Inlab Scheme Reference Manual Release 4.108

Thomas Obermair

Inlab Software GmbH

ABSTRACT

This is the reference manual of Inlab-Scheme. Inlab-Scheme is an independent implementation of the Algorithmic Language Scheme as defined by the "Revisedˆ4 Report on the Algorithmic Language Scheme" and the IEEE Standard 1178. Inlab-Scheme has in addition to the language core support for image processing of several kinds. Using the provided procedures Inlab-Scheme can be used to perform tasks such as OCR, image processing and specialized optical ’object’ recognition.

Contact Information:

Inlab Software GmbH
Josef-Wuerth-Str. 3
82031 Gruenwald
Germany
Fax: +49 89 6411160
Email: obermair@acm.org

April 29, 2006

1. Overview

Inlab-Scheme is an independent implementation of the Algorithmic Language Scheme as defined by the "Revisedˆ4 Report on the Algorithmic Language Scheme" and the IEEE Standard 1178. Inlab-Scheme has in addition to the language core support for image processing of several kinds. Using the provided procedures Inlab-Scheme can be used to perform tasks such as OCR, image processing and specialized optical 'object' recognition.

Inlab-Scheme and this manual is available at our website at <http://www.inlab.de> where additional information is available.

This Inlab-Scheme reference manual is divided in several chapters following this overview. Chapter 2 contains general information how to use Inlab-Scheme. Chapter 3 contains collected information about the data types that Inlab-Scheme uses. Chapter 4 contains the collection of all procedures and important variables/constants of Inlab-Scheme in alphabetical order.

If you find any error or if you would like to suggest something we appreciate if you tell us about it, preferably using email.

2. General Usage

2.1 Starting Inlab-Scheme

Inlab-Scheme can be started in four different ways: in interactive mode, with a file name argument, with an expression as argument and in script mode. In each mode an existing file *.schemerc* in the user's home directory (detected by the standard environment variable HOME) is loaded first after initialization. If an error occurs loading *.schemerc* Inlab-Scheme displays the error message and exits with the return code 2 (the error handler of interactive mode is not invoked).

2.1.1 Starting Inlab-Scheme in interactive mode

This is the the simple case of just typing *scheme* in your shell. Inlab-Scheme starts and initializes, prints its welcome and copyright message and starts the Inlab-Scheme top level REP (read-eval-print) loop. For further information using the top level REP loop see the chapter "Using the Top Level REP Loop" below.

The only thing that you may have to configure in your shell environment is to make sure that the binary of Inlab-Scheme is found by your shell usually by configuring your PATH environment variable. If an error occurs in interactive mode the error handler is invoked (see below).

Example (ksh):

```
$ scheme
Inlab-Scheme 4.xx
Copyright (c) ...
[1] (exit)
$
```

2.1.2 Starting Inlab-Scheme with a file name argument

If Inlab-Scheme is started with one argument supplied and this argument is regarded by *(read)* as a symbol this symbol (converted to a string) is treated as a file name. Inlab-Scheme is initialized, the Inlab-Scheme top level REP loop is not started and the filename is loaded using *(load)*. If the loading succeeds by evaluating each expression found in that file and no more expressions are found therein Inlab-Scheme internally executes an *(exit 0)* returning to the command level of the operating system (shell). If Inlab-Scheme is started in this mode no initial welcome and copyright message is displayed.

If an error occurs during load an error message is printed on *stderr* and Inlab-Scheme exits with the return code 2.

Example:

```
$ echo '(writeln "hello")' > try1.scm
$ scheme try1.scm
hello
$ echo $?
```

```
0
$
```

2.1.3 Starting Inlab-Scheme with an expression argument

If Inlab-Scheme is started with one argument supplied and this argument would be internalized to a list by *(read)* this one list is treated as an expression that is evaluated after initializing. No welcome and copyright message is displayed in this case. If the evaluation of this single expression does not explicitly cause Inlab-Scheme to exit internally an *(exit 0)* is performed after evaluating this expression.

In case of an error an error message is displayed on *stderr* and Inlab-Scheme exits with a return code of 2.

This mode may be used to use Inlab-Scheme to execute smaller programs that are supplied directly via the command line interface.

Example:

```
$ scheme '(writeln "hello")'
hello
$ echo $?
0
$
```

2.1.4 Running Inlab-Scheme scripts

Putting the path of Inlab-Scheme in the first line of a Scheme-Source preceded by *#!* enables the use of the program invocation in most standard shells. Internally we had to achieve that the evaluation of this first line is a valid scheme expression which is causing no error. This has been done by allowing the use of symbols in Inlab-Scheme that are preceded by the characters "#" and "!" and by predefining several locations (as a symbol) where a binary of Inlab-Scheme may reside to make that feature work.

To use this mechanism your Inlab-Scheme binary must be copied to one of the following locations:

```
/usr/bin/scheme
/bin/scheme
/sbin/scheme
/usr/sbin/scheme
/opt/bin/scheme
/usr/local/scheme
/usr/local/bin/scheme
```

Example (assuming a Inlab-Scheme binary on */usr/local/scheme*):

```
$ echo '#!/usr/local/scheme' > try2
$ echo '(writeln "hello")' >> try2
$ chmod +x try2
$ try2
hello
$ echo $?
0
$
```

2.2 Using the Top Level REP Loop

Inlab-Scheme provides a simple top level REP (read-eval-print) loop that can be controlled using several procedures. Generally each expression you enter interactively has an "expression number". That expression number is visualized in the Inlab-Scheme prompt enclosed in square brackets.

Expressions that you are entering interactively are stored in a expression history which holds the last 20 expressions you have entered.

The procedure *?* lists the current expression history. You can repeat a former entered expression / command using the procedure *!!*.

The procedure *!* allows you to edit an former entered expression before executing it again. The variable *the-current-editor* holds the name of the started editor in that case (vi per default). You may change that variable in your *.schemerc* file.

Example:

```
$ scheme
Inlab-Scheme 4.xx
```

```
Copyright (c) ...
[1] (define a 12)
a
[2] (define b 13)
b
[3] (?)
  [1]: (define a 12)
  [2]: (define b 13)
  [3]: (?)
ok
[4]
```

2.3 Top Level Error Handling

Inlab-Scheme in its interactive mode uses a simple error handler to display and further analyze an error. The interactive error handler displays the error message, an irritant (which caused the error in most cases) and the current content of the *expr* register of the virtual machine. Some errors will allow you to continue the flow of execution. If that is the case you will be informed.

Then you are in the error-handler indicated by a change of the prompt. Here you are able to enter a few commands. Just typing a EOF (usually Ctrl-D) or entering the command "r" returns you directly to the main read-eval-print loop.

The following commands may be entered in that mode:

```
  ? h   this help-screen
  ee    eval in error-environment
  eg    eval in global-environment
  ce    continue with value (evaluated
        in error-environment)
  cg    continue with value (evaluated
        in global-environment)
  i     continue and return '()
  d     (re-) display initial error-message
  fd    display current environment-frame
  fu    move one environment-frame
        up to the caller
  fi    restore initial environment-frame
  r     (or EOF) reset to top-level
  x     exit immediately
```

If you are evaluating an expression using the commands "ee" or "eg" a nested "sub"-error handler is started if the entered expression is causing an error again.

Displaying the environment frames may sometimes be not very informative because of the proper tail recursive implementation of Inlab-Scheme.

The current error handling of Inlab-Scheme lacks of several advanced features (like a "stack trace") and will be improved in future releases.

The following example demonstrates an error caused by an undefi ned variable and the use and continuation of program flow using the error handler:

```
[1] (define (test input)
      (let ((aa 12))
         (+ aaa input)))
test
[2] (test 12)
ERROR!
  message   : variable not found
  irritant  : aaa
  expression: aaa
  continue  : possible, continue with
              value as obtained from
              variable
Error (? for help) >> fd
aa=12
Error (? for help) >> cg
enter value to be returned: '12
24
[3]
```

2.4 Interrupting Inlab-Scheme

A running program may be interrupted by SIGINT (signal 2) which causes a BREAK error condition. This

may be done by pressing the current *intr* character on your terminal (usually Ctrl-C or sometimes DEL).

See also *disable-interrupt* and *enable-interrupt*.

2.5 Quitting Inlab-Scheme

Inlab-Scheme exits generally by calling *(exit)* returning the return code 0 to the invoking shell. You may supply explicitly a return code e.g. *(exit 3)*. If you are using the interactive mode of Inlab-Scheme you may also exit by typing the EOF character of your terminal twice (usually Ctrl-D). If Inlab-Scheme is called with one argument (an expression or an file name) and Inlab-Scheme is not forced to exit explicitly, an implicit *(exit 0)* is performed at the end of evaluating the expression or at the end of loading the supplied file.

2.6 Customizing Inlab-Scheme

Inlab Scheme allocates several different memory areas when started. The allocation of memory can be controlled and customized using ordinary environment variables. The current state of Inlab-Scheme can be displayed using the procedure *(stat)*. Please see the description in the "Expressions and Procedures" chapter below.

The following table shows the variables that can be set together with the minimum and default settings:

```
                     Min     Default
SCHEME_PSTACKSIZE    4096    16K
SCHEME_CSTACKSIZE    4096    16K
SCHEME_HASHSIZE      4096    4096
SCHEME_HEAPSIZE      2M      4M
SCHEME_AUXMEMSIZE    1M      16M
```

Inlab Scheme uses internally a two stack model. One stack is used to hold pointers to ordinary Scheme objects (that must be tracked during gc), the other is used to hold the internal continuations of the interpreter. SCHEME_PSTACKSIZE refers to the stack holding Scheme objects, SCHEME_CSTACKSIZE to the continuation stack.

The hashsize SCHEME_HASHSIZE is used to determine the size of the hash table for symbols and should be (only) increased if you are running programs with a extremely large amount of symbols.

