Ikarus Scheme User's Guide

(Preliminary Document) *Version 0.0.2*

Abdulaziz Ghuloum
November 12, 2007

Ikarus Scheme User's Guide

Contents

Chapter 1

Getting Started

1.1 Introduction

Ikarus Scheme is an implementation of the Scheme programming language. The preliminary release of Ikarus implements the majority of the features found in the current standard, the Revised^6 report on the algorithmic language Scheme[5] including full R^{6}RS library and script syntax, syntax-case, unicode strings, bytevectors, user-defined record types, exception handling, conditions, and enumerations. Over 80% of the R^{6}RS procedures and keywords are currently implemented and subsequent releases will proceed towards brining Ikarus to full R^{6}RS conformance.

The main purpose behind releasing Ikarus early is to give Scheme programmers the opportunity to experiment with the various new features that were newly introduced in R^{6}RS. The most important of such features is the ability to structure large programs into libraries; where each library extends the language through procedural and syntactic abstractions. Many useful libraries can be written using the currently supported set of R^{6}RS features including text processing tools, symbolic logic systems, interpreters and compilers, and many mathematical and scientific packages. It is my hope that this release will encourage the Scheme community to write and to share their most useful R^{6}RS libraries.

1.2 Technology Overview

Ikarus Scheme provides the programmer with many advantages:

Optimizing code generator: The compiler's backend employs state of the art technologies in code generation that produce fast efficient machine code. When developing computationally intensive programs, one is not constrained by using a slow interpreter.

Fast incremental compilation: Every library and script is quickly compiled to native machine code. When developing large software, one is not constrained by how slow the batch compiler runs.

Robust and fine-tuned standard libraries: The standard libraries are written such that they perform as much error checking as required to provide a safe and fast runtime environment.

Multi-generational garbage collector: The BiBOP[2] based garbage collector used in Ikarus allows the runtime system to expand its memory footprint as needed. The entire 32-bit virtual address space could be used and unneeded memory is released back to the operating system.

Supports many operating systems: Ikarus runs on the most popular and widely used operating systems for servers and personal computers. The supported systems include Mac OS X, GNU/Linux, FreeBSD, NetBSD, and Microsoft Windows.

1.3 System Requirements

1.3.1 Hardware

Ikarus Scheme runs on the IA-32 (*x86*) architecture supporting SSE2 extensions. This includes the Athlon 64, Sempron 64, and Turion 64 processors from AMD and the Pentium 4, Xeon, Celeron, Pentium M, Core, and Core2 processors from Intel. The system does not run on Intel Pentium III or earlier processors.

The Ikarus compiler generates SSE2 instructions to handle Scheme's IEEE floating point representation (*flonums*) for inexact numbers.

1.3.2 Operating Systems

Ikarus is tested under the following operating systems:

- Mac OS X version 10.4.
- Linux 2.6.18 (Debian, Fedora, Gentoo, and Ubuntu).
- FreeBSD version 6.2.
- NetBSD version 3.1.
- Microsoft Windows XP (using Cygwin 1.5.24).

1.3.3 Additional Software

- **GMP:** Ikarus uses the GNU Multiple Precision Arithmetic Library (GMP) for some bignum arithmetic operations. To build Ikarus from scratch, GMP version 4.2 or better must be installed along with the required header files. Pre-built GMP packages are available for most operating systems. Alternatively, GMP can be downloaded from `http://gmplib.org/`.
- **GCC:** The GNU C Compiler is required to build the Ikarus executable (e.g. the garbage collector, loader, and OS-related runtime). GCC versions 4.1 and 4.2 were successfully used to build Ikarus.
- **Autoconf and Automake:** The GNU Autoconf (version 2.59) and GNU Automake (version 1.9) tools are required if one wishes to modify the Ikarus source base. They are not required to build the official release of Ikarus.
- **XeLaTeX:** The XeLaTeX typesetting system is required for building the documentation. XeLaTeX (and XeTeX) is an implementation of the LaTeX (and TeX) typesetting system.

> *Note:* Ikarus runs in 32-bit mode only. To run it in 64-bit environments, you will have to obtain the 32-bit version of GMP, or compile it yourself after adding `ABI=32` to its configuration options.

1.4 Installation

If you are familiar with installing Unix software on your system, then all you need to know is that Ikarus uses the standard installation method found in most other Unix software. Simply run the following commands from the shell:

```
$ tar -zxf ikarus-n.n.n.tar.gz
$ cd ikarus-n.n.n
$ ./configure [--prefix=path] [CFLAGS=-I/dir] [LDFLAGS=-L/dir]
$ make
$ make install
$
```

The rest of this section describes the build process in more details. It is targeted to users who are unfamiliar with steps mentioned above.

1.4.1 Installation Details

1. Download the Ikarus source distribution. The source is distributed as a `gzip`-compressed `tar` file (`ikarus-n.n.n.tar.gz` where `n.n.n` is a 3-digit number indicating the current revision). The latest revision can be downloaded from the following URL:
   `http://www.cs.indiana.edu/~aghuloum/ikarus/`

2. Unpack the source distribution package. From your shell command, type:

   ```
   $ tar -zxf ikarus-n.n.n.tar.gz
   $
   ```

This creates the base directory `ikarus-n.n.n`.

3. Configure the build system by running the `configure` script located in the base directory. To do this, type the following commands:

   ```
   $ cd ikarus-n.n.n
   $ ./configure
   checking build system type... i386-apple-darwin8.10.1
   checking host system type... i386-apple-darwin8.10.1
   ...
   configure: creating ./config.status
   config.status: creating Makefile
   config.status: creating src/Makefile
   config.status: creating scheme/Makefile
   config.status: creating doc/Makefile
   config.status: executing depfiles commands
   $
   ```

   This configures the system to be built then installed in the system-wide location (binaries are installed in `/usr/local/bin`) . If you wish to install it in another location (e.g. in your home directory), you can supply a `--prefix` location to the `configure` script as follows:

   ```
   $ ./configure --prefix=/path/to/installation/location
   ```

   The `configure` script will fail if it cannot locate the location where GMP is installed. If running `configure` fails to locate GMP, you should supply the location in which the GMP header file, `gmp.h`, and the GMP library file, `libgmp.so`, are installed. This is done by supplying the two paths in the `CFLAGS` and `LDFLAGS` arguments:

   ```
   $ ./configure CFLAGS=-I/path/to/include LDFLAGS=-L/path/to/lib
   ```

4. Build the system by running:

   ```
   $ make
   ```

This performs two tasks. First, it builds the `ikarus` executable from the C files located in the `src` directory. It then uses the `ikarus` executable and the pre-built `ikarus.boot.orig` boot file to rebuild the Scheme boot image file `ikarus.boot` from the Scheme sources located in the `scheme` directory.

