# Ikarus Scheme User's Guide

Version 0.0.4

Abdulaziz Ghuloum

December 26, 2008

Ikarus Scheme User's Guide (Version 0.0.4)

# 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[**?**] including full R$^6$RS library and script syntax, syntax-case, unicode strings, bytevectors, user-defined record types, exception handling, conditions, and enumerations. User-define R$^6$RS libraries can be compiled in-memory on the fly or compiled to disk and loaded in subsequent runs.

In addition to supporting R$^6$RS (and most of the features found in the the previous R$^n$RS standards), Ikarus supports powerful interprocess communication facilities including support for communication with child processes via pipes and with remote processes via TCP and UDP sockets. The facility also allows for both synchronous and asynchronous communication so that a Scheme program running in Ikarus can communicate with many processes concurrently.

Ikarus also supports basic foreign function interface (FFI) facilities. This allows the programmer to define accessors and mutators for native data structures. It also allows for Scheme programs to dynamically load any library found on the host machine. Native procedures and Scheme procedures can call to each other by the call-out and call-back facilities of Ikarus.

## 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[**?**] 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.

**32-bit and 64-bit computing:** Ikarus supports both the Intel-x86 and the AMD-64 architectures. 64-bit computing allows the programmer to utilize larger address space (larger than 4GB) and provides a greater range for fixnums (61-bit fixnums). Running in 32-bit mode, however, makes more efficient utilization of resources due to the smaller memory footprint for most data structures.

**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

This section provides an overview of the hardware and software requirements needed for running Ikarus.

### 1.3.1 Hardware

Ikarus Scheme runs in 32-bit mode on the IA-32 (*x86*) architecture supporting SSE2 extensions. It also runs in 64-bit mode on platforms supporing the AMD-64 architecture. 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. Ikarus 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 and 10.5.
- 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/.
- **FFI:** The `libffi` library (version `3.0.6`) can be utilized to enable Scheme procedures to call and be called from native procedure (see Chapter 5 for

details). To enable `libffi` while building Ikarus, one must pass `--enable-libffi` to the `configure` script along with the paths to `libffi`'s include and lib directory. The `libffi` library can be obtained from http://sourceware.org/libffi/. FFI support is optional. It is not required if all one needs is writing pure Scheme code.

- **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 2.61) and GNU Automake (version 1.10) 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. XeLaTeX can be obtained from http://scripts.sil.org/xetex and is included with TeX-Live[1] and and Mac-TeX[2] distributions.

## 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] http://tug.org/texlive/
[2] http://tug.org/mactex/

### 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
```

If you wish to enable support for accessing and calling to/from foreign libraries, you need to configure Ikarus with the `--enable-libffi` option and supply the approriate `CFLAGS` and `LDFLAGS` as needed.

```
$ ./configure --enable-libffi \
     [CFLAGS=/path/to/ffi.h] \
     [LDFLAGS=/path/to/libffi.so|.dylib|.dll]
```

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 version 0.0.4
Copyright (c) 2006-2008 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 will use the default boot file. Running ikarus -h shows the location where the default boot file was installed.

  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 files ... --r6rs-script script-file arguments ...`

  The `--r6rs-script` argument instructs Ikarus that the supplied file is an R$^6$RS script. The optional list of `files` must be paths to files, each containing a set of libraries that Ikarus must load, sequentially, before running the R$^6$RS script `script-file`. 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.

## 1.6 Using `scheme-script`

Scheme scripts can be executed using the `ikarus --r6rs-script script-name` command as described in the previous section. For convenience, Ikarus follows

the R[6]RS recommendations and installs a wrapper program called `scheme-script`. Typically, a script you write would start with a `#!` line that directs your operating system to the interpreter used to evaluate the script file. The following example shows a very simple script that uses the `scheme-script` command.

```
#!/usr/bin/env scheme-script

(import (rnrs))
(display "Hello World\n")
```

If the above script was placed in a file called `hello-world`, then one can make it executable using the `chmod` Unix command.

```
$ cat hello-world
#!/usr/bin/env scheme-script

(import (rnrs))
(display "Hello World\n")

$ chmod 755 hello-world
$ ./hello-world
Hello World
$
```

> *Under Mac OS X,* if a script name ends with the `.command` extension, then it can be executed from the Finder by double-clicking on it. This brings up a terminal window in which the script is executed. The `.command` extension can be hidden from the *Get Info* item from the Finder's File menu.

## 1.7 Mapping library names to file names

The name of an R$^6$RS library consists of a non-empty list of identifiers (symbols), followed by an optional version number. All of the standard R$^6$RS libraries are built into Ikarus, thus importing any one of them does not require any special action other than listing the library name in the `import` part of a library or a script. The same holds for the `(ikarus)` library (chapter 3, page 27).

When importing a user library, Ikarus uses a simple mechanism to map library names to file names. A library name is converted to a file path by joining the library identifiers with a path separator, e.g. `"/"`.

| Library Name | ⇒ | File name |
|---|---|---|
| `(foo)` | ⇒ | `foo` |
| `(foo bar)` | ⇒ | `foo/bar` |
| `(foo bar baz)` | ⇒ | `foo/bar/baz` |

Having mapped a library name to a file path, Ikarus attempts to locate that file in one of several locations. The locations attempted depend on two settings: the search path and the file extension set (e.g., `.sls`, `.ss`, `.scm`, etc.). First, Ikarus attempts to locate the file in the current working directory from which Ikarus was invoked. In the current working directory, Ikarus enumerates all file extensions first before searching other locations. If the file is not found in the current directory, Ikarus tries to find it in the Ikarus library directory. The Ikarus library directory is determined when Ikarus is installed (based on the `--prefix` argument that was passed to the `configure` script). If Ikarus failes to locate the library file, it raises an exception and exits.

> *Tip:* Use simple library names for the libraries that you define. Library names that contain non-printable characters, complex punctuations, or unicode may pose a challenge for some operating systems. If Ikarus cannot find a library, it will raise an error listing the locations in which it looked, helping you move the library file to a place where Ikarus can find it.

## 1.8 Writing cross-implementation libraries

When searching for a library, Ikarus appends an extension (e.g., `.ss`) to the appropriate file name (e.g., `foo/bar`). The initial set of file extensions are: `/main.ikarus.sls`, `/main.ikarus.ss`, `/main.ikarus.scm`, `/main.sls`, `/main.ss`, `/main.scm`, `.ikarus.sls`, `.ikarus.ss`, `.ikarus.scm`, `.sls`, `.ss`, and `.scm`.

The list of file extensions are searched sequentially. As a consequence, files ending with the `.ikarus.*` extensions are given precedence over files that have generic Scheme extensions. The rationale for this behavior is to facilitate writing cross-implementation libraries: ones that take advantage of implementation-specific features, while at the same time provide a fail-safe alternative for other R$^6$RS implementations.

Consider for example a program which would like to use the `pretty-print` procedure to format some code, and suppose further that pretty printing is just a nice add-on (e.g., using `write` suffices, but pretty-printing is *just prettier*) Ikarus exports a good pretty-printing facility in its `(ikarus)` library. However, since `pretty-print` is not a standard procedure, a program that uses it would be rendered unportable to other R$^6$RS Scheme implementations.

The programmer can put the `.ikarus.*` extensions to use in this situation. First, the programmer writes two versions of a `(pretty-printing)` library: one for use by Ikarus, and one portable for other implementations.

