# Table of contents

1 Introduction ........................................ 1
   1.1 Overview ........................................ 1
   1.2 Further reading ................................ 1
   1.3 Acknowledgments ................................. 1

2 LispKit Base ......................................... 2

3 LispKit Box ........................................... 3
   3.1 Boxes ........................................... 3
   3.2 Mutable pairs .................................. 3

4 LispKit Bytevector .................................. 4
   4.1 Basic ........................................... 4
   4.2 Advanced ....................................... 5
   4.3 Input/Output ................................... 6

5 LispKit Char-Set ................................... 7
   5.1 Constants ....................................... 7
   5.2 Predicates ...................................... 8
   5.3 Constructors ................................... 8
   5.4 Querying character sets ........................ 9
   5.5 Character set algebra ........................... 10
   5.6 Mutating character sets ......................... 10
   5.7 Iterating over character sets .................. 11

6 LispKit Char ......................................... 14
   6.1 Predicates ....................................... 14
   6.2 Transforming characters ......................... 15
   6.3 Converting characters ............................ 16

7 LispKit Control ..................................... 17
   7.1 Sequencing ...................................... 17
   7.2 Conditionals .................................... 17
   7.3 Local bindings .................................. 19
   7.4 Local syntax bindings ............................ 22
   7.5 Iteration ........................................ 23

8 LispKit Core ......................................... 25
   8.1 Basic primitives ................................ 25
   8.2 Definition primitives ............................ 26
   8.3 Importing definitions ............................ 28
   8.4 Delayed execution ................................ 29
   8.5 Symbols .......................................... 30
   8.6 Booleans ........................................ 31
   8.7 Conditional and inclusion compilation ......... 32
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.8</td>
<td>Multiple values</td>
<td>33</td>
</tr>
<tr>
<td>8.9</td>
<td>Environments</td>
<td>33</td>
</tr>
<tr>
<td>8.10</td>
<td>Syntax errors</td>
<td>34</td>
</tr>
<tr>
<td>8.11</td>
<td>Utilities</td>
<td>34</td>
</tr>
<tr>
<td>9</td>
<td>LispKit Datatype</td>
<td>35</td>
</tr>
<tr>
<td>9.1</td>
<td>Usage</td>
<td>35</td>
</tr>
<tr>
<td>9.2</td>
<td>API</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>LispKit Date-Time</td>
<td>38</td>
</tr>
<tr>
<td>10.1</td>
<td>Time zones</td>
<td>38</td>
</tr>
<tr>
<td>10.2</td>
<td>Date-times</td>
<td>39</td>
</tr>
<tr>
<td>11</td>
<td>LispKit Disjoint-Set</td>
<td>42</td>
</tr>
<tr>
<td>12</td>
<td>LispKit Draw</td>
<td>43</td>
</tr>
<tr>
<td>12.1</td>
<td>Drawings</td>
<td>43</td>
</tr>
<tr>
<td>12.2</td>
<td>Shapes</td>
<td>47</td>
</tr>
<tr>
<td>12.3</td>
<td>Images</td>
<td>49</td>
</tr>
<tr>
<td>12.4</td>
<td>Transformations</td>
<td>50</td>
</tr>
<tr>
<td>12.5</td>
<td>Colors</td>
<td>51</td>
</tr>
<tr>
<td>12.6</td>
<td>Fonts</td>
<td>52</td>
</tr>
<tr>
<td>12.7</td>
<td>Points</td>
<td>53</td>
</tr>
<tr>
<td>12.8</td>
<td>Size</td>
<td>54</td>
</tr>
<tr>
<td>12.9</td>
<td>Rects</td>
<td>54</td>
</tr>
<tr>
<td>13</td>
<td>LispKit Draw Turtle</td>
<td>56</td>
</tr>
<tr>
<td>14</td>
<td>LispKit Dynamic</td>
<td>58</td>
</tr>
<tr>
<td>14.1</td>
<td>Dynamic bindings</td>
<td>58</td>
</tr>
<tr>
<td>14.2</td>
<td>Continuations</td>
<td>59</td>
</tr>
<tr>
<td>14.3</td>
<td>Exceptions</td>
<td>60</td>
</tr>
<tr>
<td>14.4</td>
<td>Exiting</td>
<td>63</td>
</tr>
<tr>
<td>15</td>
<td>LispKit Enum</td>
<td>64</td>
</tr>
<tr>
<td>16</td>
<td>LispKit Gvector</td>
<td>67</td>
</tr>
<tr>
<td>16.1</td>
<td>Predicates</td>
<td>67</td>
</tr>
<tr>
<td>16.2</td>
<td>Constructors</td>
<td>67</td>
</tr>
<tr>
<td>16.3</td>
<td>Managing vector state</td>
<td>68</td>
</tr>
<tr>
<td>16.4</td>
<td>Destructive growable vector operations</td>
<td>69</td>
</tr>
<tr>
<td>16.5</td>
<td>Converting growable vectors</td>
<td>71</td>
</tr>
<tr>
<td>17</td>
<td>LispKit Hashtable</td>
<td>72</td>
</tr>
<tr>
<td>17.1</td>
<td>Constructors</td>
<td>72</td>
</tr>
<tr>
<td>17.2</td>
<td>Type tests</td>
<td>73</td>
</tr>
<tr>
<td>17.3</td>
<td>Inspection</td>
<td>73</td>
</tr>
<tr>
<td>17.4</td>
<td>Hash functions</td>
<td>74</td>
</tr>
<tr>
<td>17.5</td>
<td>Procedures</td>
<td>74</td>
</tr>
<tr>
<td>17.6</td>
<td>Composition</td>
<td>76</td>
</tr>
<tr>
<td>18</td>
<td>LispKit Heap</td>
<td>77</td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>19</td>
<td>LispKit List</td>
<td>78</td>
</tr>
<tr>
<td>19.1</td>
<td>Basic constructors and procedures</td>
<td>79</td>
</tr>
<tr>
<td>19.2</td>
<td>Predicates</td>
<td>80</td>
</tr>
<tr>
<td>19.3</td>
<td>Composing and transforming lists</td>
<td>80</td>
</tr>
<tr>
<td>19.4</td>
<td>Finding and extracting elements</td>
<td>83</td>
</tr>
<tr>
<td>20</td>
<td>LispKit Markdown</td>
<td>85</td>
</tr>
<tr>
<td>20.1</td>
<td>Data Model</td>
<td>85</td>
</tr>
<tr>
<td>20.1.1</td>
<td>Blocks</td>
<td>85</td>
</tr>
<tr>
<td>20.1.2</td>
<td>Inline Text</td>
<td>86</td>
</tr>
<tr>
<td>20.2</td>
<td>Creating Markdown documents</td>
<td>86</td>
</tr>
<tr>
<td>20.3</td>
<td>Processing Markdown documents</td>
<td>87</td>
</tr>
<tr>
<td>20.4</td>
<td>API</td>
<td>87</td>
</tr>
<tr>
<td>21</td>
<td>LispKit Math</td>
<td>90</td>
</tr>
<tr>
<td>21.1</td>
<td>Numerical constants</td>
<td>90</td>
</tr>
<tr>
<td>21.2</td>
<td>Predicates</td>
<td>90</td>
</tr>
<tr>
<td>21.3</td>
<td>Exactness and rounding</td>
<td>92</td>
</tr>
<tr>
<td>21.4</td>
<td>Operations</td>
<td>93</td>
</tr>
<tr>
<td>21.5</td>
<td>Division and remainder</td>
<td>95</td>
</tr>
<tr>
<td>21.6</td>
<td>Fractional numbers</td>
<td>97</td>
</tr>
<tr>
<td>21.7</td>
<td>Complex numbers</td>
<td>97</td>
</tr>
<tr>
<td>21.8</td>
<td>String representation</td>
<td>97</td>
</tr>
<tr>
<td>21.9</td>
<td>Bitwise operations</td>
<td>98</td>
</tr>
<tr>
<td>21.10</td>
<td>Fixnum operations</td>
<td>100</td>
</tr>
<tr>
<td>21.11</td>
<td>Floating-point operations</td>
<td>103</td>
</tr>
<tr>
<td>22</td>
<td>LispKit Object</td>
<td>105</td>
</tr>
<tr>
<td>22.1</td>
<td>Introduction</td>
<td>105</td>
</tr>
<tr>
<td>22.1.1</td>
<td>Generic procedures</td>
<td>105</td>
</tr>
<tr>
<td>22.1.2</td>
<td>Objects</td>
<td>105</td>
</tr>
<tr>
<td>22.1.3</td>
<td>Inheritance</td>
<td>106</td>
</tr>
<tr>
<td>22.1.4</td>
<td>Classes</td>
<td>106</td>
</tr>
<tr>
<td>22.2</td>
<td>Procedural object interface</td>
<td>107</td>
</tr>
<tr>
<td>22.3</td>
<td>Declarative object interface</td>
<td>108</td>
</tr>
<tr>
<td>22.4</td>
<td>Procedural class interface</td>
<td>108</td>
</tr>
<tr>
<td>22.4.1</td>
<td>Instance methods</td>
<td>108</td>
</tr>
<tr>
<td>22.4.2</td>
<td>Class methods</td>
<td>109</td>
</tr>
<tr>
<td>22.5</td>
<td>Declarative class interface</td>
<td>109</td>
</tr>
<tr>
<td>23</td>
<td>LispKit Port</td>
<td>110</td>
</tr>
<tr>
<td>23.1</td>
<td>Default ports</td>
<td>110</td>
</tr>
<tr>
<td>23.2</td>
<td>Predicates</td>
<td>110</td>
</tr>
<tr>
<td>23.3</td>
<td>General ports</td>
<td>111</td>
</tr>
<tr>
<td>23.4</td>
<td>File ports</td>
<td>111</td>
</tr>
<tr>
<td>23.5</td>
<td>String ports</td>
<td>112</td>
</tr>
<tr>
<td>23.6</td>
<td>Bytevector ports</td>
<td>113</td>
</tr>
<tr>
<td>23.7</td>
<td>URL ports</td>
<td>113</td>
</tr>
<tr>
<td>23.8</td>
<td>Asset ports</td>
<td>114</td>
</tr>
<tr>
<td>23.9</td>
<td>Reading from ports</td>
<td>114</td>
</tr>
<tr>
<td>23.10</td>
<td>Writing to ports</td>
<td>116</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>24</td>
<td>LispKit Record</td>
<td>119</td>
</tr>
<tr>
<td>24.1</td>
<td>Declarative API</td>
<td>119</td>
</tr>
<tr>
<td>24.2</td>
<td>Procedural API</td>
<td>120</td>
</tr>
<tr>
<td>25</td>
<td>LispKit Regexp</td>
<td>122</td>
</tr>
<tr>
<td>25.1</td>
<td>Regular expressions</td>
<td>122</td>
</tr>
<tr>
<td>25.1.1</td>
<td>Meta-characters</td>
<td>122</td>
</tr>
<tr>
<td>25.2</td>
<td>Regular expression operators</td>
<td>123</td>
</tr>
<tr>
<td>25.2.1</td>
<td>Template Matching</td>
<td>124</td>
</tr>
<tr>
<td>25.2.2</td>
<td>Flag options</td>
<td>124</td>
</tr>
<tr>
<td>25.3</td>
<td>API</td>
<td>124</td>
</tr>
<tr>
<td>26</td>
<td>LispKit Set</td>
<td>129</td>
</tr>
<tr>
<td>26.1</td>
<td>Constructors</td>
<td>129</td>
</tr>
<tr>
<td>26.2</td>
<td>Inspection</td>
<td>129</td>
</tr>
<tr>
<td>26.3</td>
<td>Predicates</td>
<td>130</td>
</tr>
<tr>
<td>26.4</td>
<td>Procedures</td>
<td>130</td>
</tr>
<tr>
<td>26.5</td>
<td>Mutators</td>
<td>131</td>
</tr>
<tr>
<td>27</td>
<td>LispKit String</td>
<td>132</td>
</tr>
<tr>
<td>27.1</td>
<td>Basic constructors and procedures</td>
<td>132</td>
</tr>
<tr>
<td>27.2</td>
<td>Predicates</td>
<td>133</td>
</tr>
<tr>
<td>27.3</td>
<td>Composing and extracting strings</td>
<td>134</td>
</tr>
<tr>
<td>27.4</td>
<td>Manipulating strings</td>
<td>136</td>
</tr>
<tr>
<td>27.5</td>
<td>Iterating over strings</td>
<td>136</td>
</tr>
<tr>
<td>27.6</td>
<td>Converting strings</td>
<td>137</td>
</tr>
<tr>
<td>27.7</td>
<td>Input/Output</td>
<td>137</td>
</tr>
<tr>
<td>28</td>
<td>LispKit System</td>
<td>138</td>
</tr>
<tr>
<td>28.1</td>
<td>Source files</td>
<td>138</td>
</tr>
<tr>
<td>28.2</td>
<td>File operations</td>
<td>138</td>
</tr>
<tr>
<td>28.3</td>
<td>Network operations</td>
<td>141</td>
</tr>
<tr>
<td>28.4</td>
<td>Time operations</td>
<td>141</td>
</tr>
<tr>
<td>28.5</td>
<td>Locales</td>
<td>142</td>
</tr>
<tr>
<td>28.6</td>
<td>Execution environment</td>
<td>143</td>
</tr>
<tr>
<td>28.7</td>
<td>Scheme environment</td>
<td>145</td>
</tr>
<tr>
<td>29</td>
<td>LispKit Type</td>
<td>148</td>
</tr>
<tr>
<td>29.1</td>
<td>Usage of the procedural API</td>
<td>148</td>
</tr>
<tr>
<td>29.2</td>
<td>Usage of the declarative API</td>
<td>149</td>
</tr>
<tr>
<td>29.3</td>
<td>API</td>
<td>151</td>
</tr>
<tr>
<td>30</td>
<td>LispKit Vector</td>
<td>153</td>
</tr>
<tr>
<td>30.1</td>
<td>Predicates</td>
<td>153</td>
</tr>
<tr>
<td>30.2</td>
<td>Constructors</td>
<td>154</td>
</tr>
<tr>
<td>30.3</td>
<td>Managing vector state</td>
<td>155</td>
</tr>
<tr>
<td>30.4</td>
<td>Destructive vector operations</td>
<td>156</td>
</tr>
<tr>
<td>30.5</td>
<td>Converting vectors</td>
<td>157</td>
</tr>
<tr>
<td>31</td>
<td>LispPad Speech</td>
<td>159</td>
</tr>
<tr>
<td>31.1</td>
<td>Speech synthesis</td>
<td>159</td>
</tr>
<tr>
<td>31.2</td>
<td>Speakers</td>
<td>159</td>
</tr>
<tr>
<td>31.3</td>
<td>Voices</td>
<td>161</td>
</tr>
</tbody>
</table>
# LispPad Library Reference

## 32 LispPad System

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.1 Windows</td>
<td>163</td>
</tr>
<tr>
<td>32.2 Edit Windows</td>
<td>164</td>
</tr>
<tr>
<td>32.3 Graphics Windows</td>
<td>164</td>
</tr>
<tr>
<td>32.4 Utilities</td>
<td>165</td>
</tr>
</tbody>
</table>

## 33 LispPad Turtle

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>167</td>
</tr>
</tbody>
</table>

## 34 SRFI Libraries

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>169</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Overview

LispPad is an integrated development environment for Scheme on macOS. LispPad’s Scheme interpreter is based on LispKit, a R7RS-compliant implementation of Scheme which comes with a large number of pre-packaged Scheme libraries. This document is a reference manual for the core Scheme libraries coming with LispKit. The LispPad homepage provides access to frequently updated online documentation.

1.2 Further reading

There are several books which can be recommended for learning Scheme and related topics:

- “The Scheme Programming Language” by R. Kent Dybvig provides a comprehensive introduction into Scheme based on R6RS. It discusses several advanced topics and covers many Scheme libraries.
- “Simply Scheme: Introducing Computer Science” by Brian Harvey and Matthew Wright introduces Scheme slowly to beginners.
- “Structure and Interpretation of Computer Programs” by Harold Abelson and Gerald Jay Sussman is the ultimate book teaching Computer Science, all in Scheme. The book covers a broad range of Computer Science topics and should be standing on every Scheme programmers desk.
- “Essentials of Programming Languages” by Daniel P Friedman and Mitchell Wand provides a deep understanding of the essential concepts of programming languages and uses Scheme as the language to implement the concepts.

1.3 Acknowledgments

Some of this documentation is derived from existing Scheme language specifications, such as the R5RS, the R6RS, and the R7RS standards. In recent years, these standards evolved using the SRFI process, which provides access to a large number of standardized Scheme components and libraries.

2 LispKit Base

Library (lispkit base) aggregates all exported values, parameter objects, and functions from the following libraries and re-exports them.

- (lispkit box)
- (lispkit bytevector)
- (lispkit char)
- (lispkit control)
- (lispkit core)
- (lispkit dynamic)
- (lispkit hashtable)
- (lispkit list)
- (lispkit math)
- (lispkit port)
- (lispkit record)
- (lispkit string)
- (lispkit system)
- (lispkit type)
- (lispkit vector)
3 LispKit Box

LispKit is a R7RS-compliant implementation with one exception: pairs are immutable. This library provides implementations of basic mutable data structures with reference semantics: mutable one-place buffers (also called boxes) and mutable pairs.

3.1 Boxes

(box? obj)  
Returns #t if obj is a box; #f otherwise.

(box obj)  
Returns a new box object that contains obj.

(unbox box)  
Returns the current content of box. This procedure fails if box is not referring to a box.

(set-box! box obj)  
Sets the content of box to obj. This procedure fails if box is not referring to a box.

(update-box! box proc)  
Invokes proc with the content of box and stores the result of this function invocation in box. update-box! is implemented like this:

(define (update-box! box proc)  
  (set-box! box (proc (unbox box))))

3.2 Mutable pairs

(mpair? obj)  
Returns #t if v is a mutable pair (mpair); #f otherwise.

(mcons car cdr)  
Returns a new mutable pair whose first element is set to car and whose second element is set to cdr.

(mcar mpair)  
Returns the first element of the mutable pair mpair.

(mcdr mpair)  
Returns the second element of the mutable pair mpair.

(set-mcar! mpair obj)  
Sets the first element of the mutable pair mpair to obj.

(set-mcdr! mpair obj)  
Sets the second element of the mutable pair mpair to obj.
4 LispKit Bytevector

Bytevectors represent blocks of binary data. They are fixed-length sequences of bytes, where a byte is a fixnum in the range from 0 to 255 inclusive. A bytevector is typically more space-efficient than a vector containing the same values.

The length of a bytevector is the number of elements that it contains. The length is a non-negative integer that is fixed when the bytevector is created. The valid indexes of a bytevector are the exact non-negative integers less than the length of the bytevector, starting at index zero as with vectors.

Bytevectors are written using the notation \#u8\(\text{byte \ ...}\) . For example, a bytevector of length 3 containing the byte 0 in element 0, the byte 10 in element 1, and the byte 5 in element 2 can be written as follows: \#u8\(0~10~5\) . Bytevector constants are self-evaluating, so they do not need to be quoted.

4.1 Basic

\((\text{bytevector? \ obj})\)

Returns \#t if \obj\ is a bytevector; otherwise, \#f is returned.

\((\text{bytevector \ byte \ ...})\)

Returns a newly allocated bytevector containing its arguments as bytes in the given order.

\(\text{(bytevector 1 3 5 1 3 5)} \Rightarrow \#u8(1~3~5~1~3~5)\)
\(\text{(bytevector)} \Rightarrow \#u8()\)

\((\text{make-bytevector \ k})\)
\((\text{make-bytevector \ k \ byte})\)

The \text{make-bytevector} procedure returns a newly allocated bytevector of length \k. If \text{byte} is given, then all elements of the bytevector are initialized to \text{byte}, otherwise the contents of each element are unspecified.

\(\text{(make-bytevector 3 12)} \Rightarrow \#u8(12~12~12)\)

\((\text{bytevector-length \ bytevector})\)

Returns the length of bytevector in bytes as an exact integer.

\((\text{bytevector-u8-ref \ bytevector \ k})\)

Returns the \k-th byte of \text{bytevector}. It is an error if \k is not a valid index of bytevector.

\(\text{(bytevector-u8-ref \#u8(1~1~2~3~5~8~13~21) \ 5)} \Rightarrow 8\)

\((\text{bytevector-u8-set! \ bytevector \ k \ byte})\)

Stores \text{byte} as the \k-th byte of bytevector. It is an error if \k is not a valid index of bytevector.
(let ((bv (bytevector 1 2 3 4)))
  (bytevector-u8-set! bv 1 3)
  bv)  ⇒ #u8(1 3 3 4)

(bytevector-copy bytevector)
(bytevector-copy bytevector start)
(bytevector-copy bytevector start end)

Returns a newly allocated bytevector containing the bytes in bytevector between start and end. If end is not provided, it is assumed to be the length of bytevector. If start is not provided, it is assumed to be 0.

(define a #u8(1 2 3 4 5))
(define b (bytevector 10 20 30 40 50))
(bytevector-copy! b a 0 2)
b  ⇒ #u8(10 1 2 40 50)

(bytevector-copy! to at from)
(bytevector-copy! to at from start)
(bytevector-copy! to at from start end)

Copies the bytes of bytevector from between start and end to bytevector to, starting at at. The order in which bytes are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary bytevector and then into the destination. This can be achieved without allocating storage by making sure to copy in the correct direction in such circumstances.

It is an error if at is less than zero or greater than the length of to. It is also an error if (~ (bytevector-length to) at) is less than (~ (end start)).

(define a (bytevector 1 2 3 4 5))
(define b (bytevector 10 20 30 40 50))
(bytevector-copy! b a 0 2)
b  ⇒ #u8(10 1 2 40 50)

(bytevector-append bytevector …)

Returns a newly allocated bytevector whose elements are the concatenation of the elements in the given bytevectors.

(bytevector-append #u8(0 1 2) #u8(3 4 5))  ⇒ #u8(0 1 2 3 4 5)

4.2 Advanced

(utf8->string bytevector)
(utf8->string bytevector start)
(utf8->string bytevector start end)
(string->utf8 string)
(string->utf8 string start)
(string->utf8 string start end)

These procedures translate between strings and bytevectors that encode those strings using the UTF-8 encoding. The utf8->string procedure decodes the bytes of a bytevector between start and end and returns the corresponding string. The string->utf8 procedure encodes the characters of a string between start and end and returns the corresponding bytevector.
It is an error for \texttt{bytevector} to contain invalid UTF-8 byte sequences.

\begin{itemize}
  \item \texttt{(utf8->string \#u8(#x41))} ⇒ \texttt{"A"}
  \item \texttt{(string->utf8 \"λ\")} ⇒ \texttt{\#u8(#xCE #xBB)}
\end{itemize}

\texttt{(bytevector->base64 bytevector)}
\texttt{(bytevector->base64 bytevector start)}
\texttt{(bytevector->base64 bytevector start end)}

\texttt{bytevector->base64} encodes \texttt{bytevector} between \texttt{start} and \texttt{end} as a string consisting of ASCII characters using the \textit{Base64} encoding scheme. If \texttt{end} is not provided, it is assumed to be the length of \texttt{bytevector}. If \texttt{start} is not provided, it is assumed to be 0.

\texttt{(base64->bytevector str)}
\texttt{(base64->bytevector str start)}
\texttt{(base64->bytevector str start end)}

\texttt{base64->bytevector} assumes string \texttt{str} is encoded using \textit{Base64} between \texttt{start} and \texttt{end} and returns a corresponding new decoded \texttt{bytevector}.

If is an error if \texttt{str} between \texttt{start} and \texttt{end} is not a valid \textit{Base64}-encoded string. If \texttt{end} is not provided, it is assumed to be the length of \texttt{str}. If \texttt{start} is not provided, it is assumed to be 0.

\texttt{(bytevector-deflate bytevector)}
\texttt{(bytevector-deflate bytevector start)}
\texttt{(bytevector-deflate bytevector start end)}

\texttt{bytevector-deflate} encodes \texttt{bytevector} between \texttt{start} and \texttt{end} using the \textit{Deflate} data compression algorithm returning a new compressed \texttt{bytevector}. If \texttt{end} is not provided, it is assumed to be the length of \texttt{bytevector}. If \texttt{start} is not provided, it is assumed to be 0.

\texttt{(bytevector-inflate bytevector)}
\texttt{(bytevector-inflate bytevector start)}
\texttt{(bytevector-inflate bytevector start end)}

\texttt{bytevector-inflate} assumes \texttt{bytevector} is encoded using the \textit{Deflate} data compression algorithm between \texttt{start} and \texttt{end}. The procedure returns a corresponding new decoded \texttt{bytevector}.

If is an error if \texttt{bytevector}, between \texttt{start} and \texttt{end}, is not encoded using \textit{Deflate}. If \texttt{end} is not provided, it is assumed to be the length of \texttt{bytevector}. If \texttt{start} is not provided, it is assumed to be 0.

\section*{4.3 Input/Output}

\texttt{(read-binary-file path)}

\texttt{Reads the file at path and stores its content in a new bytevector which gets returned by read-binary-file.}

\texttt{(write-binary-file path bytevector)}
\texttt{(write-binary-file path bytevector start)}
\texttt{(write-binary-file path bytevector start end)}

\texttt{Writes the bytes of bytevector between start and end into a new binary file at path. If end is not provided, it is assumed to be the length of bytevector. If start is not provided, it is assumed to be 0.}
5 LispKit Char-Set

Library \texttt{(lispkit char-set)} implements efficient means to represent and manipulate sets of characters. Its design is based on SRFI 14 but the implementation is specific to the definition of characters in LispKit; i.e. library \texttt{(lispkit char-set)} assumes that characters are UTF-16 code units.

As opposed to SRFI 14, it can be assumed that the update procedures ending with “!” are mutating the corresponding character set. This means that clients of these procedures may rely on these procedures performing their functionality in terms of side effects.

In the procedure specifications below, the following conventions are used:

- A \texttt{cs} parameter is a character set.
- A \texttt{s} parameter is a string.
- A \texttt{char} parameter is a character.
- A \texttt{char-list} parameter is a list of characters.
- A \texttt{pred} parameter is a unary character predicate procedure, returning either \texttt{#t} or \texttt{#f} when applied to a character.
- An \texttt{obj} parameter may be any value at all.

Passing values to procedures with these parameters that do not satisfy these types is an error.

Unless otherwise noted in the specification of a procedure, procedures always return character sets that are distinct from the parameter character sets (unless the procedure mutates a character set and its name ends with “!”). For example, \texttt{char-set-adjoin} is guaranteed to provide a fresh character set, even if it is not given any character parameters.

Library \texttt{(lispkit char-set)} supports both mutable as well as immutable character sets. Character sets are assumed to be mutable unless they are explicitly specified to be immutable.

5.1 Constants

\begin{verbatim}
char-set:lower-case
char-set:upper-case
char-set:title-case
char-set:letter
char-set:digit
char-set:letter+digit
char-set:graphic
char-set:printing
char-set:whitespace
char-set:newlines
char-set:iso-control
char-set:punctuation
char-set:symbol
char-set:hex-digit
char-set:blank
\end{verbatim}
Library (lispkit char-set) predefines these frequently used immutable character sets.

Note that there may be characters in char-set:letter that are neither upper or lower case. The char-set:whitespaces character set contains whitespace and newline characters. char-set:blanks only contains whitespace (i.e. “blank”) characters. char-set:newlines only contains newline characters.

5.2 Predicates

(char-set? obj)
Returns #t if obj is a character set, otherwise returns #f.

(char-set-empty? cs)
Returns #t if the character set cs does not contain any characters, otherwise returns #f.

(char-set=? cs ...)
Returns #t if all the provided character sets cs ... contain the exact same characters; returns #f otherwise. For both corner cases, (char-set=? ) and (char-set=? cs ), char-set=? returns #t.

(char-set<=? cs ...)
Returns #t if every character set cs-i is a subset of character set cs-i+1; returns #f otherwise. For both corner cases, (char-set<=? ) and (char-set<=? cs ), char-set<=? returns #t.

(char-set-disjoint? cs1 cs2)
Returns #t if character sets cs1 and cs2 are disjoint, i.e. they do not share a single character; returns #f otherwise.

(char-set-contains? cs char)
Returns #t if character char is contained in character set cs; returns #f otherwise.