5. Install Ikarus by typing:

   ```
   $ make install
   ```

   If you are installing Ikarus in a system-wide location, you might need to have administrator privileges (use the `sudo` or `su` commands).

6. Test that Ikarus runs from the command line.

   ```
   $ ikarus
   Ikarus Scheme (Build 2007-10-20)
   Copyright (c) 2006-2007 Abdulaziz Ghuloum

   >
   ```

   If you get the prompt, then Ikarus was successfully installed on your system. You may need to update the `PATH` variable in your environment to contain the directory in which the `ikarus` executable was installed.

   Do not delete the `ikarus-n.n.n` directory from which you configured, built, and installed Ikarus. It will be needed if you decide at a later time to uninstall Ikarus.

1.4.2 Uninstalling Ikarus

To uninstall Ikarus, use the following steps:

```
$ cd path/to/ikarus-n.n.n
$ make uninstall
$
```

1.5 Command-line Switches

The `ikarus` executable recognizes a few command-line switches that influence how Ikarus starts.

- `ikarus -h`

  The presence of the `-h` flag causes `ikarus` to display a help message then exits. The help message summarizes the command-line switches. No further action is performed.

- `ikarus -b path/to/boot/file.boot`

  The `-b` flag (which requires an extra argument) directs `ikarus` to use the specified boot file as the initial system boot file. The boot file is a binary file that contains all the code and data of the Scheme system. In the absence of `-b` flag, the executable attempts to guess the location of the boot file using the following strategy:

  1. If `ikarus` was started by supplying an explicit location such as `/usr/local/bin/ikarus` or `./ikarus`, then the name of the boot file is the concatenation of a `.boot` prefix to the executable file name (e.g. `/usr/local/bin/ikarus.boot` or `./ikarus.boot`).
  2. Otherwise, `ikarus` assumes that it was started from a location in the `PATH` environment variable. In that case, it searches for the location of `ikarus` in the `PATH`. If `ikarus` is found in `/path/to/ikarus`, then the name of the boot file becomes `/path/to/ikarus.boot`.
  3. Failing both guesses, `ikarus` prints an error message and exits.

  The motivation for this strategy was to allow one to (1) rename the `ikarus` executable and the corresponding boot file to some new names (e.g. `my-ikarus` and `my-ikarus.boot`) without conflicting with other installed versions of Ikarus, and (2) override the location of the boot file for testing and building purposes (e.g. the installation process using one boot file to build another).

  The rest of the command-line arguments are recognized by the standard Scheme run time system. They are processed after the boot file is loaded.

- `ikarus --r6rs-script script-file-name [arguments ...]`

  The `--r6rs-script` argument instructs Ikarus that the supplied file is an R^{6}RS script. See Section 2.1 for a short introduction to writing R^{6}RS scripts. The script file name and any additional optional `arguments` can be obtained by calling the `command-line` procedure.

  ```
  $ cat test.ss
  (import (rnrs))
  (write (command-line))
  (newline)

  $ ikarus --r6rs-script test.ss hi there
  ("test.ss" "hi" "there")
  $
  ```

- `ikarus files ... [-- arguments ...]`

  The lack of an `--r6rs-script` argument causes Ikarus to start in interactive mode. Each of the `files` is first loaded, in the interaction environment. The interaction environment initially contains all the bindings exported from the `(ikarus)` library (see Chapter 3). The optional `arguments` following the `--` marker can be obtained by calling the `command-line` procedure. In interactive mode, the first element of the returned list will be the string `"*interactive*"`, corresponding to the script name in R^{6}RS-script mode.

  > *Note:* The interactive mode is intended for quickly experimenting with the built-in features. It is intended neither for developing applications nor for writing any substantial pieces of code. The main reason for this is that the interaction between R^{6}RS libraries and the interactive environment is not well understood. We hope to achieve better interaction between the two subsystems in the future.

Chapter 2

R^6RS Crash Course

The major difference between R^5RS and R^6RS is the way in which programs are loaded and evaluated.

In R^5RS, Scheme implementations typically start as an interactive session (often referred to as the REPL, or read-eval-print-loop). Inside the interactive session, the user enters definitions and expressions one at a time using the keyboard. Files, which also contain definitions and expressions, can be loaded and reloaded by calling the `load` procedure. The environment in which the interactive session starts often contains implementation-specific bindings that are not found R^5RS and users may redefine any of the initial bindings. The semantics of a loading a file depends on the state of the environment at the time the file contents are evaluated.

R^6RS differs from R^5RS in that it specifies how *whole programs*, or scripts, are compiled and evaluated. An R^6RS script is closed in the sense that all the identifiers found in the body of the script must either be defined in the script or imported from a library. R^6RS also specifies how *libraries* can be defined and used. While files in R^5RS are *loaded* imperatively into the top-level environments, R^6RS libraries can be *imported* declaratively in scripts and in other R^6RS libraries.

2.1 Writing a simple script

An R^{6}RS script is a set of definitions and expressions preceded by an `import` form. The `import` form specifies the language (i.e. the variable and keyword bindings) in which the library body is written. A very simple example of an R^{6}RS script is listed below.

```
(import (rnrs))

(display "Hello World!\n")
```

The first line imports the `(rnrs)` library. All the bindings exported from the `(rnrs)` library are made available to be used within the body of the library. The exports of the `(rnrs)` library include variables (e.g. `cons`, `car`, `display`, etc.) and keywords (e.g. `define`, `lambda`, `quote`, etc.). The second line displays the string `Hello World!` followed by a new line character.

