

[Abelson96, p. 321].

```
{unit-test
 {let ((s (stream-cons 1 [2])))
   {and
    (equal? (stream-car s)
            1)
    (equal? (stream-cdr s)
            2)}}
 }
```

5.5 stream-null

```
{define stream-null
  '()
  }
```

5.6 stream-null?

```
{define stream-null?
  null?}
```

```
{unit-test
 (stream-null?
  (stream-cdr
   (stream-cdr (stream-cons 1 [(stream-cons 2
                                             [stream-null])]))))
 }
```

5.7 list->stream

Converts a list into a stream

```
{define list->stream
  [|l|
   (if (null? l)
       [stream-null]
       [(stream-cons (car l)
                     [(list->stream
                       (cdr l))])])]}
```

```
{unit-test
 {let ((foo (list->stream '(1 2 3))))
   {and (equal? 1 (stream-car foo))
        (equal? 2 (stream-car
                   (stream-cdr foo)))
        (equal? 3 (stream-car
                   (stream-cdr
                    (stream-cdr foo))))
        (stream-null? (stream-cdr
                       (stream-cdr
                        (stream-cdr foo))))}}}
```

5.8 stream->list

Converts a stream into a list

```
{define stream->list
  [|s|
   (if (stream-null? s)
       ['()]
       [(cons (stream-car s)
              (stream->list
               (stream-cdr s)))])]}
```

```
{unit-test
 (equal? (stream->list
          (list->stream '(1 2 3)))
         '(1 2 3))
 }
```

5.9 stream-ref

The analogous procedure of “list-ref”

```
{define stream-ref
  [|s n #!key (onOutOfBounds noop)|
   (if (< n 0)
       [(onOutOfBounds)]
       [{let stream-ref ((s s) (n n))
          (if (equal? n 0)
              [(stream-car s)]
              [(if (not (stream-null? (stream-cdr s)))
                   [(stream-ref (stream-cdr s) (- n 1))]
                   [(onOutOfBounds)])])}])]}
```

[Abelson96, p. 319].

```
{unit-test
 (satisfies?
  [|i| (stream-ref (list->stream '(a b c d e)) i)]
  '(
    (-1 noop)
    (0 a)
    (4 e)
    (5 noop)
    )
  )
 (equal? (stream-ref (list->stream '(a b c d e))
                     5
                     onOutOfBounds: ['out])
         'out)}
```

5.10 integers-from

Creates an “infinite” list of integers.

```
{define integers-from
  [|n|
   (stream-cons n [(integers-from (+ n 1))])]}
```

[Abelson96, p. 326].

```
{unit-test
 (satisfies?
  [|n| (stream-ref (integers-from 0) n)]
  '(
    (0 0)
    (1 1)
    (2 2)
    ))
 (satisfies?
  [|n| (stream-ref (integers-from 5) n)]
  '(
    (0 5)
    (1 6)
    (2 7)
    ))
 }
```

5.11 stream-take

```
{define stream-take
  [|n s|
   (if {or (stream-null? s)
           (= n 0)}
       [stream-null]
       [(stream-cons (stream-car s)
                     [(stream-take (- n 1)
                                   (stream-cdr s))])])]}
```

```
{unit-test
 (satisfies?
  [|n| (stream->list
        (stream-take n (integers-from 0)))]
  '(
    (0 ())
    (1 (0))
    (2 (0 1))
    (6 (0 1 2 3 4 5))
    ))}
```

5.12 stream-filter

The analogous procedure of filter.

```
{define stream-filter
  [|p? s|
   {let stream-filter ((s s))
     (if (stream-null? s)
         [stream-null]
         [{let ((first (stream-car s)))
            (if (p? first)
                [(stream-cons
                  first
                  [(stream-filter (stream-cdr s))])]
                [(stream-filter (stream-cdr s))])}])}]}
```

```
{unit-test
 (equal?  (stream->list
           (stream-filter [|x| (not (= 4 x))]
                          (list->stream '(1 4 2 4))))
          '(1 2))
 (equal? (stream->list
          (stream-take
           10
           (stream-filter [|n|
                           (not (equal? 0
                                        (modulo n 2)))]
                          (integers-from 2))))
         '(3 5 7 9 11 13 15 17 19 21))
 }
```

5.13 primes

```
{define primes
  {let sieve-of-eratosthenes ((s (integers-from 2)))
    (stream-cons
     (stream-car s)
     [(sieve-of-eratosthenes (stream-filter
                              [|n|
                               (not (equal? 0
                                            (modulo n (stream-car s))))]
                              (stream-cdr s)))])}}
```

[Abelson96, p. 327].

```
{unit-test
 (equal? (stream->list
          (stream-take
           10
           primes))
         '(2 3 5 7 11 13 17 19 23 29))
 }
```

5.14 stream-map

The analogous procedure of “map”.

```
{define stream-map
  [|f #!rest list-of-streams|
   {let stream-map ((list-of-streams list-of-streams))
     (if (any? (map stream-null? list-of-streams))
         [stream-null]
         [(stream-cons (apply f
                              (map stream-car list-of-streams))
                       [(stream-map (map stream-cdr list-of-streams))])])}]}
```

```
{unit-test
 (equal? (stream->list
          (stream-map [|x| (+ x 1)]
                      (list->stream '(1 2 3 4 5))))
         '(2 3 4 5 6))
 (equal? (stream->list
          (stream-map [|x y| (+ x y)]
                      (list->stream '(1 2 3 4 5))
                      (list->stream '(1 1 1 1 1))))
         '(2 3 4 5 6))
 }
```

5.15 stream-enumerate-interval

```
{define stream-enumerate-interval
  [|low high #!key (step 1)|
   (if (> low high)
       [stream-null]
       [(stream-cons low
                     [(stream-enumerate-interval (+ low step)
                                                 high
                                                 step: step)])])]}
```

```
{unit-test
 (equal? (stream->list
          (stream-enumerate-interval 1 10))
         '(1 2 3 4 5 6 7 8 9 10))
 (equal? (stream->list
          (stream-enumerate-interval 1 10 step: 2))
         '(1 3 5 7 9))}
```

5.16 stream-take-while

```
{define stream-take-while
  [|p? s|
   {let stream-take-while ((s s))
     (if {or (stream-null? s)
             ((complement p?) (stream-car s))}
         [stream-null]
         [(stream-cons (stream-car s)
                       [(stream-take-while
                         (stream-cdr s))])])}]}
```

```
{unit-test
 (satisfies?
  [|s|
   (stream->list
    (stream-take-while [|n| (< n 10)]
                       s))]
  `((,(integers-from 0)               (0 1 2 3 4 5 6 7 8 9))
    (,(stream-enumerate-interval 1 4) (1 2 3 4))))
 }
```

5.17 stream-drop

```
{define stream-drop
  [|n s|
   (if {or (stream-null? s)
           (= n 0)}
       [s]
       [(stream-drop (- n 1)
                     (stream-cdr s))])]}
```

```
{unit-test
 (satisfies?
  [|n|
   (stream->list
    (stream-drop n (list->stream '(a b))))]
  '(
    (0 (a b))
    (1 (b))
    (2 ())
    (3 ())
    ))
 (equal? (stream->list
          (stream-take 10 (stream-drop 10
                                       primes)))
        '(31 37 41 43 47 53 59 61 67 71))
 }
```

5.18 stream-drop-while

```
{define stream-drop-while
  [|p? s|
   {let stream-drop-while ((s s))
     (if {or (stream-null? s)
             ((complement p?) (stream-car s))}
         [s]
         [(stream-drop-while (stream-cdr s))])}]}
```

```
{unit-test
 (satisfies?
  [|x|
   (stream->list
    (stream-drop-while [|y| (not (equal? x y))]
                       (list->stream
                        '(a b c))))]
  '(
    (a (a b c))
    (b (b c))
    (c (c))
    (d ())
    (e ())
    ))}
```

CHAPTER 6

Macros

Although many concepts first implemented in Lisp (conditional expressions, garbage collection, procedures as first-class objects) have been appropriated into mainstream languages, the one feature of Lisp which remains difficult to copy is also one of Lisp's strongest: macros. Macros are a facility by which a programmer may augment the compiler with new functionality *while the compiler is compiling.*

Mastery of macros is required to understand all subsequent chapters of this book. Should the reader have difficulty with this chapter, the author recommends reading "On Lisp" by Paul Graham [Graham94].