(char-set-every? pred cs)
(char-set-any? pred cs)
The char-set-every? procedure returns #t if predicate pred returns #t for every character in the character set cs. Likewise, char-set-any? applies pred to every character in character set cs, and returns #t if there is at least one character for which pred returns #t. If no character produces a #t value, it returns #f. The order in which these procedures sequence through the elements of cs is not specified.

5.3 Constructors

(char-set char ...)
Return a newly allocated mutable character set containing the given characters.

(immutable-char-set char ...)
Return a newly allocated immutable character set containing the given characters.

(char-set-copy cs)
(char-set-copy cs mutable?)
Returns a newly allocated copy of the character set cs. The copy is mutable by default unless parameter mutable? is provided and set to #f.
(list->char-set char-list)
(list->char-set char-list base-cs)

Return a newly allocated mutable character set containing the characters in the list of characters char-list. If character set base-cs is provided, the characters from base-cs are added to it as well.

(string->char-set s)
(string->char-set s base-cs)

Return a newly allocated mutable character set containing the characters of the string s. If character set base-cs is provided, the characters from base-cs are added to it as well.

(ucs-range->char-set lower upper)
(ucs-range->char-set lower upper base-cs)
(ucs-range->char-set lower upper limit base-cs)

Returns a newly allocated mutable character set containing every character whose ISO/IEC 10646 UCS-4 code lies in the half-open range [lower,upper). lower and upper are exact non-negative integers where lower <= upper <= limit is required to hold. limit is either an exact non-negative integer specifying the maximum upper limit, or it is #t which specifies the maximum UTF-16 code unit value. If limit is not provided, a very large default is assumed (equivalent to limit being #f).

This signature is compatible with the SRFI 16 specification which states that if the requested range includes unassigned UCS values, these are silently ignored. If the requested range includes “private” or “user space” codes, these are passed through transparently. If any code from the requested range specifies a valid, assigned UCS character that has no corresponding representative in the implementation’s character type, then

1. an error is raised if limit is #t, and
2. the code is ignored if limit is #f (the default).

If character set base-cs is provided, the characters of base-cs are included in the newly allocated mutable character set.

(->char-set x)

Coerces object x into a character set. x may be a string, character or character set. A string is converted to the set of its constituent characters; a character is converted to a singleton character set; a character set is returned as is. This procedure is intended for use by other procedures that want to provide “user-friendly”, wide-spectrum interfaces to their clients.

5.4 Querying character sets

(char-set-size cs)

Returns the number of elements in character set cs.

(char-set-count pred cs)

Apply pred to the characters of character set cs, and return the number of characters that caused the predicate to return #t.

(char-set->list cs)

This procedure returns a list of the characters of character set cs. The order in which cs's characters appear in the list is not defined, and may be different from one call to another.

(char-set->string cs)

This procedure returns a string containing the characters of character set cs. The order in which cs's characters appear in the string is not defined, and may be different from one call to another.
(char-set-hash cs)
(char-set-hash cs bound)

Compute a hash value for the character set cs. bound is a non-negative exact integer specifying the range of the hash function. A positive value restricts the return value to the range \(0, \text{bound}\). If bound is either zero or not given, a default value is used, chosen to be as large as it is efficiently practical.

### 5.5 Character set algebra

**procedure**

(char-set-adjoin cs char …)

Return a newly allocated mutable copy of cs into which the characters char … were inserted.

**procedure**

(char-set-delete cs char …)

Return a newly allocated mutable copy of cs from which the characters char … were removed.

**procedure**

(char-set-complement cs)

Return a newly allocated character set containing all characters that are not contained in cs.

**procedure**

(char-set-union cs …)

(char-set-intersection cs …)

(char-set-difference cs …)

(char-set-xor cs …)

(char-set-diff+intersection cs1 cs2 …)

These procedures implement set complement, union, intersection, difference, and exclusive-or for character sets. The union, intersection and xor operations are n-ary. The difference function is also n-ary, associates to the left (that is, it computes the difference between its first argument and the union of all the other arguments), and requires at least one argument.

Boundary cases:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(char-set-union)</td>
<td>char-set:empty</td>
</tr>
<tr>
<td>(char-set-intersection)</td>
<td>char-set:full</td>
</tr>
<tr>
<td>(char-set-xor)</td>
<td>char-set:empty</td>
</tr>
<tr>
<td>(char-set-difference cs)</td>
<td>cs</td>
</tr>
</tbody>
</table>

(char-set-diff+intersection) returns both the difference and the intersection of the arguments, i.e. it partitions its first parameter. It is equivalent to \(\text{(values } \text{(char-set-difference } cs1 \text{ cs2 } \ldots) \text{ (char-set-intersection } cs1 \text{ (char-set-union } cs2 \ldots)\})\) but can be implemented more efficiently.

**procedure**

(char-set-filter pred cs)

(char-set-filter pred cs base-cs)

Returns a new character set containing every character c in cs such that \((\text{pred } c)\) returns \#t. If character set base-cs is provided, the characters specified by pred are added to a copy of it.

### 5.6 Mutating character sets

**procedure**

(char-set-adjoin! cs char …)

Insert the characters char … into the character set cs.

**procedure**

(char-set-delete! cs char …)

Remove the characters char … from the character set cs.
(char-set-complement! cs)  
Complement the character set cs by including all characters that were not contained in cs previously and by removing all previously contained characters.

(char-set-union! cs1 cs2 ...)  
(char-set-intersection! cs1 cs2 ...)  
(char-set-difference! cs1 cs2 ...)  
(char-set-xor! cs1 cs2 ...)  
(char-set-diff+intersection! cs1 cs2 cs3 ...)  

These are update variants of the set-algebra functions mutating the first character set cs1 instead of creating a new one. char-set-diff+intersection! will perform a side-effect on both of its two required parameters cs1 and cs2.

(char-set-filter! pred cs base-cs)  
Adds every character c in cs for which (pred c) returns #t to the given character set base-cs.

(list->char-set! char-list base-cs)  
Add the characters from the character list char-list to character set base-cs and return the mutated character set base-cs.

(string->char-set! s base-cs)  
Add the characters of the string s to character set base-cs and return the mutated character set base-cs.

(ucs-range->char-set! lower upper base-cs)  
(ucs-range->char-set! lower upper limit base-cs)  

Mutates the mutable character set base-cs including every character whose ISO/IEC 10646 UCS-4 code lies in the half-open range (lower,upper). lower and upper are exact non-negative integers where lower \(<=\) upper \(<=\) limit is required to hold. limit is either an exact non-negative integer specifying the maximum upper limit, or it is #t which specifies the maximum UTF-16 code unit value. If limit is not provided, a very large default is assumed (equivalent to limit being #f).

(char-set-unfold! f p g seed base-cs)  
This is a fundamental constructor for character sets.

- g is used to generate a series of “seed” values from the initial seed: seed, (g seed), (g2 seed), (g3 seed), ...
- p tells us when to stop by returning #t when applied to one of these seed values.
- f maps each seed value to a character. These characters are added to the base character set base-cs to form the result. char-set-unfold! adds the characters by mutating base-cs as a side effect.

5.7 Iterating over character sets

(char-set-cursor cs)  
(char-set-ref cs cursor)  
(char-set-cursor-next cs cursor)  
(end-of-char-set? cursor)  

Cursors are a low-level facility for iterating over the characters in a character set cs. A cursor is a value that indexes a character in a character set. char-set-cursor produces a new cursor for a given character set. The set element indexed by the cursor is fetched with char-set-ref. A cursor index is incremented with char-set-cursor-next; in this way, code can step through every character in a character set. Stepping a cursor “past the end” of a character set produces a cursor that answers true to end-of-char-set?. It is an error to pass such a cursor to char-set-ref or to char-set-cursor-next.
A cursor value may not be used in conjunction with a different character set; if it is passed to `char-set-ref` or `char-set-cursor-next` with a character set other than the one used to create it, the results and effects are undefined. These primitives are necessary to export an iteration facility for character sets to loop macros.

```
(define cs (char-set \G \t \a \e \c \h))

;; Collect els of CS into a list.
(let lp ((cur (char-set-cursor cs)) (ans '()))
  (if (end-of-char-set? cur) ans
    (lp (char-set-cursor-next cs cur)
      (cons (char-set-ref cs cur) ans))))

;; Equivalently, using a list unfold (from SRFI 1):
(unfold-right end-of-char-set? (curry char-set-ref cs) (curry char-set-cursor-next cs) (char-set-cursor cs))
```

**(char-set-fold kons knil cs)**

This is the fundamental iterator for character sets. Applies the function `kons` across the character set `cs` using initial state value `knil`. That is, if `cs` is the empty set, the procedure returns `knil`. Otherwise, some element `c` of `cs` is chosen; let `cs'` be the remaining, unchosen characters. The procedure returns `(char-set-fold kons (kons c knil) cs')`.

```
(l lambda (cs) (char-set-fold cons '() cs))

; CHAR-SET-SIZE
(lambda (cs) (char-set-fold (lambda (c i) (+ i 1)) 0 cs))

; How many vowels in the char set?
(lambda (cs)
  (char-set-fold (lambda (c i) (if (vowel? c) (+ i 1) i)) 0 cs))
```

**(char-set-unfold f p g seed)**

**(char-set-unfold f p g seed base-cs)**

This is a fundamental constructor for character sets.

- `g` is used to generate a series of “seed” values from the initial seed: seed, (g seed), (g2 seed), (g3 seed), ...
- `p` tells us when to stop, when it returns #t when applied to one of these seed values.
- `f` maps each seed value to a character. These characters are added to a mutable copy of the base character set `base-cs` to form the result; `base-cs` defaults to an empty set.

More precisely, the following definitions hold, ignoring the optional-argument issues:

```
(define (char-set-unfold p f g seed base-cs)
  (char-set-unfold! p f g seed (char-set-copy base-cs)))

(define (char-set-unfold! p f g seed base-cs)
  (let lp ((seed seed) (cs base-cs))
    (if (p seed) cs ; P says we are done
      (lp (g seed) ; Loop on (G SEED)
        (char-set-adjoin! cs (f seed))))))
```

**Examples:**
(port->char-set p) = (char-set-unfold \texttt{eof-object?} \texttt{values} (lambda (x) (read-char p)) (read-char p))

(list->char-set lis) = (char-set-unfold \texttt{null? car cdr lis})

\textbf{(char-set-for-each proc cs)}

Apply procedure \textit{proc} to each character in the character set \textit{cs}. Note that the order in which \textit{proc} is applied to the characters in the set is not specified, and may even change from one procedure application to another.

\textbf{(char-set-map proc cs)}

\textit{proc} is a procedure mapping characters to characters. It will be applied to all the characters in the character set \textit{cs}, and the results will be collected in a newly allocated mutable character set which will be returned by \texttt{char-set-map}.
6 LispKit Char

Characters are objects that represent printed characters such as letters and digits. In LispKit, characters are UTF-16 code units.

Character literals are written using the notation \char, or \character-name, or \x hex-scalar-value. Characters written using this \ notation are self-evaluating, i.e. they do not have to be quoted.

The following standard character names are supported by LispKit:

- \alarm: U+0007
- \backspace: U+0008
- \delete: U+007F
- \escape: U+001B
- \newline: the linefeed character U+000A
- \null: the null character U+0000
- \return: the return character U+000D
- \space: the space character U+0020
- \tab: the tab character U+0009

Here are some examples using the \ notation:

- \m: lowercase letter 'm'
- \M: uppercase letter 'M'
- \( : left parenthesis ')
- \\: backslash '
- \: space character '
- \x03BB: the lambda character

Case is significant in \ character, and in \character-name, but not in \x hex-scalar-value. If character in \ character is alphabetic, then any character immediately following character cannot be one that can appear in an identifier. This rule resolves the ambiguous case where, for example, the sequence of characters \space could be taken to be either a representation of the space character or a representation of the character \s followed by a representation of the symbol pace.

Some of the procedures that operate on characters ignore the difference between upper case and lower case. The procedures that ignore case have “-ci” (for “case insensitive”) embedded in their names.

6.1 Predicates

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

(char=? char ...)  
(char<? char ...)  
(char>? char ...)  
(char<=? char ...)  
(char>=? char ...)

14
These procedures return `#t` if the results of passing their arguments to `char->integer` are respectively equal, monotonically increasing, monotonically decreasing, monotonically non-decreasing, or monotonically non-increasing. These predicates are transitive.

```lisp
(char-ci=? char ...)  
(char-ci<? char ...)  
(char-ci>? char ...)  
(char-ci<=? char ...)  
(char-ci>=? char ...)
```

These procedures are similar to `char=?` etc., but they treat upper case and lower case letters as the same. For example, `(char-ci=? #\A #\a)` returns `#t`. Specifically, these procedures behave as if `char-foldcase` were applied to their arguments before they were compared.

```lisp
(char-alphabetic? char)  
Procedure char-alphabetic? returns `#t` if its argument is an alphabetic character, otherwise it returns `#f`. Note that many Unicode characters are alphabetic but neither upper nor lower case.

(char-numeric? char)  
Procedure char-numeric? returns `#t` if its argument is a numeric character, otherwise it returns `#f`.

(char-whitespace? char)  
Procedure char-whitespace? returns `#t` if its argument is a whitespace character, otherwise it returns `#f`.

(char-upper-case? char)  
Procedure char-upper-case? returns `#t` if its argument is an upper-case character, otherwise it returns `#f`.

(char-lower-case? char)  
Procedure char-lower-case? returns `#t` if its argument is a lower-case character, otherwise it returns `#f`.

### 6.2 Transforming characters

```lisp
(char-upcase char)  
The char-upcase procedure, given an argument that is the lowercase part of a Unicode casing pair, returns the uppercase member of the pair, provided that both characters are supported by LispKit. Note that language-sensitive casing pairs are not used. If the argument is not the lowercase member of such a pair, it is returned.

(char-downcase char)  
The char-downcase procedure, given an argument that is the uppercase part of a Unicode casing pair, returns the lowercase member of the pair, provided that both characters are supported by LispKit. If the argument is not the uppercase member of such a pair, it is returned.

(char-foldcase char)  
The char-foldcase procedure applies the Unicode simple case-folding algorithm to its argument and returns the result. Note that language-sensitive folding is not used. If the argument is an uppercase letter, the result will be either a lowercase letter or the same as the argument if the lowercase letter does not exist or is not supported by LispKit.
6.3 Converting characters

Procedure \texttt{digit-value} returns the numeric value (0 to 9) of its argument if it is a numeric digit (that is, if \texttt{char-numeric?} returns \texttt{#t}), or \texttt{#f} on any other character.

\begin{verbatim}
(digit-value \#\3) ⇒ 3
(digit-value \#\x0EA6) ⇒ #f
\end{verbatim}

Procedure \texttt{char->integer} returns an exact integer between 0 and \texttt{#xD7FF} or between \texttt{#xE000} and \texttt{#x10FFFF} which is equal to the Unicode scalar value of that character. Given a non-Unicode character, it returns an exact integer greater than \texttt{#x10FFFF}.

Given an exact integer that is the value returned by a character when \texttt{char->integer} is applied to it, \texttt{integer->char} returns that character.
7 LispKit Control

7.1 Sequencing

\[(\text{begin } \text{expr} \ldots \text{exprn})\]

\text{begin} evaluates \text{expr}, \ldots, \text{exprn} sequentially from left to right. The values of the last expression \text{exprn} are returned. This special form is typically used to sequence side effects such as assignments or input and output.

7.2 Conditionals

\[(\text{if } \text{test consequent})\]

\[(\text{if } \text{test consequent alternate})\]

An if expression is evaluated as follows: first, expression test is evaluated. If it yields a true value, then expression consequent is evaluated and its values are returned. Otherwise, alternate is evaluated and its values are returned. If expression test yields a false value and no alternate expression is specified, then the result of the expression is void.

\[
\begin{align*}
& (\text{if } (> 3 2) '\text{yes }'\text{no}) \Rightarrow \text{yes} \\
& (\text{if } (> 2 3) '\text{yes }'\text{no}) \Rightarrow \text{yes} \\
& (\text{if } (> 3 2) (- 3 2) (+ 3 2)) \Rightarrow 1
\end{align*}
\]

\[(\text{when test consequent } \ldots)\]

The test expression is evaluated, and if it evaluates to a true value, the expressions consequent \ldots are evaluated in order. The result of the when expression is the value to which the last consequent expression evaluates or void if test evaluates to false.

\[
\begin{align*}
& (\text{when } (= 1 1.0)) \\
& \quad (\text{display } "1") \\
& \quad (\text{display } "2") \Rightarrow (\text{void}), \text{prints: 12}
\end{align*}
\]

\[(\text{unless test alternate } \ldots)\]

The test is evaluated, and if it evaluates to false, the expressions alternate \ldots are evaluated in order. The result of the unless expression is the value to which the last consequent expression evaluates or void if test evaluates to a true value.

\[
\begin{align*}
& (\text{unless } (= 1 1.0)) \\
& \quad (\text{display } "1") \\
& \quad (\text{display } "2") \Rightarrow (\text{void}), \text{prints nothing}
\end{align*}
\]

\[(\text{cond clause1 clause2 } \ldots)\]

Clauses like clause1 and clause2 take one of two forms, either

- \((\_\_\_\text{test expr1 }\ldots\_\_\_)\), or
• (_test_ => _expr_)  

The last clause in a cond expression can be an “else clause”, which has the form

• (else _expr1 expr2 ..._)

A cond expression is evaluated by evaluating the test expressions of successive clauses in order until one of them evaluates to a true value. When a test expression evaluates to a true value, the remaining expressions in its clause are evaluated in order, and the results of the last expression are returned as the results of the entire cond expression.

If the selected clause contains only the test and no expressions, then the value of the test expression is returned as the result of the cond expression. If the selected clause uses the => alternate form, then the expression is evaluated. It is an error if its value is not a procedure that accepts one argument. This procedure is then called on the value of the test and the values returned by this procedure are returned by the cond expression.

If all tests evaluate to #f, and there is no else clause, then the result of the conditional expression is void. If there is an else clause, then its expressions are evaluated in order, and the values of the last one are returned.

\[
\text{(cond ((> 3 2) 'greater)  
((< 3 2) 'less)) ⇒ greater  
(cond ((> 3 3) 'greater)  
((< 3 3) 'less)  
(else 'equal)) ⇒ equal  
(cond ((assv 'b '((a 1) (b 2))) => cadr)  
(else #f)) ⇒ 2)
\]

(case key clause1 clause2 …)

key can be any expression. Each clause clause1, clause2, ... has the form:

• (_datum1 ..._ _expr1 expr2 ..._)

where each datum is an external representation of some object. It is an error if any of the datums are the same anywhere in the expression. Alternatively, a clause can be of the form:

• (_datum1 ..._ => _expr_)

The last clause in a case expression can be an “else clause”, which has one of the following forms:

• (else _expr1 expr2 ..._), or
• (else => _expr_)

A case expression is evaluated as follows. Expression key is evaluated and its result is compared against each datum. If the result of evaluating key is the same, in the sense of eqv?, to a datum, then the expressions in the corresponding clause are evaluated in order and the results of the last expression in the clause are returned as the results of the case expression.

If the result of evaluating key is different from every datum, then if there is an else clause, its expressions are evaluated and the results of the last are the results of the case expression. Otherwise, the result of the case expression is void.

If the selected clause or else clause uses the => alternate form, then the expression is evaluated. It is an error, if its value is not a procedure accepting one argument. This procedure is then called on the value of the key and the values returned by this procedure are returned by the case expression.
7.3 Local bindings

The binding constructs `let`, `let*`, `letrec`, `letrec*`, `let-values`, and `let*-values` give Scheme a block structure. The syntax of the first four constructs is identical, but they differ in the regions they establish for their variable bindings. In a `let` expression, the initial values are computed before any of the variables become bound. In a `let*` expression, the bindings and evaluations are performed sequentially. While in `letrec` and `letrec*` expressions, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions. The `let-values` and `let*-values` constructs are analogous to `let` and `let*` respectively, but are designed to handle multiple-valued expressions, binding different identifiers to the returned values.

\[
\text{(let } (x \ 2) (y \ 3) \ \text{(* } x \ y)) \Rightarrow 6
\]

\[
\text{(let* } (x \ 2) (y \ 3) \ \text{let } (x \ 7) \ (z \ (\text{+ } x \ y))) \ \text{(* } z \ x)) \Rightarrow 35
\]

\[
\text{(letrec } (x \ 2) (y \ 3) \ \text{let* } (x \ 7) \ (z \ (\text{+ } x \ y))) \ \text{(* } z \ x)) \Rightarrow 70
\]
of zero or more definitions followed by a sequence of one or more expressions. It is an error for variable to appear more than once in the list of variables being bound.

The variables are bound to fresh locations holding unspecified values, the init expressions are evaluated in the resulting environment, each variable is assigned to the result of the corresponding init expression, the body is evaluated in the resulting environment, and the values of the last expression in body are returned. Each binding of a variable has the entire \texttt{letrec} expression as its scope, making it possible to define mutually recursive procedures.

\begin{verbatim}
(letrec ((even? (lambda (n)
  (if (zero? n) #t (odd? (- n 1)))))
  (odd? (lambda (n)
    (if (zero? n) #f (even? (- n 1)))))
  (even? 88)) ⇒ #t
\end{verbatim}

One restriction of \texttt{letrec} is very important: if it is not possible to evaluate each init expression without assigning or referring to the value of any variable, it is an error. The restriction is necessary because letrec is defined in terms of a procedure call where a \texttt{lambda} expression binds the variables to the values of the init expressions. In the most common uses of \texttt{letrec}, all the init expressions are lambda expressions and the restriction is satisfied automatically.

\begin{verbatim}
(letrec* ((p (lambda (x)
  (+ 1 (q (- x 1)))))
  (q (lambda (y)
    (if (zero? y) 0 (+ 1 (p (- y 1)))))
  (x (p 5))
  (y x))
y) ⇒ 5
\end{verbatim}

\begin{verbatim}
(let-values bindings
body)

bindings has the form (( variable init ...), where each init is an expression, and body is a sequence of zero or more definitions followed by a sequence of one or more expressions. It is an error for variable to appear more than once in the list of variables being bound.

The variables are bound to fresh locations, each variable is assigned in left-to-right order to the result of evaluating the corresponding init expression, the body is evaluated in the resulting environment, and the values of the last expression in body are returned. Despite the left-to-right evaluation and assignment order, each binding of a variable has the entire \texttt{letrec*} expression as its region, making it possible to define mutually recursive procedures. If it is not possible to evaluate each init expression without assigning or referring to the value of any variable or the variable of any of the bindings that follow it in bindings, it is an error. Another restriction is that it is an error to invoke the continuation of an init expression more than once.

\begin{verbatim}
(let-values ((p (lambda (x)
  (+ 1 (q (- x 1)))))
  (q (lambda (y)
    (if (zero? y) 0 (+ 1 (p (- y 1)))))
  (x (p 5))
  (y x))
y) ⇒ 5
\end{verbatim}

\begin{verbatim}
(let-values bindings
body)

bindings has the form (( variables init ...) , where each variables is a list of variables, init is an expression, and body is zero or more definitions followed by a sequence of one or more expressions. It is an error for a variable to appear more than once in variables.

The init expressions are evaluated in the current environment as if by invoking \texttt{call-with-values}, and the variables occurring in list variables are bound to fresh locations holding the values returned by the init expressions, where the variables are matched to the return values in the same way that the variables in a lambda expression are matched to the arguments in a procedure call. Then, body is evaluated in the extended environment, and the values of the last expression of body are returned. Each binding of a variable has body as its scope.

It is an error if the variables in list variables do not match the number of values returned by the corresponding init expression.
(let-values (((root rem) (exact-integer-sqrt 32)))
(* root rem)) ⇒ 35

(let*-values (bindings body)
  syntax
  bindings has the form ((formals init) ...) , where each formals is a list of variables, init is an expression, and body is zero or more definitions followed by a sequence of one or more expressions. It is an error for a variable to appear more than once.

  The let*-values construct is similar to let-values, but the init expressions are evaluated and bindings created sequentially from left to right, with the region of the bindings of each variable including the init expressions to its right as well as body. Thus the second init expression is evaluated in an environment in which the first set of bindings is visible and initialized, and so on.

  (let ((a 'a) (b 'b) (x 'x) (y 'y))
    (let*-values (((a b) (values x y))
                  ((x y) (values a b))
                  (list a b x y))) ⇒ (x y x y)

(let-optionals args (var default) ...) body ...)
  syntax
  This binding construct can be used to handle optional arguments of procedures. args refers to the rest parameter list of a procedure or lambda expression. let-optionals binds the variables var, ... to the arguments available in args, or to default, ... if there are not enough arguments provided in args. Variables are bound in parallel, i.e. all default expressions are evaluated in the current environment in which the new variables are not bound yet. Then, body is evaluated in the extended environment including all variable definitions of let-optionals, and the values of the last expression of body are returned. Each binding of a variable has body as its scope.

  (let-optionals '("one" "two")
    ((one 1) (two 2) (three 3))
    (list one two three)) ⇒ ("one" "two" 3)

(let*-optionals args (var default) ...) body ...)
  syntax
  The let*-optionals construct is similar to let-optionals, but the default expressions are evaluated and bindings created sequentially from left to right, with the scope of the bindings of each variable including the default expressions to its right as well as body. Thus the second default expression is evaluated in an environment in which the first binding is visible and initialized, and so on.

  (let*-optionals '(!10 20)
    ((one 1) (two (+ one 1)) (three (+ two 1)))
    (list one two three)) ⇒ (10 20 21)

(let-keywords args (binding ...) body ...)
  syntax
  binding has one of two forms:
  • (var default), and
  • (var keyword default)

  where var is a variable, keyword is a symbol, and default is an expression. It is an error for a variable var to appear more than once.

  This binding construct can be used to handle keyword arguments of procedures. args refers to the rest parameter list of a procedure or lambda expression. let-keywords binds the variables var, ... by name, i.e. by searching in args for the keyword argument. If an optional keyword is provided, it is used as the
name of the keyword to search for, otherwise, var is used, appending :. If the keyword is not found in args, var is bound to default.

Variables are bound in parallel, i.e. all default expressions are evaluated in the current environment in which the new variables are not bound yet. Then, body is evaluated in the extended environment including all variable definitions of let-keywords, and the values of the last expression of body are returned. Each binding of a variable has body as its scope.

```
(define (make-person . args)
  (let-keywords args ((name "J. Doe")
    (age 0)
    (occupation job: 'unknown))
    (list name age occupation)))
```

```
(let-keywords args (binding ...)) body ...

binding has one of two forms:

- (var default), and
- (var keyword default)

where var is a variable, keyword is a symbol, and default is an expression. It is an error for a variable var to appear more than once.

The let*-keywords construct is similar to let-keywords, but the default expressions are evaluated and bindings created sequentially from left to right, with the scope of the bindings of each variable including the default expressions to its right as well as body. Thus the second default expression is evaluated in an environment in which the first binding is visible and initialized, and so on.

### 7.4 Local syntax bindings

The let-syntax and letrec-syntax binding constructs are analogous to let and letrec, but they bind syntactic keywords to macro transformers instead of binding variables to locations that contain values. Syntactic keywords can also be bound globally or locally with define-syntax.

```
(let-syntax bindings body)
```

bindings has the form ((keyword transformer) ...). Each keyword is an identifier, each transformer is an instance of syntax-rules, and body is a sequence of one or more definitions followed by one or more expressions. It is an error for a keyword to appear more than once in the list of keywords being bound.

body is expanded in the syntactic environment obtained by extending the syntactic environment of the let-syntax expression with macros whose keywords are the keyword symbols bound to the specified transformers. Each binding of a keyword has body as its scope.
**(letrec-syntax bindings body)**

bindings has the form `((keyword transformer) ...)`. Each keyword is an identifier, each transformer is an instance of `syntax-rules`, and body is a sequence of one or more definitions followed by one or more expressions. It is an error for a keyword to appear more than once in the list of keywords being bound.

body is expanded in the syntactic environment obtained by extending the syntactic environment of the `letrec-syntax` expression with macros whose keywords are the keywords, bound to the specified transformers. Each binding of a keyword symbol has the transformer as well as the body within its scope, so the transformers can transcribe expressions into uses of the macros introduced by the `letrec-syntax` expression.