In addition to expressions, such as the call to `display` in the previous example, a script may define some variables. The script below defines the variable `greeting` and calls the procedure bound to it.

```
(import (rnrs))

(define greeting
  (lambda ()
    (display "Hello World!\n")))

(greeting)
```

Additional keywords may be defined within a script. In the example below, we define the `(do-times n exprs ...)` macro that evaluates the expressions `exprs` `n` times. Running the script displays `Hello World` 3 times.

```
(import (rnrs))

(define greeting
  (lambda ()
    (display "Hello World!\n")))

(define-syntax do-times
  (syntax-rules ()
    [(_ n exprs ...)
     (let f ([i n])
       (unless (zero? i)
         exprs ...
         (f (- i 1))))]))

(do-times 3 (greeting))
```

2.2 Writing simple libraries

A script is intended to be a small piece of the program—useful abstractions belong to libraries. The `do-times` macro that was defined in the previous section may be useful in places other than printing greeting messages. So, we can create a small library, `(iterations)` that contains common iteration forms.

An R^{6}RS library form is made of four essential parts: (1) the library name, (2) the set of identifiers that the library exports, (3) the set of libraries that the library imports, and (4) the body of the library.

The library name can be any non-empty list of identifiers. R^{6}RS-defined libraries includes `(rnrs)`, `(rnrs unicode)`, `(rnrs bytevectors)`, and so on.

The library exports are a set of identifiers that are made available to importing libraries. Every exported identifier must be bound: it may either be defined in the libraries or imported from another library. Library exports include variables, keywords, record names, condition names.

Library imports are similar to script imports: they specify the set of libraries whose exports are made visible within the body of the library.

The body of a library contains definitions (variable, keyword, record, condition, etc.) followed by an optional set of expressions. The expressions are evaluated for side effect when needed.

The `(iteration)` library may be written as follows:

```
(library (iteration)
  (export do-times)
  (import (rnrs))

  (define-syntax do-times
    (syntax-rules ()
      [(_ n exprs ...)
       (let f ([i n])
         (unless (zero? i)
           exprs ...
           (f (- i 1))))])))
```

To use the `(iteration)` library in our script, we add the name of the library to the script's `import` form. This makes all of `(iteration)`'s exported identifiers, e.g. `do-times`, visible in the body of the script.

```
(import (rnrs) (iteration))

(define greeting
  (lambda ()
    (display "Hello World!\n")))

(do-times 3 (greeting))
```

2.3 R^{6}RS record types

R^{6}RS provides ways for users to define new types, called record types. A record is a fixed-size data structure with a unique type (called a record type). A record may have any finite number of fields that hold arbitrary values. This section briefly describes what we expect to be the most commonly used features of the record system. Full details are in the R^{6}RS Standard Libraries document[6].

2.3.1 Defining new record types

To define a new record type, use the `define-record-type` form. For example, suppose we want to define a new record type for describing points, where a point is a data structure that has two fields to hold the point's x and y coordinates. The following definition achieves just that:

```
(define-record-type point
  (fields x y))
```

The above use of `define-record-type` defines the following procedures automatically for you:

- The constructor `make-point` that takes two arguments, `x` and `y` and returns a new record whose type is point.
- The predicate `point?` that takes an arbitrary value and returns `#t` if that value is a point, `#f` otherwise.
- The accessors `point-x` and `point-y` that, given a record of type point, return the value stored in the `x` and `y` fields.

Both the `x` and `y` fields of the `point` record type are *immutable*, meaning that once a record is created with specific `x` and `y` values, they cannot be changed later. If you want the fields to be *mutable*, then you need to specify that explicitly as in the following example.

```
(define-record-type point
  (fields (mutable x) (mutable y)))
```

This definition gives us, in addition to the constructor, predicate, and accessors, two additional procedures:

- The mutators `set-point-x!` and `set-point-y!` that, given a record of type point, and a new value, sets the value stored in the `x` field or `y` field to the new value.

> *Note:* Records in Ikarus have a printable representation in order to enable debugging programs that use records. Records are printed in the `#[type-name field-values ...]` notation. For example, `(write (make-point 1 2))` produces `#[point 1 2]`.

2.3.2 Extending existing record types

A record type may be extended by defining new variants of a record with additional fields. In our running example, suppose we want to define a `colored-point` record type that, in addition to being a `point`, it has an additional field: a *color*. A simple way of achieving that is by using the following record definition:

```
(define-record-type cpoint
  (parent point)
  (fields color))
```

Here, the definition of `cpoint` gives us:

- A constructor `make-cpoint` that takes three arguments (`x`, `y`, and `color` in that order) and returns a `cpoint` record.

- A predicate `cpoint?` that takes a single argument and determines whether the argument is a `cpoint` record.
- An accessor `cpoint-color` that returns the value of the `color` field of a `cpoint` object.

All procedures that are applicable to records of type `point` (`point?`, `point-x`, `point-y`) are also applicable to records of type `cpoint` since a `cpoint` is also a `point`.

2.3.3 Specifying custom constructors

The record type definitions explained so far use the default constructor that takes as many arguments as there are fields and returns a new record type with the values of the fields initialized to the arguments' values. It is sometimes necessary or convenient to provide a constructor that performs more than the default constructor. For example, we can modify the definition of our `point` record in such way that the constructor takes either no arguments, in which case it would return a point located at the origin, or two arguments specifying the x and y coordinates. We use the `protocol` keyword for specifying such constructor as in the following example:

```
(define-record-type point
  (fields x y)
  (protocol
    (lambda (new)
      (case-lambda
        [(x y) (new x y)]
        [()    (new 0 0)]))))
```

The protocol here is a procedure that takes a constructor procedure `new` (`new` takes as many arguments as there are fields.) and returns the desired custom constructor that we want (The actual constructor will be the value of the `case-lambda` expression in the example above). Now the constructor `make-point` would either take two arguments which constructs a `point` record as before, or no arguments, in which case `(new 0 0)` is called to construct a point at the origin.