6.1 compose

```
1  {define-macro compose
2    [|#!rest fs|
3     (if (null? fs)
4         ['identity]
5         [{let ((args (gensym)))
6            `[|#!rest ,args|
7              ,{let compose ((fs fs))
8                 (if (null? (cdr fs))
9                     [`(apply ,(car fs)
10                             ,args)]
11                    [`(,(car fs)
12                       ,(compose (cdr fs)))])}]}])]}
```

[Graham94, p. 66]

- On line 1, the libbug-private#define-macro macro[1] is invoked. Besides defining the macro, libbug-private#define-macro also exports the namespace definition and the macro definitions to external files. Libbug is a library, meant to be used by other projects. From libbug, these projects will require namespace definitions, as well as macro definitions.

```
{unit-test
 (equal? {macroexpand-1 (compose)}
         'identity)
 (equal? ((eval {macroexpand-1 (compose)}) 5)
         5)
 (equal? ((compose) 5)
         5)
```

Macro-expansions occur during compile-time, so how should a person test them? Libbug provides "macroexpand-1" which treats the macro as a procedure which transforms lists into lists, and as such is able to be tested[2].

[1]defined in section 8.10

[2]"macroexpand-1" expands the unevaluated code passed to the macro into the new form, which the compiler would have then compiled if "macroexpand-1" had not been present. But, how should "gensyms" evaluate, since by definition it creates symbols which cannot be entered into a program? During the expansion of "macroexpand-1", "gensym" is overridden into a procedure which expands into symbols like "gensymed-var1", "gensymed-var2", etc. Each call during a macro-expansion generates a new, unique symbol. Although this symbol may clash with symbols in the expanded code, this is not a problem, as these symbols are only generated in the call to "macroexpand-1"; run-time "gensym" is regular old "gensym". As such, "eval"ing code generated from "macroexpand-1" in a non-testing context is not recommended.

```
(equal? {macroexpand-1 (compose [|x| (* x 2)])}
        '[|#!rest gensymed-var1|
          (apply [|x| (* x 2)]
                 gensymed-var1)])
(equal? ((eval {macroexpand-1 (compose [|x| (* x 2)])})
         5)
        10)
(equal? ((compose [|x| (* x 2)])
         5)
        10)
```

```
(equal? {macroexpand-1 (compose [|x| (+ x 1)]
                                [|x| (* x 2)])}
        '[|#!rest gensymed-var1|
          ([|x| (+ x 1)]
           (apply [|x| (* x 2)]
                  gensymed-var1))])
(equal? ((eval {macroexpand-1 (compose [|x| (+ x 1)]
                                       [|x| (* x 2)])})
         5)
        11)
(equal? ((compose [|x| (+ x 1)]
                  [|x| (* x 2)])
         5)
        11)
```

```
(equal? {macroexpand-1 (compose [|x| (/ x 13)]
                                [|x| (+ x 1)]
                                [|x| (* x 2)])}
        '[|#!rest gensymed-var1|
          ([|x| (/ x 13)]
           ([|x| (+ x 1)]
            (apply [|x| (* x 2)]
                   gensymed-var1)))])
(equal? ((eval {macroexpand-1 (compose [|x| (/ x 13)]
                                       [|x| (+ x 1)]
                                       [|x| (* x 2)])})
         5)
        11/13)
(equal? ((compose [|x| (/ x 13)]
                  [|x| (+ x 1)]
                  [|x| (* x 2)])
         5)
        11/13)
}
```

6.2 aif

```
{define-macro aif
  [|bool body|
   `{let ((bug#it ,bool))
      (if bug#it
          [,body]
          [#f])}]}
```

Although variable capture [Graham94, p. 118-132] is generally avoided, there are instances in which variable capture is desirable [Graham94, p. 189-198]. Within libbug, varibles intended for capture are fully qualified with a namespace to ensure that the variable is captured.
[Graham94, p. 191]

```
{unit-test
 (equal? {aif (+ 5 10) (* 2 bug#it)}
         30)
 (equal? {aif #f (* 2 bug#it)}
         #f)
 (equal? {macroexpand-1 {aif (+ 5 10)
                             (* 2 bug#it)}}
         '{let ((bug#it (+ 5 10)))
            (if bug#it
                [(* 2 bug#it)]
                [#f])})
 }
```

6.3 with-gensyms

“with-gensyms” is a macro to be invoked from other macros. It is a utility for macros to minimize repetitive calls to “gensym”.

```
{define-macro with-gensyms
  [|symbols #!rest body|
   `{let ,(map [|symbol| `(,symbol {gensym})]
                symbols)
      ,@body}]}
```

[Graham94, p. 145]

```
{unit-test
 (equal? {macroexpand-1 (with-gensyms (foo bar baz)
                                      `{begin
                                         (pp ,foo)
                                         (pp ,bar)
                                         (pp ,baz)})}
         '{let ((foo (gensym))
                (bar (gensym))
                (baz (gensym)))
            `{begin
               (pp ,foo)
               (pp ,bar)
               (pp ,baz)}})
 }
```

6.4 once-only

"once-only" is a macro-creating macro which ensures that the arguments to the calling macro are evaluated only once. For more information on the problem of multiple evaluation, see [Graham94, p. 133]..

```
{define-macro once-only
  [|symbols #!rest body|
   {let ((gensyms (map [|s| (gensym)]
                       symbols)))
     (list 'list
           ''let
           (cons 'list (map [|g s| (list 'list
                                          (list 'quote g)
                                          s)]
                              gensyms
                              symbols))
           (append (list 'let
                         (map [|s g| (list s
                                            (list 'quote g))]
                               symbols
                               gensyms))
                     body))}]}
```

[Norvig92, p. 854]
Symbols with no quotes are evaluated in the first macroexpansion, symbols with one quote are evaluated in the second macroexpansion, symbols with two quotes are part of the generated code.

First Macroexpansion

```
1 {unit-test
2  (equal? {macroexpand-1 {once-only (x y) `(+ ,x ,y)}}
3         '(list 'let
4                (list (list 'gensymed-var1 x)
5                      (list 'gensymed-var2 y))
6                {let ((x 'gensymed-var1)
7                      (y 'gensymed-var2))
8                  `(+ ,x ,y)}))
```

Like "with-gensyms", "once-only" is a macro to be used by other macros. But while "with-gensyms" only wraps it's argument with a new context to be used for later macroexpansions, "once-only" needs to defer binding the variable to a "gensym-ed" variable until the second macroexpansion.