```
(library (pretty-printing) ;;; this is pretty-printing.ikarus.ss
  (export pretty-print)    ;;; can only be used by Ikarus
  (import (only (ikarus) pretty-print)))
```

```
(library (pretty-printing) ;;; this is pretty-printing.sls
  (export pretty-print)    ;;; *portable* though not very pretty.
  (import (rnrs))          ;;; for any other implementation
  (define (pretty-print x port)
    (write x port)
    (newline port)))
```

The `/main.*` extensions serve a different purpose. Often times, a set of libraries are distributed together as a package and it is convenient for the programmer to group related files in directories. If a package contains the libraries `(foo)`, `(foo core)`, and `(foo compat)`, then putting all such library files together in one directory makes it easier to package, install, and remove these libraries en masse. The layout of the package would look like:

```
foo/README              :                ignored by Ikarus
foo/COPYING             :
foo/main.ss             : (foo)          implementation independent
foo/core.ss             : (foo core)
foo/compat.ss           : (foo compat)   default R6RS library
foo/compat.ikarus.ss    :                specific for Ikarus
foo/compat.mzscheme.ss  :                specific for MzScheme
```

By default, running the `configure` script installs a set of contributed libraries into the `/usr/local/lib/ikarus` directory. If a `--prefix DIR` argument was supplied to `configure`, then the libraries are installed in the `DIR/ikarus/lib` directory.

You may install additional libraries into the Ikarus library directory. Doing so makes them available for `import` into other libraries and script regardless of where the importing code is located or the current directory in which it is executed.

## 1.9 Defining IKARUS_LIBRARY_PATH

There may be situations in which you may wish to install your own libraries into a different location. For example, you may not have sufficient administrative privileges to write to the system directory, or you may wish to keep your own libraries separate from the standard libraries. Whatever the reason is, your can store your library files in any location you want and set up the `IKARUS_LIBRARY_PATH` environment variable to point to these locations. The value of `IKARUS_LIBRARY_PATH` is a colon-separated list of directories in which Ikarus will search.

For example, suppose your script imports the `(streams derived)` library. First, Ikarus will map the library name to the file path `streams/derived.ss`. Suppose

that Ikarus was installed using the `--prefix /usr/local` configuration option, and suppose further that the value of `IKARUS_LIBRARY_PATH` is set by the user to be `/home/john/ikarus-libraries:/home/john/srfis`. Ikarus will search in the following locations in sequence until it finds the file it is looking for.

```
./streams/derived.ss
/home/john/ikarus-libraries/streams/derived.ss
/home/john/srfis/streams/derived.ss
/usr/local/lib/ikarus/streams/derived.ss
```

The method in which the value of `IKARUS_LIBRARY_PATH` is defined is typically shell dependant. If you use GNU Bash, you typically set the values of environment variables in the `~/.bash_profile` or `~/.bashrc` file by adding the following lines:

```
IKARUS_LIBRARY_PATH=/path/to/some/directory:/and/another
export IKARUS_LIBRARY_PATH
```

# Chapter 2

# R$^6$RS Crash Course

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

In R$^5$RS, 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$^5$RS and users may redefine any of the initial bindings. The semantics of loading a file depends on the state of the environment at the time the file contents are evaluated.

R$^6$RS differs from R$^5$RS in that it specifies how *whole programs*, or scripts, are compiled and evaluated. An R$^6$RS 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$^6$RS also specifies how *libraries* can be defined and used. While files in R$^5$RS are typically *loaded* imperatively into the top-level environments, R$^6$RS libraries are *imported* declaratively in scripts and in other R$^6$RS 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.

```
#!/usr/bin/env scheme-script
(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 script. 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.

```
#!/usr/bin/env scheme-script
(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.

```
#!/usr/bin/env scheme-script
(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 library or imported using the `import` form. Library exports include variables, keywords, record names, and 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.

```
#!/usr/bin/env scheme-script
(import (rnrs) (iteration))

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

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

## 2.3 R⁶RS record types

R⁶RS 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 R⁶RS Standard Libraries document[**?**].

### 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 `point-x-set!` and `point-y-set!` 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 so 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 takes 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$^6$RS provides two ways of raising exceptions: `raise` and `raise-continuable`. We describe the `raise-continuable` procedure 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 call 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. Extensions to Scheme's lexical syntax are also documented.

---

## #!ikarus — reader syntax

Ikarus extends Scheme's lexical syntax (R$^6$RS Chapter 4) in a variety of ways including:

- end-of-file marker, #!eof (page 29)
- gensym syntax, #{gensym} (page 37)
- graph syntax, #nn= #nn# (page 42)

The syntax extensions are made available by default on all input ports, until the #!r6rs token is read. Thus, reading the #!r6rs token disables all extensions to the lexical syntax on the specific port, and the #!ikarus enables them again.

If you are writing code that is intended to be portable across different Scheme implementations, we recommend adding the #!r6rs token to the top of every script

and library that you write. This allows Ikarus to alert you when using non-portable features. If you're writing code that's intended to be Ikarus-specific, we recommend adding the `#!ikarus` token in order to get an immediate error when your code is run under other implementations.

---

## port-mode **procedure**

`(port-mode ip)`

The `port-mode` procedure accepts an input port as an argument and returns one of `r6rs-mode` or `ikarus-mode` as a result. All input ports initially start in the `ikarus-mode` and thus accept Ikarus-specific reader extensions. When the `#!r6rs` token is read from a port, its mode changes to `ikarus-mode`.