```
(letrec-syntax
  ((my-or (syntax-rules ()
              ((my-or) #f)
              ((my-or e) e)
              ((my-or e1 e2 ...) (let ((temp e1))
                                    (if temp temp (my-or e2 ...)))))))
  (let (((x #f)
         (y 7)
         (temp 8)
         (let odd?))
       (if even?))
    (my-or x
           (let temp)
           (if y)
           y))) ⇒ 7
```

## 7.5 Iteration

**(do ((variable init step) ...)**

**(test res ...)**

**(command ...)**

A do expression is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition test is met (i.e. it evaluates to #t), the loop exits after evaluating the res expressions.

A do expression is evaluated as follows: The init expressions are evaluated, the variables are bound to fresh locations, the results of the init expressions are stored in the bindings of the variables, and then the iteration phase begins.

Each iteration begins by evaluating test. If the result is false, then the command expressions are evaluated in order, the step expressions are evaluated in some unspecified order, the variables are bound to fresh locations, the results of the step expressions are stored in the bindings of the variables, and the next iteration begins.

If test evaluates to #t, then the res expressions are evaluated from left to right and the values of the last res expression are returned. If no res expressions are present, then the do expression evaluates to void.

The scope of the binding of a variable consists of the entire do expression except for the init expressions. It is an error for a variable to appear more than once in the list of do variables. A step can be omitted,
in which case the effect is the same as if (variable init variable) had been written instead of (variable init).

```lisp
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i)) ⇒ #(0 1 2 3 4)

(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum)) ⇒ 25
```
8 LispKit Core

8.1 Basic primitives

(procedure? obj)
Returns #t if obj is a procedure. Otherwise, it returns #f.

(eval expr env)
If expr is an expression, it is evaluated in the specified environment and its values are returned. If it is a definition, the specified identifiers are defined in the specified environment, provided the environment is not immutable.

(apply proc arg1 … args)
The apply procedure calls proc with the elements of the list (append (list arg1 …) args) as the actual arguments.

(equal? obj1 obj2)
The equal? procedure, when applied to pairs, vectors, strings and bytevectors, recursively compares them, returning #t when the unfoldings of its arguments into possibly infinite trees are equal (in the sense of equal?) as ordered trees, and #f otherwise. It returns the same as eqv? when applied to booleans, symbols, numbers, characters, ports, procedures, and the empty list. If two objects are eqv?, they must be equal? as well. In all other cases, equal? may return either #t or #f. Even if its arguments are circular data structures, equal? must always terminate. As a rule of thumb, objects are generally equal? if they print the same.

(eqv? obj1 obj2)
The eqv? procedure defines a useful equivalence relation on objects. It returns #t if obj1 and obj2 are regarded as the same object.

(eq? obj1 obj2)
The eq? procedure is similar to eqv? except that in some cases it is capable of discerning distinctions finer than those detectable by eqv?. It always returns #f when eqv? also would, but returns #f in some cases where eqv? would return #t. On symbols, booleans, the empty list, pairs, and records, and also on non-empty strings, vectors, and bytevectors, eq? and eqv? are guaranteed to have the same behavior.

(quote datum)
(quote datum) evaluates to datum. datum can be any external representation of a LispKit object. This notation is used to include literal constants in LispKit code. (quote datum) can be abbreviated as 'datum. The two notations are equivalent in all respects. Numerical constants, string constants, character constants, vector constants, bytevector constants, and boolean constants evaluate to themselves. They need not be quoted.

(quasiquote template)
Quasiquote expressions are useful for constructing a list or vector structure when some but not all of the desired structure is known in advance. If no commas appear within template, the result of evaluating (quasiquote template) is equivalent to the result of evaluating (quote template). If a comma appears within template, however, the expression following the comma is evaluated (“unquoted”) and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed
without intervening whitespace by @, then it is an error if the following expression does not evaluate to a list; the opening and closing parentheses of the list are then “stripped away” and the elements of the list are inserted in place of the ,@ expression sequence. ,@ normally appears only within a list or vector. Quasiquote expressions can be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost quasiquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation. Comma corresponds to form unquote ,@ corresponds to form unquote-splicing.

\[ (\lambda (arg1 ...) expr ...) \]
\[ (\lambda (arg1 ... . rest) expr ...) \]
\[ (lambda rest expr ...) \]

A lambda expression evaluates to a procedure. The environment in effect when the lambda expression was evaluated is remembered as part of the procedure. When the procedure is later called with some actual arguments, the environment in which the lambda expression was evaluated will be extended by binding the variables in the formal argument list arg1 … to fresh locations, and the corresponding actual argument values will be stored in those locations. Next, the expressions in the body of the lambda expression will be evaluated sequentially in the extended environment. The results of the last expression in the body will be returned as the results of the procedure call.

\[ (case-lambda (formals expr ...) ...) \]

A case-lambda expression evaluates to a procedure that accepts a variable number of arguments and is lexically scoped in the same manner as a procedure resulting from a lambda expression. When the procedure is called, the first clause for which the arguments agree with formals is selected, where agreement is specified as for formals of a lambda expression. The variables of formals are bound to fresh locations, the values of the arguments are stored in those locations, the expressions in the body are evaluated in the extended environment, and the results of the last expression in the body is returned as the results of the procedure call. It is an error for the arguments not to agree with formals of any clause.

Here is an example showing how to use case-lambda for defining a simple accumulator:

```lisp
(define (make-accumulator n)
  (case-lambda
    (() n)
    ((m) (set! n (+ n m)))))
(define a (make-accumulator 1))
(a) ⇒ 1
(a 5) ⇒ 6
(a) ⇒ 6
```

8.2 Definition primitives

\[ (define var expr) \]
\[ (define (f arg ...) expr) \]
\[ (define-values (var ...) expr) \]

At the outermost level of a program, a definition (define var expr) has essentially the same effect as the assignment expression (set! var expr) if variable var is bound to a non-syntax value. However, if var is not bound, or is a syntactic keyword, then the definition will bind var to a new location before performing the assignment, whereas it would be an error to perform a set! on an unbound variable.

The form (define (f arg ... expr) defines a function f with arguments arg ... and body expr. It is equivalent to (define f (lambda (arg ...) expr)).

(define-values (var ...) expr)

define-values creates multiple definitions var ... from a single expression expr returning multiple values. It is allowed wherever define is allowed.
expr is evaluated, and the variables var … are bound to the return values in the same way that the formal arguments in a `lambda` expression are matched to the actual arguments in a procedure call.

It is an error if a variable var appears more than once in var ….

```
(define-values (x y) (integer-sqrt 17))
(list x y) ⇒ (4 1)
```

**(define-syntax keyword transformer)**

Syntax definitions have the form `(define-syntax keyword transformer)` . `keyword` is an identifier, and `transformer` is an instance of `syntax-rules` . Like variable definitions, syntax definitions can appear at the outermost level or nested within a body.

If the `define-syntax` occurs at the outermost level, then the global syntactic environment is extended by binding the `keyword` to the specified `transformer`, but previous expansions of any global binding for `keyword` remain unchanged. Otherwise, it is an internal syntax definition, and is local to the “body” in which it is defined. Any use of a syntax keyword before its corresponding definition is an error.

Macros can expand into definitions in any context that permits them. However, it is an error for a definition to define an identifier whose binding has to be known in order to determine the meaning of the definition itself, or of any preceding definition that belongs to the same group of internal definitions. Similarly, it is an error for an internal definition to define an identifier whose binding has to be known in order to determine the boundary between the internal definitions and the expressions of the body it belongs to.

Here is an example defining syntax for `while` loops. `while` evaluates the body of the loop as long as the predicate is true.

```
(define-syntax while
  (syntax-rules ()
    ((_ pred body ...) (let loop () (when pred body ... (loop))))))
```

**(syntax-rules (literal ...) rule ...)**

**(syntax-rules ellipsis (literal ...) rule ...)**

A `transformer spec` has one of the two forms listed above. It is an error if any of the `literal …`, or the `ellipsis` symbol in the second form, is not an identifier. It is also an error if syntax rules `rule` are not of the form

- `(pattern template)`.

The `pattern` in a `rule` is a list pattern whose first element is an identifier. In general, a `pattern` is either an identifier, a constant, or one of the following:

- `(pattern ...)`
- `(pattern pattern ... pattern)`
- `(pattern ... pattern ellipsis pattern ...)` ( `pattern ... pattern ellipsis pattern ... . pattern`)
- `#(pattern ...)`
- `#(pattern ... pattern ellipsis pattern ...)`

A `template` is either an identifier, a constant, or one of the following:

- `(element ...)`
- `(element element ... . template)` ( `ellipsis template`)
- `#(element ...)`

where an `element` is a `template` optionally followed by an `ellipsis`. An `ellipsis` is the identifier specified in the second form of `syntax-rules`, or the default identifier `...` (three consecutive periods) otherwise.

Here is an example showcasing how `when` can be defined in terms of `if`:
(define-syntax when
  (syntax-rules ()
    ((_ c e ...) (if c (begin e ...)))))

(define-library (name ...) declaration ...)
A library definition takes the following form: \(\text{define-library (name ...) declaration ...}\). \(\text{name ...}\) is a list whose members are identifiers and exact non-negative integers. It is used to identify the library uniquely when importing from other programs or libraries. It is inadvisable, but not an error, for identifiers in library names to contain any of the characters |, \, ?, *, <, ", :, >, +, [, ], /.

A declaration is any of:

- (export exportspec ...)
- (import importset ...)
- (begin statement ...)
- (include filename ...)
- (include-ci filename ...)
- (include-library-declarations filename ...)
- (cond-expand clause ...)

An export declaration specifies a list of identifiers which can be made visible to other libraries or programs. An exportspec takes one of the following forms:

- ident
- (rename ident1 ident2)

In an exportspec, an identifier ident names a single binding defined within or imported into the library, where the external name for the export is the same as the name of the binding within the library. A rename spec exports the binding defined within or imported into the library and named by ident1 in each (ident1 ident2) pairing, using ident2 as the external name.

An import declaration provides a way to import the identifiers exported by another library. It has the same syntax and semantics as an import declaration used in a program or at the read-eval-print loop.

The begin, include, and include-ci declarations are used to specify the body of the library. They have the same syntax and semantics as the corresponding expression types.

The include-library-declarations declaration is similar to include except that the contents of the file are spliced directly into the current library definition. This can be used, for example, to share the same export declaration among multiple libraries as a simple form of library interface.

The cond-expand declaration has the same syntax and semantics as the cond-expand expression type, except that it expands to spliced-in library declarations rather than expressions enclosed in begin.

(set! var expr)
Procedure set! is used to assign values to variables. expr is evaluated, and the resulting value is stored in the location to which variable var is bound. It is an error if var is not bound either in some region enclosing the set! expression or else globally. The result of the set! expression is unspecified.

8.3 Importing definitions

(import importset ...)
An import declaration provides a way to import identifiers exported by a library. Each importset names a set of bindings from a library and possibly specifies local names for the imported bindings. It takes one of the following forms:
• libraryname
• (only importset identifier … )
• (except importset identifier … )
• (prefix importset identifier )
• (rename importset (ifrom ito) … )

In the first form, all of the identifiers in the named library's export clauses are imported with the same names (or the exported names if exported with rename). The additional importset forms modify this set as follows:

• only produces a subset of the given importset including only the listed identifiers (after any renaming). It is an error if any of the listed identifiers are not found in the original set.
• except produces a subset of the given importset, excluding the listed identifiers (after any renaming). It is an error if any of the listed identifiers are not found in the original set.
• rename modifies the given importset, replacing each instance of ifrom with ito. It is an error if any of the listed identifiers are not found in the original set.
• prefix automatically renames all identifiers in the given importset, prefixing each with the specified identifier.

In a program or library declaration, it is an error to import the same identifier more than once with different bindings, or to redefine or mutate an imported binding with a definition or with set!, or to refer to an identifier before it is imported. However, a read-eval-print loop will permit these actions.

8.4 Delayed execution

(promise? obj)
The promise? procedure returns #t if argument obj is a promise, and #f otherwise.

(make-promise obj)
(eager obj)
The make-promise procedure returns a promise which, when forced, will return obj. It is similar to delay, but does not delay its argument: it is a procedure rather than syntax. If obj is already a promise, it is returned.
eager represents the same procedure like make-promise.

(delay expr)
The delay construct is used together with the procedure force to implement lazy evaluation or "call by need". (delay expr) returns an object called a promise which, at some point in the future, can be asked (by the force procedure) to evaluate expr, and deliver the resulting value.

(delay-force expr)
(lazy expr)
The expression (delay-force expr) is conceptually similar to (delay (force expr)), with the difference that forcing the result of delay-force will in effect result in a tail call to (force expr), while forcing the result of (delay (force expr)) might not. Thus iterative lazy algorithms that might result in a long series of chains of delay and force can be rewritten using delay-force to prevent consuming unbounded space during evaluation.
lazy represents the same procedure like delay-force.

(force promise)
The force procedure forces the value of a promise created by delay, delay-force, or make-promise. If no value has been computed for the promise, then a value is computed and returned. The
value of the promise must be cached (or “memoized”) so that if it is forced a second time, the previously computed value is returned. Consequently, a delayed expression is evaluated using the parameter values and exception handler of the call to \texttt{force} which first requested its value. If \texttt{promise} is not a promise, it may be returned unchanged.

\begin{verbatim}
(force (delay (+ 1 2))) ⇒ 3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p))) ⇒ (3 3)

(define integers
  (letrec ((next (lambda (n)
                  (delay (cons n (next (+ n 1)))))))
    (next 0))
(define head
  (lambda (stream) (car (force stream))))
(define tail
  (lambda (stream) (cdr (force stream))))
(quote (head (tail (tail integers)))) ⇒ 2
\end{verbatim}

The following example is a mechanical transformation of a lazy stream-filtering algorithm into Scheme. Each call to a constructor is wrapped in \texttt{delay}, and each argument passed to a deconstructor is wrapped in \texttt{force}. The use of \texttt{(delay-force ...)} instead of \texttt{(delay (force ...))} around the body of the procedure ensures that an ever-growing sequence of pending promises does not exhaust available storage, because \texttt{force} will, in effect, force such sequences iteratively.

\begin{verbatim}
(define (stream-filter p? s)
  (delay-force
    (if (null? (force s))
      (delay '())
      (let ((h (car (force s)))
            (t (cdr (force s))))
        (if (p? h)
          (delay (cons h (stream-filter p? t)))
          (stream-filter p? t))))))

(quote (head (tail (tail (stream-filter odd? integers)))))) ⇒ 5
\end{verbatim}

The following examples are not intended to illustrate good programming style, as \texttt{delay}, \texttt{force}, and \texttt{delay-force} are mainly intended for programs written in the functional style. However, they do illustrate the property that only one value is computed for a promise, no matter how many times it is forced.

\begin{verbatim}
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
            (if (> count x) count (force p)))))

(define x 5)
p ⇒ a promise
(force p) ⇒ 6
p ⇒ a promise
(begin (set! x 10) (force p)) ⇒ 6
\end{verbatim}

### 8.5 Symbols

\begin{verbatim}
(symbol? obj)
\end{verbatim}

\texttt{(symbol? obj)}

Returns \texttt{#t} if \texttt{obj} is a symbol, otherwise returns \texttt{#f}.
(gensym)
(gensym str)

Returns a new (fresh) symbol whose name consists of prefix str followed by a number. If str is not provided, "g" is used as a prefix.

(symbol=? sym …)

Returns #t if all the arguments are symbols and all have the same names in the sense of string=?.

(string->symbol str)

Returns the symbol whose name is string str. This procedure can create symbols with names containing special characters that would require escaping when written, but does not interpret escapes in its input.

(symbol->string sym)

Returns the name of symbol sym as a string, but without adding escapes.

8.6 Booleans

The standard boolean objects for true and false are written as #t and #f. Alternatively, they can be written #true and #false, respectively. What really matters, though, are the objects that the Scheme conditional expressions (if, cond, and, or, when, unless, do, when) treat as true or false. The phrase a "true value" (or sometimes just "true") means any object treated as true by the conditional expressions, and the phrase a "false value" (or "false") means any object treated as false by the conditional expressions.

Of all the Scheme values, only #f counts as false in conditional expressions. All other Scheme values, including #t, count as true. Boolean literals evaluate to themselves, so they do not need to be quoted in programs.

(boolean? obj)

The boolean? predicate returns #t if obj is either #t or #f and returns #f otherwise.

| (boolean? #f) | ⇒ #t |
| (boolean? 0) | ⇒ #f |
| (boolean? '()) | ⇒ #f |

(boolean=? obj1 obj2 …)

Returns #t if all the arguments are booleans and all are #t or all are #f.

(and test …)

The test … expressions are evaluated from left to right, and if any expression evaluates to #f, then #f is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the values of the last expression are returned. If there are no expressions, then #t is returned.

| (and (= 2 2) (> 2 1)) | ⇒ #t |
| (and (= 2 2) (< 2 1)) | ⇒ #f |
| (and 12 'c '(f g)) | ⇒ (f g) |
| (and) | ⇒ #t |

(or test …)

The test … expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to #f or if there are no expressions, then #f is returned.
(or (= 2 2) (> 2 1)) ⇒ #t
(or (= 2 2) (< 2 1)) ⇒ #t
(or #f #f #f) ⇒ #f
(or (memq 'b '(a b c)) (/ 3 0)) ⇒ (b c)

(not obj)
The not procedure returns #t if obj is false, and returns #f otherwise.

(not #t) ⇒ #f
(not 3) ⇒ #f
(not (list 3)) ⇒ #f
(not #f) ⇒ #t
(not '()) ⇒ #f
(not (list)) ⇒ #f
(not 'nil) ⇒ #f

(opt pred obj)
The opt procedure returns #t if obj is #f. If obj is not #f, opt applies predicate pred to obj and returns the result of this function application.

This function is useful to verify a given predicate pred for an optional value obj.

8.7 Conditional and inclusion compilation

(cond-expand ce-clause1 ce-clause2 …)
The cond-expand expression type provides a way to statically expand different expressions depending on the implementation. A ce-clause takes the following form:

(featurerequirement expression …)
The last clause can be an “else clause,” which has the form:

(else expression …)

A featurerequirement takes one of the following forms: - featureidentifier - (library name ) - (and featurerequirement … ) - (or featurerequirement … ) - (not featurerequirement )

LispKit maintains a list of feature identifiers which are present, as well as a list of libraries which can be imported. The value of a featurerequirement is determined by replacing each featureidentifier and (library name ) with #t, and all other feature identifiers and library names with #f, then evaluating the resulting expression as a Scheme boolean expression under the normal interpretation of and , or , and not .

A cond-expand is then expanded by evaluating the featurerequirements of successive ce-clause in order until one of them returns #t. When a true clause is found, the corresponding expression … are expanded to a begin , and the remaining clauses are ignored. If none of the listed featurerequirement evaluates to #t, then if there is an “else” clause, its expression … are included. Otherwise, the behavior of the cond-expand is unspecified. Unlike cond, cond-expand does not depend on the value of any variables. The exact features provided are defined by the implementation, its environment and host platform.

LispKit supports the following featureidentifier:

- lispkit
- r7rs
- ratios
- complex
• syntax-rules
• little-endian
• big-endian
• dynamic-loading
• modules
• 32bit
• 64bit
• macos
• macosx
• ios
• linux
• i386
• x86-64
• arm64
• arm

(include str1 str2 …)
(include-ci str1 str2 …)

Both include and include-ci take one or more filenames expressed as string literals, apply an implementation-specific algorithm to find corresponding files, read the contents of the files in the specified order as if by repeated applications of read, and effectively replace the include or include-ci expression with a begin expression containing what was read from the files. The difference between the two is that include-ci reads each file as if it began with the #!fold-case directive, while include does not.

8.8 Multiple values

(values obj …)

Delivers all of its arguments to its continuation. The values procedure might be defined as follows:

\[
\text{(define (values . things)}\text{(call-with-current-continuation (lambda (cont) (apply cont things)))))}
\]

(call-with-values producer consumer)

Calls its producer argument with no arguments and a continuation that, when passed some values, calls the consumer procedure with those values as arguments. The continuation for the call to consumer is the continuation of the call to call-with-values.

\[
\begin{align*}
\text{(call-with-values (lambda () (values 4 5)) (lambda (a b) b))} \\
& \Rightarrow 5 \\
\text{(call-with-values * -)} \\
& \Rightarrow -1
\end{align*}
\]

8.9 Environments

(environment? obj)

Returns #t if obj is an environment. Otherwise, it returns #f.
(environment list1 …)
This procedure returns an environment that results by starting with an empty environment and then importing each list, considered as an import set, into it. The bindings of the environment represented by the specifier are immutable, as is the environment itself.

(scheme-report-environment version)
If version is equal to 5, corresponding to R5RS, scheme-report-environment returns an environment that contains only the bindings defined in the R5RS library.

(null-environment version)
If version is equal to 5, corresponding to R5RS, the null-environment procedure returns an environment that contains only the bindings for all syntactic keywords defined in the R5RS library.

(interaction-environment)
This procedure returns a mutable environment which is the environment in which expressions entered by the user into a read-eval-print loop are evaluated. This is typically a superset of bindings from (lispkit base).

8.10 Syntax errors

(syntax-error message args …)
syntax-error behaves similarly to error except that implementations with an expansion pass separate from evaluation should signal an error as soon as syntax-error is expanded. This can be used as a syntax-rules template for a pattern that is an invalid use of the macro, which can provide more descriptive error messages.

message is a string literal, and args … are arbitrary expressions providing additional information. Applications cannot count on being able to catch syntax errors with exception handlers or guards.

(define-syntax simple-let
  (syntax-rules ()
    ((_ (head ... ((x . y) val) . tail) body1 body2 ...) (syntax-error "expected an identifier but got" (x . y)))
    ((_ ((name val) ...) body1 body2 ...) ((lambda (name ...) body1 body2 ...) val ...))))

8.11 Utilities

(void)
Performs no operation and returns nothing. This is often useful as a placeholder or whenever a no-op statement is needed.

(void? obj)
Returns #t if obj is the “void” value (i.e. no value); returns #f otherwise.

(identity obj)
The identity function is always returning its argument obj.
9 LispKit Datatype

Library (lispkit datatype) implements algebraic datatypes for LispKit. It provides the following functionality:

- **define-datatype** creates a new algebraic datatype consisting of a type test predicate and a number of variants. Each variant implicitly defines a constructor and a pattern.
- **define-pattern** introduces a new pattern and constructor for an existing datatype variant.
- **match** matches a value of an algebraic datatype against patterns, binding pattern variables and executing the code of the first case whose pattern matches the value.

9.1 Usage

Here is an example of a datatype defining a tree for storing and finding elements:

```lisp
(define-datatype tree tree? (empty) (node left element right) where (and (tree? left) (tree? right)))
```

The datatype `tree` defines a predicate `tree?` for checking whether a value is of type `tree`. In addition, it defines two variants with corresponding constructors `empty` and `node` for creating values of type `tree`. Variant `node` defines an invariant that prevents nodes from being constructed unless `left` and `right` are also trees.

The following line defines a new tree:

```lisp
(define t1 (node (empty) 4 (node (empty) (empty) (empty))))
```

Using `match`, values like `t1` can be deconstructed using pattern matching. The following function `elements` shows how to collect all elements from a tree in a list:

```lisp
(define (elements tree)
  (match tree
    ((empty) ())
    ((node l e r) (append (elements l) (list e) (elements r)))))
```

`match` is a special form which takes a value of an algebraic datatype and matches it against a list of cases. Each case defines a pattern and a sequence of statements which get executed if the pattern matches the value.

Cases can also optionally define a guard which is a boolean expression that gets executed if the pattern of the case matches a value. Only if the guard evaluates to true, the statements of the case get executed. Otherwise, pattern matching continues. The following function `insert` demonstrates this functionality:
(define (insert tree x)
  (match tree
    ((empty)
     (node (empty) x (empty)))
    ((node l e r) where (< x e)
     (node (insert l x) e r))
    ((node l e r)
     (node l e (insert r x))))
)

A new tree \( t_2 \), with two new elements inserted, can be created like this:

(defun t2 (insert (insert t1 2) 9))

If a pattern is used frequently containing a lot of boilerplate, it is possible to define a shortcut using the `define-pattern` syntax:

(define-pattern (single x)
  (node (empty) x (empty)))

With this declaration, it is possible to use `single` in patterns. The following example also shows that it is possible to use `else` for defining a fallback case, if no other pattern is matching.

(match t
  ((empty) #f)
  ((single x) x)
  (else (error "two many elements")))

`single` can also be used as a constructor for creating trees with a single element:

(singleton 6)

An advanced feature of `match` is the usage of pattern alternatives in a single case of a `match` construct. This can be achieved using the `or` form on the top level of a pattern:

(define (has-many-elements tree)
  (match tree
    ((empty) (single _)) #f)
    (else #t)))

The underscore in the `(single _)` subpattern is a wildcard that matches every value and that does not bind a new variable.

### 9.2 API

`define-datatype type (constr arg …) …`

`define-datatype type pred (constr arg …) …`

**(define-datatype type pred (constr arg …) where condition …)**

Defines a new datatype with a given number of datatype variants. The definition requires the symbol `type` denoting the new type, an optional symbol `pred` which gets bound to a type test function for testing whether a value is an instance of this type, and a list of constructors of the form `(constr arg1 arg2 …)`
where constr is the constructor and arg1, arg2, ... are parameter names of the constructor. A constructor can be annotated with an invariant for defining requirements the parameters need to meet. This is done via clause where expr succeeding the constructor declaration. condition is a boolean expression which gets checked when the constructor gets invoked.

**(define-pattern (constr arg ...) (impl expr ...))**

Defines a new pattern (constr arg ...) which specializes an existing pattern (impl expr ...). Such custom patterns can be used in pattern matching expressions as well as constructors for defining values of an algebraic datatype.

**(match expr case ...)**

**(match expr case ... (else stat ...))**

**match** provides a mechanism for decomposing values of algebraic datatypes via pattern matching. A match construct takes a value expr to pattern match on, as well as a sequence of cases. Each case consists of pattern alternatives, an optional guard, and a sequence of statements:

```lisp
(case = `(` patterns stat ... `)`
   | `(` `or` pattern ... `)`
(pattern = pattern
   | `(` `where` condition stat ... `)`
(pattern = ``_`); wildcard
   | var; variable
   | `#t`; literal boolean (true)
   | `#f`; literal boolean (false)
   | string; literal string
   | number; literal number
   | character; literal character
   | vector; literal vector
   | `'` expr; constant expression
   | `,` expr; value (result of evaluating expr)
   | pattern `as` var; pattern bound to variable
   | `(` `list` pattern ... `)`; list pattern
   | `(` `list` pattern ... `.` var `)`; list pattern with rest
   | `(` `list` pattern ... `.` `_` `)`; list pattern with unbound rest
   | `(` constr pattern ... `)`; variant pattern)
```

**match** iterates through the cases and executes the sequence of statements stat ... of the first case whose pattern is matching expr and whose guard condition evaluates to true. The value returned by this sequence of statements is returned by **match**.
10 LispKit Date-Time

Library (lispkit date-time) provides functionality for handling time zones, dates, and times. Time zones are represented by string identifiers referring to the region and corresponding city, e.g. "America/Los_Angeles". Dates and times are represented via date-time data structures. These encapsulate the following components:

- **timezone**: the time zone of the date
- **date**: the date consisting of its year, month, and day
- **time**: the time on date consisting of the hour (>= 0, < 24), the minute (>= 0, < 60), the second (>= 60, <60), and the nano second.

The library uses a floating-point representation of seconds since 00:00 UTC on January 1, 1970, as a means to refer to specific points in time independent of time zones. This means that, for instance, for comparing date-times with each other, a user would have to convert them to seconds and then compare the seconds instead.

For now, (lispkit date-time) assumes all dates are based on the Gregorian calendar, independent of the settings at the operating system-level.

10.1 Time zones

Time zones are represented by string identifiers referring to the region and corresponding city, e.g. "America/Los_Angeles". Procedure timezones returns a list of all supported time zone identifiers. Each time zone has a locale-specific name and an offset in seconds from Greenwich Mean Time. Some time zones also have an abbreviation which can be used as an alternative way to identify a time zone.

(timezones)
(timezones filter)

Returns a list of string identifiers for all supported time zones. If filter is provided, it can either be set to #f, in which case a list of abbreviations is returned instead, or it is a string, and only time zone identifiers which contain filter are returned.

(timezone? obj)

Returns #t if obj is a valid time zone identifier or time zone abbreviation; returns #f otherwise.

(timezone)
(timezone ident)

Returns the identifier for the time zone specified by ident. ident can either be an identifier, an abbreviation or a GMT offset as a floating-point number or integer. If ident does not refer to a supported time zone, procedure timezone will fail.

(timezone-name ts)
(timezone-name ts locale)
(timezone-name ts locale format)
Returns a locale-specific name for time zone `tz`. If `locale` is not specified, the current locale defined at the operating-system level is used. `format` specifies the name format. It can have one of the following symbolic values:

- `standard`
- `standard-short`
- `dst`
- `dst-short`
- `generic`
- `generic-short`

(timezone-abbreviation `tz`)  
Returns a string representing a time zone abbreviation for `tz`; e.g. "PDT". If the time zone `tz` does not have an abbreviation, this function returns `#f`.

(timezone-gmt-offset `tz`)  
Returns the difference in seconds between time zone `tz` and Greenwich Mean Time. The difference is returned as a floating-point number (since seconds are always represented as such by this library).

## 10.2 Date-times

(current-seconds)  
Returns a floating-point number representing the number of seconds since 00:00 UTC on January 1, 1970.

(date-time? `obj`)  
Returns `#t` if `obj` is a date-time object; returns `#f` otherwise.

(date-time)  
(date-time `year month day hour`)  
(date-time `year month day hour min`)  
(date-time `year month day hour min sec`)  
(date-time `tz`)  
(date-time `tz year month day`)  
(date-time `tz year month day hour`)  
(date-time `tz year month day hour min`)  
(date-time `tz year month day hour min sec`)  
(date-time `tz year month day hour min sec nano`)  

Constructs a date-time representation out of the given date time components. `tz` is the only string argument; it is referring to a time zone. All other arguments are numeric arguments. This procedure returns a date-time object for the specified time at the given date. If no date components are provided as arguments, procedure `date-time` returns a date-time for the current date and time.

(week->date-time `year week`)  
(week->date-time `year week wday`)  
(week->date-time `year week wday hour`)  
(week->date-time `year week wday hour min`)  
(week->date-time `year week wday hour min sec`)  
(week->date-time `year week wday hour min sec nano`)  

Chapter 10
(week->date-time tz year week wday hour min sec)
(week->date-time tz year week wday hour min sec nano)

Constructs a date-time representation out of the given date-time components. tz is the only string argument; it is referring to a time zone. All other arguments are numeric arguments. Argument wday specifies the week day in the given week. Week days are given numbers from 1 (= Monday) to 7 (= Sunday). This procedure returns a date-time object for the specified time at the given date.

The difference to date-time is that this procedure does not refer to a month and day. It rather refers to the week number as well as the weekday within this specified week number.

(seconds->date-time secs)
(seconds->date-time secs tz)

Converts the given number of seconds secs into date-time format for the given time zone tz. secs is a floating-point number. It is interpreted as the number of seconds since 00:00 UTC on January 1, 1970. secs is negative if the date-time is earlier than 00:00 UTC on January 1, 1970. If tz is missing, the current, operating-system defined time zone is used.

(string->date-time str)
(string->date-time str tz)
(string->date-time str tz locale)
(string->date-time str tz locale format)

Extracts a date and time from the given string str in the time zone tz, or the current time zone if tz is omitted. The format of the string representation is defined in terms of locale and format. format is one of the following symbols:

- none
- short
- medium
- long
- full

It is also possible to define the format for date and time separately by using a list of two values for format: (dateformat timeformat) where both dateformat and timeformat are one of the 5 symbols above. This makes it possible, for instance, to just convert a date (without time) in string form to a date-time object, e.g. by using (short none) as format.

(date-time->seconds dtime)

Returns a floating-point number representing the number of seconds since 00:00 UTC on January 1, 1970 for the given date-time object dtime.

(date-time->string dtime)
(date-time->string dtime locale)
(date-time->string dtime locale format)

Returns a string representation of the date-time object dtime. The format of the string is defined in terms of locale and format. format is either one of the following symbols: none, short, medium, long, full or it's a two element list of the form (dateformat timeformat) (just like for string->date-time).

(date-time->iso8601-string dtime)

Returns a string representation of the date-time object dtime in ISO 8601 format.

(date-time-timezone dtime)

Returns the time zone of dtime.

(date-time-year dtime)

Returns the year of dtime.
(date-time-month dt ime)  
Returns the month of dt ime.

(procedure)

(date-time-day dt ime)  
Returns the day of dt ime.

(procedure)

(date-time-hour dt ime)  
Returns the hour of dt ime.

(procedure)

(date-time-minute dt ime)  
Returns the minute of dt ime.

(procedure)

(date-time-second dt ime)  
Returns the second of dt ime.

(procedure)

(date-time-nano dt ime)  
Returns the nano-second of dt ime.

(procedure)

(date-time-weekday dt ime)  
Returns the week day of dt ime. Week days are represented as fixnums where 1 is Monday, 2 is Tuesday, ..., and 7 is Sunday.

(procedure)

(date-time-week dt ime)  
Returns the week number of dt ime according to the ISO-8601 standard. Based on this standard, weeks start on Monday. The first week of the year is the week that contains that year's first Thursday.

(procedure)

(date-time-dst-offset dt ime)  
Returns the daylight saving time offset of dt ime in seconds related to GMT. If daylight savings time is not active, date-time-dst-offset returns 0.0. The result is always a floating-point number.

(procedure)

(date-time-has-dst? dt ime)  
Returns #t if daylight saving time is active for dt ime; returns #f otherwise.

(procedure)

(next-dst-transition dt ime)  
Returns the date and time when the next daylight savings time transition takes place after dt ime. next-dst-transition returns #f if there is no daylight savings time for the time zone of dt ime.
11 LispKit Disjoint-Set

Library (lispkit disjoint-set) implements disjoint sets, a mutable union-find data structure that tracks a set of elements partitioned into disjoint subsets. Disjoint sets are based on hashtables and require the definition of an equality and a hash function.

(disjoint-set? obj)
Returns #t if obj is a disjoint set object; otherwise #f is returned.

(make-eq-disjoint-set)
Returns a new empty disjoint set using eq as equality and eq-hash as hash function.

(make-eqv-disjoint-set)
Returns a new empty disjoint set using eqv as equality and eqv-hash as hash function.

(make-disjoint-set comparator)
(make-disjoint-set hash eql)
Returns a new empty disjoint set using eql as equality and hash as hash function. Instead of providing two functions, a new disjoint set can also be created based on a comparator.

(disjoint-set-make dset x)
Adds a new singleton set x to dset if element x does not exist already in disjoint set dset.

(disjoint-set-find dset x)
(disjoint-set-find dset x default)
Looks up element x in dset and returns the set in which x is currently contained. Returns default if element x is not found. If default is not provided, disjoint-set-find uses #f instead.

(disjoint-set-union dset x y)
Unifies the sets containing x and y in disjoint set dset.

(disjoint-set-size dset)
Returns the number of sets in dset.
12 LispKit Draw

Library (lispkit draw) an API for creating drawings. A drawing is defined in terms of a sequence of instructions for drawing shapes and images. Drawings can be composed and saved as a PDF. It is also possible to draw a drawing into a bitmap and save it in formats like PNG, JPG, or TIFF. A bitmap is a special image that is not based on vector graphics.

Both drawings and shapes are based on a coordinate system whose zero point is in the upper left corner of a plane. The x and y axis extend to the right and down. Coordinates and dimensions are always expressed in terms of floating-point numbers.

12.1 Drawings

Drawings are mutable objects created via make-drawing. The functions listed in this section change the state of a drawing object and they persist drawing instructions defining the drawing. For most functions, the drawing is an optional argument. If it is not provided, the function applies to the drawing provided by the current-drawing parameter object.

current-drawing
Defines the current drawing, which is used as a default by all functions for which the drawing argument is optional. If there is no current drawing, this parameter is set to #f.

(drawing? obj)
Returns #t if obj is a drawing. Otherwise, it returns #f.

(make-drawing)
Returns a new, empty drawing. A drawing consists of a sequence of drawing instructions and drawing state consisting of the following components:

- Stroke color (set via set-color)
- Fill color (set via fill-color)
- Shadow (set via set-shadow and remove-shadow)
- Transformation (add transformation via enable-transformation and remove via disable-transformation)

(copy-drawing drawing)
Returns a copy of the given drawing.

(set-color color)

(set-color color drawing)
Sets the stroke color for the given drawing, or current-drawing if the drawing argument is not provided.

(set-fill-color color)

(set-fill-color color drawing)
Sets the fill color for the given drawing, or current-drawing if the drawing argument is not provided.

(set-line-width width)

(set-line-width width drawing)
Sets the default stroke width for the given drawing, or current-drawing if the drawing argument is not provided.

\[(\text{set-shadow} \ \text{color} \ \text{size} \ \text{blur-radius})\]
\[(\text{set-shadow} \ \text{color} \ \text{size} \ \text{blur-radius} \ \text{drawing})\]

Defines a shadow for the given drawing, or current-drawing if the drawing argument is not provided. color is the color of the shade, blur-radius defines the radius for bluring the shadow.

\[(\text{remove-shadow})\]
\[(\text{remove-shadow} \ \text{drawing})\]

Removes shadow for the subsequent drawing instructions of the given drawing, or current-drawing if the drawing argument is missing.

\[(\text{enable-transformation} \ \text{tf})\]
\[(\text{enable-transformation} \ \text{tf} \ \text{drawing})\]

Enables the transformation tf for subsequent drawing instructions of the given drawing, or current-drawing if the drawing argument is missing. Each drawing maintains an active affine transformation for shifting, rotating, and scaling the coordinate systems of subsequent drawing instructions.

\[(\text{disable-transformation} \ \text{tf})\]
\[(\text{disable-transformation} \ \text{tf} \ \text{drawing})\]

Disables the transformation tf for subsequent drawing instructions of the given drawing, or current-drawing if the drawing argument is missing.

\[(\text{draw} \ \text{shape})\]
\[(\text{draw} \ \text{shape} \ \text{width})\]
\[(\text{draw} \ \text{shape} \ \text{width} \ \text{drawing})\]

Draws shape with a given stroke width into the drawing specified via drawing or parameter object current-drawing if drawing is not provided. The default for width, in case it is not provided, is set via set-line-width. The stroke is drawn in the current stroke color of the drawing.

\[(\text{draw-dashed} \ \text{shape} \ \text{lengths} \ \text{phase})\]
\[(\text{draw-dashed} \ \text{shape} \ \text{lengths} \ \text{phase} \ \text{width})\]
\[(\text{draw-dashed} \ \text{shape} \ \text{lengths} \ \text{phase} \ \text{width} \ \text{drawing})\]

Draws shape with a dashed stroke of width width into the drawing specified via drawing or parameter object current-drawing if drawing is not provided. 1.0 is the default for width in case it is not provided. lengths specifies an alternating list of dash/space lengths. phase determines the start of the dash/space pattern. The dashed stroke is drawn in the current stroke color of the drawing.

\[(\text{fill} \ \text{shape})\]
\[(\text{fill} \ \text{shape} \ \text{drawing})\]

Fills shape with the current fill color in the drawing specified via drawing or parameter object current-drawing if drawing is not provided.

\[(\text{fill-gradient} \ \text{shape} \ \text{colors})\]
\[(\text{fill-gradient} \ \text{shape} \ \text{colors} \ \text{spec})\]
\[(\text{fill-gradient} \ \text{shape} \ \text{colors} \ \text{spec} \ \text{drawing})\]

Fills shape with a gradient in the drawing specified via argument drawing or parameter object current-drawing if drawing is not provided. The gradient is specified in terms of a list of colors and argument spec. spec can either be a number or a point. If spec is a number, this number determines an angle for a linear gradient. If spec is a point, it is the center of a radial gradient.

\[(\text{draw-line} \ \text{start} \ \text{end})\]
\[(\text{draw-line} \ \text{start} \ \text{end} \ \text{drawing})\]
Draws a line between point \textit{start} and point \textit{end} in the drawing specified via argument \textit{drawing} or parameter object \textit{current-drawing}, if \textit{drawing} is not provided. The line is drawn in the default \textit{stroke width} and the current \textit{stroke color}.

\begin{verbatim}
(draw-rect rect end)
(draw-rect rect drawing)
\end{verbatim}

Draws a rectangular given by \textit{rect} in the drawing specified via argument \textit{drawing} or parameter object \textit{current-drawing}, if \textit{drawing} is not provided. The rectangular is drawn in the default \textit{stroke width} and the current \textit{stroke color}.

\begin{verbatim}
(fill-rect rect end)
(fill-rect rect drawing)
\end{verbatim}

Fills a rectangular given by \textit{rect} with the current \textit{fill color} in the drawing specified via argument \textit{drawing} or parameter object \textit{current-drawing}, if \textit{drawing} is not provided.

\begin{verbatim}
(draw-ellipse rect end)
(draw-ellipse rect drawing)
\end{verbatim}

Draws an ellipse into the rectangular \textit{rect} in the drawing specified via argument \textit{drawing} or parameter object \textit{current-drawing}, if \textit{drawing} is not provided. The ellipse is drawn in the default \textit{stroke width} and the current \textit{stroke color}.

\begin{verbatim}
(fill-ellipse rect end)
(fill-ellipse rect drawing)
\end{verbatim}

Fills an ellipse given by rectangular \textit{rect} with the current \textit{fill color} in the drawing specified via argument \textit{drawing} or parameter object \textit{current-drawing}, if \textit{drawing} is not provided.

\begin{verbatim}
(draw-text str location font)
(draw-text str location font color)
(draw-text str location font color drawing)
\end{verbatim}

Draws string \textit{str} at \textit{location} in the given font and color in the drawing specified by argument \textit{drawing} or parameter object \textit{current-drawing} if \textit{drawing} is not provided. \textit{location} is either the left, top-most point at which the string is drawn, or it is a \textit{rect} specifying a bounding box. \textit{color} specifies the text color. If it is not provided, the text is drawn in black.

\begin{verbatim}
(text-size text)
(text-size text font)
(text-size text font dimensions)
\end{verbatim}

Returns a size object describing the width and height needed to draw string \textit{html} using \textit{font} in a space constrained by \textit{dimensions}. \textit{dimensions} is either a size object specifying the maximum width and height, or it is a number constraining the width only, assuming infinite height. If \textit{dimensions} is omitted, the maximum width and height is infinity.

\begin{verbatim}
(draw-html html location)
(draw-html html location drawing)
\end{verbatim}

Draws a string \textit{html} containing HTML source code at \textit{location} in the drawing specified by argument \textit{drawing} or parameter object \textit{current-drawing} if \textit{drawing} is not provided. \textit{location} is either the left, top-most point at which the HTML is drawn, or it is a \textit{rect} specifying a bounding box.

\begin{verbatim}
(html-size html)
(html-size html dimensions)
\end{verbatim}

Returns a size object describing the width and height needed to render the HTML in string \textit{html} in a space constrained by \textit{dimensions}. \textit{dimensions} is either a size object specifying the maximum width and height, or
it is a number constraining the width only, assuming infinite height. If dimensions is omitted, the maximum width and height is infinity.

\begin{verbatim}
(draw-image image location)
(draw-image image location opacity)
(draw-image image location opacity composition)
(draw-image image location opacity composition drawing)
\end{verbatim}

Draws image image at location with the given opacity and composition method. The image is drawn in the drawing specified by argument drawing or parameter object current-drawing if drawing is not provided. location is either the left, top-most point at which the image is drawn, or it is a rect specifying a bounding box for the image. composition is a floating-point number between 0.0 (= transparent) and 1.0 (= completely not transparent) with 1.0 being the default. composition refers to a symbol specifying a composition method. The following methods are supported (the source is the image, the destination is the drawing):

- clear: Transparency everywhere.
- copy: The source image (default).
- multiply: The source color is multiplied by the destination color.
- overlay: Source colors overlay the destination.
- source-over: The source image wherever it is opaque, and the destination elsewhere.
- source-in: The source image wherever both images are opaque, and transparent elsewhere.
- source-out: The source image wherever it is opaque and the destination is transparent, and transparent elsewhere.
- source-atop: The source image wherever both source and destination are opaque, the destination wherever it is opaque but the source image is transparent, and transparent elsewhere.
- destination-over: The destination wherever it is opaque, and the source image elsewhere.
- destination-in: The destination wherever both images are opaque, and transparent elsewhere.
- destination-out: The destination wherever it is opaque and the source image is transparent, and transparent elsewhere.
- destination-atop: The destination wherever both image and destination are opaque, the source image wherever it is opaque and the destination is transparent, and transparent elsewhere.

\begin{verbatim}
(draw-drawing other)
(draw-drawing other drawing)
\end{verbatim}

Draws drawing other into the drawing specified by argument drawing or parameter object current-drawing if drawing is not provided. This function can be used to compose drawings.

\begin{verbatim}
(clip-drawing other clippingshape)
(clip-drawing other clippingshape drawing)
\end{verbatim}

Draws drawing other into the drawing specified by argument drawing or parameter object current-drawing if drawing is not provided. This function clips the drawing using shape clippingshape; i.e. only parts within clippingshape are drawn.

\begin{verbatim}
:inline-drawing other)
:inline-drawing other drawing)
\end{verbatim}

Draws drawing other into the drawing specified by argument drawing or parameter object current-drawing if drawing is not provided. This function can be used to compose drawings in a way such that the drawing instructions from other are inlined into drawing.

\begin{verbatim}
(save-drawing path drawing size)
(save-drawing path drawing size title)
(save-drawing path drawing size title author)
\end{verbatim}
Saves drawing into a PDF file at the given filepath path. size is a size specifying the width and height of the PDF page containing the drawing in points; i.e. the media box of the page is \((\text{rect zero-point size})\). title and author are optional strings defining the title and author metadata for the generated PDF file.

\[
\text{(save-drawings path pages)}
\]

\[
\text{(save-drawings path pages title)}
\]

\[
\text{(save-drawings path pages title author)}
\]

Saves a list of pages into a PDF file at the given filepath path. A page is defined in terms of a list of two elements \((\text{drawing size})\), where drawing is a drawing for that page and size is a media box for the page. title and author are optional strings defining the title and author metadata for the generated PDF file.

\[
\text{(drawing body ...)}
\]

Creates a new empty drawing, binds parameter object current-drawing to it and executes the body statements in the dynamic scope of this binding. This special form returns the new drawing.

\[
\text{(with-drawing drawing body ...)}
\]

Binds parameter object current-drawing to drawing and executes the body statements in the dynamic scope of this binding. This special form returns the result of the last statement in the body.

\[
\text{(transform tf body ...)}
\]

This form is used in the context of drawing into current-drawing. It enables the transformation tf, executes the statements in the body and disables the transformation again.

### 12.2 Shapes

Shapes are mutable objects created via a number of constructors, including make-shape, copy-shape, line, polygon, rectangular, circle, oval, arc, and glyphs. Besides the constructors, functions like move-to, line-to and curve-to are used to extend a shape. For those functions, the affected shape is an optional argument. If it is not provided, the function applies to the shape defined by the current-shape parameter object.

**current-shape**

Defines the current shape, which is used as a default by all functions for which the shape argument is optional. If there is no current shape, this parameter is set to #f.

\[
\text{(shape? obj)}
\]

Returns #t if obj is a shape. Otherwise, it returns #f.

\[
\text{(make-shape)}
\]

\[
\text{(make-shape prototype)}
\]

Returns a new shape object. If argument prototype is provided, the new shape object will inherit from shape prototype; i.e. the new shape's definition will extend the definition of shape prototype.

\[
\text{(copy-shape shape)}
\]

Returns a copy of shape.

\[
\text{(line start end)}
\]

Returns a new line shape. start and end are the start and end points of the line.

\[
\text{(polygon point ...)}
\]

Returns a new polygon shape. The polygon is defined in terms of a sequence of points.

\[
\text{(rectangle point size)}
\]

\[
\text{(rectangle point size radius)}
\]

\[
\text{(rectangle point size xradius yradius)}
\]
Returns a new rectangular shape. The rectangle is defined in terms of the left, topmost point and a size defining both width and height of the rectangle. The optional radius, xradius and yradius arguments are used to create a rounded rectangular whose rounded edges are defined in terms of an x and y-radius. If only one radius is provided, it defines both x and y-radius.

\[
\text{\textbf{(circle point radius)}}
\]
Returns a new circle shape. The circle is defined in terms of a center point and a radius.

\[
\text{\textbf{(oval point size)}}
\]
Returns a new oval shape. The oval is defined in terms of a rectangle whose left, topmost point is provided as argument point, and whose width and height are given via argument size.

\[
\text{\textbf{(arc point radius start end)}}}
\]
\[
\text{\textbf{(arc point radius start end clockwise)}}}
\]
Returns a new arc shape. The arc is defined via the arguments point, radius, start, end and optionally clockwise. point is the starting point of the arc, radius defines the radius of the arc, start is a starting angle in radians, and end is the end angle in radians. clockwise is a boolean argument defining whether the arc is drawn clockwise or counter-clockwise. The default is #t.

\[
\text{\textbf{(glyphs str point size font)}}
\]
Returns a new “glyphs” shape. This is a shape defined by a string str rendered in the given size and font at a given point. font is a font object, size is the font size in points, and point are the start coordinates where the glyphs are drawn.

\[
\text{\textbf{(transform-shape shape tf)}}
\]
Returns a new shape derived from shape by applying transformation tf.

\[
\text{\textbf{(flip-shape shape)}}}
\]
\[
\text{\textbf{(flip-shape shape box)}}}
\]
\[
\text{\textbf{(flip-shape shape box orientation)}}}
\]
Returns a new shape by flipping/mirroring shape within box. box is a rect. If it is not provided, the bounding box of shape is used as a default. Argument orientation is a symbol defining along which axis the shape is flipped. Supported are horizontal, vertical, and mirror. Default is vertical.

\[
\text{\textbf{(interpolate points)}}}
\]
\[
\text{\textbf{(interpolate points closed)}}}
\]
\[
\text{\textbf{(interpolate points closed alpha)}}}
\]
\[
\text{\textbf{(interpolate points closed alpha method)}}}
\]
Returns a shape interpolating a list of points. closed is an optional boolean argument specifying whether the shape is closed. The default for closed is #f. alpha is an interpolation parameter in the range [0.0,1.0]; default is 0.33. method specifies the interpolation method via a symbol. The following two methods are supported: hermite and catmull-rom; default is hermite.

\[
\text{\textbf{(move-to point)}}}
\]
\[
\text{\textbf{(move-to point shape)}}}
\]
Sets the “current point” to point for the shape specified by argument shape or parameter object current-shape if shape is not provided.

\[
\text{\textbf{(line-to point ...)}}}
\]
\[
\text{\textbf{(line-to point ... shape)}}}
\]
Creates a line from the “current point” to point for the shape specified by argument shape or parameter object current-shape if shape is not provided. point becomes the new “current point”.

\[
\text{\textbf{(curve-to point ctrl1 ctrl2)}}}
\]
\[
\text{\textbf{(curve-to point ctrl1 ctrl2 shape)}}}
\]
Creates a curve from the “current point” to point for the shape specified by argument shape or parameter object current-shape if shape is not provided. cntrl1 and cntrl2 are control points defining tangents shaping the curve at the start and end points.

(relative-move-to point)  
(relative-move-to point shape)  

This function is equivalent to move-to with the exception that point is relative to the “current point”.

(relative-line-to point ...)  
(relative-line-to point ... shape)  

This function is equivalent to line-to with the exception that point is relative to the “current point”.

(relative-curve-to point cntrl1 cntrl2)  
(relative-curve-to point cntrl1 cntrl2 shape)  

This function is equivalent to curve-to with the exception that point, cntrl1 and cntrl2 are relative to the “current point”.

(add-shape other)  
(add-shape other shape)  

Adds shape other to the shape specified by argument shape or parameter object current-shape if shape is not provided. This function is typically used to compose shapes.

(shape-bounds shape)  

Returns the bounding box for the given shape as a rect.

(shape body ...)  

Creates a new empty shape, binds parameter object current-shape to it and executes the body statements in the dynamic scope of this binding. This special form returns the new drawing.

(with-shape shape body ...)  

Binds parameter object current-shape to shape and executes the body statements in the dynamic scope of this binding. This special form returns the result of the last statement in the body.

12.3 Images

Images are immutable objects representing pictures of a fixed size. Images are either loaded from image files or they are created from drawings. Images are either vector-based or bitmap-based. The current image API only allows loading vector-based images from PDF files. Bitmap-based images, on the other hand, can be loaded from PNG, JPG, GIF, etc. image files or they are created by drawing a drawing object into an empty bitmap.

(image? obj)  

Returns #t if obj is an image. Otherwise, it returns #f.

(load-image path)  

Loads an image from the file at path and returns the corresponding image object.

(load-image-asset path type)  
(load-image-asset path type dir)  

Loads an image from the file at the given relative file path and returns the corresponding image object. type refers to the default suffix of the file to load (e.g. "png" for PNG images).

load-image-asset constructs a relative file path in the following way (assuming path does not have a suffix already):
dir/path.type

where dir is "Images" if it is not provided as a parameter. It then searches the asset paths in their given order for a file matching this relative file path. Once the first matching file is found, the file is loaded as an image file and the image gets returned by load-image-asset. It is an error if no matching image was found.

(image-size image)
Returns the size of the given image object.

(bitmap? obj)
Returns #t if obj is a bitmap-based image. Otherwise, it returns #f.

(make-bitmap drawing size scale)
Creates a new bitmap-based image by drawing the object drawing into an empty bitmap of size size. scale is a factor determining the pixel density. For a scale factor of 1.0 (default), size corresponds to the number of horizontal and vertical pixels. For 2.0, the number of pixels is the double of the horizontal and vertical size, etc.

(save-bitmap path bitmap format)
Saves a given bitmap-based image bitmap in a file at filepath path. format is a symbol specifying the image file format. Supported are: png, jpg, gif, bmp, and tiff.

12.4 Transformations

A transformation is an immutable object defining an affine transformation. Transformations can be used to:

- shift,
- scale, and
- rotate coordinate systems.

Transformations are typically used in drawings to transform drawing instructions. They can also be used to transform shapes.

(transformation? obj)
Returns #t if obj is a transformation. Otherwise, it returns #f.

(transformation tf …)
Creates a new transformation by composing the given transformations tf.

(invert tf)
Inverts transformation tf and returns a new transformation object for it.

(translate dx dy)
(translate dx dy tf)
Returns a transformation for shifting the coordinate system by dx and dy. If transformation tf is provided, the translation transformation extends tf.

(scale dx dy)
(scale dx dy tf)
Returns a transformation for scaling the coordinate system by dx and dy. If transformation tf is provided, the scaling transformation extends tf.

(rotate angle)
(rotate angle tf)
Returns a transformation for rotating the coordinate system by angle (in radians). If transformation tf is provided, the rotation transformation extends tf.

## 12.5 Colors

Colors are immutable objects defining colors in terms of four components: red, green, blue and alpha. Library (lispkit draw) currently only supports RGB color spaces.

(lispkit draw) supports the concept of color lists on macOS. A color list is provided as a .plist file and stored in the “ColorLists” asset directory of LispKit. It maps color names expressed as symbols to color values. Color lists need to be loaded explicitly via procedure load-color-list.

### (color? obj)
Returns #t if obj is a color. Otherwise, it returns #f.

### (color spec)
- (color name clist)
- (color r g b)
- (color r g b alpha)

This procedure returns new color objects. If spec is provided, it either is a string containing a color description in hex format, or it is a symbol referring to the name of a color in the default color list (White Yellow Red Purple Orange Magenta Green Cyan Brown Blue Black). If a different color list should be used, its name can be specified via string clist. Procedure (available-color-lists) returns a list of all available color lists. If the color is specified via a hex string, the following formats can be used: "ccc", "#ccc", "rrggbb", and "#rrggbb".

The color can also be specified using color components r, g, b, and alpha. alpha determines the transparency of the color (0.0 = fully transparent, 1.0 = no transparency). The default value for alpha is 1.0.

### (color-red color)
Returns the red color component of color.

### (color-green color)
Returns the green color component of color.

### (color-blue color)
Returns the blue color component of color.

### (color-alpha color)
Returns the alpha color component of color.

### (color->hex color)
Returns a representation of the given color in hex form as a string.