- On line 1, "once-only" is invoked, specifying that the variables "x" and "y" shall be evaluated only once in the expanded code of "x" plus "y".

- On line 2, this first expansion of the macro sets up the second expansion's creation of a new context (line 3-4) for modified code (line 5-7)

The Second Macroexpansion

```
1  (equal? (eval `{let ((x 'x)
2                        (y 'y))
3                    ,(once-only-expand (x y)
4                                       `(+ ,x ,y))})
5          '{let ((gensymed-var1 x)
6                 (gensymed-var2 y))
7             (+ gensymed-var1 gensymed-var2)})
```

The Evaluation of the twice-expanded Code

```
1  (equal? (eval `{let ((x 5)
2                        (y 6))
3                    ,(eval `{let ((x 'x)
4                                  (y 'y))
5                              ,(once-only-expand (x y)
6                                                 `(+ ,x ,y))})})
7          11)
8  }
```

CHAPTER 7

Generalized Assignment

7.1 setf!

Lisp based systems such as Gambit do not provide raw access to memory locations via pointers, thus relieving the user of the language of direct memory management. However, pointers are very useful in that they allow a procedure to pass a memory location as a variable to another procedure, which can then indirectly access/write to that location.

Fortunatly, most data-structures in Lisp have an “accessing” procedure (for instance “car” or “stream-a”) and an “updating procedure” (“set-car!” and “stream-a-set!”). And there are only 3 patterns used to map names of accessing procedures to updating procedures.

“Rather than thinking about two distinct functions that respectively access and update a storage location somehow deduced from their arguments, we can instead simply think of a call to the access function with given arguments as a *name* for the storage location.” [Steele90, p. 123-124]

Therefore, create a macro named “setf!” which invokes the appropriate “setting” procedure, based on the given “accessing” procedure[1].

```
{define-macro setf!
  [|exp val|
   (if (not (pair? exp))
       [`{set! ,exp ,val}]
       [{case (car exp)
          ((car)  `{set-car! ,@(cdr exp) ,val})
          ((cdr)  `{set-cdr! ,@(cdr exp) ,val})
          ((caar) `{setf! (car (car ,@(cdr exp))) ,val})
          ((cadr) `{setf! (car (cdr ,@(cdr exp))) ,val})
          ((cdar) `{setf! (cdr (car ,@(cdr exp))) ,val})
          ((cddr) `{setf! (cdr (cdr ,@(cdr exp))) ,val})
```

[1]The implementation is inspired by [Kiselyov98].

```
          ((caaar) `{setf! (car (caar ,@(cdr exp))) ,val})
          ((caadr) `{setf! (car (cadr ,@(cdr exp))) ,val})
          ((cadar) `{setf! (car (cdar ,@(cdr exp))) ,val})
          ((caddr) `{setf! (car (cddr ,@(cdr exp))) ,val})
          ((cdaar) `{setf! (cdr (caar ,@(cdr exp))) ,val})
          ((cdadr) `{setf! (cdr (cadr ,@(cdr exp))) ,val})
          ((cddar) `{setf! (cdr (cdar ,@(cdr exp))) ,val})
          ((cdddr) `{setf! (cdr (cddr ,@(cdr exp))) ,val})
          ((caaaar) `{setf! (car (caaar ,@(cdr exp))) ,val})
          ((caaadr) `{setf! (car (caadr ,@(cdr exp))) ,val})
          ((caadar) `{setf! (car (cadar ,@(cdr exp))) ,val})
          ((caaddr) `{setf! (car (caddr ,@(cdr exp))) ,val})
          ((cadaar) `{setf! (car (cdaar ,@(cdr exp))) ,val})
          ((cadadr) `{setf! (car (cdadr ,@(cdr exp))) ,val})
          ((caddar) `{setf! (car (cddar ,@(cdr exp))) ,val})
          ((cadddr) `{setf! (car (cdddr ,@(cdr exp))) ,val})
          ((cdaaar) `{setf! (cdr (caaar ,@(cdr exp))) ,val})
          ((cdaadr) `{setf! (cdr (caadr ,@(cdr exp))) ,val})
          ((cdadar) `{setf! (cdr (cadar ,@(cdr exp))) ,val})
          ((cdaddr) `{setf! (cdr (caddr ,@(cdr exp))) ,val})
          ((cddaar) `{setf! (cdr (cdaar ,@(cdr exp))) ,val})
          ((cddadr) `{setf! (cdr (cdadr ,@(cdr exp))) ,val})
          ((cdddar) `{setf! (cdr (cddar ,@(cdr exp))) ,val})
          ((cddddr) `{setf! (cdr (cdddr ,@(cdr exp))) ,val})
```

```
          (else `(,((symbol-liftList
                     [|l -set! -ref|
                      (append!
                       (if (equal? (reverse -ref)
                                   (take 4 (reverse l)))
                           [(reverse (drop 4
                                           (reverse l)))]
                           [l])
                      -set!)])
                   (car exp)
                   '-set!
                   '-ref)
                 ,@(cdr exp)
                 ,val))}])]}
```

Updating a Variable Directly

```
{unit-test
 (equal? {macroexpand-1
          {setf! foo 10}}
         '{set! foo 10})
 {let ((a 5))
   {setf! a 10}
   (equal? a 10)}
```

Updating Car, Cdr, ... Through Cddddr

Test updating "car".

```
 (equal? {macroexpand-1
          {setf! (car foo) 10}}
         '{set-car! foo 10})
 {let ((foo '(1 2)))
   {setf! (car foo) 10}
   (equal? (car foo) 10)}
```

Test updating "cdr".

```
 (equal? {macroexpand-1
          {setf! (cdr foo) 10}}
         '{set-cdr! foo 10})
 {let ((foo '(1 2)))
   {setf! (cdr foo) 10}
   (equal? (cdr foo) 10)}
```

Testing all of the "car" through "cddddr" procedures will be highly repetitive. Instead, create a list which has an element at each of those accessor procedures, and test each.

```
(eval
 `{and
   ,@(map [|x| `{let ((foo '((((the-caaaar)
                               the-cadaar)
                              (the-caadar)
                              ())
                             ((the-caaadr) the-cadadr)
                             (the-caaddr)
                             ()
                             )))
                  {setf! (,x foo) 10}
                  (equal? (,x foo) 10)}]
          '(car
            cdr
            caar cadr
            cdar cddr
            caaar caadr cadar caddr
            cdaar cdadr cddar cdddr
            caaaar caaadr caadar caaddr
            cadaar cadadr caddar cadddr
            cdaaar cdaadr cdadar cdaddr
            cddaar cddadr cdddar cddddr
            ))})
```

Suffixed By -set!

Test updating procedures where the updating procedure is the name of the getting procedure, suffixed by '-set!'.

```
(equal? {macroexpand-1
         {setf! (stream-a s) 10}}
        '{stream-a-set! s 10})
{begin
  {let ((a (make-stream 1 2)))
    {setf! (stream-a a) 10}
    (equal? (make-stream 10 2)
            a)}}
```

[2]

[2]As a reminder, "stream-a", "make-stream", and "stream-a-set!" are not meant to be used directly. But for the purposes of testing "setf!", it sufficies to use them directly.

-ref Replaced By -set!