```
> (port-mode (current-input-port))
ikarus-mode
> #!r6rs (port-mode (current-input-port))
r6rs-mode
> #!ikarus (port-mode (current-input-port))
ikarus-mode
```

---

## set-port-mode! **procedure**

`(set-port-mode! ip mode)`

The `set-port-mode!` procedure modifies the lexical syntax accepted by subsequent calls to `read` on the input port. The mode is a symbol which should be one of `r6rs-mode` or `ikarus-mode`. The effect of setting the port mode is similar to that of reading the `#!r6rs` or `#ikarus` from that port.

```
> (set-port-mode! (current-input-port) 'r6rs-mode)
> (port-mode (current-input-port))
r6rs-mode
```

## #!eof

**reader syntax**

The end-of-file marker, `#!eof`, is an extension to the R$^6$RS syntax. The primary utility of the `#!eof` marker is to stop the reader (e.g. `read` and `get-datum`) from reading the rest of the file.

```
#!/usr/bin/env scheme-script
(import (ikarus))
<some code>
(display "goodbye\n")

#!eof
<some junk>
```

The `#!eof` marker also serves as a datum in Ikarus, much like `#t` and `#f`, when it is found inside other expressions.

```
> (eof-object)
#!eof
> (read (open-input-string ""))
#!eof
> (read (open-input-string "#!eof"))
#!eof
> (quote #!eof)
#!eof
> (eof-object? '#!eof)
#t
> #!r6rs #!eof
Unhandled exception
 Condition components:
   1. &error
   2. &who: tokenize
   3. &message: "invalid syntax: #!e"
> #!ikarus #!eof
$
```

## 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 mutating the variable `*screen-width*` with `set!`, we could wrap the variable `*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 the common usage pattern of parameter construction.

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

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

```
(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 uses 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 library imports

---

`import` **syntax**

```
(import import-spec* ...)
```

The `import` keyword which is exported from the `(ikarus)` library can be used anywhere definitions can occur: at a script body, library's top-level, or in internal definitions context. The syntax of the local `import` form is similar to the `import` that appears at the top of a library or a script form, and carries with it the same restrictions: no identifier name may be imported twice unless it denotes the same identifier; no identifier may be both imported and defined; and imported identifiers are immutable.

Local `import` forms are useful for two reasons: (1) they minimize the namespace clutter that usually occurs when many libraries are imported at the top level, and (2) they limit the scope of the import and thus help modularize a library's dependencies.

Suppose you are constructing a large library and at some point you realize that one of your procedures needs to make use of some other library for performing a specific task. Importing that library at top level makes it available for the entire library. Consequently, even if that library is no longer used anywhere in the code (say when the code that uses it is deleted), it becomes very hard to delete the import without first examiniming the entire library body for potential usage leaks. By locally importing a library into the appropriate scope, we gain the ability to delete the `import` form when the procedure that was using it is deleted.

## 3.3 Local modules

This section is not documented yet. Please refer to Section 10.5 of Chez Scheme User's Guide [**?**], Chapter 3 of Oscar Waddel's Ph.D Thesis [**?**], and its POPL99 paper [**?**] 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)
```

## 3.4 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. If `gensym` is passed a string, that string is used as the pretty name. If `gensym` is passed a symbol, the pretty name of the symbol is used as the pretty name of the returned gensym. See `gensym-prefix` (page 44) and `gensym-count` (page 45) 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 extend 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>)
```

## 3.5 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 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.

Note that the #b, #o, and #x numeric prefixes are not added to the output when ~b, ~o, and ~x are used.

```
> (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.

```
> (let ([n (+ (expt 2 32) #b11001)])
    (printf "~d = #b~b = #x~x\n" n n n))
4294967321 = #b100000000000000000000000011001 = #x100000019
```

---

## 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 graph notation is a way of marking and referencing parts of a data structure and, consequently, creating shared and cyclic data structures at read time instead of resorting to explicit mutation at run time. The #$n$= marks the following data structure with mark $n$, where $n$ is a nonnegative integer. The #$n$# references the data structure marked $n$. Marks can be assigned and referenced in any order but each mark must be assigned to exactly once in an expression.

```
> (let ([x '#0=(1 2 3)])
    (eq? x '#0#))
#t
> (let ([x '#0#] [y '#0=(1 2 3)])
    (eq? x y))
#t
> (eq? (cdr '(12 . #1#)) '#1=(1 2 3))
#t
> (let ([x '#1=(#1# . #1#)])
    (and (eq? x (car x))
         (eq? x (cdr x))))
#t
```

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 initial value of `gensym-count` is 0 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.

## 3.6 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
```

## 3.7 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
    24 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
    0 bytes allocated
12
```

# Chapter 4

# The `(ikarus ipc)` library

4.1

## 4.1 Environment variables

When the operating system starts a process, it starts the process in some environment that maps environment variables to values. Typical keys found in the environment are `HOME` (pointing to the home directory of the user), `PATH` (containing a colon-separated list of directories to be searched when running a command from the shell), `SHELL`, `EDITOR`, and `PAGER`. This section describes the procedures provided by Ikarus for manipulating this environment.

The environment procedures are placed in the `(ikarus ipc)` library because they provide a (limited) way for one process to communicate to a subprocess, akin to parameter passing.

---

`getenv` **procedure**