```
(color->hex (color 1.0 0.5 0.1)) ⇒ "#FF801A"
(color->hex (color "#6AF")) ⇒ "#66AAFF"
```

black  [object]
gray white red green blue yellow

Predefined color objects.

### (available-color-lists)
Returns a list of available color lists. The LispKit installation guarantees that there is at least color list "HTML" containing all named colors from the HTML 5 specification.
(available-color-lists)
⇒ ("HTML" "Web Safe Colors" "Crayons" "System" "Apple")

(load-color-list name path)
Procedure
Loads a new color list stored as a .plist file in the assets directory of LispKit at the given file path (which can also refer to color lists outside of the assets directory via absolute file paths). name is a string which specifies the name of the color list. It is added to the list of available colors if loading of the color list was successful. load-color-list returns #t if the color list could be successfully loaded, #f otherwise.

(available-colors clist)
Procedure
Returns a list of color identifiers supported by the given color list. clist is a string specifying the name of the color list.

(available-colors "HTML")
⇒ (YellowGreen Yellow WhiteSmoke White Wheat Violet Turquoise Tomato Thistle Teal Tan
  SandyBlue Snow SlateGrey SlateBlue SlateBlue SkyBlue Silver Sienna SeaShell SeaGreen
  Pink Peru PeachPuff PapayaWhip PaleVioletRed PaleTurquoise PaleGreen PaleGoldenRod Orchid
  OrangeRed Orange OliveDrab Olive OldLace Navy NavajoWhite Moccasin MistyRose MintCream
  MidnightBlue MediumVioletRed MediumTurquoise MediumSpringGreen MediumSlateBlue
  MediumSeaGreen MediumPurple MediumOrchid MediumBlue MediumAquaMarine Maroon Magenta Linen
  LimeGreen Lime LightYellow LightSteelBlue LightSlateGrey LightSlateGray LightSkyBlue
  LightSeaGreen LightSalmon LightPink LightGreen LightGreenLightGoldendrodYellow
  LightCyan LightCoral LightBlue LemonChiffon LawnGreen LavenderBlush Lavender Khaki Ivory
  Indigo IndianRed HotPink HoneyDew Grey GreenYellow Green Gray GoldenRod Gold GhostWhite
  Gainsboro Fuchsia ForestGreen FloralWhite FireBrick DodgerBlue DimGrey DimGray DeepSkyBlue
  DeepPink DarkViolet DarkTurquoise DarkSlateGrey DarkSlateGray DarkSlateBlue DarkSeaGreen
  DarkSalmon DarkRed DarkOrchid DarkOrange DarkOliveGreen DarkMagenta DarkKhaki DarkGrey
  DarkGreen DarkGray DarkGoldenRod DarkCyan DarkBlue Cyan Crimson Cornsilk CornflowerBlue
  Coral Chocolate Chartreuse CadetBlue BurlyWood Brown BlueViolet Blue BlanchedAlmond Black
  Bisque Beige Azure Aquamarine Aqua AntiqueWhite AliceBlue)

12.6 Fonts
Fonts are immutable objects defining fonts in terms of a font name and a font size (in points).

(font? obj)
Procedure
Returns #t if obj is a font. Otherwise, it returns #f.

(font fontname size)
(font familyname size weight trait …)
Procedure
If only two arguments, fontname and size, are provided, font will return a new font object for a font with the given font name and font size (in points). If more than two arguments are provided, font will return a new font object for a font with the given font family name, font size (in points), font weight, as well as a number of font traits.

The weight of a font is specified as an integer on a scale from 0 to 15. Library (lispkit draw) exports the following weight constants:

- ultralight (1)
- thin (2)
- light (3)
- book (4)
- normal (5)
- medium (6)
Font traits are specified as integer masks. The following trait constants are exported from library (lisp-kit draw):

- italic
- boldface
- unitalic
- unboldface
- narrow
- expanded
- condensed
- small-caps
- poster
- compressed
- monospace

(font-name font)
Returns the font name of font.

(font-family-name font)
Returns the font family name of font.

(font-size font)
Returns the font size of font in points.

(font-weight font)
Returns the font weight of font. See documentation of function font for details.

(font-traits font)
Returns the font traits of font as an integer bitmask. See documentation of function font for details.

(font-has-traits font trait …)
Returns #t if font has all the given traits.

(available-fonts)
Returns all the available fonts that have matching font traits.

(available-font-families)
Returns all the available font families, i.e. all font families for which there is at least one font installed.

12.7 Points

A point describes a location on a two-dimensional plane consisting of a x and y coordinate. Points are represented as pairs of floating-point numbers where the car represents the x-coordinate and the cdr represents the y-coordinate. Even though an expression like ' (3.5 . -2.0) does represent a point, it is recommended to always construct points via function point; e.g. (point 3.5 -2.0).
(point? obj)
Returns #t if obj is a valid point. Otherwise, it returns #f.

(point x y)
Returns a point for coordinates x and y.

(move-point point dx dy)
Moves point by dx and dy and returns the result as a point.

(point-x point)
Returns the x-coordinate for point.

(point-y point)
Returns the y-coordinate for point.

zero-point [object]
The zero point, i.e. (point 0.0 0.0).

12.8 Size

A size describes the dimensions of a rectangle consisting of width and height values. Sizes are represented as pairs of floating-point numbers where the car represents the width and the cdr represents the height. Even though an expression like '(5.0 3.0) does represent a size, it is recommended to always construct sizes via function size; e.g. (size 5.0 3.0).

(size? obj)
Returns #t if obj is a valid size. Otherwise, it returns #f.

(size w h)
Returns a size for the given width w and height h.

(size-width size)
Returns the width for size.

(size-height size)
Returns the height for size.

(increase-size size dx dy)
Returns a new size object whose width is increased by dx and whose height is increased by dy.

(scale-size size factor)
Returns a new size object whose width and height is multiplied by factor.

12.9 Rects

A rect describes a rectangle in terms of an upper left point and a size. Rects are represented as pairs whose car is a point and whose cdr is a size. Even though an expression like '((1.0 . 2.0) . (3.0 4.0)) does represent a rect, it is recommended to always construct rects via function rect; e.g. (rect (point 1.0 2.0) (size 3.0 4.0)).

(rect? obj)
Returns #t if obj is a valid rect. Otherwise, it returns #f.

(rect point size)
(rect x y width height)
Returns a rect either from the given point and size, or from x-coordinate x, y-coordinate y, width w, and height h.

(move-rect rect dx dy)
Moves rect by dx and dy and returns the result.

(rect-point rect)
Returns the upper left corner point of rect.

(rect-size rect)
Returns the size of the rect.

(rect-x rect)
Returns the x-coordinate of the upper left corner point of rect.

(rect-y rect)
Returns the y-coordinate of the upper left corner point of rect.

(rect-size rect)
Returns the size of rect as a size, i.e. as a pair of floating-point numbers where the car represents the width and the cdr represents the height of rect.

(rect-width rect)
Returns the width of rect.

(rect-height rect)
Returns the height of rect.
13 LispKit Draw Turtle

Library (lispkit draw turtle) defines a simple “turtle graphics” API. The API provides functionality for creating turtles and for moving turtles on a plane generating drawings as a side-effect. A drawing is a data structure defined by library (lispkit draw).

A turtle is defined in terms of the following components: - A position \((x, y)\) defining the coordinates where the turtle is currently located within a coordinate system defined by parameters used to create the turtle via \texttt{make-turtle} - A heading \textit{angle} which defines the direction in degrees into which the turtle is moving - A boolean flag \textit{pen down} which, if set to \#t, will make the turtle draw lines on the graphics plane when moving. - A \textit{line width} defining the width of lines drawn by the turtle - A \textit{color} defining the color of lines drawn by the turtle - A \textit{drawing} which records the moves of the turtle while the pen is down.

Turtles are mutable objects created via \texttt{make-turtle}. The functions listed below change the state of a turtle. In particular, they generate a drawing as a side-effect which can be accessed via \texttt{turtle-drawing} . For most functions, the turtle is an optional argument. If it is not provided, the function applies to the turtle provided by the \texttt{current-turtle} parameter object.

\texttt{current-turtle}

Defines the \texttt{current turtle}, which is used as a default by all functions for which the turtle argument is optional. If there is no current turtle, this parameter is set to \#f.

\texttt{(turtle? obj)}

Returns \#t if \textit{obj} is a turtle. Otherwise, it returns \#f.

\texttt{(make-turtle x y scale)}

Returns a new turtle object. \textit{x} and \textit{y} determine the “home point” of the turtle. This is equivalent to the zero point of the coordinate system in which the turtle navigates. \textit{scale} is a scaling factor.

\texttt{(turtle-drawing turtle)}

Returns the drawing associated with the given \textit{turtle}.

\texttt{(pen-up)}
\texttt{(pen-up turtle)}

Lifts \textit{turtle} from the plane. If \textit{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used. Subsequent \texttt{forward} and \texttt{backward} operations don’t lead to lines being drawn. Only the current coordinates are getting updated.

\texttt{(pen-down)}
\texttt{(pen-down turtle)}

Drops \textit{turtle} onto the plane. If \textit{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used. Subsequent \texttt{forward} and \texttt{backward} operations will lead to lines being drawn.

\texttt{(pen-color color)}
\texttt{(pen-color color turtle)}

Sets the drawing color of \textit{turtle} to \textit{color}. If \textit{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used. \textit{color} is a color object as defined by library (lispkit draw).

\texttt{(pen-size size)}
\texttt{(pen-size size turtle)}
Sets the pen size of \texttt{turtle} to \textit{size}. If \texttt{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used. The pen size corresponds to the width of lines drawn by \texttt{forward} and \texttt{backward}.

\begin{verbatim}
(home)
(home turtle)
\end{verbatim}

Moves \texttt{turtle} to its home position. If \texttt{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used.

\begin{verbatim}
(move x y)
(move x y turtle)
\end{verbatim}

Moves \texttt{turtle} to the position described by the coordinates \textit{x} and \textit{y}. If \texttt{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used.

\begin{verbatim}
(heading angle)
(heading angle turtle)
\end{verbatim}

Sets the heading of \texttt{turtle} to \textit{angle}. If \texttt{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used. \textit{angle} is expressed in terms of degrees.

\begin{verbatim}
(turn angle)
(turn angle turtle)
\end{verbatim}

Adjusts the heading of \texttt{turtle} by \textit{angle} degrees. If \texttt{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used.

\begin{verbatim}
(right angle)
(right angle turtle)
\end{verbatim}

Adjusts the heading of \texttt{turtle} by \textit{angle} degrees. If \texttt{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used.

\begin{verbatim}
(left angle)
(left angle turtle)
\end{verbatim}

Adjusts the heading of \texttt{turtle} by -\textit{angle} degrees. If \texttt{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used.

\begin{verbatim}
(forward distance)
(forward distance turtle)
\end{verbatim}

Moves \texttt{turtle} forward by \textit{distance} units drawing a line if the pen is down. If \texttt{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used.

\begin{verbatim}
(backward distance)
(backward distance turtle)
\end{verbatim}

Moves \texttt{turtle} backward by \textit{distance} units drawing a line if the pen is down. If \texttt{turtle} is not provided, the turtle defined by \texttt{current-turtle} is used.
# 14 LispKit Dynamic

## 14.1 Dynamic bindings

**(make-parameter init)**

**(make-parameter init converter)**

Returns a newly allocated parameter object, which is a procedure that accepts zero arguments and returns the value associated with the parameter object. Initially, this value is the value of (converter init), or of init if the conversion procedure converter is not specified. The associated value can be temporarily changed using parameterize. The default associated value can be changed by invoking the parameter object as a function with the new value as the only argument.

Parameter objects can be used to specify configurable settings for a computation without the need to pass the value to every procedure in the call chain explicitly.

**(parameterize ((param value) ...) body)**

A parameterize expression is used to change the values returned by specified parameter objects param during the evaluation of body. The param and value expressions are evaluated in an unspecified order. The body is evaluated in a dynamic environment in which calls to the parameters return the results of passing the corresponding values to the conversion procedure specified when the parameters were created. Then the previous values of the parameters are restored without passing them to the conversion procedure. The results of the last expression in the body are returned as the results of the entire parameterize expression.

```lisp
(define radix
  (make-parameter 10 (lambda (x)
                        (if (and (exact-integer? x) (<= 2 x 16))
                            x
                            (error "invalid radix")))))

(define (f n) (number->string n (radix)))
(f 12) ⇒ "12"
(parameterize ((radix 2)) (f 12)) ⇒ "1100"
(f 12) ⇒ "12"
(radix 16)
(parameterize ((radix 0)) (f 12)) ⇒ error: invalid radix
```

**(make-dynamic-environment)**

Returns a newly allocated copy of the current dynamic environment. Dynamic environments are represented as mutable hashtables.

**(dynamic-environment)**

Returns the current dynamic environment represented as mutable hashtables.

**(set-dynamic-environment! hashtable)**

Sets the current dynamic environment to the given dynamic environment object. Dynamic environments are modeled as hashtables.
14.2 Continuations

(continuation? obj)
Returns #t if obj is a continuation procedure, #f otherwise.

(call-with-current-continuation proc)
(call/cc proc)

The procedure call-with-current-continuation (or its equivalent abbreviation call/cc) packages the current continuation as an “escape procedure” and passes it as an argument to proc. It is an error if proc does not accept one argument.

The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead use the continuation that was in effect when the escape procedure was created. Calling the escape procedure will cause the invocation of before and after thunks installed using dynamic-wind.

The escape procedure accepts the same number of arguments as the continuation to the original call to call-with-current-continuation. Most continuations take only one value. Continuations created by the call-with-values procedure (including the initialization expressions of define-values, let-values, and let*-values expressions), take the number of values that the consumer expects. The continuations of all non-final expressions within a sequence of expressions, such as in lambda, case-lambda, begin, let, let*, letrec, letrec*, let-values, let*-values, let-syntax, letrec-syntax, parameterize, guard, case, cond, when, and unless expressions, take an arbitrary number of values because they discard the values passed to them in any event. The effect of passing no val- ues or more than one value to continuations that were not created in one of these ways is unspecified.

The escape procedure that is passed to proc has unlimited extent just like any other procedure in Scheme. It can be stored in variables or data structures and can be called as many times as desired. However, like the raise and error procedures, it never returns to its caller.

The following examples show only the simplest ways in which call-with-current-continuation is used. If all real uses were as simple as these examples, there would be no need for a procedure with the power of call-with-current-continuation.

(dynamic-wind before thunk after)
Calls thunk without arguments, returning the result(s) of this call. before and after are called, also without arguments, as required by the following rules. Note that, in the absence of calls to continuations captured using call-with-current-continuation, the three arguments are called once each, in order. before is called whenever execution enters the dynamic extent of the call to thunk and after is called whenever it
exists that dynamic extent. The dynamic extent of a procedure call is the period between when the call is initiated and when it returns. The before and after thunks are called in the same dynamic environment as the call to dynamic-wind. In Scheme, because of call-with-current-continuation, the dynamic extent of a call is not always a single, connected time period. It is defined as follows:

- The dynamic extent is entered when execution of the body of the called procedure begins.
- The dynamic extent is also entered when execution is not within the dynamic extent and a continuation is invoked that was captured (using call-with-current-continuation) during the dynamic extent.
- It is exited when the called procedure returns.
- It is also exited when execution is within the dynamic extent and a continuation is invoked that was captured while not within the dynamic extent.

If a second call to dynamic-wind occurs within the dynamic extent of the call to thunk and then a continuation is invoked in such a way that the afters from these two invocations of dynamic-wind are both to be called, then the after associated with the second (inner) call to dynamic-wind is called first.

If a second call to dynamic-wind occurs within the dynamic extent of the call to thunk and then a continuation is invoked in such a way that the before from these two invocations of dynamic-wind are both to be called, then the before associated with the first (outer) call to dynamic-wind is called first.

If invoking a continuation requires calling the before from one call to dynamic-wind and the after from another, then the after is called first.

The effect of using a captured continuation to enter or exit the dynamic extent of a call to before or after is unspecified.

```lisp
(let ((path '()))
  (c #f)
  (let ((add (lambda (s)
                (set! path (cons s path))))
        (lambda () (add 'connect))
        (lambda () (add (call-with-current-continuation
                         (lambda (c0) (set! c c0) 'talk1))))
        (lambda () (add 'disconnect)))
    (if (< (length path) 4)
      (c 'talk2)
      (reverse path)))
  (connect talk1 disconnect connect talk2 disconnect))

(return obj)
Returns to the top-level of the read-eval-print loop with obj as the result (or terminates the program with obj as its return value).

14.3 Exceptions

(with-exception-handler handler thunk)

The with-exception-handler procedure returns the results of invoking thunk. handler is installed as the current exception handler in the dynamic environment used for the invocation of thunk. It is an error if handler does not accept one argument. It is also an error if thunk does not accept zero arguments.
After printing, the second example then raises another exception: “exception handler returned”.

**(guard (var cond-clause …) body)**

The body is evaluated with an exception handler that binds the raised object to var and, within the scope of that binding, evaluates the clauses as if they were the clauses of a **cond** expression. That implicit **cond** expression is evaluated with the continuation and dynamic environment of the **guard** expression. If every **cond-clause's test** evaluates to **#f** and there is no “else” clause, then **raise-continuable** is invoked on the raised object within the dynamic environment of the original call to **raise** or **raise-continuable**, except that the current exception handler is that of the **guard** expression.

Please note that each **cond-clause** is as in the specification of **cond**.

**(make-error message irrilist)**

Returns a newly allocated custom error object consisting of **message** as its error message and the list of irritants **irrilist**.

**(make-assertion-error procline expr)**

Returns a newly allocated assertion error object referring to a procedure of name **procline** and an expression **expr** which triggered the assertion. Assertion errors that were raised should never be caught as they indicate a violation of an invariant.

**(raise obj)**

Raises an exception by invoking the current exception handler on **obj**. The handler is called with the same dynamic environment as that of the call to **raise**, except that the current exception handler is the one that was in place when the handler being called was installed. If the handler returns, a secondary exception is raised in the same dynamic environment as the handler. The relationship between **obj** and the object raised by the secondary exception is unspecified.

**(raise-continuable obj)**

Raises an exception by invoking the current exception handler on **obj**. The handler is called with the same dynamic environment as the call to **raise-continuable**, except that: (1) the current exception handler is the one that was in place when the handler being called was installed, and (2) if the handler being called returns, then it will again become the current exception handler. If the handler returns, the values it returns become the values returned by the call to **raise-continuable**.
(lambda ()
  (+ (raise-continuable "should be a number") 23)))
prints: should be a number
⇒ 65

(error message obj ...)
 Raises an exception as if by calling raise on a newly allocated error object which encapsulates
the information provided by message, as well as any obj, known as the irritants. The procedure error-
object? must return #t on such objects. message is required to be a string.

(define (null-list? l)
  (cond ((pair? l) #f)
        ((null? l) #t)
        (else (error "null-list?: argument out of domain" l))))

(assertion expr)
 Raises an exception as if by calling raise on a newly allocated assertion error object encapsulating expr
as the expression which triggered the assertion failure and the current procedure's name. Assertion errors
that are raised via assertion should never be caught as they indicate a violation of a critical invariant.

(define (null-list? l)
  (cond ((pair? l) #f)
        ((null? l) #t)
        (else (assertion '((list? l))))))

(assert expr0 expr1 ...)
 Executes expr0, expr1, ... in the given order and raises an assertion error as soon as the first expression is
evaluating to #f. The raised assertion error encapsulates the expression that evaluated to #f and the
name of the procedure in which the assert statement was placed.

(define (drop-elements xs n)
  (assert (list? xs) (fixnum? n) (not (negative? n)))
  (if (or (null? xs) (zero? n)) xs (drop-elements (cdr xs) (fx1- n))))

(error-object? obj)
 Returns #t if obj is an error object, #f otherwise. Error objects are either implicitly created via error
or they are created explicitly with procedure make-error.

(error-object-message err)
 Returns the message (which is a string) encapsulated by the error object err.

(error-object-irritants err)
 Returns a list of the irritants encapsulated by the error object err.

(error-object-stacktrace err)
 Returns a list of procedures representing the stack trace encapsulated by the error object err. The stack
trace reflects the currently active procedures at the time the error object was created (either implicitly via
error or explicitly via make-error).

(read-error? obj)
 This error type predicate returns #t if obj is an error object raised by the read procedure; otherwise, it
returns #f.

(file-error? obj)
 This error type predicate returns #t if obj is an error object raised by the inability to open an input or
output port on a file; otherwise, it returns #f.
14.4 Exiting

(exit)
(exit obj)

Runs all outstanding `dynamic-wind` after procedures, terminates the running program, and communicates an exit value to the operating system. If no argument is supplied, or if `obj` is `#t`, the `exit` procedure should communicate to the operating system that the program exited normally. If `obj` is `#f`, the `exit` procedure will communicate to the operating system that the program exited abnormally. Otherwise, `exit` should translate `obj` into an appropriate exit value for the operating system, if possible. The `exit` procedure must not signal an exception or return to its continuation.

(emergency-exit)
(emergency-exit obj)

Terminates the program without running any outstanding `dynamic-wind` “after procedures” and communicates an exit value to the operating system in the same manner as `exit`. 


15 LispKit Enum

Library (lispkit enum) provides an implementation of enumerated values and sets of enumerated values based on the API defined by R6RS.

Enumerated values are represented by ordinary symbols, while finite sets of enumerated values form a separate type, known as enumeration set. The enumeration sets are further partitioned into sets that share the same universe and enumeration type. These universes and enumeration types are created by the make-enumeration procedure. Each call to that procedure creates a new enumeration type.

In the descriptions of the following procedures, enum-set ranges over the enumeration sets, which are defined as the subsets of the universes that can be defined using make-enumeration.

(make-enumeration symbol-list) procedure
Argument symbol-list must be a list of symbols. The make-enumeration procedure creates a new enumeration type whose universe consists of those symbols (in canonical order of their first appearance in the list) and returns that universe as an enumeration set whose universe is itself and whose enumeration type is the newly created enumeration type.

(enum-set-universe enum-set) procedure
Returns the set of all symbols that comprise the universe of its argument enum-set, as an enumeration set.

(enum-set-indexer enum-set) procedure
Returns a unary procedure that, given a symbol that is in the universe of enum-set, returns its 0-origin index within the canonical ordering of the symbols in the universe; given a value not in the universe, the unary procedure returns #f.

(let* ((e (make-enumeration '(red green blue)))
       (i (enum-set-indexer e)))
  (list (i 'red) (i 'green) (i 'blue) (i 'yellow)))
⇒ (0 1 2 #f)

The enum-set-indexer procedure could be defined as follows using the memq procedure:

(let* ((symbols (enum-set->list (enum-set-universe set)))
       (cardinality (length symbols)))
  (lambda (x)
    (cond ((memq x symbols) =>
           (lambda (probe) (- cardinality (length probe))))
          (else #f)))))

(enum-set-constructor enum-set) procedure
Returns a unary procedure that, given a list of symbols that belong to the universe of enum-set, returns a subset of that universe that contains exactly the symbols in the list. The values in the list must all belong to the universe.

(enum-set->list enum-set) procedure
Returns a list of the symbols that belong to its argument, in the canonical order of the universe of enum-set.
(let* ((e (make-enumeration '(red green blue)))
       (c (enum-set-constructor e)))
  (enum-set->list (c '(blue red)))
⇒ (red blue)

**Procedure**

### enum-set-member? symbol enum-set

**Procedure**

### enum-set-subset? enum-set1 enum-set2

**Procedure**

### enum-set=? enum-set1 enum-set2

The `enum-set-member?` procedure returns `#t` if its first argument is an element of its second argument, `#f` otherwise.

The `enum-set-subset?` procedure returns `#t` if the universe of `enum-set1` is a subset of the universe of `enum-set2` (considered as sets of symbols) and every element of `enum-set1` is a member of `enum-set2`. It returns `#f` otherwise.

The `enum-set=?` procedure returns `#t` if `enum-set1` is a subset of `enum-set2` and vice versa, as determined by the `enum-set-subset?` procedure. This implies that the universes of the two sets are equal as sets of symbols, but does not imply that they are equal as enumeration types. Otherwise, `#f` is returned.

(let* ((e (make-enumeration '(red green blue)))
       (c (enum-set-constructor e)))
  (list
    (enum-set-member? 'blue (c '(red blue)))
    (enum-set-member? 'green (c '(red blue)))
    (enum-set-subset? (c '(red blue)) e)
    (enum-set-subset? (c '(red blue)) (c '(blue red)))
    (enum-set-subset? (c '(red blue)) (c '(red)))
    (enum-set=? (c '(red blue)) (c '(blue red))))
⇒ (#t #f #t #t #f #t)

**Procedure**

### enum-set-union enum-set1 enum-set2

### enum-set-intersection enum-set1 enum-set2

### enum-set-difference enum-set1 enum-set2

Arguments `enum-set1` and `enum-set2` must be enumeration sets that have the same enumeration type.


(let* ((e (make-enumeration '(red green blue)))
       (c (enum-set-constructor e)))
  (list
    (enum-set->list (enum-set-union (c '(blue)) (c '(red))))
    (enum-set->list
      (enum-set-intersection (c '(red green)) (c '(red blue))))
    (enum-set->list
      (enum-set-difference (c '(red green)) (c '(red blue))))
⇒ ((red blue) (red) (green))

**Procedure**

### enum-set-complement enum-set

Returns `enum-set`'s complement with respect to its universe.

(let* ((e (make-enumeration '(red green blue)))
       (c (enum-set-constructor e)))
  (enum-set->list (enum-set-complement (c '(red))))
⇒ (green blue)
(enum-set-projection enum-set1 enum-set2)

Projects enum-set1 into the universe of enum-set2, dropping any elements of enum-set1 that do not belong to the universe of enum-set2. If enum-set1 is a subset of the universe of its second, no elements are dropped, and the injection is returned.

(let ((e1 (make-enumeration '(red green blue black)))
     (e2 (make-enumeration '(red black white))))
  (enum-set->list (enum-set-projection e1 e2)))
⇒ (red black)

(define-enumeration type-name (symbol …) constructor)

The define-enumeration form defines an enumeration type and provides two macros for constructing its members and sets of its members. A define-enumeration form is a definition and can appear anywhere any other definition can appear.

(type-name) is an identifier that is bound as a syntactic keyword; symbol … are the symbols that comprise the universe of the enumeration (in order).

(type-name symbol) checks whether the name of symbol is in the universe associated with type-name. If it is, (type-name symbol) is equivalent to symbol. It is a syntax violation if it is not.

(constructor) is an identifier that is bound to a syntactic form that, given any finite sequence of the symbols in the universe, possibly with duplicates, expands into an expression that evaluates to the enumeration set of those symbols.

(constructor symbol …) checks whether every … is in the universe associated with type-name. It is a syntax violation if one or more is not. Otherwise (constructor symbol …) is equivalent to ((enum-set-constructor (constructor-syntax)) '(symbol …)).

Here is a complete example:

(define-enumeration color (black white purple maroon) color-set)
(color black) ⇒ black
(color purpel) ⇒ error: symbol not in enumeration universe
(enum-set->list (color-set)) ⇒ ()
(enum-set->list (color-set maroon white)) ⇒ (white maroon)
This library defines an API for *growable vectors*. Just like regular vectors, *growable vectors* are heterogeneous sequences of elements which are indexed by a range of integers. Unlike for regular vectors, the length of a *growable vector* is not fixed. Growable vectors may expand or shrink in length. Nevertheless, growable vectors are fully compatible to regular vectors and all operations from library *(lispkit vector)* may also be used in combination with growable vectors. The main significance of library *(lispkit gvector)* is in providing functions to construct growable vectors. Growable vectors are always *mutable* by design.

Just like for vectors with a fixed length, the valid indexes of a growable vector are the exact, non-negative integers less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the growable vector.

Two growable vectors are *equal*? if they have the same length, and if the values in corresponding slots of the vectors are *equal*?. A growable vector is never *equal*? a regular vector of fixed length.

Growable vectors are written using the notation #g(obj ...). For example, a growable vector of initial length 3 containing the number one as element 0, the list (8 16 32) as element 1, and the string “Scheme” as element 2 can be written as follows: #g(1 (8 16 32) "Scheme").

Growable vector constants are self-evaluating, so they do not need to be quoted in programs.

### 16.1 Predicates

**function**

(\text{gvector?} \ \text{obj})

Returns \#t if \text{obj} is a growable vector; otherwise returns \#f.