Test updating procedures where the updating procedure is the name of the getting procedure, removing the suffix of '-ref', and adding a suffix of '-set!'.

```
(equal? {macroexpand-1
          {setf! (string-ref s 0) #\q}}
         '{string-set! s 0 #\q})
{let ((s "foobar"))
  {setf! (string-ref s 0) #\q}
  (equal? s "qoobar")}
(equal? {macroexpand-1
          {setf! (vector-ref v 2) 4}}
         '{vector-set! v 2 4})
{let ((v (vector 1 2 '() "")))
  {setf! (vector-ref v 2) 4}
  (equal? v
           (vector 1 2 4 ""))}
}
```

7.2 mutate!

Like “setf!”, “mutate!” takes a generalized variable as input, but it additionally takes a procedure. The procedure is applied to the value of that generalized variable, and is then stored back into it.

```
{define-macro mutate!
  [|exp f|
   (if (symbol? exp)
       [`{setf! ,exp (,f ,exp)}]
       [{let* ((args (cdr exp))
                (syml (map [|s| (gensym)]
                           args)))
          `{let ,(zip syml args)
             {setf! (,(car exp) ,@syml) (,f (,(car exp) ,@syml))}}}])]}
```

```
{unit-test
 (equal? {macroexpand-1 {mutate! foo not}}
         '{setf! foo (not foo)})
 {let ((foo #t))
   {and
    {begin
      {mutate! foo not}
      (equal? foo #f)}
    {begin
      {mutate! foo not}
      (equal? foo #t)}}}
```

[3]

```
 (equal? {macroexpand-1 (mutate! foo [|n| (+ n 1)])}
         '{setf! foo ([|n| (+ n 1)] foo)})
 {let ((foo 1))
   (mutate! foo [|n| (+ n 1)])
   (equal? foo
           2)}
```

[3]“mutate!” is used in similar contexts as Common Lisp’s “define-modify-macro” would be, but it is more general, as it allows the new procedure to remain anonymous, as compared to making a new name like “toggle” [Graham94, p. 169].

```
(equal? {macroexpand-1 {mutate! (vector-ref foo 0) [|n| (+ n 1)]}}
        '{let ((gensymed-var1 foo)
               (gensymed-var2 0))
           {setf! (vector-ref gensymed-var1
                              gensymed-var2)
                  ([|n| (+ n 1)] (vector-ref gensymed-var1
                                             gensymed-var2))}})
{let ((foo (vector 0 0 0)))
  {mutate! (vector-ref foo 0) [|n| (+ n 1)]}
  (equal? foo
          (vector 1 0 0))}
{let ((foo (vector 0 0 0)))
  {mutate! (vector-ref foo 2) [|n| (+ n 1)]}
  (equal? foo
          (vector 0 0 1))}
```

```
(equal? {macroexpand-1
         {mutate! (vector-ref foo {begin
                                    {setf! index (+ 1 index)}
                                    index})
                  [|n| (+ n 1)]}}
        '{let ((gensymed-var1 foo)
               (gensymed-var2 {begin
                                {setf! index (+ 1 index)}
                                index}))
           {setf! (vector-ref gensymed-var1
                              gensymed-var2)
                  ([|n| (+ n 1)] (vector-ref gensymed-var1
                                             gensymed-var2))}})
{let ((foo (vector 0 0 0))
      (index 1))
  {mutate! (vector-ref foo {begin
                             {setf! index (+ 1 index)}
                             index})
           [|n| (+ n 1)]}
  {and (equal? foo
               (vector 0 0 1))
       (equal? index
               2)}}
}
```

7.3 destructuring-bind

“destructuring-bind” is a generalization of “let”, in which multiple variables may be bound to values based on their positions within a list. Look at the tests at the end of the section for an example.

“destructuring-bind” is a complicated macro which can be decomposed into a regular procedure named “tree-of-accessors”, and the macro “destructuring-bind”[4].

```
{define tree-of-accessors
  [|pat lst #!key (gensym gensym) (n 0)|
   {cond ((null? pat)                 '())
         ((symbol? pat)               `((,pat (drop ,n ,lst))))
         ((equal? (car pat) '#!rest) `((,(cadr pat) (drop ,n
                                                          ,lst))))
         (else
          (cons {let ((p (car pat)))
                  (if (symbol? p)
                      [`(,p (list-ref ,lst ,n))]
                      [{let ((var (gensym)))
                         (cons `(,var (list-ref ,lst ,n))
                               (tree-of-accessors p
                                                  var
                                                  gensym: gensym
                                                  n: 0))}])}
                (tree-of-accessors (cdr pat)
                                   lst
                                   gensym: gensym
                                   n: (+ 1 n))))}]}
```

```
{unit-test
 (equal? (tree-of-accessors '() '(1 2))
         '())
 (equal? (tree-of-accessors 'a '(1 2))
         '((a (drop 0 (1 2)))))
 (equal? (tree-of-accessors '(#!rest d) '(1 2))
         '((d (drop 0 (1 2)))))
 (equal? (tree-of-accessors '(a) '(1 2))
         '((a (list-ref (1 2) 0))))
 (equal? (tree-of-accessors '(a . b) '(1 2))
         '((a (list-ref (1 2) 0))
           (b (drop 1 (1 2)))))
```

[4]This poses a small problem. “tree-of-accessors” is not macroexpanded as it a not a macro, therefore it does not have access to the compile-time “gensym” procedure which allows macroexpansions to be tested. To allow “tree-of-accessors” to be tested independently, as well as part of “destructuring-bind”, “tree-of-accessors” takes a procedure named “gensym” as an argument, defaulting to whatever value “gensym” is by default in the environment.

```
(equal? (tree-of-accessors '(a (b c))
                           '(1 (2 3))
                           gensym: ['gensymed-var1])
       '((a (list-ref (1 (2 3)) 0))
         ((gensymed-var1 (list-ref (1 (2 3)) 1))
          (b (list-ref gensymed-var1 0))
          (c (list-ref gensymed-var1 1)))))
}
```

Although “tree-of-accessors” appears to be victim of the multiple-evaluation problem that macros may have, “tree-of-accessors” is completely safe to use as long as the caller does not directly pass a list to “tree-of-accessors”. The only caller of “tree-of-accessors” is “destructuring-bind”, which passes a symbol to “tree-of-accessors”. Therefore “destructuring-bind” does not fall victim to unintended multiple evaluations.

```
{define-macro destructuring-bind
  [|pat lst #!rest body|
   {let ((glst (gensym)))
     `{let ((,glst ,lst))
        ,{let create-nested-lets ((bindings
                                   (tree-of-accessors
                                    pat
                                    glst
                                    gensym: gensym)))
           (if (null? bindings)
               [`{begin ,@body}]
               [`{let ,(map [|b| (if (pair? (car b))
                                     [(car b)]
                                     [b])]
                            bindings)
                   ,(create-nested-lets (flatmap [|b| (if (pair? (car b))
                                                          [(cdr b)]
                                                          ['()])]
                                                 bindings))}])}}}]}
```