```
(getenv key)
```

The procedure `getenv` retrieves the value associated with `key` (which must be a string) in the environment. The value returned is a (utf8-decoded) string, or `#f` if there is no mapping for `key` in the environment.

---

`setenv` **procedure**

```
(setenv key value)
(setenv key value overwrite?)
```

The procedure `setenv` sets the mapping of `key` to `value` in the environment. Both `key` and `value` must be strings. If the `overwrite?` argument is provided and is `#f`, `setenv` does not overwrite a value associated with `key` if one already exists. The procedure `setenv` may raise an exception if the operating system cannot allocate enough memory to hold the new mapping.

---

`unsetenv` **procedure**

```
(unsetenv key)
```

The procedure `unsetenv` removes `key` and its associated value (if one exists) from the environment.

> *Caveat:* The underlying system procedure `setenv` may leak some memory in some operating systems when passed some values. Ikarus has no way of getting around this system limitation and thus may leak some memory for some calls to `setenv`. Use sparingly.

## 4.2 Subprocess communication

---

`system` **procedure**

`(system string)`

The `system` procedure takes a string represeting an external shell command and arguments and invokes the shell (typically `sh` on Unix systems) on this command. The returned value from `system` is the exit status of the external command.

Ikarus's `system` procedure is a thin wrapper around the `system` procedure in the Standard C Library `libc`.

```
> (system "ls M*")
Makefile        Makefile.am     Makefile.in
0
```

---

`process` **procedure**

`(process program-name args ...)`

The `process` procedure takes as input a string representing the path to an external program and a set of strings that are the arguments to the external program. It invokes the program with the given arguments, and returns four values: (1) a process identifier (`pid`), (2) an output port which pipes to the process's `stdin`, (3) an input port wired to the process's `stdout`, and (4) an input port wired to the process's `stderr`. All three ports are blocking: reading and writing to any one of them blocks Ikarus until the some bytes are available for reading or writing. Attempting to read from the process's `stdout` port may block indefinitely if the external

program does not write anything (e.g., if it attempts to read from `stdin` instead). Communicating with an external process must therefore be done according to the protocol in which the external process communicates.

---

## process-nonblocking **procedure**

```
(process-nonblocking program-name args ...)
```

The procedure `process-nonblocking` is similar to the `process` procedure except that the three returned ports are put in nonblocking mode. Attempting to perform a read or write operation on a nonblocking port in which bytes are not available for reading or writing causes Ikarus to enqueue the port with the continuation in which the read/write operation occurs and attempt to dispatch previously enqueued ports on which some bytes are ready for read or write.

---

## waitpid **procedure**

```
(waitpid)
(waitpid pid)
(waitpid pid block?)
(waitpid pid block? want-error?)
```

The `waitpid` procedure waits for the process with the given `pid` to terminate and, if successful, returns a `wstatus` object encapsulating the wait status of the process. Without arguments, `waitpid` defaults the `pid` to `-1` which allows one to wait for any child process to exit. If the `block?` argument is true (the default), `waitpid` blocks indefinitely waiting for a child process to exit. When `block?` is false, `waitpid` returns immediately regardless of whether or not a child process has exited. The `want-error?` controls what happens if `block?` was specified to be `#f` and no child had exited. If `want-error?` is true (the default), an error is signaled. Otherwise, `waitpid` returns `#f` if no process has exited. Operations on the wait status result are listed below.

---

## wstatus-pid **procedure**

```
(wstatus-pid wstatus)
```

The `wstatus-pid` returns the `pid` of the process whose status is recorded in the

`wstatus` object. This `pid` is most useful when the default `pid` of `-1` is given to `waitpid` and thus the pid of the exiting process is not known beforehand.

---

### wstatus-exit-status **procedure**

```
(wstatus-exit-status wstatus)
```

The procedure `wstatus-exit-status` returns the exit status of the child process. It is typically `0` if the child exited normally and has other numeric values if the child process encountered an error.

---

### wstatus-received-signal **procedure**

```
(wstatus-received-signal wstatus)
```

The procedure `wstatus-received-signal` returns the name of the signal (or the number of the signal if the name is not known) that caused the child process to exit.

The signal name is one of the following symbols:

```
SIGABRT  SIGALRM  SIGBUS   SIGCHLD  SIGCONT    SIGFPE   SIGHUP
SIGILL   SIGINT   SIGKILL  SIGPIPE  SIGQUIT    SIGSEGV  SIGSTOP
SIGTERM  SIGTSTP  SIGTTIN  SIGTTOU  SIGUSR1    SIGUSR2  SIGPOLL
SIGPROF  SIGSYS   SIGTRAP  SIGURG   SIGVTALRM  SIGXCPU  SIGXFSZ
```

---

### kill **procedure**

```
(kill pid signal-name)
```

The `kill` procedure takes a `pid` and a signal name (a symbol from the list above) and asks the operating system to send the signal to the given process.

## 4.3 TCP and UDP sockets

TODO

---

### tcp-connect **procedure**

```
(tcp-connect host-name service-name)
```

TODO

---

### tcp-connect-nonblocking **procedure**

```
(tcp-connect-nonblocking host-name service-name)
```

TODO

---

### udp-connect **procedure**

```
(udp-connect host-name service-name)
```

TODO

---

### udp-connect-nonblocking **procedure**

```
(udp-connect-nonblocking host-name service-name)
```

TODO

---

### tcp-server-socket **procedure**

```
(tcp-server-socket port-number)
```

TODO

---

### tcp-server-socket-nonblocking **procedure**

```
(tcp-server-socket-nonblocking port-number)
```

TODO

---

### accept-connection **procedure**

```
(accept-connection tcp-server)
```

TODO

---

## accept-connection-nonblocking **procedure**

```
(accept-connection-nonblocking tcp-server)
```

TODO

---

## close-tcp-server-socket **procedure**