Another reason why one might want to use custom constructors is to precompute the initial values of some fields based on the values of other fields. An example of this case is adding a `distance` field to the record type which is computed as $d = \sqrt{x^2 + y^2}$. The protocol in this case may be defined as:

```
(define-record-type point
  (fields x y distance)
  (protocol
    (lambda (new)
      (lambda (x y)
        (new x y (sqrt (+ (expt x 2) (expt y 2))))))))
```

Note that derived record types need not be modified when additional fields are added to the parent record type. For example, our `cpoint` record type still works unmodified even after we added the new `distance` field to the parent. Calling `(point-distance (make-cpoint 3 4 #xFF0000))` returns `5.0` as expected.

2.3.4 Custom constructors for derived record types

Just like how base record types (e.g. `point` in the running example) may have a custom constructor, derived record types can also have custom constructors that do other actions. Suppose that you want to construct `cpoint` records using an optional color that, if not supplied, defaults to the value 0. To do so, we supply a `protocol` argument to `define-record-type`. The only difference here is that the procedure `new` is a *curried* constructor. It first takes as many arguments as the constructor of the parent record type, and returns a procedure that takes the initial values of the new fields.

In our example, the constructor for the `point` record type takes two arguments. `cpoint` extends `point` with one new field. Therefore, `new` in the definition below first takes the arguments for `point`'s constructor, then takes the initial color value. The definition below shows how the custom constructor may be defined.

```
(define-record-type cpoint
  (parent point)
  (fields color)
  (protocol
    (lambda (new)
      (case-lambda
        [(x y c) ((new x y) c)]
        [(x y)   ((new x y) 0)]))))
```

2.4 Exception Handling

The procedure `with-exception-handler` allows the programmer to specify how to handle exceptional situations. It takes two procedures as arguments:

- An exception handler which is a procedure that take a single argument, the object that was raised.
- A body thunk which is a procedure with no arguments whose body is evaluated with the exception handler installed.

In addition to installing exception handlers, R^6RS provides two ways of raising exceptions: `raise` and `raise-continuable`. We describe the procedure `raise-continuable` first since it's the simpler of the two. For the code below, assume that `print` is defined as:

```
(define (print who obj)
  (display who)
  (display ": ")
  (display obj)
  (newline))
```

The first example, below, shows how a simple exception handler is installed. Here, the exception handler prints the object it receives and returns the symbol `there`. The call to `raise-continuable` calls the exception handler, passing it the symbol `here`. When the handler returns, the returned value becomes the value of the calls to `raise-continuable`.

```
(with-exception-handler
  (lambda (obj)                       ;;; prints
    (print "handling" obj)            ;;;   handling: here
    'there)                           ;;;   returned: there
  (lambda ()
    (print "returned" (raise-continuable 'here))))
```

Exceptional handlers may nest, and in that case, if an exception is raised while evaluating an inner handler, the outer handler is called as the following example illustrates:

```
(with-exception-handler
  (lambda (obj)                       ;;; prints
    (print "outer" obj)               ;;;   inner: here
    'outer)                           ;;;   outer: there
  (lambda ()                          ;;;   returned: outer
    (with-exception-handler
      (lambda (obj)
        (print "inner" obj)
        (raise-continuable 'there))
      (lambda ()
        (print "returned" (raise-continuable 'here))))))
```

In short, `with-exception-handler` binds an exception handler within the dynamic context of evaluating the thunk, and `raise-continuable` calls it.

The procedure `raise` is similar to `raise-continuable` except that if the handler returns, a new exception is raised, calling the next handler in sequence until the list of handlers is exhausted.

```
(call/cc                                ;;; prints
  (lambda (escape)                      ;;;   inner: here
    (with-exception-handler             ;;;   outer: #[condition ---]
      (lambda (obj)                     ;;; returns
        (print "outer" obj)             ;;;   12
        (escape 12))
      (lambda ()
        (with-exception-handler
          (lambda (obj)
            (print "inner" obj)
            'there)
          (lambda ()
            (print "returned" (raise 'here))))))))
```

Here, the call to `raise` calls the inner exception handler, which returns, causing `raise` to re-raise a non-continuable exception to the outer exception handler. The outer exception handler then calls the escape continuation.

The following procedure provides a useful example of using the exception handling mechanism. Consider a simple definition of the procedure `configuration-option` which returns the value associated with a key where the key/value pairs are stored in an association list in a configuration file.

```
(define (configuration-option filename key)
  (cdr (assq key (call-with-input-file filename read))))
```

Possible things may go wrong with calling `configuration-option` including errors opening the file, errors reading from the file (file may be corrupt), error in `assq` since what's read may not be an association list, and error in `cdr` since the key may not be in the association list. Handling all error possibilities is tedious and error prone. Exceptions provide a clean way of solving the problem. Instead of guarding against all possible errors, we install a handler that suppresses all errors and returns a default value if things go wrong. Error handling for `configuration-option` may be added as follows:

```
(define (configuration-option filename key default)
  (define (getopt)
    (cdr (assq key (call-with-input-file filename read))))
  (call/cc
    (lambda (k)
      (with-exception-handler
        (lambda (_) (k default))
        getopt))))
```

Chapter 3

The `(ikarus)` library

In addition to the libraries listed in the R^{6}RS standard, Ikarus contains the `(ikarus)` library which provides additional useful features. The `(ikarus)` library is a composite library—it exports a superset of all the supported bindings of R^{6}RS. While not all of the exports of `(ikarus)` are documented at this time, this chapter attempts to describe a few of these useful extensions.

3.1 Parameters

Parameters in Ikarus[1] are intended for customizing the behavior of a procedure during the dynamic execution of some piece of code. Parameters are first class entities (represented as procedures) that hold the parameter value. A parameter procedure accepts either zero or one argument. If given no arguments, it returns the current value of the parameter. If given a single argument, it must set the state to the value of the argument. Parameters replace the older concept of using starred `*global*` customization variables. For example, instead of writing:

```
(define *screen-width* 72)
```

and then mutate the variable `*screen-width*` with `set!`, we could wrap `*screen-width*` with a `screen-width` parameter as follows:

```
(define *screen-width* 72)
(define screen-width
  (case-lambda
    [()  *screen-width*]
    [(x) (set! *screen-width* x)]))
```

The value of `screen-width` can now be passed as argument, returned as a value, and exported from libraries.