The SCHEME_HEAPSIZE determines the total size of the heap in bytes. Please note that on a 64-bit machine the default-value of 4M means actually half the number of pointers that can be hold in the heap compared to a 32-bit machine. Internally the heap is divided into two parts and is collected using a traditional stop and copy algorithm.

The SCHEME_AUXMEMSIZE environment variable controls the size of the auxiliary memory that will trigger a garbage collection with the hope to reclaim auxiliary memory. Auxiliary memory is not allocated in advance, SCHEME_AUXMEMSIZE specifies only the amount of auxiliary memory that will trigger a GC. The auxiliary memory is currently used for memory containing the actual contents of images (that may be very large).

The values of the variables can be abbreviated using 'K' or 'M' as the last character of the value indicating multiplication with 1024 or 1048576 respectively.

The following shows an example dialog using a Bourne shell type environment which increases the size of the heap to 16 megabytes:

```
$ export SCHEME_HEAPSIZE=16M
$ scheme
  ...
```

A further customization possibility is the use of the file *.schemerc* in your home directory. If this file exists and Inlab-Scheme is started this file is loaded using *load* in every mode.

3. Data Types

Inlab-Scheme is implemented using a "pure indirect memory model" what actually means that every Inlab-Scheme object is really a pointer to a tagged object.

3.1 Constants

Pointer to constants are pointing to a statically memory area which identifies each individual constant. In Inlab-Scheme the following constants exist:

3.1.1 #t

The boolean constant "true".

3.1.2 #f

The boolean constant false, which is the only object that counts as false in conditional expressions like *if* and *cond*.

3.1.3 #u

The "unassigned constant" which may be bound to a variable. If a variable bound to this constant is used an error is signaled. This constant is used to detect some errors in derived expressions.

3.1.4 ()

The empty list *()* is actually implemented as a constant and distinct from *#f*.

3.2 Symbols

Symbols are parsed (internalized) by *read* in lower case. Symbols have a name that has all properties of a string. It is guaranteed that a symbol with the same name is always the same in the sense of *eq?*.

3.3 Numbers

Inlab-Scheme support inexact reals (floating points) and exact integers as numeric data types.

3.3.1 Exact Integers

Exact Integers can be as large as the native word size of the machine on which Inlab-Scheme is compiled on (32 bit or 64 bit signed). Internally some static space is allocated for "often used" integers, integers lying outside the "often used" range are allocated dynamically. This actually means that *eq?* on integers may fail, even is two integers contain the same value. You should - as always - use *eqv?* to compare integer values for equality.

3.3.2 Inexact Reals

Inexact real numeric values are implemented using floating point standard arithmetic as provided by the host machine.

3.4 Characters

Inlab-Scheme supports numeric character values from 0 to 255. Characters are static allocated objects what guarantees *eq?* to return *#t* for the same character.

As specified in the R4RS the following special character names are supported:

```
#\space
#\newline
```

A "#\" followed by a 3 digit decimal value with leading zeroes may be used to specify each possible value from 0 to 255.

Example:

```
[1] #\048
#\0
[2] #\078
#\N
```

Characters are self evaluating. As specified in the R4RS case is significant in *#<character>* but not in *#<character name>*.

3.5 Strings

Strings are kept in the main heap and are dynamically allocated. Strings may contain any sequence of 8 bit characters (including 0), which allows storing any binary data using strings. To enter any possible 8 bit character value a followed by a three digit decimal value in the range 0 ... 255 may be used. A special character that cannot be represented another way is also printed using that notation.

Example:

```
[1] "abc\\def"
"abc\\def"
[2] "\078"
"N"
```

Although the length of a string internally has no special limit (depends on the current heap size and the architecture of the machine) there is a limit in *(read)* which limits the length of a token to a maximum of 10KB. This actually means that a single string constant (e.g. in a program) larger than 10KB will cause a parsing error in Inlab-Scheme.

3.6 Lists

Lists are implemented the usual way using "cons cells", there is no special to tell about them.

3.7 Vectors

Vectors are implemented as specifi ed by the R4RS as a "heterogeneous structure" allocated in heap. Vectors are not self-evaluating and an error is signaled (artifi cially) when a vector is about to be evaluated.

3.8 Ports

The port concept of R4RS is extended by string-input-ports, string-output-ports, Unix popen-ports (reading and writing) and by providing access to UNIX *stderr* via *current-error-port*.

3.9 Procedures

Three types of procedures are implemented: Primitive procedures (implemented in C), compound procedures (written in Scheme) and syntax procedures which for the low level macro facility of Inlab-Scheme (also written in Scheme).

3.9.1 Primitive Procedures

Primitive procedures have a name associated with it. Primitive procedures are printed as *#<primitive-procedure <name>>*.

Example:

```
[1] car
#<primitive-procedure car>
[2] (define a car)
a
[3] a
#<primitive-procedure car>
```

3.9.2 Compound Procedures

Compound procedures may have a name associated with it for debugging purposes. Compound procedures with a name may be generated using *named-lambda* or using *(define (<name> <args>) <body>)*.

Compound procedures with a name are printed as *#<compound-procedure <name>>*, anonymous compound procedures (as returned by *lambda*) as *#<compound-procedure>*.

Example:

```
[1] (define (plus1 n)
      (+ n 1))
plus1
[2] plus1
#<compound-procedure plus1>
[3] (lambda (n) (+ n 1))
#<compound-procedure>
[4]
```

3.9.3 Syntax Procedures

Inlab-Scheme implements its macro facility using syntax procedures, which are first class objects as ordinary compound procedures are.

A similar naming scheme to compound procedures is implemented by *syntax-lambda* and *named-syntax-lambda*.

3.10 Graphical Data Types

3.10.1 Bitmaps

A bitmap is a data structure that allows to store bilevel (B/W) images in a compact form in memory. Each line (row) is stored occupying one bit per pixel and is filled up to a byte boundary at the end.

A 0 represents a white, a 1 a black pixel. The origin of a bitmap (0,0) is in the upper left. Addressing generally follows a row/column scheme (row as the first, column as the second parameter).

Reading and writing bitmap to/from the following graphic formats on disk is supported: TIFF G4, TIFF G4 multi-page, XBM and PNG.

The internal structure of a bitmap has the following fields:

h the actual height of the bitmap

w the actual width of the bitmap

orow the original row position of that bitmap in another bitmap, -1 if undefined

ocol the original column position of that bitmap in another bitmap, -1 if undefined

xres the X resolution in DPI (floating point value), -1 if undefined

yres the Y resolution in DPI (floating point value), -1 if undefined

The contents of these fields are accessible / settable using the appropriate procedures. Printing a bitmap shows the actual contents of these fields.

Example:

```
[1] (define a
      (bitmap-create 100 100))
a
[2] a
#<bitmap orow:-1 ocol:-1 xres:-1 yres:-1
         h:100 w:100 b:13>
[3]
```

3.10.2 Greymaps

A greymap is a data structure that allows the storage of greylevel images in memory. One byte (256 greylevels, 8 bit depth) is allocated for each pixel.

0 represents white, 255 a black pixel, all grey levels are in between. The origin of a greymap (0,0) is in the upper left. Generally the same row/column addressing scheme as being used for bitmaps is used.

Reading greymaps from any PNG file is supported (performing a suitable conversion if necessary) , greymaps may be written in greylevel format either interlaced or not.

The internal structure of a greymap has the following fields (similar to a bitmap):

h the actual height of the greymap

w the actual width of the greymap

orow the original row position of that greymap in another greymap, -1 if undefined

ocol the original column position of that greymap in another greymap, -1 if undefined

xres the X resolution in DPI (fbating point value), -1 if undefi ned

yres the Y resolution in DPI (fbating point value), -1 if undefi ned

The contents of these fi elds are accessible / settable using the appropriate procedures. Printing a greymap shows the actual contents of these fi elds.

Example:

```
[1] (define a
      (greymap-create 100 100))
a
[2] a
#<greymap orow:-1 ocol:-1 xres:-1 yres:-1
         h:100 w:100>
[3]
```

4. Expressions and Procedures

In this chapter all procedures are listed alphabetically sorted. a *(+)* following the name of the procedure indicates a non standard extension of Inlab-Scheme, a *(!)* indicates an compatible extension to a standard procedure or an important note to the implementation of the procedure in Inlab-Scheme (sometimes it indicates that a particular procedure isn't implemented at all).

The R4RS is referenced very often and you will have to have it handy to fi nd the actual description of many (standard) procedures. We fi nally decided to do so in this cases instead of just copying the content of the R4RS.

4.1 ! (+)

This procedure allows either to edit and re-execute the last expression entered in interactive mode (entering *(!)*) or a selected expression referenced by its number (entering e.g. *(! 2)*). The editor in the variable *the-current-editor* is used to edit the expression.

4.2 !! (+)

(!!) repeats the previous entered expression by re-evaluating it in interactive mode of Inlab-Scheme and returns the result of this evaluation.

(!! <expression-no>) repeats the previous entered expression with the given expression number by reevaluating it and returns the result of the evaluation.

4.3 #f

#f is the boolean constant "false" which is the only value that counts as false in conditional expressions.

#f is not the same as *()*, the empty list.

4.4 #t

#t is the boolean constant "true".

*4.5 **

(<z1> ...)* returns the product of its arguments. No argument may be supplied, in that case + will return *1*. If any argument is inexact, then the result will also be inexact.

4.6 +

(+ <z1> ...) returns the sum of its arguments. No argument may be supplied, in that case + will return *0*. If any argument is inexact, then the result will also be inexact.

4.7 -

(- <z1> ...) given two or more arguments - returns the difference of its arguments, associating to the left. Given one argument - returns the "additive inverse" of its argument.

4.8 /

(/ <z1> ...) given two or more arguments / returns the quotient of its arguments, associating to the left. Given one argument / returns the "multiplicative inverse" of its argument.

4.9 =

(= <x1> <x2> <x3> ...) returns *#t* if the arguments are equal, *#f* otherwise. Using it on inexact arguments may be unreliable because a small inaccuracy may affect the result. At least 2 parameters are required.