```
(close-tcp-server-socket tcp-server)
```

TODO

# Chapter 5

# The (ikarus foreign) library

This chapter describes the facilities through which Ikarus interfaces with the host operating system and other external libraries. The facilities of the (ikarus foreign) library give the Scheme program unrestricted access to the computer memory, allowing one to allocate, access, modify, and free memory as needed. The facilities also allow the Scheme program to *call out* to system procedures as well as allow the native procedures to *call back* into Scheme.

This chapter is organized as follows: Section 5.1 gives an overview of the basic concepts such as shared libraries, external symbols, foreign data types, pointers, and procedures. Section 5.3 describes the primitives that (ikarus foreign) provides for direct manipulation of memory. Section **??** deals with loading external libraries and calling out to native library procedures and calling back into Scheme. To demonstrate the usefulness of the foreign facilities, Ikarus ships with two libraries that also serve as extended examples for using the system. Section **??** describes The OpenGL library (ikarus opengl) which allows the programmer to produce 2D and 3D computer graphics. Section **??** describes the (ikarus objc) which allows the programmer to access libraries and frameworks written in the Objective-C programming language and thus provides full access to the Mac OS X system (e.g., making graphical user interfaces with Cocoa and drawing graphics with Quartz all from Scheme).

Ikarus version 0.0.4 is the first version of Ikarus to support the described foreign interfaces.

## 5.1 Overview

In order to make full use of the computer, it is important for a programming environment (e.g., Ikarus Scheme) to facilitate access to the underlying architecture on which it runs. The underlying architecture includes the API provided by the host operating system kernel (e.g., Linux), the system libraries (e.g., `libc`), and other site-installed libraries (e.g., `sqlite3`). Providing direct access to such API from within Scheme allows the programmer to write Scheme libraries that have few or no dependencies on external programs (such as `C` development toolchain). When dealing with system libraries, the programmer must have a thorough understanding of many aspects of the targetted system. This section attempts to provide answers to many questions that are frequently encountered when interfacing to external libraries.

## 5.2 Memory management

Ikarus Scheme is a managed environment. Like in many programming environments, Ikarus manages its own memory. Scheme objects are allocated in a special memory region (the Scheme heap) and have type-specific object layout that allows the run time system to distinguish object types and allows the garbage collector to locate all potentially live objects and reclaim the memory of dead objects. Scheme objects are also *opaque* in the sense that the data structures used to represent Scheme objects (e.g., pairs) are not exposed to the programmer, who can only interact with objects through an interface (e.g., `car`, `cdr`).

Unmanaged environments, such as the operating system on which Ikarus runs, require that the programmer manages the allocation and deallocation of system resources herself. Memory regions, file handles, external devices, the screen, etc., are all examples of resources whose management must be coordinated among the different parts of the system, and this becomes the responsibility of the programmer who is wiring the different subsystems together.

Memory, from a system's point of view, is *transparent*. A pointer is an integer denoting an address of memory. This memory address may contain a value that requires interpretation. At the lowest-level, each byte of memory contains eight bits, each of which may be toggled on or off. A level higher, contiguous sequences of bytes

are grouped together and are interpreted as integers, floating point numbers, or pointers to other memory addresses. These are the basic data types that are often interpreted atomically. Yet a level higher, groups of basic types form data structures such as arrays, linked lists, trees, and so on. Objects, as found in object-oriented programming languages, are at an even higher level of abstraction since they are treated as opaque references that retain state and know how to respond to messages.

The procedures in the `(ikarus foreign)` library are meant to provide a way to interface with the low level memory operations such as setting and getting bytes from specific locations in memory. Although they do not provide high-level operations, the basic procdures make implementing high-level operations (such as the Objective-C system presented in Chapter **??**) possible. Programmers are encouraged to define their own abstractions that are most suitable for the specific target library rather than using the low-level operations directly. This results in writing more robust and more easily maintainable libraries. To put it more boldly: **Do not sprinkle your code with low-level memory operations**.

## 5.3 Memory operations

---

`malloc` **procedure**

`(malloc n)`

The `malloc` procedure allocates `n` bytes of memory and returns a pointer to the allocated memory. The `malloc` Scheme procedure is implemented using the host-provided `malloc` system procedure (often found in `libc`). The number of bytes, `n`, must be a positive exact integer.

```
> (malloc 10)
#<pointer #x00300320>
> (malloc 10000)
#<pointer #x01800400>
```

## free **procedure**

```
(free p)
```

The `free` procedure takes a pointer and frees the memory region at the given address. The memory region must be allocated with `malloc`, `calloc`, or a similar system procedure. Once freed, memory operations on the given address are invalid and may cause the system to crash at unpredictable times. Ikarus cannot check for such errors since the memory may be freed by procedures that are external to Ikarus.

## pointer->integer **procedure**

```
(pointer->integer p)
```

The procedure `pointer->integer` converts the value of the pointer `p` to an exact integer value. The result may be a fixnum or a bignum depending on the pointer.

## integer->pointer **procedure**

```
(integer->pointer n)
```

The procedure `integer->pointer` converts the exact integer `n` to a pointer value. The lower 32 bits (or 64 bits on 64-bit systems) of the value of `n` are significant in computing the pointer value. It is guaranteed that `(integer->pointer (pointer->integer p))` points to the same address as `p`.

## pointer? **procedure**

```
(pointer? x)
```

The predicate `pointer?` returns `#t` if the value of `x` is a pointer, and returns `#f` otherwise.

> *Note:* The result of calling the procedures `eq?`, `eqv?` and `equal?` on pointer values is unspecified.

## pointer-set-c-char! **procedure**

```
(pointer-set-c-char! p i n)
```

The procedure `pointer-set-c-char!` sets a single byte of memory located at offset `i` from the pointer `p` to the value of `n`. The pointer `p` must be a valid pointer. The index `i` must be an exact integer. The value of `n` must be an exact integer. Only the 8 lowermost bits of `n` are used in the operation and the remaining bits are ignored.

## pointer-set-c-short! **procedure**

```
(pointer-set-c-short! p i n)
```

The procedure `pointer-set-c-char!!` sets two bytes located at offset `i` and `(+ i 1)` to the 16 lowermost bits of the exact integer `n`. Note that the offset `i` is a byte offset; `pointer-set-c-short!` does not perform any pointer arithmetic such as scaling the offset by the size of the memory location.

## pointer-set-c-int! **procedure**