`make-parameter` **procedure**

`(make-parameter x)`
`(make-parameter x f)`

As parameters are common in Ikarus, the procedure `make-parameter` is defined to model common usage pattern of parameter construction.

---

[1] Parameters are found in many Scheme implementations such as Chez Scheme and MzScheme.

`(make-parameter x)` constructs a parameter with `x` as the initial value. For example, the code above could be written succinctly as:

```
(define screen-width (make-parameter 72))
```

`(make-parameter x f)` constructs a parameter which filters the assigned values through the procedure `f`. The initial value of the parameter is the result of calling `(f x)`. Typical used of the filter procedure include checking some constraints on the passed argument or converting it to a different data type. The `screen-width` parameter may be constructed more robustly as:

```
(define screen-width
  (make-parameter 72
    (lambda (w)
      (assert (and (integer? w) (exact? w)))
      (max w 1))))
```

This definition ensures, through `assert`, that the argument passed is an exact integer. It also ensures, through `max` that the assigned value is always positive.

`parameterize` **syntax**

`(parameterize ([lhs* rhs*] ...) body body* ...)`

Parameters can be assigned to by simply calling the parameter procedure with a single argument. The `parameterize` syntax is used to set the value of a parameter within the dynamic extent of the `body body* ...` expressions.

The `lhs* ...` are expressions, each of which must evaluate to a parameter. Such parameters are not necessarily constructed by `make-parameter`—any procedure that follows the parameters protocol works.

The advantage of using `parameterize` over explicitly assigning to parameters (same argument applies to global variables) is that you're guaranteed that whenever control exits the body of a `parameterize` expression, the value of the parameter is reset back to what it was before the body expressions were entered. This is true even in the presence of `call/cc`, errors, and exceptions.

The following example shows how to set the text property of a terminal window. The parameter `terminal-property` sends an ANSI escape sequence to the terminal whenever the parameter value is changed. The use of `terminal-property` within `parameterize` changes the property before `(display "RED!")` is called and resets it back to normal when the body returns.

```
(define terminal-property
  (make-parameter "0"
    (lambda (x)
      (display "\x1b;[")
      (display x)
      (display "m")
      x)))

(display "Normal and ")
(parameterize ([terminal-property "41;37"])
  (display "RED!"))
(newline)
```

3.2 Local Modules

This section is not documented yet. Please refer to Section 10.5 of Chez Scheme User's Guide [1], Chapter 3 of Oscar Waddel's Ph.D Thesis [7], and its POPL99 paper [8] for details on using the `module` and `import` keywords. Ikarus's internal module system is similar in spirit to that of Chez Scheme.

`module` **syntax**

```
(module M definitions ... expressions ...)
(module definitions ... expressions ...)
```

`import` **syntax**

```
(import M)
```

3.3 Gensyms

Gensym stands for a *generated symbol*—a fresh symbol that is generated at run time and is guaranteed to be *not* `eq?` to any other symbol present in the system. Gensyms are useful in many applications including expanders, compilers, and interpreters when generating an arbitrary number of unique names is needed.

Ikarus is similar to Chez Scheme in that the readers (including the `read` procedure) and writers (including `write` and `pretty-print`) maintain the read/write invariance on gensyms. When a gensym is written to an output port, the system automatically generates a random unique identifier for the gensym. When the gensym is read back though the `#{gensym}` read syntax, a new gensym is *not* regenerated, but instead, it is looked up in the global symbol table.

A gensym's name is composed of two parts: a *pretty* string and a *unique* string. The Scheme procedure `symbol->string` returns the pretty string of the gensym and not its unique string. Gensyms are printed by default as `#{pretty-string unique-string}`.

`gensym` **procedure**

`(gensym)`
`(gensym string)`
`(gensym symbol)`

The procedure `gensym` constructs a new gensym. If passed no arguments, it constructs a gensym with no pretty name. The pretty name is constructed when and if the pretty name of the resulting gensym is needed. See `gensym-prefix` (page 35) and `gensym-count` (page 35) for details.

```
> (gensym)
#{g0 |y0zf>GlFvcTJE0xw|}
> (gensym)
#{g1 |U%X&sF6kX!YC8LW=|}
> (eq? (gensym) (gensym))
#f
```

`(gensym string)` constructs a new gensym with `string` as its pretty name. Similarly, `(gensym symbol)` constructs a new gensym with the pretty name of `symbol`, if it has one, as its pretty name.

```
> (gensym "foo")
#{foo |>VgOllCM&$dSvRN=|}
> (gensym 'foo)
#{foo |!TqQLmtw2hoEYfU>|}
> (gensym (gensym 'foo))
#{foo |N2C>5O0>C?OROUBU|}
```

`gensym?` **procedure**

`(gensym? x)`

The `gensym?` predicate returns `#t` if its argument is a gensym, and returns `#f` otherwise.

```
> (gensym? (gensym))
#t
> (gensym? 'foo)
#f
> (gensym? 12)
#f
```

`gensym->unique-string` **procedure**

`(gensym->unique-string gensym)`

The `gensym->unique-string` procedure returns the unique name associated with the gensym argument.

```
> (gensym->unique-string (gensym))
"YukrolLMgP?%ElcR"
```

#{gensym} **reader syntax**

`#{unique-name}`
`#{pretty-name unique-name}`
`#:pretty-name`

Ikarus's `read` and `write` procedures extends the lexical syntax of Scheme by the ability to read and write gensyms using one of the three forms listed above.