**function**

(\text{gvector-empty?} \ \text{obj})

Returns \#t if \text{obj} is a growable vector of length zero; otherwise returns \#f.

### 16.2 Constructors

**function**

(make-gvector)

Returns a newly allocated growable vector of capacity \text{c}. The capacity is used to pre-allocate space for up to \text{c} elements.

**function**

(make-gvector \text{c})

Returns a newly allocated growable vector whose elements contain the given arguments.

\[(\text{gvector} \ a \ b \ c) \Rightarrow \#g(a \ b \ c)\]

**function**

(list->gvector \text{list})

The \text{list->gvector} procedure returns a newly created growable vector initialized to the elements of the list \text{list} in the order of the list.
(list->gvector '(a b c)) ⇒ #g(a b c)

(vector->gvector vector)

procedure

Returns a newly allocated growable vector initialized to the elements of the vector vector in the order of vector.

(gvector-copy vector)

(gvector-copy vector start)

(gvector-copy vector start end)

procedure

Returns a newly allocated copy of the elements of the given growable vector between start and end, but excluding the element at index end. The elements of the new vector are the same (in the sense of eqv?) as the elements of the old.

(gvector-append vector ...)

procedure

Returns a newly allocated growable vector whose elements are the concatenation of the elements of the given vectors.

(gvector-append #((a b c) #g(d e f)) ⇒ #g(a b c d e f)

(gvector-concatenate vector xs)

procedure

Returns a newly allocated growable vector whose elements are the concatenation of the elements of the vectors in xs. xs is a proper list of vectors.

(gvector-concatenate '#((a b c) #(d) #g(e f))) ⇒ #g(a b c d e f)

(gvector-map f vector1 vector2 ...)

procedure

Constructs a new growable vector of the shortest size of the vector arguments vector1, vector2, etc. Each element at index i of the new vector is mapped from the old vectors by (f (vector-ref vector1 i) (vector-ref vector2 i) ...). The dynamic order of the application of f is unspecified.

(gvector-map + #((1 2 3 4 5) #g(10 20 30 40)) ⇒ #g(11 22 33 44)

(gvector-map/index f vector1 vector2 ...)

procedure

Constructs a new growable vector of the shortest size of the vector arguments vector1, vector2, etc. Each element at index i of the new vector is mapped from the old vectors by (f i (vector-ref vector1 i) (vector-ref vector2 i) ...). The dynamic order of the application of f is unspecified.

(gvector-map/index (lambda (i x y) (cons i (+ x y))) #g(1 2 3) #g(10 20 30)) ⇒ #g((0 . 11) (1 . 22) (2 . 33))

16.3 Managing vector state

(gvector-length vector)

procedure

Returns the number of elements in growable vector vector as an exact integer.

(gvector-ref vector k)

procedure

The gvector-ref procedure returns the contents of element k of vector. It is an error if k is not a valid index of vector or if vector is not a growable vector.
(gvector-ref '#g(1 2 3 5 8 13 21) 5) ⇒ 8
(gvector-ref '#g(1 2 3 5 8 13 21) (exact (round (* 2 (acos -1))))) ⇒ 13

(gvector-set! vector k obj)procedure
The vector-set! procedure stores obj in element k of growable vector vector. It is an error if k is not a valid index of vector or if vector is not a growable vector.

(let ((vec (gvector 0 '(2 2 2 2) "Anna")))
  (gvector-set! vec 1 '("Sue" "Sue"))
  vec)
⇒ #g(0 (“Sue" "Sue") "Anna")

(gvector-add! vector obj …)procedure
Appends the values obj, … to growable vector vector. This increases the length of the growable vector by the number of obj arguments.

(let ((vec (gvector 0 '(2 2 2 2) "Anna")))
  (gvector-add! vec "Micha")
  vec)
⇒ #g(0 (2 2 2 2) "Anna" "Micha")

(gvector-insert! vector k obj)procedure
Inserts the value obj into growable vector vector at index k. This increases the length of the growable vector by one.

(let ((vec (gvector 0 '(2 2 2 2) "Anna")))
  (gvector-insert! vec 1 "Micha")
  vec)
⇒ #g(0 "Micha" (2 2 2 2) "Anna")

(gvector-remove! vector k)procedure
Removes the element at index k from growable vector vector. This decreases the length of the growable vector by one.

(let ((vec (gvector 0 '(2 2 2 2) "Anna")))
  (gvector-remove! vec 1)
  vec)
⇒ #g(0 "Anna")

(gvector-remove-last! vector)procedure
Removes the last element of the growable vector vector. This decreases the length of the growable vector by one.

(let ((vec (gvector 0 '(2 2 2 2) "Anna")))
  (gvector-remove-last! vec)
  vec)
⇒ #g(0 (2 2 2 2))

16.4 Destructive growable vector operations

Procedures which operate only on a part of a growable vector specify the applicable range in terms of an index interval [start; end]; i.e. the end index is always exclusive.
(gvector-copy! to at from)
(gvector-copy! to at from start)
(gvector-copy! to at from start end)

Copies the elements of vector from between start and end to growable vector to, starting at at. The order in which elements are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary vector and then into the destination. start defaults to 0 and end defaults to the length of vector.

It is an error if at is less than zero or greater than the length of to. It is also an error if (- (gvector-length to) at) is less than (- end start).

(define a (vector 1 2 3 4 5))
(define b (gvector 10 20 30 40 50))
(gvector-copy! b 1 a 0 2)
b ⇒ #g(10 1 2 40 50)

(gvector-reverse! vector)
(gvector-reverse! vector start)
(gvector-reverse! vector start end)

Procedure gvector-reverse! destructively reverses the contents of growable vector between start and end. start defaults to 0 and end defaults to the length of vector.

(define a (gvector 1 2 3 4 5))
(vector-reverse! a)
a ⇒ #g(5 4 3 2 1)

(gvector-sort! pred vector)

Procedure gvector-sort! destructively sorts the elements of growable vector vector using the "less than" predicate pred.

(define a (gvector 7 4 9 1 2 8 5))
(gvector-sort! < a)
a ⇒ #g(1 2 4 5 7 8 9)

(gvector-map! f vector1 vector2 ...)

Similar to gvector-map which maps the various elements into a new vector via function f, procedure gvector-map! destructively inserts the mapped elements into growable vector vector1. The dynamic order in which f gets applied to the elements is unspecified.

(define a (gvector 1 2 3 4))
(gvector-map! + a #(10 20 30))
a ⇒ #g(11 22 33 4)

(gvector-map/index! f vector1 vector2 ...)

Similar to gvector-map/index which maps the various elements together with their index into a new vector via function f, procedure gvector-map/index! destructively inserts the mapped elements into growable vector vector1. The dynamic order in which f gets applied to the elements is unspecified.

(define a #g(1 2 3 4))
(gvector-map/index! (lambda (i x y) (cons i (+ x y))) a #(10 20 30))
a ⇒ #g((0 . 11) (1 . 22) (2 . 33) 4)
16.5 Converting growable vectors

**(gvector->list vector)**
**(gvector->list vector start)**
**(gvector->list vector start end)**

The `gvector->list` procedure returns a newly allocated list of the objects contained in the elements of growable vector between `start` and `end` in the same order as in `vector`.

```
(gvector->list '#g(dah dah didah)) ⇒ (dah dah didah)
(gvector->list '#g(dah dah didah) 1 2) ⇒ (dah)
```

**(gvector->vector vector)**
**(gvector->vector vector start)**
**(gvector->vector vector start end)**

The `gvector->list` procedure returns a newly allocated list of the objects contained in the elements of growable vector `vector` between `start` and `end` in the same order as in `vector`.

```
(gvector->list '#(dah dah didah)) ⇒ error since the argument is not a gvector
(gvector->list '#g(dah dah didah) 1 2) ⇒ (dah)
```
17 LispKit Hashtable

Library (lispkit hashtable) provides a native implementation of hash tables based on the API defined by R6RS.

A hashtable is a data structure that associates keys with values. Any object can be used as a key, provided a hash function and a suitable equivalence function is available. A hash function is a procedure that maps keys to exact integer objects. It is the programmer's responsibility to ensure that the hash function is compatible with the equivalence function, which is a procedure that accepts two keys and returns true if they are equivalent and #f otherwise. Standard hash tables for arbitrary objects based on the eq?, eqv?, , and equal? predicates are provided. Also, hash functions for arbitrary objects, strings, and symbols are included.

The specification below uses the hashtable parameter name for arguments that must be hashtables, and the key parameter name for arguments that must be hashtable keys.

17.1 Constructors

(make-eqHashtable) procedure
(make-eqHashtable k)
Returns a newly allocated mutable hashtable that accepts arbitrary objects as keys and compares those keys with eq?. If an argument is given, the initial capacity of the hashtable is set to approximately k elements.

(make-eqvHashtable) procedure
(make-eqvHashtable k)
Returns a newly allocated mutable hashtable that accepts arbitrary objects as keys and compares those keys with eqv?. If an argument is given, the initial capacity of the hashtable is set to approximately k elements.

(make-equalHashtable) procedure
(make-equalHashtable k)
Returns a newly allocated mutable hashtable that accepts arbitrary objects as keys and compares those keys with equal?. If an argument is given, the initial capacity of the hashtable is set to approximately k elements.

(makeHashtable hash equiv) procedure
(makeHashtable hash equiv k)
Returns a newly allocated mutable hashtable using hash as the hash function and equiv as the equivalence function for comparing keys. If a third argument k is given, the initial capacity of the hashtable is set to approximately k elements.

hash and equiv must be procedures. hash should accept a key as an argument and should return a non-negative exact integer object. equiv should accept two keys as arguments and return a single boolean value. Neither procedure should mutate the hashtable returned by makeHashtable. Both hash and equiv should behave like pure functions on the domain of keys. For example, the string-hash and
string?= procedures are permissible only if all keys are strings and the contents of those strings are never changed so long as any of them continues to serve as a key in the hashtable. Furthermore, any pair of keys for which equiv returns true should be hashed to the same exact integer objects by hash.

(alist->eq-hashtable alist) (alist->eq-hashtable alist k)
Returns a newly allocated mutable hashtable consisting of the mappings contained in the association list alist. Keys are compared with eq?. If argument k is given, the capacity of the returned hashtable is set to at least k elements.

(alist->eqv-hashtable alist) (alist->eqv-hashtable alist k)
Returns a newly allocated mutable hashtable consisting of the mappings contained in the association list alist. Keys are compared with eqv?. If argument k is given, the capacity of the returned hashtable is set to at least k elements.

(alist->equal-hashtable alist) (alist->equal-hashtable alist k)
Returns a newly allocated mutable hashtable consisting of the mappings contained in the association list alist. Keys are compared with equal?. If argument k is given, the capacity of the returned hashtable is set to at least k elements.

(hashtable-copy hashtable) (hashtable-copy hashtable mutable)
Returns a copy of hashtable. If the mutable argument is provided and is true, the returned hashtable is mutable; otherwise it is immutable.

(hashtable-empty-copy hashtable)
Returns a new mutable hashtable that uses the same hash and equivalence functions like hashtable.

17.2 Type tests

(hashtable? obj)
Returns #t if obj is a hashtable. Otherwise, it returns #f.

(eq-hashtable? obj)
Returns #t if obj is a hashtable which uses eq? for comparing keys. Otherwise, it returns #f.

(eqv-hashtable? obj)
Returns #t if obj is a hashtable which uses eqv? for comparing keys. Otherwise, it returns #f.

(equal-hashtable? obj)
Returns #t if obj is a hashtable which uses equal? for comparing keys. Otherwise, it returns #f.

17.3 Inspection

(hashtable-equivalence-function hashtable)
Returns the equivalence function used by hashtable to compare keys. For hashtables created with make-eq-hashtable, make-eqv-hashtable, and make-equal-hashtable, returns eq?, eqv?, and equal? respectively.
(hashtable-hash-function hashtable)  
(declare (function hashtable-hash-function (hashtable) (hashtable-force?))  
 Returns the hash function used by hashtable. For hashtables created by make-eq-hashtable and make-eqv-hashtable, #f is returned. This behavior can be disabled if boolean parameter force? is being provided and set to #t. In this case, hashtable-hash-function will also return hash functions for eq and eqv-based hashtables.

(hashtable-mutable? hashtable)  
(declare (function hashtable-mutable? (hashtable))  
 Returns #t if hashtable is mutable, otherwise #f.

17.4 Hash functions

The equal-hash, string-hash, and string-ci-hash procedures are acceptable as the hash functions of a hashtable only, if the keys on which they are called are not mutated while they remain in use as keys in the hashtable.

(equal-hash obj)  
(declare (function equal-hash (obj))  
 Returns an integer hash value for obj, based on its structure and current contents. This hash function is suitable for use with equal? as an equivalence function. Like equal?, the equal-hash procedure must always terminate, even if its arguments contain cycles.

(eqv-hash obj)  
(declare (function eqv-hash (obj))  
 Returns an integer hash value for obj, based on obj’s identity. This hash function is suitable for use with eqv? as an equivalence function.

(eq-hash obj)  
(declare (function eq-hash (obj))  
 Returns an integer hash value for obj, based on obj’s identity. This hash function is suitable for use with eq? as an equivalence function.

(string-hash str)  
(declare (function string-hash (str))  
 Returns an integer hash value for str, based on its current characters. This hash function is suitable for use with string=? as an equivalence function.

(string-ci-hash str)  
(declare (function string-ci-hash (str))  
 Returns an integer hash value for str based on its current characters, ignoring case. This hash function is suitable for use with string-ci=? as an equivalence function.

(symbol-hash sym)  
(declare (function symbol-hash (sym))  
 Returns an integer hash value for symbol sym.

17.5 Procedures

(hashtable-size hashtable)  
(declare (function hashtable-size (hashtable))  
 Returns the number of keys contained in hashtable as an exact integer object.

(hashtable-load hashtable)  
(declare (function hashtable-load (hashtable))  
 Returns the load factor of the hashtable. The load factor is defined as the ratio between the number of keys and the number of hash buckets of hashtable.

(hashtable-ref hashtable key default)  
(declare (function hashtable-ref (hashtable key default))  
 Returns the value in hashtable associated with key. If hashtable does not contain an association for key, default is returned.
(hashtable-get hashtable key)
Returns a pair consisting of a key matching key and associated value from hashtable. If hashtable does not contain an association for key, hashtable-get returns #f.

For example, for a hashtable ht containing the mapping 3 to "three", (hashtable-get ht 3) will return (3 . "three").

(hashtable-set! hashtable key obj)
Changes hashtable to associate key with obj, adding a new association or replacing any existing association for key.

(hashtable-delete! hashtable key)
Removes any association for key within hashtable.

(hashtable-add! hashtable key obj)
Changes hashtable to associate key with obj, adding a new association for key. The difference to hashtable-set! is that existing associations of key will remain in hashtable, whereas hashtable-set! replaces an existing association for key.

(hashtable-remove! hashtable key)
Removes the association for key within hashtable which was added last, and returns it as a pair consisting of the key matching key and its associated value. If there is no association of key in hashtable, hashtable-remove! will return #f.

(alist->hashtable! hashtable alist)
Adds all the associations from alist to hashtable using hashtable-add!.

(hashtable-contains? hashtable key)
Returns #t if hashtable contains an association for key, #f otherwise.

(hashtable-update! hashtable key proc default)
hashtable-update! applies proc to the value in hashtable associated with key, or to default if hashtable does not contain an association for key. The hashtable is then changed to associate key with the value returned by proc. proc is a procedure which should accept one argument, it should return a single value, and should not mutate hashtable. The behavior of hashtable-update! is equivalent to the following code:

(hashtable-set! hashtable key
  (proc (hashtable-ref hashtable key default)))

(hashtable-clear! hashtable)
(hashtable-clear! hashtable k)
Removes all associations from hashtable. If a second argument k is given, the current capacity of the hashtable is reset to approximately k elements.

(hashtable-keys hashtable)
Returns an immutable vector of all keys in hashtable.

(hashtable-values hashtable)
Returns an immutable vector of all values in hashtable.

(hashtable-entries hashtable)
Returns two values, an immutable vector of the keys in hashtable, and an immutable vector of the corresponding values.

(hashtable-key-list hashtable)
Returns a list of all keys in hashtable.
(hashtable-value-list hashtable)
Returns a list of all values in hashtable.

(hashtable->alist hashtable)
Returns a list of all associations in hashtable as an association list. Each association is represented as a pair consisting of the key and the corresponding value.

(hashtable-for-each proc hashtable)
Applies proc to every association in hashtable. proc should be a procedure accepting two values, a key and a corresponding value.

(hashtable-map! proc hashtable)
Applies proc to every association in hashtable. proc should be a procedure accepting two values, a key and a corresponding value, and returning one value. This value and the key will replace the existing binding.

17.6 Composition

(hashtable-union! hashtable1 hashtable2)
Includes all associations from hashtable2 in hashtable1 if the key of the association is not already contained in hashtable1.

(hashtable-intersection! hashtable1 hashtable2)
Removes all associations from hashtable1 for which the key of the association is not contained in hashtable2.

(hashtable-difference! hashtable1 hashtable2)
Removes all associations from hashtable1 for which the key of the association is contained in hashtable2.
18 LispKit Heap

Library (lispkit heap) provides an implementation of a priority queue in form of a binary max heap. A max heap is a tree-based data structure in which for any given node C, if P is a parent node of C, then the value of P is greater than or equal to the value of C. Heaps as implemented by (lispkit heap) are mutable objects.

(make-heap pred<?)
Returns a new empty binary max heap with pred<? being the associated ordering function.

(heap-empty? hp)
Returns #t if the heap hp is empty, otherwise #f is returned.

(heap-max hp)
Returns the largest item in heap hp, i.e. the item which is larger than all others according to the comparison function of hp. Note, heap-max does not remove the largest item as opposed to heap-delete-max!.
If there are no items on the heap, an error is signaled.

(heap-add! hp e1 ...)
Inserts an item into the heap. The same item can be inserted multiple times.

(heap-delete-max! hp)
Returns the largest item in heap hp, i.e. the item which is larger than all others according to the comparison function of hp, and removes the item from the heap. If there are no items on the heap, an error is signaled.

(heap-clear! hp)
Removes all items from hp. After this procedure has been executed, the heap is empty.

(heap-copy hp)
Returns a copy of heap hp.

(heap->vector hp)
Returns a new vector containing all items of the heap hp in descending order. This procedure does not mutate hp.

(heap->list hp)
Returns a list containing all items of the heap hp in descending order.

(heap->reversed-list hp)
Returns a list containing all items of the heap hp in ascending order.

(list->heap! hp items)
Inserts all the items from list items into heap hp.

(list->heap items pred<?)
Creates a new heap for the given ordering predicate pred<? and inserts all the items from list items into it. list->heap returns the new heap.

(vector->heap vec pred<?)
Creates and returns a new heap for the given ordering predicate pred<? and inserts all the items from vector vec into it.
19 LispKit List

Lists are heterogeneous data structures constructed out of pairs and an empty list object.

A pair consists of two fields called car and cdr (for historical reasons). Pairs are created by the procedure cons. The car and cdr fields are accessed by the procedures car and cdr. As opposed to most other Scheme implementations, lists are immutable in LispKit. Thus, it is not possible to set the car and cdr fields of an already existing pair.

Pairs are used primarily to represent lists. A list is defined recursively as either the empty list or a pair whose cdr is a list. More precisely, the set of lists is defined as the smallest set $X$ such that

- The empty list is in $X$
- If list is in $X$, then any pair whose cdr field contains list is also in $X$.

The objects in the car fields of successive pairs of a list are the elements of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The length of a list is the number of elements, which is the same as the number of pairs.

The empty list is a special object of its own type. It is not a pair, it has no elements, and its length is zero.

The most general notation (external representation) for Scheme pairs is the “dotted” notation $(c_1 . c_2)$ where $c_1$ is the value of the car field and $c_2$ is the value of the cdr field. For example $(4 . 5)$ is a pair whose car is 4 and whose cdr is 5. Note that $(4 . 5)$ is the external representation of a pair, not an expression that evaluates to a pair.

A more streamlined notation can be used for lists: the elements of the list are simply enclosed in parentheses and separated by spaces. The empty list is written $(\)$. For example,

- $(a \ b \ c \ d \ e)$

and

- $(a . (b . (c . (d . (e . ())))))$

are equivalent notations for a list of symbols.

A chain of pairs not ending in the empty list is called an improper list. Note that an improper list is not a list. The list and dotted notations can be combined to represent improper lists:

- $(a \ b \ c \ . \ d)$

is equivalent to

- $(a . (b . (c . d)))$
19.1 Basic constructors and procedures

(cons x y)
Returns a pair whose car is x and whose cdr is y.

(car xs)
Returns the contents of the car field of pair xs. Note that it is an error to take the car of the empty list.

(cdr xs)
Returns the contents of the cdr field of pair xs. Note that it is an error to take the cdr of the empty list.

(caar xs)
(cadr xs)
(cdar xs)
(cddr xs)

These procedures are compositions of car and cdr as follows:

(define (caar x) (car (car x)))
(define (cadr x) (car (cdr x)))
(define (cdar x) (cdr (car x)))
(define (cddr x) (cdr (cdr x)))

( caaar xs )
( caadr xs )
( cadar xs )
( caddr xs )
( cdaar xs )
( cdaadr xs )
( cdadar xs )
( cddar xs )
( cdddr xs )

These eight procedures are further compositions of car and cdr on the same principles. For example, caddr could be defined by (define caddr (lambda (x) (car (cdr (cdr x))))). Arbitrary compositions up to four deep are provided.

(caaaaar xs)
(caaadr xs)
(cadadr xs)
(caddar xs)
(cadaaar xs)
(cdaadr xs)
(cdadad dr xs)
(cdadaar xs)
(cddaadr xs)
(cddaddr xs)
(cdddadar xs)
(cdddddr xs)

These sixteen procedures are further compositions of car and cdr on the same principles. For example, cadddr could be defined by (define cadddr (lambda (x) (car (cdr (cdr (cdr x)))))). Arbitrary compositions up to four deep are provided.
(make-list k)
(make-list k fill)

Returns a list of k elements. If argument fill is given, then each element is set to fill. Otherwise the content of each element is the empty list.

(list x …)

Returns a list of its arguments, i.e. (x …).

(list 'a (+ 3 4) 'c) ⇒ (a 7 c)
(list) ⇒ ()

(length xs)

Returns the length of list xs.

(length '(a b c)) ⇒ 3
(length '(a (b) (c d e))) ⇒ 3
(length '()) ⇒ 0

19.2 Predicates

(pair? obj)

Returns #t if obj is a pair, #f otherwise.

(null? obj)

Returns #t if obj is an empty list, #f otherwise.

(list? obj)

Returns #t if obj is a proper list, #f otherwise. A chain of pairs ending in the empty list is called a proper list.

(every? pred xs …)

Applies the predicate pred across the lists xs …, returning #t if the predicate returns #t on every application. If there are n list arguments xs1 … xsn, then pred must be a procedure taking n arguments and returning a single value, interpreted as a boolean. If an application of pred returns #f, then every? returns #f immediately without applying pred further anymore.

(any? pred xs …)

Applies the predicate pred across the lists xs …, returning #t if the predicate returns #t for at least one application. If there are n list arguments xs1 … xsn, then pred must be a procedure taking n arguments and returning a single value, interpreted as a boolean. If an application of pred returns #t, then any? returns #t immediately without applying pred further anymore.

19.3 Composing and transforming lists

(append xs …)

Returns a list consisting of the elements of the first list xs followed by the elements of the other lists. If there are no arguments, the empty list is returned. If there is exactly one argument, it is returned. The last argument, if there is one, can be of any type. An improper list results if the last argument is not a proper list.
(append '(x) '(y)) ⇒ (x y)
(append '(a) '(b c d)) ⇒ (a b c d)
(append '(a (b)) '((c))) ⇒ (a (b) (c))
(append '(a b) '(c . d)) ⇒ (a b c . d)
(append '() 'a) ⇒ a

(concatenate xss)
This procedure appends the elements of the list of lists xss. That is, concatenate returns (apply append xss).

(reverse xs)
Procedure reverse returns a list consisting of the elements of list xs in reverse order.

(filter pred xs)
Returns all the elements of list xs that satisfy predicate pred. Elements in the result list occur in the same order as they occur in the argument list xs.

(remove pred xs)
Returns a list without the elements of list xs that satisfy predicate pred: (lambda (pred list) (filter (lambda (x) (not (pred x))) list)). Elements in the result list occur in the same order as they occur in the argument list xs.

(partition pred xs)
Partitions the elements of list xs with predicate pred returning two values: the list of in-elements (i.e. elements from xs satisfying pred) and the list of out-elements. Elements occur in the result lists in the same order as they occur in the argument list xs.

(map f xs ...)
The map procedure applies procedure proc element-wise to the elements of the lists xs ... and returns a list of the results, in order. If more than one list is given and not all lists have the same length, map terminates when the shortest list runs out. The dynamic order in which proc is applied to the elements of the lists is unspecified.

It is an error if proc does not accept as many arguments as there are lists xs ... and return a single value.
Maps $f$ over the elements of the lists $xs \ldots$, just as in function $\text{map}$. However, the results of the applications are appended together to determine the final result. $\text{append-map}$ uses $\text{append}$ to append the results together. The dynamic order in which the various applications of $f$ are made is not specified. At least one of the list arguments $xs \ldots$ must be finite.

This is equivalent to $(\text{apply} \ \text{append} \ (\text{map} \ f \ xs \ldots))$.

\[
\begin{align*}
(\text{append-map} \ f \ x1 \ x2 \ldots) \Rightarrow \\
(\text{apply} \ \text{append} \ (\text{map} \ f \ x1 \ x2 \ldots))
\end{align*}
\]

This function works like $\text{map}$, but only values differently from $\#f$ are being included in the resulting list. The dynamic order in which the various applications of $f$ are made is not specified. At least one of the list arguments $xs \ldots$ must be finite.

\[
\begin{align*}
(\text{filter-map} \ f \ x1 \ x2 \ldots) \Rightarrow \\
(\text{apply} \ \text{filter} \ (\text{map} \ f \ x1 \ x2 \ldots))
\end{align*}
\]

The arguments to $\text{for-each}$ $xs \ldots$ are like the arguments to $\text{map}$, but $\text{for-each}$ calls $\text{proc}$ for its side effects rather than for its values. Unlike $\text{map}$, $\text{for-each}$ is guaranteed to call $\text{proc}$ on the elements of the lists in order from the first element to the last. If more than one list is given and not all lists have the same length, $\text{for-each}$ terminates when the shortest list runs out.

\[
\begin{align*}
(\text{for-each} \ f \ x1 \ x2 \ldots) \Rightarrow \\
(\text{apply} \ \text{for-each} \ (\text{map} \ f \ x1 \ x2 \ldots))
\end{align*}
\]

Fundamental list recursion operator applying $f$ to the elements $x1 \ldots xn$ of list $xs$ in the following way: $(f \ldots (f (f z (x1) x2) \ldots xn))$. In other words, this function applies $f$ recursively based on the following rules, assuming one list parameter $xs$:

\[
\begin{align*}
(\text{fold-left} \ f \ z \ xs) \Rightarrow \ (\text{apply} \ \text{fold-left} \ (\text{map} \ f \ z \ xs) \ (\text{cdr} \ xs)) \\
(\text{fold-left} \ f \ z \ '(())) \Rightarrow \ z
\end{align*}
\]

If $n$ list arguments are provided, then function $f$ must take $n + 1$ parameters: one element from each list, and the “seed” or fold state, which is initially $z$ as its very first argument. The $\text{fold-left}$ operation terminates when the shortest list runs out of values.

\[
\begin{align*}
(\text{fold-left} \ (\lambda x y \ (\text{cons} y x)) \ '() \ '(1 2 3 4)) \Rightarrow \ (4 3 2 1) \\
(\text{define} \ (\lambda x y \ z \ (\text{cons} y x) \ (\text{cons} (z \ a \ b) \ (\text{cons} (a \ b \ c \ d)))) \ (\text{apply} \ \text{fold-left} \ (\lambda x y \ z \ (\text{cons} y x) \ (\text{cons} (a \ b \ c \ d) \ (\text{cons} (a \ b \ c \ d)))) \ (4 3 2 1)) \Rightarrow \ (4 3 3 2 2 11)
\end{align*}
\]

Please note, compared to function $\text{fold}$ from library $(\text{srfi} \ 1)$, this function applies the “seed”/fold state always as its first argument to $f$.

\[
\begin{align*}
(\text{fold-right} \ f \ z \ xs \ldots) \Rightarrow \\
(\text{apply} \ \text{fold-right} \ (\text{map} \ f \ z \ xs \ldots))
\end{align*}
\]

Fundamental list recursion operator applying $f$ to the elements $x1 \ldots xn$ of list $xs$ in the following way: $(f \ x1 \ (f \ x2 \ldots \ (f \ xn \ z)))$. In other words, this function applies $f$ recursively based on the following rules, assuming one list parameter $xs$:
(fold-right f z xs) ⇒ (f (car xs) (fold-right f z (cdr xs)))

(fold-right f z '()) ⇒ z

(define (xcons xs x) (cons x xs))
(fold-left xcons '()' '(1 2 3 4)) ⇒ (4 3 2 1)

If \( n \) list arguments \( xs \) ... are provided, then function \( f \) must take \( n + 1 \) parameters: one element from each list, and the “seed” or fold state, which is initially \( z \). The fold-right operation terminates when the shortest list runs out of values.