```
(pointer-set-c-int! p i n)
```

The procedure `pointer-set-c-int!` sets four bytes located at offset `i` to `(+ i 3)` to the 32 lowermost bits of the exact integer `n`. Like `pointer-set-c-short!`, `pointer-set-c-int!` does not scale the offset `i`.

## pointer-set-c-long! **procedure**

```
(pointer-set-c-long! p i n)
```

On 64-bit systems, the procedure `pointer-set-c-long!` sets eight bytes located at offset `i` to `(+ i 7)` to the 64 lowermost bits of the exact integer `n`. Like the previous procedures, `pointer-set-c-long!` does not scale the offset `i`. On 32-bit systems, `pointer-set-c-long!` performs the same task as `pointer-set-c-int!`.

## pointer-set-c-float! **procedure**

```
(pointer-set-c-float! p i fl)
```

The procedure `pointer-set-c-float!` converts the Scheme floating point number `fl` (represented in Ikarus as an IEEE-754 double precision floating point number) to a float (an IEEE-754 single precision floating point number) and stores the result in the four bytes at offset `i` of the pointer `p`.

---

## pointer-set-c-double! **procedure**

`(pointer-set-c-double! p i fl)`

The procedure `pointer-set-c-double!` stores the double precision IEEE-754 floating point value of the Scheme flonum `fl` in the eight bytes at offset `i` of the pointer `p`.

---

## pointer-set-c-pointer! **procedure**

`(pointer-set-c-pointer! p i pv)`

On 64-bit systems, the procedure `pointer-set-c-pointer!` sets eight bytes located at offset `i` to `(+ i 7)` to the 64-bit pointer value of `pv`. On 32-bit systems, the procedure `pointer-set-c-pointer!` sets four bytes located at offset `i` to `(+ i 3)` to the 32-bit pointer value of `pv`. Like the previous procedures, `pointer-set-c-pointer!` does not scale the offset `i`.

---

## pointer-ref-c-signed-char **procedure**

`(pointer-ref-c-signed-char p i)`

The procedure `pointer-ref-c-signed-char` loads a single byte located at offset `i` from the pointer `p` and returns an exact integer representing the sign-extended integer value of that byte. The resulting value is in the range of $[-128, 127]$ inclusive.

---

## pointer-ref-c-unsigned-char **procedure**

`(pointer-ref-c-unsigned-char p i)`

The procedure `pointer-ref-c-unsigned-char` loads a single byte located at offset `i` from the pointer `p` and returns an exact integer representing the unsigned integer

value of that byte. The resulting value is in the range $[0, 255]$ inclusive.

The following example shows the difference between `pointer-ref-c-signed-char` and `pointer-ref-c-unsigned-char`.

```
> (let ([p (malloc 3)])
    (pointer-set-c-char! p 0 #b01111111)
    (pointer-set-c-char! p 1 #b10000000)
    (pointer-set-c-char! p 2 #b11111111)
    (let ([result
           (list (pointer-ref-c-signed-char p 0)
                 (pointer-ref-c-signed-char p 1)
                 (pointer-ref-c-signed-char p 2)
                 (pointer-ref-c-unsigned-char p 0)
                 (pointer-ref-c-unsigned-char p 1)
                 (pointer-ref-c-unsigned-char p 2))])
      (free p)
      result))
(127 -128 -1 127 128 255)
```

## pointer-ref-c-signed-short **procedure**

`(pointer-ref-c-signed-short p i)`

The procedure `pointer-ref-c-signed-short` loads two bytes located at offsets `i` and `(+ i 1)` from the pointer `p` and returns an exact integer representing the sign-extended integer value of the sequence. The resulting value is in the range $[-32768, 32767]$ inclusive.

## pointer-ref-c-unsigned-short **procedure**

`(pointer-ref-c-unsigned-short p i)`

The procedure `pointer-ref-c-unsigned-short` loads two bytes located at offsets `i` and `(+ i 1)` from the pointer `p` and returns an exact integer representing the unsigned integer value of the sequence. The resulting value is in the range $[0, 65535]$ inclusive.

## pointer-ref-c-signed-int **procedure**

(pointer-ref-c-signed-int p i)

The procedure pointer-ref-c-signed-int loads four bytes starting at offset i of pointer p and returns an exact integer in the range of $[-2^{31}, 2^{31} - 1]$ inclusive.

## pointer-ref-c-unsigned-int **procedure**

(pointer-ref-c-unsigned-int p i)

The procedure pointer-ref-c-unsigned-int loads four bytes starting at offset i of pointer p and returns an exact integer in the range of $[0, 2^{32} - 1]$ inclusive.

## pointer-ref-c-signed-long **procedure**

(pointer-ref-c-signed-long p i)

On 64-bit systems, the procedure pointer-ref-c-signed-long loads eight bytes starting at offset i of pointer p and returns an integer in the range of $[-2^{63}, 2^{63} - 1]$ inclusive. On 32-bit systems, the procedure pointer-ref-c-signed-long performs the same task as pointer-ref-c-signed-int.

## pointer-ref-c-unsigned-long **procedure**

(pointer-ref-c-unsigned-long p i)

On 64-bit systems, the procedure pointer-ref-c-unsigned-long loads eight bytes starting at offset i of pointer p and returns an integer in the range of $[0, 2^{64} - 1]$ inclusive. On 32-bit systems, the procedure pointer-ref-c-unsigned-long performs the same task as pointer-ref-c-unsigned-int.

## pointer-ref-c-float **procedure**

(pointer-ref-c-float p i)

The procedure pointer-ref-c-float returns the four-byte float (represented as IEEE-754 single precision floating point number) stored at offset i of the pointer p. The value is extended to an IEEE-754 double precision floating point number that

Ikarus uses to represent inexact numbers.

---

### pointer-ref-c-double **procedure**

```
(pointer-ref-c-double p i)
```

The procedure `pointer-ref-c-double` returns the eight-byte float (represented as IEEE-754 double precision floating point number) stored at offset `i` of the pointer `p`.

---

### pointer-ref-c-pointer **procedure**