4.10 ? (+)

This procedure is used with no arguments only in interactive mode to display the content of the current history. The history is keeping the last 20 expressions entered in interactive mode.

4.11 <

(< <x1> <x2> <x3> ...) returns *#t* if the arguments are monotonically increasing, *#f* otherwise. Using it on inexact arguments may be unreliable because a small inaccuracy may affect the result. At least 2 parameters are required.

4.12 >

(> <x1> <x2> <x3> ...) returns *#t* if the arguments are monotonically decreasing, *#f* otherwise. Using it on inexact arguments may be unreliable because a small inaccuracy may affect the result. At least 2 parameters are required.

4.13 <=

(<= <x1> <x2> <x3> ...) returns *#t* if the arguments are monotonically nondecreasing, *#f* otherwise. Using it on inexact arguments may be unreliable because a small inaccuracy may affect the result. At least 2 parameters are required.

4.14 >=

(>= <x1> <x2> <x3> ...) returns *#t* if the arguments are monotonically nonincreasing, *#f* otherwise. Using it on inexact arguments may be unreliable because a small inaccuracy may affect the result. At least 2 parameters are required.

4.15 <variable>

An expression consisting of a variable is a variable reference. The value of of the variable reference is the value stored in the location to which the variable is bound. It is an error to reference an unbound variable.

4.16 abs

(abs <x>) returns the magnitude of its argument.

4.17 acos

(acos <z>) computes the principal value of the arc cosine of *<z>*. A domain error occurs for arguments not in the range [-1, +1]. The arc cosine in the range [0, pi] radians is returned.

4.18 and

(and <test1> ...) is implemented as described in the R4RS as a syntax procedure.

4.19 angle (!)

angle is not implemented in Inlab-Scheme.

4.20 append

(append <list> ...) returns a list consisting of the elements of the first list followed by the elements of the other lists. The resulting list is always newly allocated, except that it shares structure with the last list argument. The last list argument may actually be any object; an improper list results if the last argument is not a proper list.

4.21 apply

(apply <proc> <args>) calls *<proc>* with the elements of *<args>* (a list) as the actual arguments and returns the result of the call.

(apply <proc> <arg1> ... <args>) calls *<proc>* with the elements of the list *(append (list <arg1> ...) <args>)* as the actual arguments and returns the result of the call.

4.22 argument-vector (+)

(argument-vector) returns the current Unix *argv[]* of the interpreter invocation as a vector of strings.

4.23 argv (+)

(argv) returns the current *argv[]* of the interpreter invocation as a list of strings. *argv* is defined using *argument-vector* for compatibility reasons.

4.24 asin

(asin <z>) computes the principal value of the arc sine of *<z>*. A domain error occurs for arguments not in the range [-1, +1]. The arc sine in the range [-pi/2, +pi/2] radians is returned.

4.25 assq

(assq <obj> <alist>) Alist (for "association list") must be a list of pairs. This procedure finds the first pair in *<alist>* whose car field is *<obj>* and returns that pair. If no pair in *<alist>* has *<obj>* as its car then *#f* (not the empty list) is returned. This procedure uses *eq?* to compare *<obj>* with the car fields of the pairs in *<alist>*.

4.26 assv

(assv <obj> <alist>) works as *assq* using *eqv?* to compare *<obj>* with the car fields of the pairs in *<alist>*.

4.27 assv

(assoc <obj> <alist>) works as *assq* using *equal?* to compare *<obj>* with the car fields of the pairs in *<alist>*.

4.28 atan

(atan <z>) computes the principal value of the arc tangent of *<z>*. The arc tangent in the range [-pi/2, +pi/2] radians is returned.

(atan <y> <x>) computes *(angle (make-rectangular <x> <y>))* according to the R4RS.

4.29 auxmem-maxsize (+)

(auxmem-maxsize) returns the size of auxiliary memory that can be used without triggering a garbage collection when allocating more auxiliary memory.

Auxiliary memory is a notion of a memory that is used for several types of objects (currently bitmaps and greymaps) and that does not imply copying its contents during gc. See also *auxmem-size*.

4.30 auxmem-size (+)

(auxmem-size) returns the currently used portion of the auxiliary memory in bytes as an integer. See *auxmem-maxsize*.

4.31 begin

(begin <expression1> <expression2> ...) is implemented as described in the R4RS.

4.32 bitmap->greymap (+)

(bitmap->greymap <bitmap>) converts the supplied bitmap to a greymap and returns this new generated greymap. Each black pixel in the bitmap becomes a greymap pixel with greylevel 255, each white pixel

becomes a greymap pixel with greylevel 0.

4.33 bitmap->string (+)

(bitmap->string <bitmap>) returns the bitmap pixel data of the given bitmap in a new created string. The string contains all scanlines of the bitmap concatenated while each line is filled up to the byte boundary with binary 0.

4.34 bitmap-appendtiff (+)

(bitmap-appendtiff <bitmap> <filename>) appends a new page to the end of the supplied tiff file containing the given bitmap in G4 encoding. If the supplied file does not already exist this function creates the file writing the bitmap to the first page.

4.35 bitmap-clearpixel! (+)

(bitmap-clearpixel! <bitmap> <row> <column>) sets the pixel at the given row and column to zero (white). The bitmap is modified and returned.

4.36 bitmap-compare (+)

(bitmap->compare <bitmap1> <bitmap2>) compares two bitmaps and returns a dotted pair of two integers. The *car* component contains the number of pixels found to be equal, the *cdr* component contains the number of pixels in total.

To form an equality quotient of two bitmaps the *car* value can be divided by the *cdr* value.

The number of total pixels becomes important if the height's and width's of the two given bitmaps differ. It is computed by multiplying the maximum width with the maximum height of the two bitmaps.

4.37 bitmap-copy (+)

(bitmap-copy <bitmap>) creates a copy of the given bitmap and returns that copy.

4.38 bitmap-create (+)

(bitmap-create <height> <width>) creates a new bitmap with the given height and width, initializes each pixel to white and returns that newly created bitmap.

4.39 bitmap-crop (+)

(bitmap-crop <bitmap> <row> <col> <height> <width>) returns a copy of a region of a bitmap. The region is defined by its position (*<row>* and *<col>*) and by the desired *<height>* and *<width>*). It is an error if a pixel of the requested region lies outside the bitmap where it should be copied from.

4.40 bitmap-double (+)

(bitmap-double <bitmap>) creates a new bitmap with double height and double width of the given bitmap. The new bitmap is initialized by filling a square of 2 by 2 pixels for each original pixel in the original bitmap with the value of the corresponding original pixel. The resulting bitmap is returned. If a X or Y resolution is defined in the original bitmap structure the resolution is doubled and stored in the internal structure of the resulting bitmap.

4.41 bitmap-drawline! (+)

(bitmap-drawline! <bitmap> <x1> <y1> <x2> <y2>) draws a line from the pixel at *x1.y1* to the pixel *x2.y2* using Bresenham's algorithm. The bitmap is destructively modified and returned. It is an error if the coordinates are outside the given bitmap.

4.42 bitmap-equality (+)

(bitmap-equality <bitmap1> <bitmap2>) returns the equality of two given bitmaps as a flonum using the result of *bitmap-compare*.

4.43 bitmap-explode (+)

(bitmap-explode <bitmap>) applys the functions *bitmap-nextblack* and *bitmap-extract!* on a copy of the given bitmap until no more black pixels can be found in that copy. All generated subbitmaps are collected in a list which is returned by that function. This function can generally be used to divide a scanned page of a document or something similar in several subbitmaps in order to start a further analysis like character or object recognition.

4.44 bitmap-extract! (+)

(bitmap-extract! <bitmap> <row> <column>) extracts a subbitmap from the given bitmap by recursively determining all its black pixels starting at the given black pixel. The subbitmap is erased in the original bitmap, the generated subbitmap is returned. The dimensions of the subbitmap are just as big to hold all found pixels (bounding box). The original position of the subbitmap is kept in the bitmap structure itself (orow and ocol) and can be accessed afterwards using the functions *bitmap-orow* and *bitmap-ocol*.

During the recursive process of finding all pixels of the subbitmap a black pixel is considered to be a neighbor of a given black pixel if it is directly located above, under, left or right of a given black pixel. A good starting point to extract an unknown bitmap i.e. from a scanned page is the result of *bitmap-nextblack*.

This function is used to implement *bitmap-explode*. The code for this primitive is derived from the fbod-fill operator of the X11 "bitmap" editor.

4.45 bitmap-getpixel (+)

(bitmap-getpixel <bitmap> <row> <column>) returns *#t* if the pixel at the given coordinates is set (black), otherwise *#f*.

4.46 bitmap-grey! (+)

(bitmap-grey! <bitmap>) modifies the given bitmap anding each odd line with with a bitstream of 0xAA bytes and each even line with a bitstream of 0x55 bytes. This leads to a chessboard-like 50% grey effect on 100% black parts of a bitmap.The original content of the bitmap is destroyed. The modified bitmap is returned.

4.47 bitmap-half (+)

(bitmap-half <bitmap>) does a "downscaling" of the given bitmap to a bitmap with half the original height and half the original width. Each second pixel horizontally and vertically is taken from the original bitmap to form the destination bitmap.

It is an error to supply a bitmap which contains only 1 pixel. The resulting bitmap is returned. Each defined resolution in the original bitmap (X or Y) is multiplied by 0.5 and stored in the internal structure of the resulting bitmap.

4.48 bitmap-half4 (+)

(bitmap-half4 <bitmap>) does a downscaling of the given bitmap to a bitmap with half the original height and half the original width just as *bitmap-half* with the difference, that for each resulting pixel 4 pixels in a square of 2 by 2 are considered in the source bitmap.

If one of the considered pixels is black the resulting pixel will become black. This leads to a downscaling algorithm that tends to preserve little black structures. The resulting bitmap is returned. It is an error to supply a bitmap that contains only one pixel.

Each defined resolution in the original bitmap (X or Y) is multiplied by 0.5 and stored in the internal structure of the resulting bitmap.