[Graham94, p. 232]

```
{unit-test
 (equal? {macroexpand-1
          {destructuring-bind (a (b . c) #!rest d)
                              '(1 (2 3) 4 5)
                              (list a b c d)}}
         '{let ((gensymed-var1 '(1 (2 3) 4 5)))
            {let ((a (list-ref gensymed-var1 0))
                  (gensymed-var2 (list-ref gensymed-var1 1))
                  (d (drop 2 gensymed-var1)))
              {let ((b (list-ref gensymed-var2 0))
                    (c (drop 1 gensymed-var2)))
                {begin (list a b c d)}}}})
 (equal? {destructuring-bind (a (b . c) #!rest d)
                             '(1 (2 3) 4 5)
                             (list a b c d)}
         '(1 2 (3) (4 5)))
 }
```

7.4 The End of Compilation

At the beginning of the book, in chapter 2, “bug-language.scm” was imported, so that “libbug-private#define”, and “libbug-private#define-macro” can be used. This chapter is the end of the file “main.bug.scm”. However, as will be shown in the next chapter, “bug-languge.scm” opened files for writing during compile-time, and they must be closed, accomplished executing “at-end-of-compilation”.

```
{at-compile-time
 (at-end-of-compilation)}
```

Part II

Foundations Of Libbug

CHAPTER 8

Computation At Compile-Time

This chapter, which was evaluated before the previous chapters, provides the foundation for computation at compile-time. Although the most prevalent code which executed at compile-time in the previous chapters was code for testing, many other computations occurred during compile-time transparently to the reader. These other computations produced output files for namespace mappings and for macro definitions, to be used by other programs which link against libbug.

8.1 at-compile-time

"at-compile-time" is a macro which "eval"'s the form during macroexpansion, but evaluates to the symbol "noop", thus not affecting run-time [Feeley12]. Evaling during macro-expansion is how the compiler may be augmented with new procedures, thus treating the compiler as an interpreter.

```
1 {##namespace ("bug#" at-compile-time)}
2 {##define-macro at-compile-time
3   [|#!rest forms|
4    (eval `{begin
5            ,@forms})
6    `{quote noop}]}
```

- On line 4, the unevaluated code which is passed to "at-compile-time" is evaluated during macro-expansion, thus at compile-time. The macro-expansion expands into "quote noop", so the form will not evaluate at runtime.

8.2 at-both-times

"at-both-times", like "at-compile-time", "eval"'s the forms in the compile-time environment, but also in the run-time environment.

```
1 {##namespace ("bug#" at-both-times)}
2 {##define-macro at-both-times
3   [|#!rest forms|
4    (eval `{begin
5             ,@forms})
6    `{begin
7       ,@forms}]}
```

- On lines 4-5, evaluation in the compile-time environment
- On lines 6-7, evaluation in the run-time environment. The forms are returned unaltered to Gambit's compiler, thus ensuring that they are defined in the run-time environment.

8.3 at-compile-time-expand

"at-compile-time-expand" allows any procedure to act as a macro.

```
{##namespace ("bug#" at-compile-time-expand)}
{##define-macro at-compile-time-expand
  [|#!rest forms|
   (eval `{begin
            ,@forms})]}
```

8.4 Create Files for Linking Against Libbug

Libbug is a collection of procedures and macros. Building libbug results in a dynamic library and a "loadable" library (a .o1 file, for loading in the Gambit interpreter). But programs which link against libug will require libbug's macro definitions and namespace declarations, both of which are not compiled into the libraries. Rather than manually copying all of them to external files, why not generate them during compile-time?

Open one file for the namespaces, "libbug#.scm", and one for the macros, "libbug-macros.scm". These files will be pure Gambit scheme code, no libbug-syntax enhancements, and they are not intended to be read by a person.

```
{at-compile-time
```

8.4.1 Create File for Namespaces

The previous three macros are currently namespaced within libbug, but external projects which will use libbug may need these namespace mappings as well. Towards that goal, open a file during compile-time and then write those namespace mappings to the file.

```
{##define libbug-headers-file
  (open-output-file '(path: "libbug#.scm" append: #f))}
(display
 ";;; Copyright 2014-2016 - William Emerison Six
  ;;;  All rights reserved
  ;;;  Distributed under LGPL 2.1 or Apache 2.0
  {##namespace (\"bug#\" at-compile-time)}
  {##namespace (\"bug#\" at-both-times)}
  {##namespace (\"bug#\" at-compile-time-expand)}
  "
 libbug-headers-file)
```

8.4.2 Create File for Macro Definitions

The previous three macros are currently available throughout the definition of libbug, but not to programs which link against libbug. To rectify that, open a file during compile-time, and write those macro definitions to the file.

```
 1  (##include "config.scm")
 2  {##define bug-configuration#libbugsharp
 3    (string-append bug-configuration#prefix "/include/bug/libbug#.scm")}
 4
 5  {##define libbug-macros-file
 6    (open-output-file '(path: "libbug-macros.scm" append: #f))}
 7  (display
 8   (string-append
 9    ";;; Copyright 2014-2016 - William Emerison Six
10     ;;;  All rights reserved
11     ;;;  Distributed under LGPL 2.1 or Apache 2.0
12     (##include \"~~lib/gambit#.scm\")
13     (##include \"" bug-configuration#libbugsharp "\")
14     {##define-macro at-compile-time
15       [|#!rest forms|
16        (eval `{begin
17                 ,@forms})
18        `{quote noop}]}
19     {##define-macro at-both-times
20       [|#!rest forms|
21        (eval `{begin
22                 ,@forms})
23        `{begin
24           ,@forms}]}
25     {##define-macro at-compile-time-expand
26       [|#!rest forms|
27        (eval `{begin
28                 ,@forms})]}
29    ")
30   libbug-macros-file)
```

- On line 1-3, include the “config.scm” file which was preprocessed by Autoconf, so that the installation directory of libbug is known at compile-time.
- On line 13, “libbug#.scm” is imported, so that the generated macros are namespaced correctly in external projects which import libbug. In the previous section, this file is created at compile-time. Remember that when “libbug-macros.scm” will be imported by an external project, “libbug#.scm” will exist with all of the namespaces defined in libbug[1].

8.4.3 Close Files At Compile-Time

At the end of compilation, these open files will need to be closed, and the namespace needs to be reset.

[1] Marty: “Well Doc, we can scratch that idea. I mean we can’t wait around a year and a half for this thing to get finished.” Doc Brown: “Marty it’s perfect, you’re just not thinking fourth-dimensionally. Don’t you see, the bridge will exist in 1985.” -Back to the Future 3

```
{define at-end-of-compilation
  [(display
    "
    (##namespace (\"\"))"
    libbug-macros-file)
   (force-output libbug-headers-file)
   (close-output-port libbug-headers-file)
   (force-output libbug-macros-file)
   (close-output-port libbug-macros-file)]}
;; close the call to at-compile-time
}
```

8.5 libbug-private#write-and-eval

Now that those files are open, namespaces will be written to “libbug#.scm” and macro definitions to “libbug-macros.scm”. However, the code shouldn’t have be to duplicated for each context, as was done for the previous three macros.

Create a macro named “write-and-eval” which will write the unevaluated form plus a newline to the file, and then return the form so that the compiler actually evaluate it[2].

```
{##define-macro libbug-private#write-and-eval
  [|port form|
   (eval `{begin
            (write ',form ,port)
            (newline ,port)})
   form]}
```

“write-and-eval” writes the form to a file, and only evaluates the form in the run-time context. For namespaces in libbug, namespaces should be valid at compile-time too.

8.6 libbug-private#namespace

Namespaces for procedures in libbug need to be available at compile-time, run-time, and in the namespace file for inclusion in projects which link to libbug.