```
(pointer-ref-c-pointer p i)
```

The procedure `pointer-ref-c-pointer` returns the pointer stored at offset `i` from the pointer `p`. The size of the pointer (also the number of bytes loaded) depends on the architecture: it is 4 bytes on 32-bit systems and 8 bytes on 64-bit systems.

## 5.4 Accessing foreign objects from Scheme

---

### dlopen **procedure**

```
(dlopen)
(dlopen library-name)
(dlopen library-name lazy? global?)
```

The procedure `dlopen` takes a string `library-name` represented a system library and calls the system procedure `dlopen` which dynamically loads the given library into the running process. The name of the library is system-dependent and must include the appropriate suffix (e.g., `*.so` on Linux, `*.dylib` on Darwin and `*.dll` on Cygwin). The `library-name` may include a full path which identifies the location of the library, or may be just the name of the library in which case the system will lookup the library name using the `LD_LIBRARY_PATH` environment variable.

The argument `lazy?` specifies how library dependencies are loaded. If true, `dlopen` delays the resolution and loading of dependent libraries until they are actually used.

If false, all library dependencies are loaded before the call to dlopen returns.

The argument global? specifies how the scope of the symbols exported from the loaded library. If true, all exported symbols become part of the running image, and subsequent dlsym calls may not need to specify the library from which the symbol is loaded. If false, the exported symbols are not global and the library pointer needs to be specified for dlsym.

Calling (dlopen library-name) is equivalent to (dlopen library-name #f #f). Calling (dlopen) without arguments returns a pointer to the current process.

If succesful, dlopen returns a pointer to the external library which can be used subsequently by dlsym and dlclose. If the library cannot be loaded, dlopen returns #f and the procedure dlerror can be used to obtain the cause of the failure.

Consult the dlopen(3) page in your system manual for further details.

---

## dlclose **procedure**

```
(dlclose library-pointer)
```

The procedure dlclose is a wrapped around the system procedure with the same name. It receives a library pointer (e.g., one obtained from dlopen) and releases the resources loaded from that library. Closing a library renders all symbols and static data structures that the library exports invalid and the program may crash or corrupt its memory if such symbols are used after a library is closed.

Most system implementations of dynamic loading employ reference counting for dlopen and dlclose in that library resources are not freed until the number of calls to dlclose matches the number of calls to dlopen.

The procedure dlclose returns a boolean value indicating whether the success status of the operation. If dlclose returns #f, the procedure dlerror can be used to obtain the cause of the error.

Consult the dlclose(3) page in your system manual for further details.

---

## dlerror **procedure**

```
(dlerror)
```

If any of the dynamic loading operations (i.e., `dlopen`, `dlclose`, `dlsym`) fails, the cause of the error can be obtained by calling `dlerror` which returns a string describing the error. The procedure `dlerror` returns `#f` if there was no dynamic loading error.

Consult the `dlerror(3)` page in your system manual for further details.

---

`dlsym` **procedure**

`(dlsym library-pointer string)`

The procedure `dlsym` takes a library pointer (e.g., one obtained by a call to `dlopen`) and a string representing the name of a symbol that the library exports and returns a pointer to the location of that symbol in memory. If `dlsym` fails, it returns `#f` and the cause of the error can be obtained using the procedure `dlerror`.

Consult the `dlsym(3)` page in your system manual for further details.

## 5.5 Calling out to foreign procedures

Ikarus provides the means to call out from Scheme to foreign procedures. This allows the programmers to extend Ikarus to access system-specific facilities that is available on the host machine.

In order to call out to a foreign procedure, one must provide two pieces of information: the signature of the foreign procedure (e.g., its type declaration if it is a `C` procedure) and the address of the procedure in memory. The address of the procedure can be easily obtained using `dlsym` if the name of the procedure and its exporting library are known. The signature of the procedure cannot, in general, be obtained dynamically, and therefore must be hard coded into the program.

The signature of the foreign procedure is required for proper linkeage between the Scheme system and the foreign system. Using the signature, Ikarus determines how Scheme values are converted into native values, and where (e.g., in which registers and stack slots) to put these arguments. The signature also determines where the returned values are placed and how they are converted from the system data types to the corresponding Scheme data types.

A procedure's signature is composed of two parts: the return type and the parameter types. The return type is a symbol that can be any one of the type specifiers listed in Figure 5.1, page 71. The parameter types is a list of type specifier symbols. The symbol `void` can appear as a return type but cannot appear as a parameter type.

---

### `make-c-callout` **procedure**

`((make-c-callout return-type parameter-types) native-pointer)`

The procedure `make-c-callout` is the primary facility for making foreign procedures callable from Scheme. It works as follows. First, `make-c-callout` receives two arguments denoting the signature of the procedure to be called. It prepares a bridge that converts from Scheme's calling conventions and data structures to their foreign counterparts. It returns a procedure $p_1$. Second, the procedure $p_1$ accepts a pointer to a foreign procedure (e.g., one obtained from `dlsym`) and returns a Scheme procedure $p_2$ that encapsulates the foreign procedure. The final procedure $p_2$ can be called with as many arguments as the ones specified in the `parameter-types`. The parameters supplies to $p_2$ must match the types supplied as the `parameter-types` according to the "Valid Scheme types" column in the table in Figure 5.1. The procedure $p_2$ converts the parameters from Scheme types to native types, calls the foreign procedure, obtains the result, and converts it to the appropriate Scheme value (depending on the `return-type`).

The interface of `make-c-callout` is broken down into three stages in order to accomodate common usage patterns. Often types, a function signature can be used by many foreign procedures and therefore, `make-c-callout` can be called once per signature and each signature can be used multiple times. Similarly, separating the foreign procedure preparation from parameter passing allows for preparing the foreign procedure once and calling it many times.

The types listed in Figure 5.1 are restricted to basic types and provide no automatic conversion from composite Scheme data structures (such as strings, symbols, vectors, and lists) to native types. The restriction is intentional in order for Ikarus to avoid making invalid assumptions about the memory management of the targeted library. For example, while Ikarus *can* convert a Scheme string to a native byte array (e.g., using `string->bytevector` to decode the string, then using `malloc` to allocate a temporary buffer, and copying the bytes from the bytevector to the allocated memory), it cannot decide when this allocated byte array is no longer needed

and should be freed. This knowledge is library-dependent and is often procedure-dependent. Therefore, Ikarus leaves it to the programmer to manage all memory related issues.

Outgoing parameters to foreign procedures are checked against the declared types. For example, if a callback is prepared to expect a parameter of type `signed-int`, only exact integers are allowed to be passed out. For integer types, only a fixed number of bits is used and the remaining bits are ignored. For floating point types, the argument is checked to be a Scheme flonum. No implicit conversion between exact and inexact numbers is performed.

The following example illustrates the use of the `make-c-callout` procedure in combination with `dlopen` and `dlsym`. The session was run on a 32-bit Ikarus running under Mac OS X 10.4. First, the `libc.dylib` foreign library is loaded and is bound to the variable `libc`. Next, we obtain a pointer to the `atan` foreign procedure that is defined in `libc`. The native procedure `atan` takes a `double` as an argument and re-

| Type specifier | Size | Valid Scheme types | Corresponding `C` types |
|---|---|---|---|
| `signed-char` | 1 byte | exact integer | `char` |
| `unsigned-char` | 1 byte | exact integer | `unsigned char` |
| `signed-short` | 2 bytes | exact integer | `short` |
| `unsigned-short` | 2 bytes | exact integer | `unsigned short` |
| `signed-int` | 4 bytes | exact integer | `int` |
| `unsigned-int` | 4 bytes | exact integer | `unsigned int` |
| `signed-long` | 4/8 bytes | exact integer | `long` |
| `unsigned-long` | 4/8 bytes | exact integer | `unsigned long` |
| `float` | 4 bytes | flonum | `float` |
| `double` | 8 bytes | flonum | `double` |
| `pointer` | 4/8 bytes | pointer | `void*`, `char*`, `int*`, `int**`, `int(*)(int,int,int)`, etc. |
| `void` | – | – | `void` |

Figure 5.1: The above table lists valid type specifiers that can be used in callout and callback signatures. Specifiers with “4/8 bytes” have size that depends on the system: it is 4 bytes on 32-bit systems and 8 bytes on 64-bit systems. The `void` specifier can only be used as a return value specifier to mean “no useful value is returned”.

turns a double and that's the signature that we use for make-c-callout. Finally, we call the foreign procedure interface with one argument, 1.0, which is a flonum and thus matches the required parameter type. The native procedure returns a double value which is converted to the Scheme flonum with value 0.7853981633974483.