`#{unique-name}` constructs, at read time, a gensym whose unique name is the one specified. If a gensym with the same unique name already exists in the system's symbol table, that gensym is returned.

```
> '#{some-long-name}
#{g0 |some-long-name|}
> (gensym? '#{some-long-unique-name})
#t
> (eq? '#{another-unique-name} '#{another-unique-name})
#t
```

The two-part `#{pretty-name unique-name}` gensym syntax is similar to the syntax shown above with the exception that if a new gensym is constructed (that is, if the gensym did not already exist in the symbol table), the pretty name of the constructed gensym is set to `pretty-name`.

```
> '#{foo unique-identifier}
#{foo |unique-identifier|}
> '#{unique-identifier}
#{foo |unique-identifier|}
> '#{bar unique-identifier}
#{foo |unique-identifier|}
```

The `#:pretty-name` form constructs, at read time, a gensym whose pretty name is `pretty-name` and whose unique name is fresh. This form guarantees that the resulting gensym is not `eq?` to any other symbol in the system.

```
> '#:foo
#{foo |j=qTGlEwS/Zlp2Dj|}
> (eq? '#:foo '#:foo)
#f
```

generate-temporaries **example**

The `(rnrs syntax-case)` library provides a `generate-temporaries` procedure, which takes a syntax object (representing a list of things) and returns a list of fresh identifiers. Using `gensym`, that procedure can be defined as follows:

```
(define (generate-temporaries* stx)
  (syntax-case stx ()
    [(x* ...)
     (map (lambda (x)
            (datum->syntax #'unimportant
              (gensym
                (if (identifier? x)
                    (syntax->datum x)
                    't))))
          #'(x* ...))]))
```

The above definition works by taking the input `stx` and destructuring it into the list of syntax objects `x* ...`. The inner procedure maps each `x` into a new syntax object (constructed with `datum->syntax`). The datum is a gensym, whose name is the same name as `x` if `x` is an identifier, or the symbol `t` if `x` is not an identifier. The output of `generate-temporaries*` generates names similar to their input counterpart:

```
> (print-gensym #f)
> (generate-temporaries* #'(x y z 1 2))
(#<syntax x> #<syntax y> #<syntax z> #<syntax t> #<syntax t>)
```

3.4 Printing

`pretty-print` **procedure**

`(pretty-print datum)`
`(pretty-print datum output-port)`

The procedure `pretty-print` is intended for printing Scheme data, typically Scheme programs, in a format close to how a Scheme programmer would write it. Unlike `write`, which writes its input all in one line, `pretty-print` inserts spaces and new lines in order to produce more pleasant output.

```
(define fact-code
 '(letrec ([fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))])
    (fact 5)))

> (pretty-print fact-code)
(letrec ((fact
          (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
  (fact 5))
```

The second argument to `pretty-print`, if supplied, must be an output port. If not supplied, the `current-output-port` is used.

> *Limitations:* As shown in the output above, the current implementation of `pretty-print` does not handle printing of square brackets properly.

`pretty-width` **parameter**

`(pretty-width)`
`(pretty-width n)`

The parameter `pretty-width` controls the number of characters after which the `pretty-print` starts breaking long lines into multiple lines. The initial value of `pretty-width` is set to 60 characters, which is suitable for most terminals and printed material.

```
> (parameterize ([pretty-width 40])
     (pretty-print fact-code))
(letrec ((fact
           (lambda (n)
             (if (zero? n)
                 1
                 (* n (fact (- n 1)))))))
  (fact 5))
```

Note that `pretty-width` does not guarantee that the output will not extend beyond the specified number. Very long symbols, for examples, cannot be split into multiple lines and may force the printer to go beyond the value of `pretty-width`.

`format` **procedure**

`(format fmt-string args ...)`

The procedure `format` produces a string formatted according to the value of `fmt-string` and the supplied arguments. The format string contains markers in which the string representation of each argument is placed. The markers include:

`"~s"` instructs the formatter to place the next argument as if the procedure `write` has printed it. If the argument contains a string, the string will be quoted and all quotes and backslashes in the string will be escaped. Similarly, characters will be printed using the `#`
`x` notation.

`"~a"` instructs the formatter to place the next argument as if the procedure `display` has printed it. Strings and characters are placed as they are in the output.

`"˜b"` instructs the formatter to convert the next argument to its binary (base 2) representation. The argument must be an exact number. Note that the `#b` numeric prefix is not produced in the output.

`"˜o"` is similar to `"˜b"` except that the number is printed in octal (base 8).

`"˜x"` is similar to `"˜b"` except that the number is printed in hexadecimal (base 16).

`"˜d"` outputs the next argument, which can be an exact or inexact number in its decimal (base 10) representation.

`"˜˜"` instructs the formatter to place a tilde character, ˜, in the output without consuming an argument.

```
> (format "message: ~s, ~s, and ~s" 'symbol "string" #\c)
"message: symbol, \"string\", and #\\c"

> (format "message: ~a, ~a, and ~a" 'symbol "string" #\c)
"message: symbol, string, and c"
```

---

printf

procedure

`(printf fmt-string args ...)`

The procedure `printf` is similar to `format` except that the output is sent to the `current-output-port` instead of being collected in a string.

```
> (printf "message: ~s, ~s, and ~s\n" 'symbol "string" #\c)
message: symbol, "string", and #\c

> (printf "message: ~a, ~a, and ~a\n" 'symbol "string" #\c)
message: symbol, string, and c
```

---

fprintf

procedure

`(fprintf output-port fmt-string args ...)`

The procedure `fprintf` is similar to `printf` except that the output port to which the output is sent is specified as the first argument.

`print-graph` **parameter**

```
(print-graph)
(print-graph #t)
(print-graph #f)
```

The `print-graph` parameter controls how the writers (e.g. `pretty-print` and `write`) handle shared and cyclic data structures. In Ikarus, all writers detect cyclic data structures and they all terminate on all input, cyclic or otherwise.

If the value of `print-graph` is set to `#f` (the default), then the writers does not attempt to detect shared data structures. Any part of the input that is shared is printed as if no sharing is present.