(fold-right (lambda (x l) (if (even? x) (cons x l) l)) '() '(1 2 3 4 5 6)) ⇒ (2 4 6)

As opposed to fold-left, procedure fold-right is not tail-recursive.

(sort less xs) procedure
Returns a sorted list containing all elements of \( xs \) such that for every element \( x_i \) at position \( i \), \( (\text{less } x_j \ x_i) \) returns \#t for all elements \( x_j \) at position \( j \) where \( j < i \).

(merge less xs ys) procedure
Merges two lists \( xs \) and \( ys \) which are both sorted with respect to the total ordering predicate less and returns the result as a list.

(tabulate count proc) procedure
Returns a list with \( count \) elements. Element \( i \) of the list, where \( 0 \leq i < count \), is produced by \( (\text{proc } i) \).

(tabulate 4 fx1+) ⇒ (1 2 3 4)

(iota count) procedure
(iota count start)
(iota count start step)
Returns a list containing the elements \( (\text{start} \ \text{start}+\text{step} \ldots \ \text{start}+(\text{count}-1)\times\text{step}) \). The \( \text{start} \) and \( \text{step} \) parameters default to 0 and 1.

(iota 5) ⇒ (0 1 2 3 4)
(iota 5 0 -0.1) ⇒ (0 -0.1 -0.2 -0.3 -0.4)

19.4 Finding and extracting elements

(list-tail xs k) procedure
Returns the sublist of list \( xs \) obtained by omitting the first \( k \) elements. Procedure list-tail could be defined by

(define (list-tail xs k)
  (if (zero? k) xs (list-tail (cdr xs) (- k 1))))

(list-ref xs k) procedure
Returns the \( k \)-th element of list \( xs \). This is the same as the car of \( (\text{list-tail } xs \ k) \).

(memq obj xs) procedure
(memv obj xs)
(member obj xs)  
(member obj xs compare)

These procedures return the first sublist of xs whose car is obj, where the sublists of xs are the non-empty lists returned by (list-tail xs k) for k less than the length of xs. If obj does not occur in xs, then #f is returned. The memq procedure uses eq? to compare obj with the elements of xs, while memv uses eqv? and member uses compare, if given, and equal? otherwise.

(delq obj xs)  
(delv obj xs)  
(delete obj xs)  
(delete obj xs compare)

Returns a copy of list xs with all entries equal to element obj removed. delq uses eq? to compare obj with the elements in list xs, delv uses eqv?, and delete uses compare if given, and equal? otherwise.

(assq obj alist)  
(assv obj alist)  
(assoc obj alist)  
(assoc obj alist compare)

alist must be an association list, i.e. a list of key/value pairs. This family of procedures 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 is returned. The assq procedure uses eq? to compare obj with the car fields of the pairs in alist, while assv uses eqv? and assoc uses compare if given, and equal? otherwise.

(define e '((a 1) (b 2) (c 3)))

(assq 'a e) ⇒ (a 1)  
(assq 'b e) ⇒ (b 2)  
(assq 'd e) ⇒ #f  
(assq (list 'a) '(((a)) (((b)) ((c))))) ⇒ #f  
(assoc (list 'a) '(((a)) (((b)) ((c))))) ⇒ ((a))  
(assq 5 '(((2 3) (5 7) (11 13)))) ⇒ (#f)  
(assv 5 '(((2 3) (5 7) (11 13)))) ⇒ (3 5)

(alist-delq obj alist)  
(alist-delv obj alist)  
(alist-delete obj alist)  
(alist-delete obj alist compare)

Returns a copy of association list alist with all entries removed whose car is equal to element obj. alist-delq uses eq? to compare obj with the first elements of all members of list xs, alist-delv uses eqv?, and alist-delete uses compare if given, and equal? otherwise.

(key xs)  
(key xs default)

Returns (car xs) if xs is a pair, otherwise default is being returned. If default is not provided as an argument, #f is used instead.

(value xs)  
(value xs default)

Returns (cdr xs) if xs is a pair, otherwise default is being returned. If default is not provided as an argument, #f is used instead.
20 LispKit Markdown

Library (lispkit markdown) provides an API for programmatically constructing Markdown documents, for parsing strings in Markdown format, as well as for mapping Markdown documents into corresponding HTML. The Markdown syntax supported by this library is based on the CommonMark Markdown specification.

20.1 Data Model

Markdown documents are represented using an abstract syntax that is implemented by three algebraic datatypes block, list-item, and inline, via define-datatype of library (lispkit datatype).

20.1.1 Blocks

At the top-level, a Markdown document consist of a list of blocks. The following recursively defined datatype shows all the supported block types as variants of type block.

```
(define-datatype block markdown-block?
  (document blocks)
  (blockquote blocks)
  (list-items start tight items)
  (paragraph text)
  (heading level text)
  (indented-code lines)
  (fenced-code lang lines)
  (html-block lines)
  (reference-def label dest title)
  (thematic-break))
```

(document blocks) represents a full Markdown document consisting of a list of blocks. (blockquote blocks) represents a blockquote block which itself has a list of sub-blocks. (list-items start tight items) defines either a bullet list or an ordered list. start is #f for bullet lists and defines the first item number for ordered lists. tight is a boolean which is #f if this is a loose list (with vertical spacing between the list items). items is a list of list items of type list-item as defined as follows:

```
(define-datatype list-item markdown-list-item?
  (bullet ch tight? blocks)
```

85
The most frequent Markdown block type is a paragraph. (paragraph text) represents a single paragraph of text where text refers to a list of inline text fragments of type inline (see below). (heading level text) defines a heading block for a heading of a given level, where level is a number starting with 1 (up to 6). (indented-code lines) represents a code block consisting of a list of text lines each represented by a string. (fenced-code lang lines) is similar: it defines a code block with code expressed in the given language lang. (html lines) defines a HTML block consisting of the given lines of text. (reference-def label dest title) introduces a reference definition consisting of a given label, a destination URI dest, as well as a title string. Finally, (thematic-break) introduces a thematic break block separating the previous and following blocks visually, often via a line.

### 20.1.2 Inline Text

Text is represented as lists of inline text segments, each represented as an object of type inline. inline is defined as follows:

```lisp
(define-datatype inline markdown-inline?
  (text str)
  (code str)
  (emph text)
  (strong text)
  (link text uri title)
  (auto-link uri)
  (email-auto-link email)
  (image text uri title)
  (html tag)
  (line-break hard?))
```

(text str) refers to a text segment consisting of string str. (code str) refers to a code string str (often displayed as verbatim text). (emph text) represents emphasized text (often displayed as italics). (strong text) represents text in boldface. (link text uri title) represents a hyperlink with text linking to uri and title representing a title for the link. (auto-link uri) is a link where uri is both the text and the destination URI. (email-auto-link email) is a “mailto:” link to the given email address email. (image text uri title) inserts an image at uri with image description text and image link title title. (html tag) represents a single HTML tag of the form <tag >. Finally, (line-break #f) introduces a “soft line break”, whereas (line-break #t) inserts a “hard line break”.

### 20.2 Creating Markdown documents

Markdown documents can either be constructed programmatically via the datatypes introduced above, or a string representing a Markdown documents gets parsed into the internal abstract syntax representation via function markdown.
For instance, (markdown "# My title

This is a paragraph.") returns a markdown document consisting of two blocks: a header block for header “My title” and a paragraph block for the text “This is a paragraph”:

```
(markdown "# My title

This is a paragraph.")
⇒ #block: (document (#block: (heading 1 (#inline: (text "My title"))) #block: (paragraph (#inline: (text "This is a paragraph.")))))
```

The same document can be created programmatically in the following way:

```
(document
  (list
    (heading 1 (list (text "My title")))
    (paragraph (list (text "This is a paragraph.")))))
⇒ #block: (document (#block: (heading 1 (#inline: (text "My title"))) #block: (paragraph (#inline: (text "This is a paragraph.")))))
```

### 20.3 Processing Markdown documents

Since the abstract syntax of Markdown documents is represented via algebraic datatypes, pattern matching can be used to deconstruct the data. For instance, the following function returns all the top-level headers of a given Markdown document:

```
(import (lispkit datatype)) ; this is needed to import `match`

(define (top-headings doc)
  (match doc
    ((document blocks)
     (filter-map (lambda (block)
                   (match block
                     ((heading 1 text) (text->raw-string text))
                     (else #f))
                   blocks)))))
```

An example for how `top-headings` can be applied to this Markdown document:

```
# *header* 1
Paragraph.
# __header__ 2
## header 3
The end.
```

is shown here:

```
(top-headings (markdown "# *header* 1
Paragraph.
# __header__ 2
## header 3
The end."))
⇒ (#"header 1" "header 2")
```

### 20.4 API

(markdown-blocks? obj)

Returns \#t\ if \obj\ is a proper list of objects \o\ for which \(markdown-block?\ \o\) returns \#t\; otherwise it returns \#f\.
(markdown-block? obj)
Returns #t if obj is a variant of algebraic datatype block.

(markdown-block=? lhs rhs)
Returns #t if markdown blocks lhs and rhs are equals; otherwise it returns #f.

(markdown-list? obj)
Returns #t if obj is a proper list of list items i for which (markdown-list-item? i) returns #t; otherwise it returns #f.

(markdown-list-item? obj)
Returns #t if obj is a variant of algebraic datatype list-item.

(markdown-list-item=? lhs rhs)
Returns #t if markdown list items lhs and rhs are equals; otherwise it returns #f.

(markdown-text? obj)
Returns #t if obj is a proper list of objects o for which (markdown-inline? o) returns #t; otherwise it returns #f.

(markdown-text? obj)
Returns #t if obj is a variant of algebraic datatype inline.

(markdown-inline? obj)
Returns #t if markdown inline text lhs and rhs are equals; otherwise it returns #f.

(markdown? obj)
Returns #t if obj is a valid markdown document, i.e. an instance of the document variant of datatype block; returns #f otherwise.

(markdown=? lhs rhs)
Returns #t if markdown documents lhs and rhs are equals; otherwise it returns #f.

(markdown str)
 Parses the text in Markdown format in str and returns a representation of the abstract syntax using the algebraic datatypes block, list-item, and inline.

(markdown->html md)
Converts a Markdown document md into HTML, represented in form of a string. md needs to satisfy the markdown? predicate.

(blocks->html bs)
(blocks->html bs tight)
Converts a Markdown block or list of blocks bs into HTML, represented in form of a string. tight? is a boolean and should be set to true if the conversion should consider tight typesetting (see CommonMark specification for details).

(text->html txt)
Converts Markdown inline text or list of inline texts txt into HTML, represented in form of a string.

(markdown->html-doc md)
(markdown->html-doc md style)
(markdown->html-doc md style codestyle)
(markdown->html-doc md style codestyle cblockstyle)
(markdown->html-doc md style codestyle cblockstyle colors)
Converts a Markdown document md into a styled HTML document, represented in form of a string. md needs to satisfy the markdown? predicate. style is a list with up to three elements: (size font color). It specifies the default text style of the document. size is the point size of the font, font is a font name, and color is a HTML color specification (e.g. "#FF6789"). codestyle specifies the style of inline code in the
same format. *colors* is a list of HTML color specifications for the following document elements in this order: the border color of code blocks, the color of blockquote “bars”, the color of H1, H2, H3 and H4 headers.

\[(text->string \text{ text})\]  
\text{procedure}  
\text{Converts given inline text} \text{ text} \text{ into a string representation which encodes markup in} \text{ text} \text{ using Markdown syntax.} \text{ text} \text{ needs to satisfy the} markdown-text? \text{ predicate.}

\[(text->raw-string \text{ text})\]  
\text{procedure}  
\text{Converts given inline text} \text{ text} \text{ into a string representation ignoring markup in} \text{ text}. \text{ text} \text{ needs to satisfy the} markdown-text? \text{ predicate.}
21 LispKit Math

Library (lispkit math) defines functions on numbers. Numbers are arranged into a tower of subtypes in which each level is a subset of the level above it:

- number
- complex number
- real number
- rational number
- integer

For example, 3 is an integer. Therefore 3 is also a rational, a real, and a complex number. These types are defined by the predicates number?, complex?, real?, rational?, and integer?.

There is no simple relationship between a number's type and its representation inside a computer. Scheme's numerical operations treat numbers as abstract data, as independent of their representation as possible.

21.1 Numerical constants

pi
The constant pi.

e
Euler's number, i.e. the base of the natural logarithm.

21.2 Predicates

(number? obj)  procedure
(complex? obj)
(real? obj)
(rational? obj)
(integer? obj)

These numerical type predicates can be applied to any kind of argument, including non-numbers. They return #t if the object is of the named type, and otherwise they return #f. In general, if a type predicate is true of a number then all higher type predicates are also true of that number. Consequently, if a type predicate is false of a number, then all lower type predicates are also false of that number.

If \( z \) is a complex number, then (real? \( z \)) is true if and only if (zero? (imag-part \( z \))) is true. If \( x \) is an inexact real number, then (integer? \( x \)) is true if and only if (= \( x \) (round \( x \))).

The numbers +inf.0, -inf.0, and +nan.0 are real but not rational.
(complex? 3+4i) ⇒ #t
(complex? 3) ⇒ #t
(real? 3) ⇒ #t
(real? -2.5+0i) ⇒ #t
(real? -2.5+0.0i) ⇒ #f
(real? #e1e10) ⇒ #t
(real? +inf.0) ⇒ #t
(real? +nan.0) ⇒ #t
(rational? -inf.0) ⇒ #f
(rational? 3.5) ⇒ #t
(rational? 6/10) ⇒ #t
(rational? 6/3) ⇒ #t
(integer? 3+0i) ⇒ #t
(integer? 3.0) ⇒ #t
(integer? 8/4) ⇒ #t

(fixnum? obj)
procedure
Returns #t if object obj is a fixnum; otherwise returns #f. A fixnum is an exact integer that is small enough to fit in a machine word. LispKit fixnums are 64-bit words. Fixnums are signed and encoded using 2's complement.

(ratnum? obj)
procedure
Returns #t if object obj is a fractional number, i.e. a rational number which isn't an integer.

(bignum? obj)
procedure
Returns #t if object obj is a large integer number, i.e. an integer which isn't a fixnum.

(flonum? obj)
procedure
Returns #t if object obj is a floating-point number.

(cflonum? obj)
procedure
Returns #t if object obj is a complex floating-point number.

(exact? obj)
(inexact? obj)
These numerical predicates provide tests for the exactness of a quantity. For any Scheme number, precisely one of exact? and inexact? is true.

(exact? 3.0) ⇒ #f
(exact? #e3.0) ⇒ #t
(inexact? 3.) ⇒ #t

(exact-integer? obj)
procedure
Returns #t if obj is both exact and an integer; otherwise returns #f.

(exact-integer? 32) ⇒ #t
(exact-integer? 32.0) ⇒ #f
(exact-integer? 32/5) ⇒ #f

(finite? obj)
procedure
The finite? procedure returns #t on all real numbers except +inf.0, -inf.0, and +nan.0, and on complex numbers if their real and imaginary parts are both finite. Otherwise it returns #f.

(finite? 3) ⇒ #t
(finite? +inf.0) ⇒ #f
(finite? 3.0+inf.0i) ⇒ #f
The `infinite?` procedure returns `#t` on the real numbers `+inf.0` and `+inf.0`, and on complex numbers if their real or imaginary parts or both are infinite. Otherwise it returns `#f`.

- `(infinite? 3)` ⇒ `#f`
- `(infinite? +inf.0)` ⇒ `#t`
- `(infinite? +nan.0)` ⇒ `#f`
- `(infinite? 3.0+inf.0i)` ⇒ `#t`

The `nan?` procedure returns `#t` on `+nan.0`, and on complex numbers if their real or imaginary parts or both are `+nan.0`. Otherwise it returns `#f`.

- `(nan? +nan.0)` ⇒ `#t`
- `(nan? 32)` ⇒ `#f`
- `(nan? +nan.0+5.0i)` ⇒ `#t`
- `(nan? 2+2i)` ⇒ `#f`

Returns `#t` if number `x` is positive, i.e. `x > 0`.

Returns `#t` if number `x` is negative, i.e. `x < 0`.

Returns `#t` if number `z` is zero, i.e. `z = 0`.

Returns `#t` if the integer number `n` is even.

Returns `#t` if the integer number `n` is odd.

### 21.3 Exactness and rounding

Scheme distinguishes between numbers that are represented exactly and those that might not be. This distinction is orthogonal to the dimension of type. A number is exact if it was written as an exact constant or was derived from exact numbers using only exact operations. A number is inexact if it was written as an inexact constant, if it was derived using inexact ingredients, or if it was derived using inexact operations.

Rational operations such as `+` should always produce exact results when given exact arguments. If the operation is unable to produce an exact result, then it either reports the violation of an implementation restriction or it silently coerces its result to an inexact value.

The procedure `inexact` returns an inexact representation of `z`. The value returned is the inexact number that is numerically closest to the argument. For inexact arguments, the result is the same as the argument. For exact complex numbers, the result is a complex number whose real and imaginary parts are the result of applying inexact to the real and imaginary parts of the argument, respectively. If an exact argument has no reasonably close inexact equivalent (in the sense of `=equivalent`), then a violation of an implementation restriction may be reported.

The procedure `exact` returns an exact representation of `z`. The value returned is the exact number that is numerically closest to the argument. For exact arguments, the result is the same as the argument.
For inexact non-integral real arguments, the function may return a rational approximation. For inexact complex arguments, the result is a complex number whose real and imaginary parts are the result of applying exact to the real and imaginary parts of the argument, respectively. If an inexact argument has no reasonably close exact equivalent, (in the sense of \( = \)), then a violation of an implementation restriction may be reported.

These procedures implement the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range.

(approximate \( x \) \( \delta \))

Procedure \texttt{approximate} approximates floating-point number \( x \) returning a rational number which differs at most \( \delta \) from \( x \).

(rationalize \( x \) \( y \))

The \texttt{rationalize} procedure returns the simplest rational number differing from \( x \) by no more than \( y \). A rational number \( r_1 \) is simpler than another rational number \( r_2 \) if \( r_1 = p_1/q_1 \) and \( r_2 = p_2/q_2 \) (in lowest terms) and \( |p_1| \leq |p_2| \) and \( |q_1| \leq |q_2| \). Thus \( 3/5 \) is simpler than \( 4/7 \). Although not all rationals are comparable in this ordering (consider \( 2/7 \) and \( 3/5 \)), any interval contains a rational number that is simpler than every other rational number in that interval (the simpler \( 2/5 \) lies between \( 2/7 \) and \( 3/5 \)). Note that \( 0 = 0/1 \) is the simplest rational of all.

\[
\begin{align*}
\text{(rationalize (exact .3) 1/10)} & \Rightarrow 1/3
\end{align*}
\]

(floor \( x \))
(ceiling \( x \))
(truncate \( x \))
(round \( x \))

These procedures return integers. \texttt{floor} returns the largest integer not larger than \( x \). \texttt{ceiling} returns the smallest integer not smaller than \( x \). \texttt{truncate} returns the integer closest to \( x \) whose absolute value is not larger than the absolute value of \( x \). \texttt{round} returns the closest integer to \( x \), rounding to even when \( x \) is halfway between two integers.

If the argument to one of these procedures is inexact, then the result will also be inexact. If an exact value is needed, the result can be passed to the \texttt{exact} procedure. If the argument is \texttt{infinite} or a \texttt{NaN}, then it is returned.

\[
\begin{align*}
\text{(floor -4.3)} & \Rightarrow -5.0 \\
\text{(ceiling -4.3)} & \Rightarrow -4.0 \\
\text{(truncate -4.3)} & \Rightarrow -4.0 \\
\text{(round -4.3)} & \Rightarrow -4.0 \\
\text{(floor 3.5)} & \Rightarrow 3.0 \\
\text{(ceiling 3.5)} & \Rightarrow 4.0 \\
\text{(truncate 3.5)} & \Rightarrow 3.0 \\
\text{(round 3.5)} & \Rightarrow 4.0 \ ; \text{inexact} \\
\text{(round 3/2)} & \Rightarrow 4 \ ; \text{exact} \\
\text{(round ?)} & \Rightarrow 7
\end{align*}
\]

21.4 Operations

(+ \( z \) …)
(* \( z \) …)

These procedures return the sum or product of their arguments.
( + 34) \Rightarrow 7
(* 3) \Rightarrow \text{3}
(*) \Rightarrow 0
(* 4) \Rightarrow 4
(* ) \Rightarrow 1

(- z)

(- z1 z2 ...)  

**/( z)**  

(/ z1 z2 ...)

With two or more arguments, these procedures return the difference or quotient of their arguments, associating to the left. With one argument, however, they return the additive or multiplicative inverse of their argument.

It is an error if any argument of / other than the first is an exact zero. If the first argument is an exact zero, the implementation may return an exact zero unless one of the other arguments is a NaN.

(- 3 4) \Rightarrow -1
(- 3 4 5) \Rightarrow -6
(- 3) \Rightarrow -3
(/ 3 4 5) \Rightarrow 3/20
(/ 3) \Rightarrow 1/3

(= x ...)
(< x ...)
(> x ...)
(<= x ...)
(>= x ...)

These procedures return #t if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically non-decreasing, or monotonically non-increasing, and #f otherwise. If any of the arguments are +nan.0, all the predicates return #f. They do not distinguish between inexact zero and inexact negative zero.

(max x1 x2 ...)  

(min x1 x2 ...)  

These procedures return the maximum or minimum of their arguments.

If any argument is inexact, then the result will also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If min or max is used to compare numbers of mixed exactness, and the numerical value of the result cannot be represented as an inexact number without loss of accuracy, then the procedure reports an implementation restriction.

(abs x)

The abs procedure returns the absolute value of its argument x.

(square z)

Returns the square of z. This is equivalent to (* z z).

(square 42) \Rightarrow 1764
(square 2.0) \Rightarrow 4.0

(sqrt z)

Returns the principal square root of z. The result will have either a positive real part, or a zero real part and a non-negative imaginary part.
(sqrt 9) ⇒ 3
(sqrt -1) ⇒ +i

(exact-integer-sqrt k)
Returns two non-negative exact integers s and r where \( k = s^2 + r \) and \( k < (s+1)^2 \).

(expt z1 z2)
Returns \( z1 \) raised to the power \( z2 \). For non-zero \( z1 \), this is \( z1^{z2} = e^{(z2 \log z1)} \). The value of \( 0^{z} \) is 1 if \( \text{zero? } z \), 0 if \( \text{real-part } z \) is positive, and an error otherwise. Similarly for \( 0.0^z \), with inexact results.

(exp z)
(log z)
(log z1 z2)
(sin z)
(cos z)
(tan z)
(asin z)
(acos z)
(atan z)
(atan y x)

These procedures compute the usual transcendental functions. The \( \log \) procedure computes the natural logarithm of \( z \) (not the base-ten logarithm) if a single argument is given, or the base-\( z2 \) logarithm of \( z1 \) if two arguments are given. The \( \text{asin} \), \( \text{acos} \), and \( \text{atan} \) procedures compute arc-sine, arc-cosine, and arc-tangent, respectively. The two-argument variant of \( \text{atan} \) computes \( \text{angle} \ (\text{make-rectangular } x \ y) \).

21.5 Division and remainder

(gcd n …)
(lcm n …)

These procedures return the greatest common divisor (\( \text{gcd} \)) or least common multiple (\( \text{lcm} \)) of their arguments. The result is always non-negative.

(truncate/ n1 n2.)
(truncate-quotient n1 n2)
(truncate-remainder n1 n2)

These procedures implement number-theoretic integer division. It is an error if \( n2 \) is zero. \( \text{truncate/} \) returns two integers; the other two procedures return an integer. All the procedures compute a quotient \( nq \) and remainder \( nr \) such that \( n1 = n2 \times nq + nr \). The three procedures are defined as follows:
The remainder \( n_r \) is determined by the choice of integer \( n_q \): \( n_r = n_1 - n_2 \times n_q \) where \( n_q = \text{truncate}(n_1/n_2) \).

For any of the operators, and for integers \( n_1 \) and \( n_2 \) with \( n_2 \) not equal to 0:

\[
\begin{align*}
\text{quotient} & \equiv n_1 \\
(\times (\times n_2 (\text{truncate-quotient} n_1 n_2)) \times n_2) & \Rightarrow \#t
\end{align*}
\]

provided all numbers involved in that computation are exact.

\[
\begin{align*}
(\text{truncate}/ 5 2) & \Rightarrow 2 1 \\
(\text{truncate}/ -5 2) & \Rightarrow -2 -1 \\
(\text{truncate}/ 5 -2) & \Rightarrow -2 1 \\
(\text{truncate}/ -5 -2) & \Rightarrow 2 -1 \\
(\text{truncate}/ -5.0 -2) & \Rightarrow 2.0 -1.0
\end{align*}
\]

\[(\text{floor}/ 5 2) \quad \text{procedure}\]
\[(\text{floor-quotient} n_1 n_2) \quad \text{procedure}\]
\[(\text{floor-remainder} n_1 n_2) \quad \text{procedure}\]

These procedures implement number-theoretic integer division. It is an error if \( n_2 \) is zero. \text{floor}/ returns two integers; the other two procedures return an integer. All the procedures compute a quotient \( n_q \) and remainder \( n_r \) such that \( n_1 = n_2 \times n_q + n_r \). The three procedures are defined as follows:

\[
\begin{align*}
(\text{floor}/ n_1 n_2) & \Rightarrow n_q n_r \\
(\text{floor-quotient} n_1 n_2) & \Rightarrow n_q \\
(\text{floor-remainder} n_1 n_2) & \Rightarrow n_r
\end{align*}
\]

The remainder \( n_r \) is determined by the choice of integer \( n_q \): \( n_r = n_1 - n_2 \times n_q \) where \( n_q = \text{floor}(n_1/n_2) \).

For any of the operators, and for integers \( n_1 \) and \( n_2 \) with \( n_2 \) not equal to 0:

\[
\begin{align*}
\text{quotient} & \equiv n_1 \\
(\times (\times n_2 (\text{floor-quotient} n_1 n_2)) \times n_2) & \Rightarrow \#t
\end{align*}
\]

provided all numbers involved in that computation are exact.

\[
\begin{align*}
(\text{floor}/ 5 2) & \Rightarrow 2 1 \\
(\text{floor}/ -5 2) & \Rightarrow -3 1 \\
(\text{floor}/ 5 -2) & \Rightarrow -3 -1 \\
(\text{floor}/ -5 -2) & \Rightarrow 2 -1
\end{align*}
\]

\[(\text{quotient} n_1 n_2) \quad \text{procedure}\]
\[(\text{remainder} n_1 n_2) \quad \text{procedure}\]
\[(\text{modulo} n_1 n_2) \quad \text{procedure}\]

The quotient and remainder procedures are equivalent to \text{truncate-quotient} and \text{truncate-remainder}, respectively, and \text{modulo} is equivalent to \text{floor-remainder}. These procedures are provided for backward compatibility with earlier versions of the Scheme language specification.
21.6 Fractional numbers

\[(\text{numerator } q)\]
\[(\text{denominator } q)\]

These procedures return the numerator or denominator of their rational number \( q \). The result is computed as if the argument was represented as a fraction in lowest terms. The denominator is always positive. The denominator of 0 is defined to be 1.