```
> (import (ikarus foreign))
> (define libc (dlopen "libc.dylib"))
> libc
#<pointer #x00100770>
> (define libc-atan-ptr (dlsym libc "atan"))
> libc-atan-ptr
#<pointer #x9006CB1F>
> (define libc-atan
    ((make-c-callout 'double '(double)) libc-atan-ptr))
> libc-atan
#<procedure>
> (libc-atan 1.0)
0.7853981633974483
> (libc-atan 1)
Unhandled exception
 Condition components:
    1. &assertion
    2. &who: callout-procedure
    3. &message: "argument does not match type double"
    4. &irritants: (1)
```

## 5.6 Calling back to Scheme

In order to provide full interoperability with native procedures, Ikarus allows native procedures to call back into Scheme just as it allows Scheme to call out to native procedures. This is important for many system libraries that provide graphical user interfaces with event handling (e.g., Cocoa, GTK+, GLUT, etc.), database engines (e.g., libsqlite, libmysql, etc.), among others.

The native calling site for the call back is compiled with a specific callback signature encoding the expected parameter types and return type. Therefore, a Scheme

procedure used for a call back must be wrapped with a proper adapter that converts the incoming parameters from native format to Scheme values as well as convert the value that the Scheme procedure returns back to native format. The signature format is similar to the one used for call outs (see Figure 5.1 on page 71 for details).

---

`make-c-callback` **procedure**

`((make-c-callback return-type parameter-types) scheme-procedure)`

The procedure `make-c-callback` is similar to the procedure `make-c-callout` except that it provides a bridge from native procedures back into Scheme. While the procedure `make-c-callout` takes a native pointer and returns a Scheme procedure, `make-c-callback` takes a Scheme procedure and returns a native pointer. The native pointer can be called by foreign procedures. The native parameters are converted to Scheme data (according to `parameter-types`), the Scheme procedure is called with these parameters, and the returned value is converted back into native format (according to `return-type`) before control returns to the native call site.

Note that the native procedure pointer obtained from `make-c-callback` is indistinguishable from other native procedures that are obtained using `dlsym` or similar means. In particular, such native pointers can be passed to `make-c-callout` resulting in a Scheme procedure that calls out to the native procedure that in turn calls back into Scheme. The following segment illustrates a very inefficient way of extracting the lowermost 32 bits from an exact integer.

```
> (format "~x"
    (((make-c-callout 'unsigned-int '(unsigned-int))
      ((make-c-callback 'unsigned-int '(unsigned-int))
       values))
     #xfedcba09876543210fedcba09876543210))
"76543210"
```

> *Caveat emptor:* Preparing each call out and call back procedure leaks a small amount of memory. This is because the system cannot track such pointers that go into native code (which may retain such pointers indefinitely). Use judiciously.

# Chapter 6

# Missing Features

Ikarus does not fully conform to R$^6$RS yet. Although it implements most of R$^6$RS's macros and procedures, some are still missing. This section summarizes the set of missing features and procedures.

- `number->string` does not accept the third argument (precision). Similarly, `string->number` and the reader do not recognize the `|p` notation.
- The following procedures are missing from `(rnrs arithmetic bitwise)`:

  ```
  bitwise-reverse-bit-field bitwise-rotate-bit-field
  ```

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

  ```
  fxreverse-bit-field   fxrotate-bit-field
  ```

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

  ```
  equal-hash
  ```

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

  ```
  make-custom-binary-input/output-port
  make-custom-textual-input/output-port
  open-file-input/output-port
  ```