4.49 bitmap-hash (+)

(bitmap-hash <bitmap> <arraysize>) computes an integer value out of the contents of the bitmap and out of the dimensions of the bitmap that fits into a given array-size. This function can be use to build up hashtables of bitmaps.

4.50 bitmap-height (+)

(bitmap-height <bitmap>) returns the height of the given bitmap as an integer.

4.51 bitmap-hstripe! (+)

(bitmap-hstripe! <bitmap>) modifies the given bitmap setting each even horizontal line of the bitmap to white. The original content of the bitmap is destroyed. The modified bitmap is returned.

4.52 bitmap-implant! (+)

(bitmap-implant! <subbitmap> <bitmap>) "implants" the given subbitmap in the given bitmap modifying the bitmap that way. "Implanting" is done by transferring each black pixel from the subbitmap to the original position in the bitmap. If the dimensions of the bitmap are too small, redundant pixels are not transferred. The original position of the subbitmap has to be defined. (*bitmap-orow* and *bitmap-ocol* each must not return *-1* on the subbitmap). This function is used to implement *bitmap-implode*.

4.53 bitmap-implantxor! (+)

(bitmap-implantxor! <subbitmap> <bitmap>) does the same as *bitmap-implant!* with the difference that the supplied bitmap is "implanted" using a xor-operation for each pixel in the subbitmap and the destination bitmap.

4.54 bitmap-implode (+)

(bitmap-implode <list> <rows> <cols>) creates a new bitmap with the given dimensions. Next the supplied list of subbitmaps is processed restoring each subbitmap into it's original position using *bitmap-implant!*. The original position of each subbitmap in the list has to be defined (*bitmap-orow* and *bitmap-ocol* must both not returning *-1*). The resulting bitmap is returned.

bitmap-implode called with a list generated by *bitmap-explode* and with the original dimensions results in a bitmap whose equality to the original bitmap (using *bitmap-equality*) is 1.

4.55 bitmap-invert (+)

(bitmap-invert <bitmap>) inverts all pixels in a copy of the given bitmap. The new inverted bitmap is returned. The original bitmap is left unchanged.

4.56 bitmap-invert! (+)

(bitmap-invert! <bitmap>) inverts the given bitmap destroying the original content. The modified bitmap is returned.

4.57 bitmap-list->tifffile (+)

(bitmap-list->tifffile <list> <filename>) creates the given filename and places the list of bitmaps in it using a 'page' for each element.

4.58 bitmap-nextblack (+)

(bitmap-nextblack <bitmap> <startrow> <startcol>) looks for the next black pixel in the given bitmap starting at the given coordinates rowwise downwards. The search is done line by line from left to right. If a black pixel is found that way it's position is returned in a dotted pair whose *car* component contains the row and the *cdr* component the column. If no more black pixel is found in the given bitmap a dotted pair which prints *(-1 . -1)* is returned. This function is used to implement *bitmap-explode*.

4.59 bitmap-nextblackup (+)

(bitmap-nextblackup <bitmap> <startrow> <startcol>) does the same as *bitmap-nextblack* but decreases the row after each line search.

4.60 bitmap-ocol (+)

(bitmap-ocol <bitmap>) returns the original column of this bitmap as an integer. If this value has not been set *-1* is returned. This value is used to keep the position of an extracted bitmap internally in the bitmap

structure.

4.61 bitmap-ocol-set! (+)

(bitmap-ocol-set! <bitmap> <column>) sets the value of the "original column" kept in the bitmap structure itself to the specified value as *<column>*. *<column>* must be an integer. The value of *<column>* is returned.

4.62 bitmap-orow (+)

(bitmap-orow <bitmap>) returns the original row of this bitmap as an integer. If this value has not been set *-1* is returned. This value is used to keep the position of an extracted bitmap internally in the bitmap structure.

4.63 bitmap-orow-set! (+)

(bitmap-orow-set! <bitmap> <row>) sets the value of the "original row" kept in the bitmap structure itself to the specified value as *<row>*. *<row>* must be an integer. The value of <row> is returned.

4.64 bitmap-readmtiff (+)

(bitmap-readtiff <filename> <page>) does the same as *bitmap-readtiff* after seeking to the supplied page in an multipage TIFF file. The first page has the number 0. If the seek to the supplied page fails this function returns *#f*.

4.65 bitmap-readpng (+)

(bitmap-readpng <filename>) reads the bitmap stored in *<filename>* and returns that bitmap. This procedure is restricted to B/W (1 bit) greyscale PNG format in *<filename>*. *bitmap-readpng* accepts interlaced greyscale/1 bit PNG format.

4.66 bitmap-readtiff (+)

(bitmap-readtiff <filename>) reads a file in TIFF format, creates a bitmap out of it and returns that bitmap. Supported compression schemes are none, G3 and G4. It is an error to read any image which is not a bitmap (which depth per pixel not equals 1).

4.67 bitmap-readxbm (+)

(bitmap-readxbm <filename>) reads a bitmap in xbm format, creates a bitmap object out of it and returns that bitmap.

4.68 bitmap-rotate-left (+)

(bitmap-rotate-left <bitmap>) creates a new bitmap by rotating the given bitmap 90 degrees to the left and returns the new bitmap.

4.69 bitmap-rotate-right (+)

(bitmap-rotate-right <bitmap>) creates a new bitmap by rotating the given bitmap 90 degrees to the right and returns the new bitmap.

4.70 bitmap-rowbytes (+)

(bitmap-rowbytes <bitmap>) returns the number of bytes which is used internally to store one row of pixels as an integer. This is a low level function.

4.71 bitmap-scale (+)

(bitmap-scale <bitmap> <row-factor> <col-factor>) scales the given bitmap by a row and col factor returning the new bitmap.

4.72 bitmap-scale-absolute (+)

(bitmap-scale-absolute <bitmap> <rows> <cols>) scales the given bitmap to the absolute given dimensions.

4.73 bitmap-setpixel! (+)

(bitmap-setpixel! <bitmap> <row> <column>) sets the pixel at the given coordinates to one (black). The bitmap is modified and returned.

4.74 bitmap-size (+)

(bitmap-size <bitmap>) returns the number of pixels in a given bitmap as an integer.

4.75 bitmap-tifffile->list (+)

(bitmap-tifffile->list <filename>) Opens the file and returns a list of bitmaps with represents each 'page' in the multipage TIFF file. Please consider that reading a bitmap means decompressing it to auxiliary memory; using that function on large multipage tiff files may cause allocation problems and/or heavy swapping/paging.

4.76 bitmap-tiffpage (+)

(bitmap-tiffpage <processor> <infile> [<outfile>]) processes a given multipage TIFF file page by page. The processor has to be a function which expects 2 arguments: pagenumber (starting with 0) and the actual bitmap of that page. The processor is called for each page with its arguments. If you supply the optional *<outfile>* parameter the processor must return a (processed) bitmap that is placed in the given file in *<outfile>* in multipage TIFF format.

4.77 bitmap-tiffpages (+)

(bitmap-tiffpages <tifffile>) returns the size of the directory of the supplied tiff file as an integer indicating the absolute number of "pages".

4.78 bitmap-vglue (+)

(bitmap-vglue <bitmap1> <bitmap2>) creates a new bitmap appending *<bitmap2>* at the end of *<bitmap1>* in vertical direction. The width of the resulting bitmap is the maximum of the individual widths of the two bitmaps, the height is the sum of each individual height of the two bitmaps.

4.79 bitmap-vstripe! (+)

(bitmap-vstripe! <bitmap>) modifies the given bitmap setting each even vertical line of the bitmap to white. The original content of the bitmap is destroyed. The modified bitmap is returned.

4.80 bitmap-width (+)

(bitmap-width <bitmap>) returns the width of the given bitmap as an integer.

4.81 bitmap-writeipng (+)

(bitmap-writeipng <bitmap> <filename>) stores the *<bitmap>* in PNG format (GREYSCALE, B/W) in the given filename in interlaced format.

4.82 bitmap-writepng (+)

(bitmap-writepng <bitmap> <filename>) writes the given bitmap in PNG format to the given filename.

4.83 bitmap-writetiff (+)

(bitmap-writetiff <bitmap> <filename>) writes the contents of the given bitmap out to a file whose name is given as a string. The format written is TIFF with G4 compression. The returned value is unspecified. If the X and Y resolution is defined in the bitmap object the appropriate tag in the generated TIFF file is used to store this information.

4.84 bitmap-writexbm (+)

(bitmap-writexbm <bitmap> <filename>) writes the contents of the given bitmap to a file whose name is given as a string. The format written is the X11 xbm format which can be used by the editor *bitmap* which is part of the X11 distribution. The returned value is unspecified.

4.85 bitmap-xres (+)

(bitmap-xres <bitmap>) returns the current X resolution of the supplied bitmap as a fbat. A return of *-1.0* is considered an undefi ned X resolution.

4.86 bitmap-xres-set! (+)

(bitmap-xres-set! <bitmap> <resolution>) sets the current X resolution of the supplied bitmap to the supplied resolution. The resolution must be supplied as an inexact fbat. The resolution supplied is returned.

4.87 bitmap-yres (+)

(bitmap-yres <bitmap>) returns the current Y resolution of the supplied bitmap as a fbat. A return of *-1.0* is considered an undefi ned X resolution.

4.88 bitmap-yres-set! (+)

(bitmap-yres-set! <bitmap> <resolution>) sets the current Y resolution of the supplied bitmap to the supplied resolution. The resolution must be supplied as an inexact fbat. The resolution supplied is returned.

4.89 bitmap? (+)

(bitmap? <obj>) returns *#t* if the given object *<obj>* is a bitmap, otherwise *#f*.

4.90 boolean?

(boolean? <obj>) returns *#t* is the object *<obj>* is either *#f* or *#t* and returns *#f* otherwise.

4.91 caar ... cddddr

These procedures are the usual compositions of *car* and *cdr* and are implemented as specifi ed in the R4RS. Arbitrary compositions, up to four deep, are provided.