[2]any procedure which is namespaced to “libbug-private” is not exported to the namespace file nor the macro file

```
{##define-macro libbug-private#namespace
  [|#!rest to-namespace|
   {begin
     (eval `{##namespace ("bug#" ,@to-namespace)})
     `{begin
        {libbug-private#write-and-eval
         libbug-headers-file
         {##namespace ("bug#" ,@to-namespace)}}}}]}
```

8.7 if

In the following, a new version of ”if” is defined named “bug#if”, where “bug#if” takes two zero-argument procedures, treating them as Church Booleans. bug#if was first used and described in section 2.3.

```
1  {libbug-private#namespace if}
2  {libbug-private#write-and-eval
3   libbug-macros-file
4   {at-both-times
5    {##define-macro if
6      [|pred ifTrue ifFalse|
7       {##if {and (list? ifTrue)
8                  (list? ifFalse)
9                  (equal? 'lambda (car ifTrue))
10                 (equal? 'lambda (car ifFalse))}
11            (list '##if pred
12                  `{begin ,@(cddr ifTrue)}
13                  `{begin ,@(cddr ifFalse)})
14            (error "bug#if requires two lambda expressions")}]}}}
```

- On line 7, “##if” is called. In Gambit’s system of namespacing, “##’ is prefixed to a variable name to specify to use the global namespace for that variable. “bug#if” is built on Gambit’s implementation of “if”, but since line 1 set the namespace of “if” to “bug#if”, “##if” must be used.
- On lines 7-10, check that the caller of “bug#if” is passing lambdas, i.e. has not forgetten that “if” is namespaced to “bug”.
- On line 14, if the caller of “bug#if” has not passed lambdas, error at compile-time.
- On line 11-13, evaluate the body of the appropriate lambda, depending on whether the predicate is true or false.

8.8 unit-test

Given that the reader now knows how to evaluate at compile-time, implementing a macro to execute tests at compile-time is trivial.

- Make a macro called “with-tests”, which takes an unevaluated definition and an unevaluated list of tests.
- “eval” the definiton at compile-time, “eval” the tests at compile-time. If any test evaluates to false, force the compiler to exit in error, producing and appropriate error message. If all of the tests pass, the Gambit compiler will proceed with compiling the definition.

```
{libbug-private#namespace unit-test}
{libbug-private#write-and-eval
 libbug-macros-file
 {##define-macro unit-test
   [|#!rest tests|
    (eval
     `(if {and ,@tests}
          [’noop]
          [(for-each pp (list "Test Failed" ’,tests))
           (error "Tests Failed")]))]}}
```

8.9 libbug-private#define

“libbug-private#define” is the main procedure-defining procedure used throughout libbug. “libbug-private#define” takes a variable name, a value to be stored in the variable, and an optional suite of tests.

```
1 {##define-macro
2   libbug-private#define
3   [|name body|
4    `{begin
5       {libbug-private#namespace ,name}
6       {at-both-times
7        {##define ,name ,body}}}]}
```

“libbug-private#define” defines the procedure/data at compile-time at run-time, and exports the namespace mapping to the appropriate file. “libbug-private#define” itself is not exported to the macros file.

On line 5-7, “with-tests” is applied to test at compile-time, thus also ensuring that the definition is available to other procedures both at compile-time and at run-time.

8.10 libbug-private#define-macro

Like “libbug-private#define” is built upon “##define”, “libbug-private#define-macro” is built upon “##define-macro”. Like “libbug-private#define”, “libbug-private#define-macro” uses “with-tests”, thus ensuring that the macro is available both at run-time and at compile-time. But, macros do not get compiled into libraries, so for other projects to use them, they must be exported to file.

The steps will be as follows:

- Write the macro to file

- Write the macro-expander to file
- Define the macro-expander within libbug
- Define the macro using “with-tests”.

```
{##define-macro libbug-private#define-macro
  [|name lambda-value #!rest tests|
```

8.10.1 Write Macros to File

```
   (write
    `{at-both-times
```

Macro Definition

The macro definition written to the file will be imported as text by other projects, which themselves may have different namespace mappings than libbug. To ensure that the macro works correctly in other contexts, the appropriate namespace mappings must be loaded for the definition of this macro definition.

```
1        {##define-macro
2          ,name
3          (lambda ,(cadr lambda-value)
4            ,(list 'quasiquote
5                   `{##let ()
6                      (##include "~~lib/gambit#.scm")
7                      (##include ,bug-configuration#libbugsharp)
8                      ,(if {and (pair? (caddr lambda-value))
9                                (equal? 'quasiquote
10                                       (caaddr lambda-value))}
11                          [(car (cdaddr lambda-value))]
12                          [(append (list 'unquote)
13                                   (cddr lambda-value))])})))}
```

- On line 3, the written-to-file lambda value shall have the same argument list as the argument list passed to “libbug-private#define-macro”

  ```
  > (cadr '[|foo bar| (quasiquote 5)])
  (foo bar)
  ```

  (Remember, all REPL sessions in this book assume that “bug-gsi” is running, and as such, the lambda literals will be preprocessed.

```
(cadr '[|foo bar| (quasiquote 5)])
```

expands to

```
(cadr '(lambda (foo bar) (quasiquote 5)))
```

- On line 4, the unevaluated form in argument "lambda-value" may or may not be quasiquoted. Either way, write a quasiquoted form to the file. In the case that the "lambda-value" argument was not actually intended to be quasiquoted, unquote the lambda's body (which is done on line 12-13), thereby negating the quasi-quoting from line 4.
- On line 4-5, rather than nesting quasiquotes, line 4 uses a technique of replacing a would-be nested quasiquote with ",(list 'quasiquote `(...)". This makes the code more readable [Norvig92, p. 854]. Should the reader be interested in learning more about nested quasiquotes, Appendix C of [Steele90, p. 960] is a great reference.
- On line 5-7, ensure that the currently unevaluated form will be evaluated in a context in which the namespaces resolve consistently as they were written in this book. Line 5 create a bounded context for namespace mapping. Line 6 sets standard Gambit namespace mappings, line 7 sets libbug's mappings.
- On line 8-10, check to see if the unevaluated form is quasiquoted.

  ```
  > (caaddr '[|foo bar| (quasiquote 5)])
  quasiquote
  ```

- On line 11, it is quasiquoted, as such, grab the content of the list minus the quasiquoting.

  ```
  > (car (cdaddr '[|foo bar| (quasiquote 5)]))
  5
  ```

  Remember that this value gets wrapped in a quasiquote from line 5

  ```
  > (list 'quasiquote (car (cdaddr '[|foo bar|
                                      (quasiquote 5)])))
  `5
  ```

- On line 12-13, since this is not a quasi-quoted form, just grab the form, and "unquote" it.

  ```
  > (append (list 'unquote ) (cddr '[|foo bar| (+ 5 5)]))
  ,(+ 5 5)
  ```

  Remember, this value gets wrapped in a quasiquote from line 4

  ```
  > (list 'quasiquote (append (list 'unquote )
                              (cddr '[|foo bar|
                                        (+ 5 5)])))
  `,(+ 5 5)
  ```