If the value of `print-graph` is set to `#t`, all sharing of data structures is marked using the `#n=` and `#n#` notation.

```
> (parameterize ([print-graph #f])
    (let ([x (list 1 2 3 4)])
      (pretty-print (list x x x))))
((1 2 3 4) (1 2 3 4) (1 2 3 4))

> (parameterize ([print-graph #t])
    (let ([x (list 1 2 3 4)])
      (pretty-print (list x x x))))
(#0=(1 2 3 4) #0# #0#)

> (parameterize ([print-graph #f])
    (let ([x (list 1 2)])
      (let ([y (list x x x x)])
        (set-car! (last-pair y) y)
        (pretty-print (list y y)))))
(#0=((1 2) (1 2) (1 2) #0#) #0#)
```

```
> (parameterize ([print-graph #t])
    (let ([x (list 1 2)])
      (let ([y (list x x x x)])
        (set-car! (last-pair y) y)
        (pretty-print (list y y)))))
(#0=(#1=(1 2) #1# #1# #0#) #0#)
```

---

print-gensym **parameter**

```
(print-gensym)
(print-gensym #t)
(print-gensym #f)
(print-gensym 'pretty)
```

The parameter `print-gensym` controls how gensyms are printed by the various writers.

If the value of `print-gensym` is `#f`, then gensym syntax is suppressed by the writers and only the gensyms' pretty names are printed. If the value of `print-gensym` is `#t`, then the full `#{pretty unique}` syntax is printed. Finally, if the value of `print-gensym` is the symbol `pretty`, then gensyms are printed using the `#:pretty` notation.

```
> (parameterize ([print-gensym #f])
    (pretty-print (list (gensym) (gensym))))
(g0 g1)

> (parameterize ([print-gensym #t])
    (pretty-print (list (gensym) (gensym))))
(#{g2 |KR1M2&CTt1<B0n/m|} #{g3 |FBAb&7NC6&=c82!O|})

> (parameterize ([print-gensym 'pretty])
    (pretty-print (list (gensym) (gensym))))
(#:g4 #:g5)
```

The initial value of `print-gensym` is `#t`.

`gensym-prefix` **parameter**

`(gensym-prefix)`
`(gensym-prefix string)`

The parameter `gensym-prefix` specifies the string to be used as the prefix to generated pretty names. The default value of `gensym-prefix` is the string `"g"`, which causes generated strings to have pretty names in the sequence `g0`, `g1`, `g2`, etc.

```
> (parameterize ([gensym-prefix "var"] [print-gensym #f])
     (pretty-print (list (gensym) (gensym) (gensym))))
(var0 var1 var2)
```

Beware that the `gensym-prefix` controls how pretty names are generated, and has nothing to do with how `gensym` constructs a new gensym. In particular, notice the difference between the output in the first example with the output of the examples below:

```
> (pretty-print
    (parameterize ([gensym-prefix "var"] [print-gensym #f])
      (list (gensym) (gensym) (gensym))))
(g3 g4 g5)

> (let ([ls (list (gensym) (gensym) (gensym))])
    (parameterize ([gensym-prefix "var"] [print-gensym #f])
      (pretty-print ls)))
(var5 var6 var7)
```

`gensym-count` **parameter**

`(gensym-count)`
`(gensym-count n)`

The parameter `gensym-count` determines the number which is attached to the `gensym-prefix` when gensyms' pretty names are generated. The value of `gensym-count` starts at 0 when the system starts and is incremented every time a pretty name is generated. It might be set to any non-negative integer value.

```
> (let ([x (gensym)])
    (parameterize ([gensym-count 100] [print-gensym #f])
      (pretty-print (list (gensym) x (gensym)))))
(g100 g101 g102)
```

Notice from all the examples so far that pretty names are generated in the order at which the gensyms are printed, not in the order in which gensyms were created.

3.5 Tracing

`trace-define` **syntax**

`(trace-define (name . args) body body* ...)`
`(trace-define name expression)`

The `trace-define` syntax is similar to `define` except that the bound value, which must be a procedure, becomes a traced procedure. A traced procedure prints its arguments when it is called and prints its values when it returns.

```
> (trace-define (fact n)
    (if (zero? n) 1 (* n (fact (- n 1)))))
> (fact 5)
|(fact 5)
| (fact 4)
| |(fact 3)
| | (fact 2)
| | |(fact 1)
| | | (fact 0)
| | | 1
| | |1
| | 2
| |6
| 24
|120
120
```

The tracing facility in Ikarus preserves and shows tail recursion and distinguishes it from non-tail recursion by showing tail calls starting at the same line in which their parent was called.

```
> (trace-define (fact n)
    (trace-define (fact-aux n m)
      (if (zero? n) m (fact-aux (- n 1) (* n m))))
```

```
    (fact-aux n 1))
> (fact 5)
|(fact 5)
|(fact-aux 5 1)
|(fact-aux 4 5)
|(fact-aux 3 20)
|(fact-aux 2 60)
|(fact-aux 1 120)
|(fact-aux 0 120)
|120
120
```

Moreover, the tracing facility interacts well with continuations and exceptions.

```
> (call/cc
    (lambda (k)
      (trace-define (loop n)
        (if (zero? n)
            (k 'done)
            (+ (loop (- n 1)) 1)))
      (loop 5)))
|(loop 5)
| (loop 4)
| |(loop 3)
| | (loop 2)
| | |(loop 1)
| | | (loop 0)
done
```

`trace-lambda` **syntax**

`(trace-lambda name args body body* ...)`

The `trace-lambda` macro is similar to `lambda` except that the resulting procedure is traced: it prints the arguments it receives and the results it returns.

`make-traced-procedure` **procedure**

`(make-traced-procedure name proc)`

The procedure `make-traced-procedure` takes a name (typically a symbol) and a procedure. It returns a procedure similar to `proc` except that it traces its arguments and values.

```
> (define (fact n)
    (if (zero? n)
        (lambda (k) (k 1))
        (lambda (k)
          ((fact (- n 1))
           (make-traced-procedure `(k ,n)
             (lambda (v)
               (k (* v n))))))))
> (call/cc
    (lambda (k)
      ((fact 5) (make-traced-procedure 'K k))))
|((k 1) 1)
|((k 2) 1)
|((k 3) 2)
|((k 4) 6)
|((k 5) 24)
|(K 120)
120
```

3.6 Timing

This section describes some of Ikarus's timing facilities which may be useful for benchmarking and performance tuning.