4.92 call-with-current-continuation

(call-with-current-continuation <proc>) is fully implemented as specifi ed in the R4RS.

4.93 call-with-input-fi le

(call-with-input-fi le <string> <proc>) is implemented as specifi ed in the R4RS calling *<proc>* with one open input port referencing to the opened fi le given in *<string>*.

4.94 call-with-output-fi le

(call-with-output-fi le <string> <proc>) is implemented as specifi ed in the R4RS calling *<proc>* with one open output port referencing to the opened (truncated or new created) fi le given in *<string>*.

4.95 call/cc (+)

(call/cc <proc>) is an abbreviation of *(call-with-current-continuation <proc>)* implemented as a syntax procedure.

4.96 car

(car <pair>) returns the contents of the car fi eld of *pair*. It is an error to take the car of the empty list.

4.97 case

(case <key> <clause1> <clause2> ...) is implemented as described in the R4RS.

4.98 cdr

(cdr <pair>) returns the contents of the cdr fi eld of *pair*. It is an error to take the cdr of the empty list.

4.99 ceiling

(ceiling <x>) returns the smallest integer not larger than *<x>*.

4.100 char->integer (!)

(char->integer <char>) returns an exact integers representation of the given char. Note that in Inlab-Scheme a character may contain any 8 bit value, so the returned integer is in the interval [0 ... 255].

4.101 char-alphabetic?

(char-alphabetic? <char>) returns *#t* if the given char is an alphabetic character in the US ASCII character set, otherwise *#f*.

4.102 char-ci=?

(char-ci=? <char1> <char2>) is the case insensitive variant of *char=?*. Only US ASCII alphabetic characters are handled case insensitive.

4.103 char-ci<=?

(char-ci<=? <char1> <char2>) is the case insensitive variant of *char<=?*. Only US ASCII alphabetic characters are handled case insensitive.

4.104 char-ci>=?

(char-ci>=? <char1> <char2>) is the case insensitive variant of *char>=?*. Only US ASCII alphabetic characters are handled case insensitive.

4.105 char-ci<?

(char-ci<? <char1> <char2>) is the case insensitive variant of *char<?*. Only US ASCII alphabetic characters are handled case insensitive.

4.106 char-ci>?

(char-ci>? <char1> <char2>) is the case insensitive variant of *char>?*. Only US ASCII alphabetic characters are handled case insensitive.

4.107 char-lower-case?

(char-lower-case? <char>) returns *#t* if the given char is a lower case alphabetic character in the US ASCII character set, otherwise *#f*.

4.108 char-numeric?

(char-numeric? <char>) returns *#t* if the given char is a numeric character in the US ASCII character set, otherwise *#f*.

4.109 char-ready? (!)

is currently not implemented in Inlab-Scheme.

4.110 char-upper-case?

(char-upper-case? <char>) returns *#t* if the given char is an upper case alphabetic character in the US ASCII character set, otherwise *#f*.

4.111 char-whitespace?

(char-whitespace? <char>) returns *#t* if the given char is a whitespace character, otherwise *#f*. Space, tab, line feed, form feed and carriage return in the US ASCII character set are whitespace characters in Inlab-Scheme.

4.112 char=?

(char=? <char1> <char2>) returns *#t* if the both characters are the same, otherwise *#f*.

4.113 char?

(char? <obj>) returns *#t* if *<obj>* is a character, otherwise returns *#f*.

4.114 char<=?

(char<=? <char1> <char2>) returns *#t* if the numerical value of *<char1>* is smaller than or equal to that of *<char2>*, otherwise *#f*.

4.115 char>=?

(char>=? <char1> <char2>) returns *#t* if the numerical value of *<char1>* is greater than or equal to that of *<char2>*, otherwise *#f*.

4.116 char<?

(char<? <char1> <char2>) returns *#t* if the numerical value of *<char1>* is smaller than that of *<char2>*, otherwise *#f*.

4.117 char>?

(char>? <char1> <char2>) returns *#t* if the numerical value of *<char1>* is greater than that of *<char2>*, otherwise *#f*.

4.118 close (+)

(close <port>) closes the given port regardless if it is an input or an output port.

4.119 close-input-port

(close-input-port <port>) closes the input port *<port>*. The value returned is unspecified.

4.120 close-output-port

(close-output-port <port>) closes the output port *<port>*. The value returned is unspecified.

4.121 complex?

(complex? <obj>) returns *#t* if *<obj>* is a complex number, which applies to both numerical types in Inlab-Scheme, *#f* otherwise.

4.122 cond

cond is implemented as described in the R4RS including the alternative *<clause>* syntax of *(<test> => <recipient>)*. In that case *<recipient>* is evaluated and must evaluate to a procedure of one argument. This procedure is then invoked with the value of the *<test>*.

4.123 cons

(cons <obj1> <obj2>) returns a newly allocated pair whose *car* is *<obj1>* and whose *cdr* is *<obj2>*. The pair is guaranteed to be different (in the sense of *eqv?*) from any existing object.

4.124 cos

(cos <z>) computes the cosine of *<z>* (measured in radians). A large magnitude argument may yield a result with little or no significance.

4.125 current-input-port

(current-input-port) returns the current default input port.

4.126 current-output-port

(current-output-port) returns the current default output port.

4.127 define

(define ...) is implemented as described in the R4RS supporting the following forms

- *(define <variable> <expression>)*

- *(define (<variable> <formals>) <body>)*
- *(define (<variable> . <formal>) <body>)*
- *(define <variable>)*

Top level definitions are supported, as well as internal definitions are. In fact Inlab-Scheme permits a definition at any place in a program structure.

4.128 delay

(delay <expression>) is implemented as described in the R4RS. See also *force*.

4.129 denominator (!)

denominator is not implemented in Inlab-Scheme.

4.130 disable-interrupt (+)

(disable-interrupt) disables Inlab-Scheme to be interrupted by a received SIGINT signal.

4.131 display

(display <obj>) writes a representation of the given <*obj*> to the current output port. Strings and characters are written not for reinternalization by *read* (strings and characters are sent to the output port as they are).

(display <obj> <port>) does the same to the given output port.

4.132 do

(do ...) is implemented as described in the R4RS. The "named let" variant of looping is also implemented.

4.133 dynamic-wind (+)

(dynamic-wind <before-thunk> <action-thunk> <after-thunk>) is a generalization on Common Lisp *unwind-protect*, designed to take into account that continuations produced by *call-with-current-continuation* ma be reentered. The arguments <*before-thunk*>, <*action-thunk*> and <*after-thunk*> must all be procedures of no arguments (thunks).

dynamic-wind behaves as follows: first <*before-thunk*> is called. Then <*action-thunk*> is called. Whenever control leaves <*action-thunk*> , wether by normal return or by invocation of a continuation (a non local exit), <*after-thunk*> is executed. If a continuation is created during the execution of <*action-thunk*> (by use of *call-with-current-continuation*) , and that continuation is invoked from outside the *dynamic-wind*, it is guaranteed that <*before-thunk*> will be executed before that continuation gains control.

dynamic-wind returns the value of <*action-thunk*>.

As an extension to this behavior Inlab-Scheme does not call <*exit-thunk*> and again <*init-thunk*> when changing control using continuations if it can prove that both belong to the same dynamic-wind expression and do occur in the same depth and order of control flow.

4.134 eload (e)

4.135 enable-interrupt (+)

(enable-interrupt) enables the running Inlab-Scheme to be interrupted by a received SIGINT signal.

4.136 eof-object?

(eof-object? <obj>) returns *#t* if the given object <*obj*> is the end of file object, otherwise returns *#f*.

4.137 eq?

(eq? <obj1> <obj2>) returns *#t* if both objects <*obj1*> and <*obj2*> are exactly the same thing, otherwise *#f*.

Since Inlab-Scheme uses a pure indirect memory model where everything is a pointer to a tagged object

eq? is implemented as a pure pointer comparison.

4.138 equal?

(equal <obj1> <obj2>) is implemented as described in the R4RS. *equal?* may fail to terminate if its arguments are circular data structures.

4.139 eqv?

(eqv? <obj1> <obj2>) is implemented as described in the R4RS. Please note that it is important to use *eqv?* on numbers (integers) since two integers of the same number may be not the same object causing *eq?* returning *#f*.

4.140 escheme? (e)

4.141 even?

(even? <number>) returns *#t* if the given exact integer is even, *#f* otherwise. Passing an inexact number causes an error.

4.142 exact->inexact

(exact->inexact <z>) returns an inexact representation of *<z>*.

4.143 exact?

(exact? <obj>) returns *#t* if *<obj>* is an exact number, which applies to exact integers only in Inlab-Scheme, *#f* otherwise.

4.144 exit (+)

(exit) immediately exits the interpreter returning 0 as returncode to the Unix environment. *(exit <returncode>)* exits immediately returning the supplied returncode.

4.145 exp

(exp <z>) returns the exponential value of the given argument *<z>*.

4.146 expt

(expt <z1> <z2>) return1 *<z1>* raised to the power *<z2>*. *(expt 0 0)* is defined to be *1*.

4.147 file-exists? (+)

(file-exists? <filename-spec>) returns *#t* if the supplied filename-spec is an Unix object that can be successfully opened using *fopen()* for "reading" in the current invocation environment of the interpreter, otherwise *#f*.

4.148 floor

(floor <x>) returns the largest integer not larger than *<x>*.

4.149 fluid-let (+)

(fluid let ((<variable> <init>) ...) expression ...) is an implementation of dynamic binding as found in MIT-Scheme, The syntax of this special form is similar to that of *let*.

Unlike *let*, *fluid-let* creates no new bindings; instead it assigns the values of each init to the binding (determined by the rules of lexical scoping) of its corresponding variable.

fluid-let then evaluates the expressions in the body sequentially, restores the original values of the variables, and returns the value of the last expression. *fluid-let* assigns values in an unspecified order; because of this, it must be possible to evaluate each init without referring to any variable.