Macro to expand macro invocations

In order to be able to test the macro transformation before evaluation of the expanded code, create the macro with “-expand” suffixed to the “name”, with the same procedure body as the “lambda-value”’s body, but “quote” the result of the macro-expansion so that the compiler will not compile it. Locally define “gensym” in this generated procedure, so that tests may be written[3].

```
      {##define-macro
        ,(string->symbol (string-append (symbol->string name)
                                        "-expand"))
        (lambda ,(cadr lambda-value)
          {let ((gensym-count 0))
            {let ((gensym
                   [{begin
                      {set! gensym-count
                            (+ 1 gensym-count)}
                      (string->symbol
                       (string-append
                        "gensymed-var"
                        (number->string gensym-count)))}]))
              (list 'quote ,@(cddr lambda-value))}})}}
```

```
    libbug-macros-file)
   (newline libbug-macros-file)
```

8.10.2 Define Macro and Run Tests

Now that the macro has been exported to a file, now the macro must be defined within libbug itself. Firstly, create the expander.

```
   {let ((gensym-count (gensym)))
     `{begin
```

Namespace the procedure and the expander.

```
        {libbug-private#namespace ,name}
        {libbug-private#namespace
         ,(string->symbol
           (string-append (symbol->string name)
                          "-expand"))}
```

Create the expander similarly to the previous section.

[3]“##gensym”, by definition, creates a unique symbol which the programmer could never input, which is why it needs to be overridden for testing macro-expansions.

```
        {at-both-times
         {##define-macro
           ,(string->symbol
             (string-append (symbol->string name)
                            "-expand"))
           (lambda ,(cadr lambda-value)
             {let ((,gensym-count 0))
               {let ((gensym
                      [{begin
                         {set! ,gensym-count
                               (+ 1 ,gensym-count)}
                         (string->symbol
                          (string-append
                           "gensymed-var"
                           (number->string ,gensym-count)))}]))
                 (list 'quote ,@(cddr lambda-value))}})}}
```

Now that the macroexpander procedure has been defined, define the macro and execute the compile-time tests.

```
        {at-both-times
         {##define-macro
           ,name
           ,lambda-value}
         ,@tests}}}]}
```

8.11 macroexpand-1

"macroexpand-1" allows the programmer to test macro-expansion by writing

```
(equal? {macroexpand-1 (aif (+ 5 10)
                            (* 2 it))}
       '{let ((it (+ 5 10)))
          (if it
              [(* 2 it)]
              [#f])})
```

instead of

```
(equal? {aif-expand (+ 5 10)
                    (* 2 it))}
       '{let ((it (+ 5 10)))
          (if it
              [(* 2 it)]
              [#f])})
```

```
{libbug-private#define-macro
 macroexpand-1
 [|form|
  {let* ((m (car form))
         (the-expander (string->symbol
                        (string-append (symbol->string m)
                                       "-expand"))))
    `(,the-expander ,@(cdr form))}]}
```

8.12 libbug-private#define-structure

Like “##define-structure”, but additionally writes the namespaces to file.

```
{##define-macro
  libbug-private#define-structure
  [|name #!rest members|
   `{begin
      {libbug-private#namespace
       ,(string->symbol
         (string-append "make-"
                        (symbol->string name)))
       ,(string->symbol
         (string-append (symbol->string name)
                        "?"))
       ,@(map [|m|
               (string->symbol
                (string-append (symbol->string name)
                               "-"
                               (symbol->string m)))]
              members)
       ,@(map [|m|
               (string->symbol
                (string-append (symbol->string name)
                               "-"
                               (symbol->string m)
                               "-set!"))]
              members)}
      {at-both-times
       {##namespace (""
                     define
                     define-structure
                     )}
       {define-structure ,name ,@members}
       {##namespace ("libbug-private#"
                     define
                     )}
       {##namespace ("bug#"
                     define-structure
                     )}}}]}
```

Appendices

APPENDIX A

Compile-Time Language

This appendix[1] provides a quick tour of computer language which is interpreted by the compiler but which is absent in the generated machine code. Examples are provided in well-known languages to illustrate that many compilers are also interpreters for a subset of the language. This appendex provides a baseline understanding of compile-time computation so that the reader may contrast these languages' capabilities with libbug's. But first, let's discuss was is meant by the words "language","compiler", and "interpreter".

In "Introduction to Automata Theory, Languages, and Computation", Hopcroft, Motwani, and Ullman define language as "A set of strings all of which are chosen from some $\Sigma^\star$, where Σ is a particular alphabet, is called a language" [Hopcroft01, p. 30]. Plainly, that means an "alphabet" is a set of characters (for instance, ASCII), and that a computer "language" is defined as all of the possible sequences of characters from that alphabet which match some criterion.

An "interpreter" is a pre-existing computer program which takes a specific computer language as input, and does some action based on it. A "compiler" is a pre-existing computer program which takes computer language as input, but rather than immediately executing the actions specified by the input, instead it translates the input language into another computer language (typically machine code), which is then output to a file for interpretation[2] at a later time.

In practice though, the distinction is not clear cut. Most compilers do not exclusively translate from an input language to an output language; instead, they also interpret a subset of the input language as part of the process of generating the output language. Well, what types of computations can be performed by this subset of language, and how do they vary in expressive power?

A.1 C

Consider the following C code:

[1]Examples in the appendix will have boxes and line numbers around the code, but they are not part of libbug.
[2]the CPU can be viewed as an interpreter which takes machine code as its input

```
 1 #include <stdio.h>
 2 #define square(x) ((x) * (x))
 3 int fact(unsigned int n);
 4 int main(int argc, char* argv[]){
 5 #ifdef DEBUG
 6   printf("Debug - argc = %d\n", argc);
 7 #endif
 8  printf("%d\n",square(fact(argc)));
 9   return 0;
10 }
11 int fact(unsigned int n){
12   return n == 0
13     ? 1
14     : n * fact(n-1);
15 }
```

- On line 1, the #include preprocessor command is language that the compiler is intended to interpret, instructing the compiler to read the file "stdio.h" from the filesystem and to splice the content into the current C file. The #include command itself has no representation in the generated machine code, although the contents of the included file may.
- Line 2 defines a C macro. C macros are procedure definitions which are not to be translated into the output language, instead it is a procedure to be interpreted by the compiler at compile-time only. C macros take a text string as input and transforms it into a new text string as output. This expansion happens before the compiler does much else. For example, using GCC as a compiler, if you run the C preprocessor "cpp" on the above C code, you'll see that

  ```
    printf("%d\n",square(fact(argc)));
  ```

  expands into

  ```
    printf("%d\n",((fact(argc)) * (fact(argc))));
  ```

  before compilation.
- Line 3 defines a procedure prototype, so that the compiler knows the argument types and return type for a procedure not yet defined called "fact". It is language interpreted by the compiler to determine the types for the procedure call to "fact" on line 8.
- Lines 4 through 10 are a procedure definition, which will be translated into instructions in the generated machine code. Line 5 however, is language to be interpreted by the compiler, referencing a variable which is defined only during compilation, to detemine whether or not line 6 should be compiled.

A.2 C++

C++ inherits C's macros, but with the additional introduction of templates, C++'s compile-time language incidentally became Turing complete. This means that anything that can be calculated by a computer can be calculated using template expansion at compile-time. That's great! So how does a programmer begin to use this new expressive power?