`time` **syntax**

`(time expression)`

The `time` macro performs the following: it evaluates `expression`, then prints a summary of the run time statistics, then returns the values returned by `expression`. The run-time summary includes the number of bytes allocated, the number of garbage collection runs, and the time spent in both the mutator and the collector.

```
> (let ()                                  ;;; 10 million
    (define ls (time (vector->list (make-vector 10000000))))
    (time (append ls ls))
    (values))
running stats for (vector->list (make-vector 10000000)):
    3 collections
    672 ms elapsed cpu time, including 547 ms collecting
    674 ms elapsed real time, including 549 ms collecting
    120012328 bytes allocated
running stats for (append ls ls):
    4 collections
    1536 ms elapsed cpu time, including 1336 ms collecting
    1538 ms elapsed real time, including 1337 ms collecting
    160000040 bytes allocated
```

Note: The output listed above is *just a sample* that was taken at some point on some machine. The output on your machine at the time you read this may vary.

`time-it` **procedure**

`(time-it who thunk)`

The procedure `time-it` takes a datum denoting the name of the computation and a thunk (i.e. a procedure with no arguments), invokes the thunk, prints the stats, and returns the values obtained from invoking the thunk. If the value of `who` is non-false, `who` is used when displaying the run-time statistics. If the value of `who` is `#f`, then no name for the computation is displayed.

```
> (time-it "a very fast computation"
    (lambda () (values 1 2 3)))
running stats for a very fast computation:
    no collections
    0 ms elapsed cpu time, including 0 ms collecting
    0 ms elapsed real time, including 0 ms collecting
    56 bytes allocated
1
2
3

> (time-it #f (lambda () 12))
running stats:
    no collections
    0 ms elapsed cpu time, including 0 ms collecting
    0 ms elapsed real time, including 0 ms collecting
    32 bytes allocated
12
```

Chapter 4

Missing Features

Ikarus does not fully conform to R^6RS yet. Although it implements the most immediately useful features of R^6RS including more than 80% of R^6RS's macros and procedures, some areas are still lacking. This section summarizes the set of missing features and procedures.

- Numeric tower is complete except for complex numbers.
  Consequences:
  – Reader does not recognize complex number notation (e.g. `5-7i`).
  – Procedures that may construct complex numbers from non-complex arguments may signal an error or return an incorrect value (for example, `(sqrt -1)` should *not* be `+nan.0`).
- Reader does not recognize `#!r6rs` syntax. It should be modified to accept both `#!r6rs` and `#!ikarus` so that Ikarus-specific reader features (gensym syntax, record syntax, shared graphs, fasl objects, etc.) can be enabled/disabled as needed.
- The procedure `equal?` may not terminate on `equal?` infinite (circular) input.
- Representation of I/O ports is missing a transcoder field.

4.1 List of missing R^{6}RS procedures

The following procedures are missing from `(rnrs base)`:

```
angle magnitude make-polar make-rectangular
```

The following procedures are missing form `(rnrs bytevectors)`:

```
string->utf16     string->utf32    utf16->string     utf32->string
```

The following procedures are missing from `(rnrs unicode)`:

```
string-downcase   string-foldcase   string-titlecase   string-upcase
string-normalize-nfc                string-normalize-nfd
string-normalize-nfkc               string-normalize-nfkd
```

The following procedures are missing from `(rnrs arithmetic bitwise)`:

```
bitwise-ior  bitwise-xor   bitwise-if
bitwise-copy-bit-field     bitwise-bit-set?       bitwise-copy-bit
bitwise-first-bit-set      bitwise-bit-count     bitwise-bit-field
bitwise-reverse-bit-field  bitwise-rotate-bit-field  bitwise-length
```

The following procedures are missing from `(rnrs arithmetic fixnum)`:

```
fxbit-count  fxbit-field  fxbit-set?  fxcopy-bit  fxcopy-bit-field
fxfirst-bit-set fxlength  fxreverse-bit-field   fxrotate-bit-field
```

The following procedures are missing from `(rnrs hashtables)`:

```
hashtable-copy
make-eqv-hashtable         make-hashtable
hashtable-hash-function    hashtable-equivalence-function
equal-hash   string-hash   string-ci-hash   symbol-hash
```

The following procedures are missing from `(rnrs io ports)`:

```
call-with-bytevector-output-port     call-with-string-output-port
binary-port?   textual-port?    port-eof?
port-has-port-position?         port-position
port-has-set-port-position!?    set-port-position!
call-with-port        close-port
get-bytevector-all    get-bytevector-some
get-bytevector-n      get-bytevector-n!
get-char              put-char      lookahead-char
get-u8                lookahead-u8  put-u8
get-string-all        get-string-n  get-string-n!  put-string
get-datum             put-datum
make-custom-binary-input-port         make-custom-binary-input/output-port
make-custom-binary-output-port        make-custom-textual-input-port
make-custom-textual-input/output-port make-custom-textual-output-port
open-bytevector-input-port            open-bytevector-output-port
open-file-input-port   open-file-input/output-port open-file-output-port
open-string-input-port open-string-output-port
output-port-buffer-mode
transcoded-port       port-transcoderput-bytevector
standard-error-port   standard-input-port   standard-output-port
string->bytevector    bytevector->string
```

Bibliography

[1] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005.

[2] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BiBOP: Flexible and efficient storage management for dynamically-typed languages. Technical Report 400, Indiana University, March 1994.

[3] Abdulaziz Ghuloum and R. Kent Dybvig. Generation-friendly Eq hash tables. In *Proceedings of the 2007 Workshop on Scheme and Functional Programming*, pages 27–35. Université Laval Technical Report DIUL-RT-0701, 2007.

[4] Abdulaziz Ghuloum and R. Kent Dybvig. Implicit phasing for R6RS libraries. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 303–314, New York, NY, USA, 2007. ACM.

[5] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton Van Straaten (Editors). Revised^6 report on the algorithmic language Scheme. 2007.

[6] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton Van Straaten (Editors). Revised^6 report on the algorithmic language Scheme–standard libraries. 2007.

[7] Oscar Waddell. *Extending the Scope of Syntactic Abstraction*. PhD thesis, Indiana University Computer Science Department, August 1999.

[8] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–213, January 1999.

Index