The implementation of *fluid-let* is implemented in Inlab-Scheme to be save to any use of continuations.

4.150 for-each

(for-each <proc> <list1> <list2> ...) works like *map* but collects no list of results. The return value is unspecified.

4.151 force

(force <promise>) forces the value of *<promise>* (see *delay*). If no value has been computed for the promise, then a value is computed and returned. The value of the promise is cached so that if it is forced a second time, the previously computed value is returned.

4.152 gc (+)

(gc) performs a garbage collection and returns an unspecified value.

4.153 gcd

(gcd n1 ...) returns the greatest common divisor of its arguments. The result is always non-negative. *gcd* called with no arguments returns *0*.

4.154 getenv (+)

(getenv <environment-variable>) returns the value of the given environment variable as a string or *#f* if the variable is not defined in the current environment. The environment variable must be given as a string.

Example:

```
[1] (getenv "HOME")
"/usr/home/lisp"
[2] (getenv "UNDEFINED")
#f
```

4.155 getpid (+)

(getpid) returns the process id of the interpreter as an integer.

4.156 greymap->bitmap (+)

(greymap->bitmap <greymap>) converts the *<greymap>* into a bitmap and returns that bitmap. Each pixel whose greylevel is 128 and above is considered to be black in the resulting bitmap, 127 and below converts to white.

4.157 greymap->bitmap/t (+)

(greymap->bitmap/t <greymap> <threshold>) works as *greymap->bitmap* but with a variable threshold that must be in the range between 0 and 255 inclusive. All pixel equal and above the given threshold are considered to be black.

4.158 greymap-copy (+)

(greymap-copy <greymap>) returns a copy of the given greymap.

4.159 greymap-create (+)

(greymap-create <row> <columns>) creates a new greymap containing white pixels and returns that greymap.

4.160 greymap-crop (+)

(greymap-crop <greymap> <row> <col> <height> <width>) returns a copy of a region of a greymap. The region is defined by its position (*<row>* and *<col>*) and by the desired *<height>* and *<width>*). It is an error if a pixel of the requested region lies outside the greymap where it should be copied from.

4.161 greymap-equal? (+)

(greymap-equal? <greymap1> <greymap2>) returns *#t* if the contents of the two greymaps are identical, *#f* otherwise.

4.162 greymap-getpixel (+)

(greymap-getpixel <greymap> <row> <column>) returns the content of the referenced pixel as an integer in the range of 0 ... 255.

4.163 greymap-height (+)

(greymap-height <greymap>) returns the actual height of a greymap.

4.164 greymap-invert! (+)

(greymap-invert! <greymap>) inverts the given greymap such destructively modifying the greymap. If you want to preserve the original content of the bitmap use *greymap-copy* to preserve it. The greymap is returned after the inversion of all pixels.

4.165 greymap-ocol (+)

(greymap-ocol <greymap>) returns the integer currently in the "original column" field of the internal greymap structure. A return value of *-1* means "undefined".

4.166 greymap-ocol-set! (+)

(greymap-ocol-set! <greymap> <value>) sets the "original column" to the given integer value in the internal greymap structure.

4.167 greymap-orow (+)

(greymap-orow <greymap>) returns the integer currently in the "original row" field of the internal greymap structure. A return value of *-1* means "undefined".

4.168 greymap-orow-set! (+)

(greymap-orow-set! <greymap> <value>) sets the "original row" to the given integer value in the internal greymap structure.

4.169 greymap-readpng (+)

(greymap-readpng <filename>) reads the PNG file given in *<filename>* and returns a greymap. *greymap-readpng* should read any PNG file format to greyscale doing an implicit to greyscale conversion if needed. *greymap-readpng* reads interlaced PNG files as well as non interlaces PNG files. Note that during read 16 bytes per pixel are allocated temporarily using the auxiliary memory allocation scheme of Inlab-Scheme.

4.170 greymap-scale (+)

(greymap-scale <bitmap> <row-factor> <col-factor>) scales the given greymap by a row and col factor returning the new bitmap. Scaling is done simply by skipping or inserting suitable rows and columns. If you want do to a scaling with better looking results you should apply *greymap-smooth* or *greymap-smooth/v* before.

4.171 greymap-setpixel! (+)

(greymap-setpixel! <greymap> <row> <column> <color>) sets a pixel at the given position to the specified color. The color has to be an integer in the range 0 ... 2555 representing a greylevel (0 = white, 255 = black).

4.172 greymap-smooth (+)

(greymap-smooth <greymap>) Performs a smoothing operation as *greymap-smooth/v* with a square-height of 3.

4.173 greymap-smooth-region (+)

(greymap-smooth-region <greymap> <row> <col> <height> <width> <value>) performs a smoothing operation on a given region of a greymap and returns that region. The region's origin is *<row>* and *<col>*, its height and width are given in *<height>* and *<width>* respectively. *<value>* is the odd height of a

square that average greymap-value is computed for each pixel.

4.174 greymap-smooth/v (+)

(greymap-smooth/v <greymap> <value>) performs a smoothing operation on the given greymap computing the average greymap-value of an square of the given height and width in value. Since every pixel has to be the center of the square *<value>* has to be an positive, odd value. A new, smoothed greymap is returned.

4.175 greymap-width (+)

(greymap-width <greymap>) returns the actual width of a greymap.

4.176 greymap-writeipng (+)

(greymap-writeipng <greymap> <filename>) stores the *<greymap>* in PNG format (GREYSCALE) in *<filename>* in interlaced format.

4.177 greymap-writepng (+)

(greymap-writepng <greymap> <filename>) the supplied greymap is written out in PNG-format to the given file name.

4.178 greymap-xres (+)

(greymap-xres <greymap>) returns the floating point value in the "x resolution" field of the internal greymap structure. A return value of *-1.0* means "undefined". The resolution unit is DPI (dots per inch).

4.179 greymap-xres-set! (+)

(greymap-xres-set! <greymap> <float>) sets the "x resolution" field in the internal greymap structure to the given value, which must be an inexact floating point value.

4.180 greymap-yres (+)

(greymap-yres <greymap>) returns the floating point value in the "y resolution" field of the internal greymap structure. A return value of *-1.0* means "undefined". The resolution unit is DPI (dots per inch).

4.181 greymap-yres-set! (+)

(greymap-yres-set! <greymap> <float>) sets the "y resolution" field in the internal greymap structure to the given value, which must be an inexact floating point value.

4.182 greymap? (+)

(greymap? <obj>) return *#t* if the supplied object *<obj>* is a greymap, otherwise *#f*.

4.183 if

if is implemented as described in the R4RS supporting the following two forms:

- *(if <test> <consequent> <alternate>)*
- *(if <test> <consequent>)*

4.184 imag-part (!)

imag-part is not implemented in Inlab-Scheme.

4.185 inexact->exact

(inexact->exact <z>) returns an exact representation of *<z>*.

4.186 inexact?

(exact? <obj>) returns *#t* if *<obj>* is an inexact number, which applies to inexact floating point numbers only in Inlab-Scheme, *#f* otherwise.

4.187 input-port?

(input-port? <obj>) returns *#t* if *<obj>* is an input port, otherwise returns *#f*.

4.188 integer->char (!)

(integer->char <n>) returns the char whose integer representation is given in *<n>*. Note that *<n>* may be supplied in the range [0 ... 255] since any 8 bit representable value can be converted into a char.

4.189 integer?

(integer? <obj>) returns *#t* if *<obj>* is an integer number, which applies to exact integers and inexact fbats which could reasonably be converted to an exact integer in in Inlab-Scheme, *#f* otherwise. This behavior is unreliable on inexact fbating point numbers, since any inaccuracy may affect the result.

4.190 lambda (!)

(lambda <formals> <body>) is implemented as described in the R4RS. The following forms in *<formals>* are supported:

- *(<variable1> ...)*
- *<variable>*
- *<variable1> ... <variableN-1> . <variableN>*

Inlab-Scheme implements *lambda* using *named-lambda* with a name of *'()* which means "unnamed".

Inlab-Scheme does currently not detect a multiple appearance of the same variable in *<formals>* and does not signal an error in this case. This may be fi xed in a future release.

4.191 lcm

(lcm n1 ...) returns the least common multiple of its arguments. The result is always non-negative. *lcm* called with no arguments returns *1*.

4.192 length

(length <list>) returns the length of the supplied list.

4.193 let (!)

(let <bindings> <body>) is implemented as described in the R4RS as a syntax procedure. Also supported is the "named let" syntax variant *(let <variable> <bindings> <body>)* as described in chapter 4.2.1 in the R4RS to provide a "more general looping construct" than *do*.

*4.194 let**

(let <bindings> <body>)* is implemented as described in the R4RS as a syntax procedure based on *let*.

4.195 letrec

(letrec <bindings> <body>) is implemented as described in the R4RS as a syntax procedure. The restriction on *letrec* forcing an implementation to generate an error if an unassigned variable is referenced in an *<init>* is implemented using the special constant *#u* (unassigned constant) of Inlab-Scheme.

4.196 list

(list <obj> ...) returns a newly allocated list of its arguments. *(list)* returns the empty list.

4.197 list->string

(list->string <chars>) returns a newly allocated string formed from the characters in the list of characters *<chars>*.

4.198 list->vector

(list->vector <list>) returns a newly created vector initialized to the elements of the list *<list>*.

4.199 list-ref

(list-ref <list> <k>) returns the <*k*>'th element of <*list*>.

4.200 list-tail

(list-tail <list> <k>) returns the sublist of list obtained by omitting the first <*k*> elements.

4.201 list-transform-negative (+)

(list-transform-positive <list> <predicate>) generates a list of all elements in the given list for which the given predicate is false. The order of the elements is kept.

4.202 list-transform-positive (+)

(list-transform-positive <list> <predicate>) generates a list of all elements in the given list for which the given predicate is true. The order of the elements is kept.

4.203 list?