The following is an example of calculating the factorial of 3, using C++ procedures for run-time calulation, and C++'s templates for compile-time calculation.

```
 1  #include <iostream>
 2  template <unsigned int n>
 3  struct factorial {
 4      enum { value = n * factorial<n - 1>::value };
 5  };
 6  template <>
 7  struct factorial<0> {
 8      enum { value = 1 };
 9  };
10  int fact(unsigned int n){
11    return n == 0
12      ? 1
13      : n * fact(n-1);
14  }
15  int main(int argc, char* argv[]){
16    std::cout << factorial<3>::value << std::endl;
17    std::cout << fact(3) << std::endl;
18    return 0;
19  }
```

- Lines 10-14 are the run-time calculation of "fact", identical to the previous version in C.
- Lines 2-9 are the template code for the compile-time calculation of "factorial".
- On line 16, "factorial<3>::value" is an language to be interpreted by the compiler via template expansions. Template expansions conditionally match patterns based on types (or values in the case of integers). For iteration, instead of loops, templates expand recursively. In this case, "factorial<3>::value" expands to "3 * factorial<3 - 1>::value". The compiler does the subtraction during compile-time, so "factorial<3>::value" expands to "3 * factorial<2>::value". This recursion will terminate on "factorial<0>::value" on line 7. (Even though the base case of "factorial<0>" is specified after the more general case of "factorial<n>", template expansion expands to the most specific case first. So the compiler will terminate.)
- On line 17, a run-time call to "fact", defined on line 10, is declared.

A.2.1 Disassembling the Object File

The drastic difference in the generated code can be observed by using "objdump -D".

```
400850: be 06 00 00 00   mov    $0x6,%esi
400855: bf c0 0d 60 00   mov    $0x600dc0,%edi
40085a: e8 41 fe ff ff   callq  4006a0 <_ZNSolsEi@plt>
.......
.......
.......
40086c: bf 03 00 00 00   mov    $0x3,%edi
400871: e8 a0 ff ff ff   callq  400816 <_Z4facti>
400876: 89 c6            mov    %eax,%esi
400878: bf c0 0d 60 00   mov    $0x600dc0,%edi
40087d: e8 1e fe ff ff   callq  4006a0 <_ZNSolsEi@plt>
```

- The instructions at memory locations 400850 through 40085a correspond to the printing of the compile-time expanded call to factorial<3>::value. The immediate value 6 is loaded into the esi register; then the second two lines call the printing routine[3].
- The instructions at locations 40086c through 40087d correspond to the printing of the run-time calculation to fact(3). The immediate value 3 is loaded into the edi register, fact is invoked, the result of calling fact is moved from the eax register to the esi register, and then printing routine is called.

The compile-time computation worked!

A.3 libbug

Like C++, libbug's compile-time language is Turing complete. Here is how to solve the same problem with libbug.

```
1 {at-both-times
2  {define fact
3    [|n| (if (= n 0)
4             [1]
5             [(* n (fact (- n 1)))])]}}
6
7 (pp (at-compile-time-expand (fact 3)))
8 (pp (fact 3))
```

- On line 1, the "at-both-times" macro is invoked, taking the unevaluated definition of "fact" as as argument, interpreting it at compile-time, and compiling it for use at runtime.
- On lines 2-5, the definition of the "fact".
- On line 7, "at-compile-time-expand" is a macro which takes unevaluated code, evaluates it to some result which is then compiled by the compiler. The code will expand at compile-time to "(pp 6)".
- On line 8, the run-time calculation of "(fact 3)".

[3]at least I assume, because I don't completely understand how C++ name-mangling works

A.3.1 Inspecting the Gambit VM Bytecode

By compiling the Scheme source to the “gvm” intermediate representation, the stated behavior can be verified.

```
 1    r1 = '6
 2    r0 = #4
 3    jump/safe fs=4 global[pp] nargs=1
 4  #4 fs=4 return-point
 5    r1 = '3
 6    r0 = #5
 7    jump/safe fs=4 global[fact] nargs=1
 8  #5 fs=4 return-point
 9    r0 = frame[1]
10    jump/poll fs=4 #6
11  #6 fs=4
12    jump/safe fs=0 global[pp] nargs=1
```

- Lines 1-4 correspond to “(pp (at-compile-time-expand (fact 3)))”. The precomputed value of “(fact 3)” is 6, which is directly stored into a GVM register, and then the “pp” routine is called to print it out.
- Lines 5-12 correspond to “(pp (fact 3))”. 3 is stored in a GVM register, “fact” is called, the result of which is passed to “pp”.

A.4 Comparison of Power

Although C++’s and libbug’s compile-time languages are both Turing complete, they vary drastically in actual real-world programming power. The language used for compile-time calculation of “fact” in C++ is a drastically different language than the one used for run-time. Although not fully demonstrated in this book, C++ template metaprogramming relies exclusively on recursion for repetition (it has no looping construct), it has no mutable state, and it lacks the ability to do input/output (I/O)[4]

In contrast, the compile-time language in libbug is the same exact language as the one that the compiler is compiling, complete with state and I/O! How can that power be used? This book is the beginning of an answer.

[4]For the masochist who wants to know more about C++’s compile-time language, I recommend [Abrahams2004]

APPENDIX B

Related Work

- Jonathan Blow. https://www.youtube.com/watch?v=UTqZNujQOlA
- "Compile-time Unit Testing", Aron Barath and Zoltan Porkolab, Eotvos Lorand University, http://ceur-ws.org/Vol-1375/SQAMIA2015_Paper1.pdf

Bibliography

[Abelson96] Abelon, Harold, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*, The MIT Press, Massachusetts, Second Edition, 1996.

[Abrahams2004] Abrahams, David and Aleksey Gurtovoy *C++ Template Metaprogramming*, Addison Wesley 2004.

[Church51] Church, Alonzo *The Calculi of Lambda-Conversion*, Princeton University Press, New Jersey, Second Printing, 1951.

[Dybvig03] Dybvig, R. Kent. *The Scheme Programming Language*, The MIT Press, Massachusetts, Third Edition, 2003.

[Feeley12] Feeley, Marc. https://mercure.iro.umontreal.ca/pipermail/gambit-list/2012-April/005917.html, 2012

[Friedman96] Friedman, Daniel P., and Matthias Felleisen *The Scheme Programming Language*, The MIT Press, Massachusetts, Fourth Edition, 1996.

[Graham94] Graham, Paul. *On Lisp*, Prentice Hall, New Jersey, 1994.

[Graham96] Graham, Paul. *ANSI Common Lisp*, Prentice Hall, New Jersey, 1996.

[Harvey01] Harvey, Brian and Matthew Wright. *Simply Scheme - Introducing Computer Science*, The MIT Press, Massachusetts, Second Edition, 2001.

[Hopcroft01] Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Massachusetts, Second Edition, 2001.

[Kiselyov98] Kiselyov, Oleg. http://okmij.org/ftp/Scheme/setf.txt , 1998.

[Knuth97] Knuth, Donald E. *The Art Of Computer Programming, Volume 1*, Addison Wesley, Massachusetts, Third Edition, 1997.

[Norvig92] Norvig, Peter *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, San Francisco, CA 1992.

[Pierce02] Pierce, Benjamin C. *Types and Programming Languages*, The MIT Press Cambridge, Massachusetts 2002.

[Stallings03] Stallings, William *Cryptography and Network Security*, Pearson Education, Upper Saddle River, New Jersey, Third Edition, 2002.

[Steele90] Steele Jr, Guy L. *Common Lisp the Language*, Digital Press, 1990.

Index