(list? <obj>) returns *#t* if *obj* is a list, otherwise returns *#f*. By definition, all lists have finite length and are terminated by the empty list (this actually means that circular lists are recognized as not being a list).

4.204 load

(load <filename>) reads expressions and definitions from the file and evaluates them sequentially. The results are not printed by Inlab-Scheme. An unspecified value is returned.

(load) loads the file specified in the variable *the-current-filename*.

4.205 log

(log <z>) returns the value of the natural logarithm of argument <*z*>.

4.206 magnitude (!)

magnitude is not implemented in Inlab-Scheme.

4.207 make-polar (!)

make-polar is not implemented in Inlab-Scheme.

4.208 make-rectangular (!)

make-rectangular is not implemented in Inlab-Scheme.

4.209 make-string

(make-string <k>) creates a new string with <*k*> elements. The contents of the string are unspecified in this case but actually all blanks in Inlab-Scheme. The string is returned.

(make-string <k> <char>) creates a new string which is <*k*> in length and contains in every position the given char. The string is returned.

4.210 make-vector

(make-vector <k>) returns a newly allocated vector of <*k*> elements. The initial contents of each element is left unspecified in R4RS, Inlab-Scheme actually initializes each member being the empty list in this case.

(make-vector <k> <fill>) returns a newly allocated vector of <*k*> elements, each element is initialized to <*fill*>.

4.211 map

(map <proc> <list1> <list2> ...) applies <*proc*> element-wise to the elements of the lists and returns a list of the results, in order from left to right. The <*list*>'s must be lists, and <*proc*> must be a procedure taking as many arguments as there are lists. If more than one list is given they must all be the same length. The dynamic order in which <*proc*> is supplied to the elements of the lists in unspecified (but in Inlab-

Scheme actually left to right).

4.212 max

(max <x1> <x2> ...) returns the maximum of its arguments. If any argument is inexact, then the result will also be inexact. At least one argument must be supplied.

4.213 member

(member <obj> <list>) returns the first sublist of list whose car is *<obj>* in the sense of *equal?* or *#f* if *<obj>* does not occur.

4.214 memq

(memq <obj> <list>) returns the first sublist of list whose car is *<obj>* in the sense of *eq?* or *#f* if *<obj>* does not occur.

4.215 memv

(memv <obj> <list>) returns the first sublist of list whose car is *<obj>* in the sense of *eqv?* or *#f* if *<obj>* does not occur.

4.216 min

(min <x1> <x2> ...) returns the minimum of its arguments. If any argument is inexact, then the result will also be inexact. At least one argument must be supplied.

4.217 modulo

(modulo <n1> <n2>) implements the modulo operator according to the R4RS. The value returned has always the sign of the divisor.

4.218 named-syntax-lambda (+)

(named-syntax-lambda (<name> <expr> <env>) <body>) is a special form that creates a syntax-procedure as a first class object. This is the low level macro facility of Inlab Scheme. Currently no high level macro facility is implemented.

If a syntax-procedure is about to be executed the body of the syntax-lambda form is executed in an environment that holds the original expression to be transformed in *<expr>* and the invocation environment in *<env>*.

The expression that is returned by the evaluation of a syntax-procedure is evaluated in place of the original expression. If the original expression was a pair and the substitute is also a pair the original code is modified destructively to prevent repetitive macro expansion.

The *<name>* parameter has the purpose to give a syntax-procedure a name for debugging purposes and is not evaluated.

4.219 negative?

(negative? <number>) returns *#t* if the given number is less that zero, *#f* otherwise.

4.220 newline

(newline) writes an end of line to the current output port. Inlab-Scheme does this writing a line feed character. The returned value is unspecified.

(newline <port>) does the same writing to the given output port.

4.221 not

(not <obj>) is implemented as described in the R4RS.

4.222 null?

(null? <obj>) returns *#t* if *obj* is the empty list, otherwise returns *#f*.

4.223 number->string (!)

(number->string <number>) converts a number to its string representation in the radix 10.

(number->string <number> <radix>) converts a number to its string representation in the given radix. Inlab-Scheme supports the radices 2,8,10 and 16 on exact numbers (exact integers) and 10 on inexact reals (floating points). An unsupported radix signals an error.

4.224 number?

(number? <obj>) returns *#t* if *<obj>* is a number, *#f* otherwise. Inlab-Scheme supports inexact floating point and exact integer numbers.

4.225 numerator (!)

numerator is not implemented in Inlab-Scheme.

4.226 odd?

(odd? <number>) returns *#t* if the given exact integer is odd, *#f* otherwise. Passing an inexact number causes an error.

4.227 open-input-file

(open-input-file <filename>) takes a string naming an existing file and returns an input port capable of delivering characters from the file. If the file cannot be opened an error is signaled.

4.228 open-output-file (!)

(open-output-file <filename>) takes a string naming an output file to be created and returns an output port capable of writing characters to a new (or truncated) file by that name. If the file cannot be opened an error is signaled. If a file with the given name already exists the file is truncated to length zero and an output port is returned.

4.229 or

(or <test1> ...) is implemented as described in the R4RS as a syntax procedure.

4.230 output-port?

(output-port? <obj>) returns *#t* if *<obj>* is an output port, otherwise returns *#f*.

4.231 pair?

(pair? <obj>) returns *#t* if object *<obj>* is a pair, otherwise returns *#f*.

4.232 patimg-to-tiff (+)

(patimg-to-tiff <patimg-file> <tiff-file>) invokes a built in file format converter which converts the USAPat PATIMG file format as found on the newer USAPat CD-ROM series of the US-PTO to multipage tiff.

The resolutions found in the PATIMG file are preserved using the appropriate tiff tags. Both filenames must be specified as strings. An unspecified value is returned. If any problem occurs during conversion an error is signalled.

Converting is done without decompressing the G3/G4 streams.

4.233 peek-char

(peek-char) returns the next character from the current input port without updating the port to point to the following character. If no character is available the end of file object is returned.

(peek-char <port>) does the same with the given input port.

4.234 popen/r (+)

(popen/r <command>) forks the supplied command and connects an Inlab-Scheme input-stream on its stdout.

4.235 popen/w (+)

(popen/w <command>) forks the supplied command and connects a Inlab-Scheme output-stream to it which is returned. The output stream can be closed by *close* or *close-output-stream*.

4.236 positive?

(positive? <number>) returns *#t* if the given number is greater that zero, *#f* otherwise.

4.237 pp (+)

(pp <obj> [<port>]) pretty prints the given object *<obj>* to the current-output-port or to the supplied (optional) output-port. Compound and syntax-procedures can be supplied as the object resulting in pretty printing the actual code of the procedures.

4.238 procedure?

(procedure <obj>) returns *#t* if *<obj>* is a procedure, otherwise returns *#f*. Objects in Inlab-Scheme that return *#t* are compound procedures generated by *lambda* or *named-lambda* or primitive procedures. *procedure?* returns *#f* for objects generated by *syntax-lambda* or *named-syntax-lambda*.

4.239 quasiquote

(quasiquote <template>) is implemented as described in the R4RS and may be an abbreviated using a backquote (‘).

unquote and *unquote-splicing* are bound to procedures that generate an error if called outside an *quasiquote* template.

4.240 quote (!)

(quote <datum>) evaluates to *<datum>*. *(quote <datum>)* may be abbreviated as *’<datum>*. Numerical constants, string constants, character constants and boolean constants evaluate "to themselves", they need not be quoted.

In contrast to the R4RS Inlab-Scheme does not signal an error if a constant is changed using a mutation procedure like *set-car!* or *string-set!*. This may be changed in a future release.

4.241 quotient

(quotient <n1> <n2>) implements number theoretic integer division according to the R4RS. The returned value always has the sign of the product of its arguments.

4.242 rand (+)

(rand) returns a random integer like *rand()* does.

4.243 rational?

(rational? <obj>) returns *#t* if *<obj>* is a rational number, which applies to exact integers in Inlab-Scheme, *#f* otherwise.

4.244 rationalize (!)

rationalize is not implemented in Inlab-Scheme.

4.245 read

(read) returns the next object parsable from the current input port and returns that object.

(read <port>) returns the next object parsable from the given input port and returns that object.

4.246 read-char

(read-char) returns the next character from the current input port. If no character is available the end of file object is returned.

(read-char <port>) does the same with the given input port.

4.247 real-part (!)

real-part is not implemented in Inlab-Scheme.

4.248 real?

(real? <obj>) returns *#t* if *<obj>* is a real number, which applies to both numerical types in Inlab-Scheme, *#f* otherwise.

4.249 remainder

(remainder <n1> <n2>) implements a different modulo operator according to the R4RS. The value returned is either zero or has the sign of the dividend.

4.250 reverse

(reverse <list>) returns a newly allocated list consisting of the elements of *<list>* in reverse order.

4.251 round

(round <x>) returns the closes integer to *<x>*, rounding to even when *<x>* is halfway between two integers. Round does that according to the IEEE fbating point standard (default rounding mode).

4.252 set!

(set! <variable> <expression>) evaluates *<expression>* and stores the resulting value in the location to which *<variable>* is bound. The result is unspecifi ed.

4.253 set-car!

(set-car! <pair> <obj>) Stores *obj* in the car fi eld of *pair*. The value returned is unspecifi ed.

4.254 set-cdr!

(set-cdr! <pair> <obj>) Stores *obj* in the cdr fi eld of *pair*. The value returned is unspecifi ed.

4.255 sin

(sin <z>) computes the sine of *<z>* (measured in radians). A large magnitude argument may yield a result with little or no signifi cance.

4.256 sqrt

(sqrt <z>) returns the principal square root of *<z>*.

4.257 st33-to-tiff (+)

(st33-to-tiff <list> <tiff-fi le) invokes a built in fi le format converter which converts the WIPO ST.33 fi le format as found on the european ESPACE CD-ROM series to multipage tiff.

st33-to-tiff expects a list of input fi lenames each specifi ed as a string and the name of the output tiff fi le as a string.

st33-to-tiff creates the output tiff fi le by converting and appending each page in the specifi ed list from left to right. The resolution tags generated are always 300DPI.

An unspecifi ed value is returned. If any problem occurs during conversion an error is signalled.

Example:

```
[1] (st33-to-tiff '("page0001" "page0002")
                  "tmp.tiff")
ok
[2] (define a (bitmap-readtiff "tmp.tiff"))
a
[3]
```

4.258 stat (+)

(stat) performs a garbage collection and displays several useful values concerning the status of the currently

running Inlab-Scheme.

Example:

```
[1] (stat)
created: Jan  2 1997 15:59:53
destructive macro expansion is ON.
running on a 32-bit machine.
Heap   : 4.0 MB, 43586 DWords of 524288
         used, 91.6866 % free.
AuxMem : 0 bytes of 8388608 used,
         100 % free.
Symbols: 899 Symbols of 1024 used,
         125 available.
Hash   : 812 entries of 4096 used, 3284
         free, max. length of bucket-
         list: 3
Stacks : 11/16384 (pstack) 5/16384 (cstack)
ok
[4]
```

The creation date is displayed. The "destructive macro expansion" is always ON (it can be switched off internally for testing purposes).

The "native word size" of the machine on which Inlab-Scheme runs is indicated in the third line.

The size of Auxiliary Memory is displayed next (this memory is NOT allocated and just a notion of size when Inlab-Scheme should perform a garbage collection). Auxiliary Memory is used by Inlab-Scheme to store all "advanced objects" like bitmaps and greymaps.

The current number of symbols is displayed. The symbol memory is extended automatically if the current chunk of space is exceeded. The hash table information may be an indicator to increase it if the "max. length of bucket list" is obviously too large (e.g. > 4). This could happen if your program uses very large amounts of symbols.

Finally the two stacks are displayed with its current position (stack pointer) and the total size.

Note: If you are running a restricted binary of Inlab-Scheme you will not be able to execute *(stat)* because the garbage collection invoked by it terminates the program.

We used *stat* to prove the proper tail-recursive implementation of Inlab-Scheme regarding stack allocations.

4.259 string

(string <char1> ...) returns a newly allocated string composed of the arguments (which must all be chars). *(string)* returns the empty string.

4.260 string->bitmap (+)

(string->bitmap <string> <row> <col>) creates a new bitmap with the given dimensions and initializes it with the bits given in a string. Each line given in the string has to be filled up to a byte boundary. The length of the given string must match with the given dimensions. The generated bitmap is returned.

The implementation relays on the fact that in Inlab-Scheme strings can contain arbitrary bytes in the range of 0 to 255.

4.261 string->input-port (+)

(string->input-port <string>) creates an input port out of the supplied string.

4.262 string->list

(string->list <string>) returns a newly allocated list of the characters that make up the given string.

4.263 string->number (!)

(string->number <string>) converts the given external representation of a number given in *<string>* to its numerical equivalent and returns that. An explicit radix prefix may be supplied, supported explicit radix prefixes are "#b" for binary, "#o" for octal, "#d" for decimal and "#x" for hexadecimal. An exponent may be given as an "e" causing generation of an inexact real. If the conversion can not be done *#f* is returned.

(string->number <string> <radix>) allows to specify a radix, supported are 2.8.10 and 16 for exact integers, 10 for inexact reals.

4.264 string->symbol

(string->symbol <string>) returns the symbol whose name is *<string>*. This procedure can create symbols with names containing special characters or letters in the non-standard case (which is lower case for Inlab-Scheme). Note that Inlab-Scheme allows any 8 bit value being a character of a string and that the name of a symbol has the same properties.

4.265 string-append

(string-append <string> ...) returns a newly allocated string whose characters form the concatenation of the given strings.

(string-append) returns the empty string.

4.266 string-ci=?

(string-ci=? <string1> <string2>) works like *string=?* but treats upper and lower case letters as though they were the same characters (US ASCII).

4.267 string-ci>=?

string-ci>=? is the case insensitive variant of *string>=?*.

4.268 string-ci<=?

string-ci<=? is the case insensitive variant of *string<=?*.

4.269 string-ci<?

string-ci<? is the case insensitive variant of *string<?*.

4.270 string-ci>?

string-ci>? is the case insensitive variant of *string>?*.

4.271 string-copy

(string-copy <string>) returns a newly allocated copy of the given string.

4.272 string-fill!

(string-fill! string char) stores *<char>* in every element of the given *<string>* such modifying it and returns an unspecified value.

4.273 string-length

(string-length <string>) returns the number of characters in the given string.

4.274 string-ref

(string-ref <string> <k>) returns the character at the position *<k>* in *<string>* using zero-origin indexing.

4.275 string-set!

(string-set! <string> <k> <char>) stores *<char>* at position *<k>* in *<string>* and returns an unspecified value. Zero-origin indexing is used.

4.276 string=?

(string=? <string1> <string2>) returns *#t* if the two strings are of the same length and contain the same characters in the same positions, otherwise returns *#f*.

4.277 string?

(string? <obj>) returns *#t* is *<obj>* is a string, otherwise returns *#f*.

4.278 string<=?

(string<=? <string1> <string2>) returns *#t* if *<string1>* is lexicographic less than or equal to *<string2>* *#f* otherwise.

4.279 string>=?

(string>=? <string1> <string2>) returns *#t* if *<string1>* is lexicographic greater than or equal to *<string2>* *#f* otherwise.

4.280 string<?

(string<? <string1> <string2>) returns *#t* if *<string1>* is lexicographic less than *<string2>* *#f* otherwise.

4.281 string>?

(string>? <string1> <string2>) returns *#t* if *<string1>* is lexicographic greater than *<string2>* *#f* otherwise.

4.282 substring

(substring <string> <start> <end>) returns a newly allocated string formed from the characters of *<string>* beginning with index *<start>* (inclusive) and ending with index *<end>* (exclusive) using zero-origin indexing.

4.283 symbol->string (!)

(symbol->string <symbol>) returns the name of the symbol as a string. Since Inlab-Scheme delivers a copy of the actual name of the symbol in the returned string, mutation procedures as *string-set!* generate no error if applied to the string.

4.284 symbol?

(symbol? <obj>) returns *#t* if *<obj>* is a symbol, otherwise returns *#f*.

4.285 syntax-lambda (+)

The special form *(syntax-lambda (<expr> <env>) <body>)* creates an anonymous syntax-procedure the same way as *named-syntax-lambda* does with the difference that no name is specified. *syntax-lambda* is defined as follows:

```
(define syntax-lambda
  (named-syntax-lambda
    (syntax-lambda expr env)
     (cons
       'named-syntax-lambda
       (cons (cons '() (car (cdr expr)))
          (cdr (cdr expr))))))
```

4.286 syntax-procedure? (+)

(syntax-procedure? <obj>) returns *#t* if *<obj>* is a syntactic procedure, otherwise *#f*.

4.287 system (+)

(system <command>) Executes the supplied Unix command *<command>* which has to be a string and returns the Unix exit code as an integer.

(system) invokes a subshell and returns the Unix return code of this shell invocation as an integer.

4.288 tan

(tan <z>) computes the tangent of *<z>* (measured in radians). A large magnitude argument may yield a result with little or no significance.

4.289 the-current-editor (+)

This is a variable that contains the name (path) of the editor of choice for editing expressions. This variable is used by *(!)*, *(edit)*and *(load)*.

4.290 transcript-off (!)

transcript-off is not implemented in Inlab-Scheme.

4.291 transcript-on (!)

transcript-on is not implemented in Inlab-Scheme.

4.292 truncate

(truncate <x>) returns the integer closest to *<x>* whose absolute value is not larger than the absolute value of *<x>*.

4.293 vector

(vector <obj> ...) returns a newly allocated vector whose elements contain the given arguments.

(vector) returns the empty vector.

4.294 vector->list

(vector->list <vector>) returns a newly allocated list of the objects contained in the elements of *<vector>*.

4.295 vector-fill!

(vector-fill! <vector> <fill>) stores *<fill>* in each element of *<vector>*. The value returned is unspecified.

4.296 vector-length

(vector-length <vector>) returns the number od elements in *<vector>*.

4.297 vector-ref

(vector-ref <vector> <k>) returns the contents of element *<k>* of *<vector>* using zero-origin indexing.

4.298 vector-set!

(vector-set! <vector> <k> <fill>) stores *<fill>* in element *<k>* of *<vector>* using zero-origin indexing. The value returned is unspecified.

4.299 vector?

(vector? <obj>) returns *#t* if *<obj>* is a vector, otherwise *#f*.

4.300 with-input-from-file

(with-input-from-file <string> <thunk>) is implemented as specified in the R4RS calling *<thunk>* with the default input port "pointing" to the file specified in *<string>*.

4.301 with-output-to-file

(with-output-to-file <string> <thunk>) is implemented as specified in the R4RS calling *<thunk>* with the default output port "pointing" to the file specified in *<string>* (which is newly created or truncated if it already exists). All output directed to the redefined standard output port is finally collected in the file *<string>*.

4.302 write

(write <obj>) writes a written representation of *<obj>* to the current output port in a way that generally allows internalization of the written representation using *read*.

(write <obj> <port>) does the same writing to the given output port.

4.303 write-char

(write-char <char>) writes the given character to the current output port and returns an unspecified value.

(write-char <char> <port>) does the same writing to the given port.

4.304 zero?

(zero? <number>) returns *#t* if the given number is zero, *#f* otherwise. Using *zero?* on inexact arguments may be unreliable because a small inaccuracy may affect the result.

References

[1] Revisedˆ4 Report on the Algorithmic Language Scheme, November 1991.

[2] Draft Standard for the Scheme Programming Language P1178/D4, March 7, 1990.

[3] MIT-Scheme Reference, Scheme Release 7, Draft: October 18, 1988.

[4] Abelson, G.J. Sussman, J. Sussman: Structure and Interpretation of Computer Programs (1st edition), MIT Press 1985, ISBN 0-262-01077-1.

CONTENTS