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(numerator (/ 6 4))</td>
<td>3</td>
</tr>
<tr>
<td>(denominator (/ 6 4))</td>
<td>2</td>
</tr>
<tr>
<td>(denominator (inexact (/ 6 4)))</td>
<td>2.0</td>
</tr>
</tbody>
</table>

21.7 Complex numbers

\[(\text{make-rectangular } x1 \, x2)\]

Returns the complex number \( x1 + x2 \times i \). Since in LispKit, all complex numbers are inexact, \text{make-rectangular} returns an inexact complex number for all \( x1 \) and \( x2 \).

\[(\text{make-polar } x1 \, x2)\]

Returns a complex number \( z \) such that \( z = x1 \times e^{(x2 \times i)} \), i.e. \( x1 \) is the magnitude of the complex number. The \text{make-polar} procedure may return an inexact complex number even if its arguments are exact.

\[(\text{real-part } z)\]

Returns the real part of the given complex number \( z \).

\[(\text{imag-part } z)\]

Returns the imaginary part of the given complex number \( z \).

\[(\text{magnitude } z)\]

Returns the magnitude of the given complex number \( z \). Assuming \( z = x1 \times e^{(x2 \times i)} \), \text{magnitude} returns \( x1 \). The \text{magnitude} procedure is the same as \text{abs} for a real argument.

\[(\text{angle } z)\]

Returns the angle of the given complex number \( z \). The angle is a floating-point number between \(-\pi\) and \(\pi\).

21.8 String representation

\[(\text{number->string } z)\]
\[(\text{number->string } z \, \text{radix})\]

It is an error if \( \text{radix} \) is not one of 2, 8, 10, or 16. The procedure \text{number->string} takes a number \( z \) and a \text{radix} and returns as a string an external representation of the given number in the given radix such that

\[
\text{let } ((\text{number number})
(\text{radix radix}))
(\text{eqv? number (string->number (number->string number radix) radix))})
\]

is true. It is an error if no possible result makes this expression true. If omitted, \text{radix} defaults to 10.
If \( z \) is inexact, the radix is 10, and the above expression can be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent and trailing zeroes) needed to make the above expression true. Otherwise, the format of the result is unspecified. The result returned by \texttt{number->string} never contains an explicit radix prefix.

The error case can occur only when \( z \) is not a complex number or is a complex number with a non-rational real or imaginary part. If \( z \) is an inexact number and the radix is 10, then the above expression is normally satisfied by a result containing a decimal point. The unspecified case allows for infinities, NaNs, and unusual representations.

\begin{verbatim}
(string->number str)
(string->number str radix)
\end{verbatim}

Returns a number of the maximally precise representation expressed by the given string \( str \). It is an error if \( radix \) is not 2, 8, 10, or 16. If supplied, \( radix \) is a default radix that will be overridden if an explicit radix prefix is present in string (e.g. "#o177"). If \( radix \) is not supplied, then the default radix is 10. If string \( str \) is not a syntactically valid notation for a number, or would result in a number that cannot be represented, then \texttt{string->number} returns \texttt{#f}. An error is never signaled due to the content of string.

\begin{verbatim}
(string->number "100")  ⇒ 100
(string->number "100" 16)  ⇒ 256
(string->number "1e2")  ⇒ 100.0
\end{verbatim}

21.9 Bitwise operations

The following bitwise functions operate on integers including fixnums and bignums.

\begin{verbatim}
(bitwise-not n)
(bitwise-and n …)
(bitwise-ior n …)
(bitwise-xor n …)
(bit-count n)
\end{verbatim}

Returns the bitwise complement of \( n \); i.e. all 1 bits are changed to 0 bits and all 0 bits to 1 bits.

Returns the bitwise and of the given integer arguments \( n … \).

Returns the bitwise inclusive or of the given integer arguments \( n … \).

Returns the bitwise exclusive or (xor) of the given integer arguments \( n … \).

Returns the population count of 1’s if \( n >= 0 \), or 0’s, if \( n < 0 \). The result is always non-negative. The R6RS analogue \texttt{bitwise-bit-count} procedure is incompatible as it applies \texttt{bitwise-not} to the population count before returning it if \( n \) is negative.

\begin{verbatim}
(bit-count 0)  ⇒ 0
(bit-count -1)  ⇒ 0
(bit-count 7)  ⇒ 3
(bit-count 13)  ⇒ 3
(bit-count -13)  ⇒ 2
(bit-count 30)  ⇒ 4
(bit-count -30)  ⇒ 4
(bit-count (expt 2 100))  ⇒ 1
(bit-count (- (expt 2 100)))  ⇒ 100
(bit-count (- (+ 1 (expt 2 100))))  ⇒ 1
\end{verbatim}
(integer-length n)
Returns the number of bits needed to represent n, i.e.

\[
\text{procedure}
\]

\[
\text{(ceiling} / \text{(log} \text{if} (\text{negative?} \text{integer})
\text{(-} \text{integer})
\text{(+} \text{1} \text{integer}))\text{)
(\text{log 2))})
\]

The result is always non-negative. For non-negative n, this is the number of bits needed to represent n in an unsigned binary representation. For all n, (+ 1 (integer-length i)) is the number of bits needed to represent n in a signed two's-complement representation.

(first-bit-set n)
Returns the index of the least significant 1 bit in the two's complement representation of n. If n is 0, then −1 is returned.

\[
\text{procedure}
\]

\[
\text{(first-bit-set 0)} \Rightarrow -1
\]
\[
\text{(first-bit-set 1)} \Rightarrow 0
\]
\[
\text{(first-bit-set -4)} \Rightarrow 2
\]

(bit-set? n k)
k must be non-negative. The bit-set? procedure returns #t if the k-th bit is 1 in the two's complement representation of n, and #f otherwise. This is the result of the following computation:

\[
\text{procedure}
\]

\[
\text{(not} \text{(zero?} \text{(bitwise-and} \text{(bitwise-arithmetic-shift-left 1 k) n))))
\]

(copy-bit n k b)
k must be non-negative, and b must be either 0 or 1. The copy-bit procedure returns the result of replacing the k-th bit of n by the k-th bit of b, which is the result of the following computation:

\[
\text{procedure}
\]

\[
\text{(bitwise-if} \text{(bitwise-arithmetic-shift-left 1 k)}
\text{(bitwise-arithmetic-shift-left b k) n)}
\]

(arithmetic-shift n count)
If count > 0, shifts integer n left by count bits; otherwise, shifts fixnum n right by count bits. In general, this procedure returns the result of the following computation: (floor (* n (expt 2 count))).

\[
\text{procedure}
\]

\[
\text{(arithmetic-shift -6 -1)} \Rightarrow -3
\]
\[
\text{(arithmetic-shift -5 -1)} \Rightarrow -3
\]
\[
\text{(arithmetic-shift -4 -1)} \Rightarrow -2
\]
\[
\text{(arithmetic-shift -3 -1)} \Rightarrow -2
\]
\[
\text{(arithmetic-shift -2 -1)} \Rightarrow -1
\]
\[
\text{(arithmetic-shift -1 -1)} \Rightarrow -1
\]

(arithmetic-shift-left n count)
Returns the result of arithmetically shifting n to the left by count bits. count must be non-negative. The arithmetic-shift-left procedure behaves the same as arithmetic-shift.

\[
\text{procedure}
\]

(arithmetic-shift-right n count)
Returns the result of arithmetically shifting n to the right by count bits. count must be non-negative. (arithmetic-shift-right n m) behaves the same as (arithmetic-shift n (fx- m)).

\[
\text{procedure}
\]
21.10 Fixnum operations

LispKit supports arbitrarily large exact integers. Internally, it has two different representations, one for smaller integers and one for the rest. These are colloquially known as **fixnums** and **bignums** respectively. In LispKit, a *fixnum* is represented as a 64 bit signed integer which is encoded using two-complement.

Fixnum operations perform integer arithmetic on their fixnum arguments. If any argument is not a fixnum, or if the mathematical result is not representable as a fixnum, it is an error. In particular, this means that fixnum operations may return a mathematically incorrect fixnum in these situations without raising an error.

```
(integer->fixnum n)
```

*integer-*→*fixnum* coerces a given integer *n* into a fixnum. If *n* is a fixnum already, *n* is returned by *integer-*→*fixnum*. If *n* is a bignum, then the first word of the bignum is returned as the result of *integer-*→*fixnum*.

```
(fx+ n m)
(fx- n m)
(fx* n m)
(fx/ n m)
```

These procedures return the sum, the difference, the product and the quotient of their two fixnum arguments *n* and *m*. These procedures may overflow without reporting an error.

```
(fx= n m)
(fx< n m)
(fx> n m)
(fx<= n m)
(fx>= n m)
```

These procedures implement the comparison predicates. They return #t if *n = m*, *n < m*, *n > m*, *n <= m*, or *n >= m* respectively.

```
(fx1+ n)
```

Increments the fixnum *n* by one and returns the value. This procedure may overflow without raising an error.

```
(fx1- n)
```

Decrement the fixnum *n* by one and returns the value. This procedure may overflow without raising an error.

```
(fxzero? n)
```

Returns #t if fixnum *n* equals to 0.

```
(fxppositive? n)
```

Returns #t if fixnum *n* is positive, i.e. *n > 0*.

```
(fxnegative? n)
```

Returns #t if fixnum *n* is negative, i.e. *n < 0*.

```
(fxabs n)
```

Returns the absolute value of its fixnum argument *n*.

```
(fxremainder n m)
```

This procedure returns a value *r* such that the following equation holds: *n = m * q + r* where *q* is the largest number of multiples of *m* that will fit inside *n*. The sign of *m* gets ignored. This means that *(fxremainder n m)* and *(fxremainder n (- m)) always return the same answer.
(fxremainder 13 5) ⇒ 3
(fxremainder 13 -5) ⇒ 3
(fxremainder -13 5) ⇒ -3
(fxremainder -13 -5) ⇒ -3

(fxmodulo n m)
This procedure computes a remainder similar to (fxremainder n m), but when (fxremainder n m) has a different sign than m, (fxmodulo n m) returns (+ (fxremainder n m) m) instead.

(fxmodulo 13 5) ⇒ 3
(fxmodulo 13 -5) ⇒ -2
(fxmodulo -13 5) ⇒ 2
(fxmodulo -13 -5) ⇒ -3

(fxsqrt n)
Approximates the square root s of fixnum n such that s is the biggest fixnum for which s × s ≤ n.

(fxnot n)
Returns the bitwise-logical inverse for fixnum n.

(fxand n m)
Returns the bitwise-logical and for n and m.

(fxior n m)
Returns the bitwise-logical inclusive or for n and m.

(fxxor n m)
Returns the bitwise-logical exclusive or (xor) for n and m.

(fxf mask n m)
Merges the bit sequences n and m, with bit sequence mask determining from which sequence to take each bit. That is, if the k-th bit of mask is 1, then the k-th bit of the result is the k-th bit of n, otherwise it’s the k-th bit of m.

(fxf 3 1 8) ⇒ 9
(fxf 3 8 1) ⇒ 0
(fxf 1 1 2) ⇒ 3
(fxf #b00111000 #b11100000 #b00001111) ⇒ #b00110011 = 51

fxif can be implemented via (fxior (fxand mask n) (fxand (fxnot mask) m)).

(fxarithmetic-shift n count)
If count > 0, shifts fixnum n left by count bits; otherwise, shifts fixnum n right by count bits. The absolute value of count must be less than (fixnum-width).
(fxarithmetic-shift 8 2) ⇒ 32
(fxarithmetic-shift 4 0) ⇒ 4
(fxarithmetic-shift 8 -1) ⇒ 4
(fxarithmetic-shift -1 62) ⇒ -4611686018427387904

fxarithmetic-shift can be implemented via \((\text{floor } (\text{fx* } n \ (\text{expt } 2 m)))\) if this computes to a fixnum.

(fxarithmetic-shift-left n count)

(fxarithmetic-shift-right n count)

fxbit-count n

(fxbit-set? n k)

k must be non-negative and less than (fixnum-width). The fxbit-set? procedure returns #t if the
k-th bit is 1 in the two's complement representation of n, and #f otherwise. This is the fixnum result of the following computation:

\[
\text{(not (fxzero? (fxand n (fxarithmetic-shift-left 1 k)))))}
\]

(fxcopy-bit n k b)  
\(k\) must be non-negative and less than (fixnum-width). \(b\) must be 0 or 1. The \text{fxcopy-bit} procedure returns the result of replacing the k-th bit of \(n\) by \(b\), which is the result of the following computation:

\[
\text{(fxif (fxarithmetic-shift-left 1 k) (fxarithmetic-shift-left b k) n)}
\]

(fxmin n m)  
Returns the minimum of fixnums \(n\) and \(m\).

(fxmax n m)  
Returns the maximum of fixnums \(n\) and \(m\).

(fxrandom max)  
Returns a random number between \(min\) (inclusive) and \(max\) (exclusive). If \(min\) is not provided, then 0 is assumed to be the minimum bound. \(min\) is required to be non-negative, \(max\) is required to be bigger than \(min\).

(fixnum-width)  
Returns the number of bits for representing fixnums. The current implementation of LispKit always returns 64.

(least-fixnum)  
Returns the smallest possible fixnum. The current implementation of LispKit always returns -9223372036854775808.

(greatest-fixnum)  
Returns the greatest possible fixnum. The current implementation of LispKit always returns 9223372036854775807.

### 21.11 Floating-point operations

(real->flonum x)  
Returns the best floating-point (flonum) representation of \(x\).

(fl+ x y)  
(fl- x y)  
These procedures return the flonum sum or product of their flonum arguments \(x\) and \(y\). In general, they return the flonum that best approximates the mathematical sum or product.

(fl- x y)  
(fl/ x y)  
These procedures return the flonum difference or quotient of their flonum arguments \(x\) and \(y\). In general, they return the flonum that best approximates the mathematical difference or quotient.

(flzero? x)  
Returns \#t if \(x = 0.0\), \#f otherwise.
(flpositive? x)
Returns #t if x > 0.0, #f otherwise.

(flnegative? x)
Returns #t if x < 0.0, #f otherwise.

(fl= x y)
(fl< x y)
(fl> x y)
(fl<= x y)
(fl>= x y)
These procedures return #t if their flonum arguments x and y are respectively: equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing, #f otherwise.

(fl= +inf.0 +inf.0) ⇒ #t
(fl= -inf.0 +inf.0) ⇒ #f
(fl= -inf.0 -inf.0) ⇒ #t
(fl= 0.0 -0.0) ⇒ #t
(fl< 0.0 -0.0) ⇒ #f
(fl= +nan.0 123.0) ⇒ #f
(fl< +nan.0 123.0) ⇒ #f

(flabs x)
Returns the absolute value of x as a flonum.

(flmin x y)
Returns the minimum value of x and y.

(flmax x y)
Returns the maximum value of x and y.
22 LispKit Object

Library (lispkit object) implements a simple, delegation-based object system for LispKit. It provides procedural and declarative interfaces for objects and classes. The class system is optional. It mostly provides means to define and manage new object types and construct objects using object constructors.

22.1 Introduction

Similar to other Scheme and Lisp-based object systems, methods of objects are defined in terms of object/class-specific specializations of generic procedures. A generic procedure consists of methods for the various objects/classes it supports. A generic procedure performs a dynamic dispatch on the first parameter (the self parameter) to determine the applicable method.

22.1.1 Generic procedures

Generic procedures can be defined using the define-generic form. Here is an example which defines three generic methods, one with only a self parameter, and two with three parameters self, x and y. The last generic procedure definition includes a default method which is applicable to all objects for which there is no specific method. When a generic procedure without default is applied to an object that does not define its own method implementation, an error gets signaled.

```latex
(define-generic (point-coordinates self))
(define-generic (set-point-coordinates! self x y))
(define-generic (point-move! self x y)
  (let ((coord (point-coordinate self)))
    (set-point-coordinate! self (+ (car coord) x) (+ (cdr coord) y))))
```

22.1.2 Objects

An object encapsulates a list of methods each implementing a generic procedure. These methods are regular closures which can share mutable state. Objects do not have an explicit notion of a field or slot as in other Scheme or Lisp-based object systems. Fields/slots need to be implemented via generic procedures and method implementations sharing state. Here is an example explaining this approach:

```latex
(define (make-point x y) (object ()
  ((point-coordinates self) (cons x y))
  ((set-point-coordinates! self nx ny) (set! x nx) (set! y ny))
  ((object->string self) (string-append (object->string x) "/" (object->string y))))
```

This is a function creating new point objects. The x and y parameters of the constructor function are used for representing the state of the point object. The created point objects implement three generic procedures: point-coordinates, set-point-coordinates, and object->string. The latter procedure is defined directly by the library and, in general, used for creating a string representation.
of any object. By implementing the `object->string` method, the behavior gets customized for the object.

The following lines of code illustrate how point objects can be used:

```lisp
(define pt (make-point 25 37))
pt => #object:<box (…)>
(object->string pt) => "25/37"
(point-coordinates pt) => (25 . 37)
(set-point-coordinates! pt 5 6)
(object->string pt) => "5/6"
(point-coordinates pt) => (5 . 6)
```

### 22.1.3 Inheritance

The LispKit object system supports inheritance via delegation. The following code shows how colored points can be implemented by delegating all point functionality to the previous implementation and by simply adding only color-related logic.

```lisp
(define-generic (point-color self) #f)
(define (make-colored-point x y color)
  (object ((super (make-point x y)))
    ((point-color self) color)
    ((object->string self)
      (string-append (object->string color) ":" (invoke (super object->string) self))))))
```

The object created in function `make-colored-point` inherits all methods from object `super` which gets set to a new point object. It adds a new method to generic procedure `point-color` and redefines the `object->string` method. The redefinition is implemented in terms of the inherited `object->string` method for points. The form `invoke` can be used to refer to overridden methods in delegatee objects. Thus, `(invoke (super object->string) self)` calls the `object->string` method of the super object but with the identity (self) of the colored point.

The following interaction illustrates the behavior:

```lisp
(define cpt (make-colored-point 100 50 'red))
(point-color cpt) => red
(point-coordinates cpt) => (100 . 50)
(set-point-coordinates! cpt 101 51)
(object->stringcpt) => "red:101/51"
```

Objects can delegate functionality to multiple delegatees. The order in which they are listed determines the methods which are being inherited in case there are conflicts, i.e. multiple delegatees implement a method for the same generic procedure.

### 22.1.4 Classes

Classes add syntactic sugar, simplifying the creation and management of objects. They play the following role in the object-system of LispKit:

1. A class defines a constructor for objects represented by this class.
2. Each class defines an object type, which can be used to distinguish objects created by the same constructor and supporting the same methods.
3. A class can inherit functionality from several other classes, making it easy to reuse functionality.
4. Classes are first-class objects supporting a number of class-related procedures.

The following code defines a `point` class with similar functionality as above:

```
(define-class (point x y) ()
  (object ()
    ((point-coordinates self) (cons x y))
    ((set-point-coordinates! self nx ny) (set! x nx) (set! y ny))
    ((object->string self) (string-append (object->string x) "/" (object->string y))))
)
```

Instances of this class are created by using the generic procedure `make-instance` which is implemented by all class objects:

```
(define pt2 (make-instance point 82 10))
(pt2) => #point:#<box (...>)
(object->string pt2) => "82/10"
```

Each object created by a class implements a generic procedure `object-class` referring to the class of the object. Since classes are objects themselves we can obtain their name with generic procedure `class-name`:

```
(object-class pt2) => #class:#<box (...>)
(class-name (object-class pt2)) => point
(instance-of? point pt2) => #t
(instance-of? point pt)
```

Generic procedure `instance-of?` can be used to determine whether an object is a direct or indirect instance of a given class. The last two lines above show that `pt2` is an instance of `point`, but `pt` is not, even though it is functionally equivalent.

The following definition re-implements the colored point example from above using a class:

```
(define-class (colored-point x y color) (point)
  (if (or (< x 0) (< y 0))
      (error "coordinates are negative: ($0; $1)" x y))
  (object ((super (make-instance point x y)))
    ((point-color self) color)
    ((object->string self)
      (string-append (object->string color) ":" (invoke (super object->string) self))))
)
```

The following lines illustrate the behavior of `colored-point` objects vs `point` objects:

```
(point-color cpt2) => blue
(point-coordinates cpt2) => (128 . 256)
(set-point-coordinates! cpt2 64 32)
(object->string cpt2) => "blue:64/32"
(instance-of? point cpt2) => #t
(instance-of? colored-point cpt2) => #f
(instance-of? colored-point cpt) => #f
(class-name (object-class cpt2)) => colored-point
```

#### 22.2 Procedural object interface

```
(object? expr)  procedure
```
(make-object)  
(procedure)

(make-object delegate …)  
(procedure)

(method obj generic)  
(procedure)

(object-methods obj)  
(procedure)

(add-method! obj generic method)  
(procedure)

(delete-method! obj generic)  
(procedure)

(make-generic-procedure …)  
(procedure)

22.3 Declarative object interface

(object …)  
(syntax)

(define-generic …)  
(syntax)

(invoke …)  
(syntax)

22.4 Procedural class interface

(class? expr)  
(procedure)

root  
(object)

(make-class name superclasses constructor)  
(procedure)

22.4.1 Instance methods

(object-class self)  
(generic procedure)

(object-equal? self obj)  
(generic procedure)

(object->string self)  
(generic procedure)
22.4.2 Class methods

(class-name self)  
(class-direct-superclasses self)  
(subclass? self other)  
(make-instance self . args)  
(instance-of? self obj)

22.5 Declarative class interface

(define-class ...)
23 LispKit Port

Ports represent abstractions for handling input and output. They are used to access files, devices, and similar things on the host system on which LispKit is running.

An input port is a LispKit object that can deliver data upon command, while an output port is an object that can accept data. In LispKit, input and output port types are disjoint, i.e. a port is either an input or an output port.

Different port types operate on different data. LispKit provides two different types of ports: textual ports and binary ports. Textual ports and binary ports are disjoint, i.e. a port is either textual or binary.

A textual port supports reading or writing of individual characters from or to a backing store containing characters using read-char and write-char, and it supports operations defined in terms of characters, such as read and write.

A binary port supports reading or writing of individual bytes from or to a backing store containing bytes using read-u8 and write-u8 below, as well as operations defined in terms of bytes.

23.1 Default ports

current-output-port
current-input-port
current-error-port

These parameter objects represent the current default input port, output port, or error port (an output port), respectively. These parameter objects can be overridden with parameterize.

default-output-port
default-input-port

These two ports are the initial values of current-output-port and current-input-port when LispKit gets initialized. They are typically referring to the default output and input device of the system on which LispKit is running.

23.2 Predicates

(port? obj)  procedure
Returns #t if obj is a port object; otherwise #f is returned.

(input-port? obj) procedure
(output-port? obj) procedure

These predicates return #t if obj is an input port or output port; otherwise they return #f.

(textual-port? obj) procedure
(binary-port? obj) procedure

These predicates return #t if obj is a textual or a binary port; otherwise they return #f.
(input-port-open? port)  
(output-port-open? port)

Returns #t if port is still open and capable of performing input or output, respectively, and #f otherwise.

(eof-object? obj)

Returns #t if obj is an end-of-file object, otherwise returns #f.

### 23.3 General ports

(close-port port)  
(close-input-port port)  
(close-output-port port)

Closes the resource associated with port, rendering the port incapable of delivering or accepting data. It is an error to apply close-input-port and close-output-port to a port which is not an input or output port, respectively. All procedures for closing ports have no effect if the provided port has already been closed.

(with-input-from-port port thunk)  
(with-output-to-port port thunk)

The given port is made to be the value returned by current-input-port or current-output-port (as used by (read), (write obj), and so forth). The thunk is then called with no arguments. When the thunk returns, the port is closed and the previous default is restored. It is an error if thunk does not accept zero arguments. Both procedures return the values yielded by thunk. If an escape procedure is used to escape from the continuation of these procedures, they behave exactly as if the current input or output port had been bound dynamically with parameterize.

(call-with-port port proc)

The call-with-port procedure calls proc with port as an argument. It is an error if proc does not accept one argument.

If proc returns, then the port is closed automatically and the values yielded by proc are returned. If proc does not return, then the port will not be closed automatically unless it is possible to prove that the port will never again be used for a read or write operation.

This is necessary, because LispKit's escape procedures have unlimited extent and thus it is possible to escape from the current continuation but later to resume it. If LispKit would be permitted to close the port on any escape from the current continuation, then it would be impossible to write portable code using both call-with-current-continuation and call-with-port.

### 23.4 File ports

(open-input-file filepath)  
(open-binary-input-file filepath)

Takes a filepath referring to an existing file and returns a textual input port or binary input port that is capable of delivering data from the file. If the file does not exist or cannot be opened, an error that satisfies file-error? is signaled.

(open-output-file filepath)  
(open-binary-output-file filepath)
Takes a filepath referring to an output file to be created and returns a textual output port or binary output port that is capable of writing data to the new file. If a file with the given name exists already, the effect is unspecified. If the file cannot be opened, an error that satisfies file-error? is signaled.

(with-input-from-file filepath thunk)
(with-output-to-file filepath thunk)

The file determined by filepath is opened for input or output as if by open-input-file or open-output-file, and the new port is made to be the value returned by current-input-port or current-output-port (as used by (read), (write obj), and so forth). The thunk is then called with no arguments. When the thunk returns, the port is closed and the previous default is restored. It is an error if thunk does not accept zero arguments. Both procedures return the values yielded by thunk. If an escape procedure is used to escape from the continuation of these procedures, they behave exactly as if the current input or output port had been bound dynamically with parameterize.

(call-with-input-file filepath proc)
(call-with-output-file filepath proc)

These procedures create a textual port obtained by opening the file referred to by filepath (a string) for input or output as if by open-input-file or open-output-file. This port and proc are then passed to a procedure equivalent to call-with-port. It is an error if proc does not accept one argument.

### 23.5 String ports

(open-input-string str)

Takes a string and returns a textual input port that delivers characters from the string. If the string is modified, the effect is unspecified.

(open-output-string)

Returns a textual output port that will accumulate characters for retrieval by get-output-string.

```lisp
(parameterize ((current-output-port (open-output-string)))
  (display "piece")
  (display " by piece ")
  (display "by piece.")
  (get-output-string (current-output-port)))
⇒ "piece by piece by piece."
```

(get-output-string port)

It is an error if port was not created with open-output-string.

Returns a string consisting of the characters that have been output to port so far in the order they were output.

```lisp
(parameterize ((current-output-port (open-output-string)))
  (display "piece")
  (display " by piece ")
  (display "by piece.")
  (newline)
  (get-output-string (current-output-port)))
⇒ "piece by piece by piece."
```

(with-input-from-string str thunk)

String str is opened for input as if by open-input-string, and the new textual string port is made to be the value returned by current-input-port. The thunk is then called with no arguments. When the
thunk returns, the port is closed and the previous default is restored. It is an error if thunk does not accept zero arguments. with-input-from-string returns the values yielded by thunk. If an escape procedure is used to escape from the continuation of these procedures, they behave exactly as if the current input port had been bound dynamically with parameterize.

(with-output-to-string thunk)
A new string output port is created as if by calling open-output-string, and the new port is made to be the value returned by current-output-port. The thunk is then called with no arguments. When the thunk returns, the port is closed and the previous default is restored. It is an error if thunk does not accept zero arguments. Both procedures return the values yielded by thunk. If an escape procedure is used to escape from the continuation of these procedures, they behave exactly as if the current input or output port had been bound dynamically with parameterize.

(call-with-output-string proc)
The procedure proc is called with one argument, a textual output port. The values yielded by proc are ignored. When proc returns, call-with-output-string returns the port's accumulated output as a string.

This procedure is defined as follows:

```lisp
(define (call-with-output-string procedure)
  (let ((port (open-output-string)))
    (procedure port)
    (get-output-string port)))
```

### 23.6 Bytevector ports

(open-input-bytevector bvector)
Takes a bytevector bvector and returns a binary input port that delivers bytes from the bytevector bvector.

(open-output-bytevector)
Returns a binary output port that will accumulate bytes for retrieval by get-output-bytevector.

(get-output-bytevector port)
It is an error if port was not created with open-output-bytevector. get-output-bytevector returns a bytevector consisting of the bytes that have been output to the port so far in the order they were output.

(call-with-output-bytevector proc)
The procedure proc gets called with one argument, a binary output port. The values yielded by procedure proc are ignored. When it returns, call-with-output-bytevector returns the port's accumulated output as a newly allocated bytevector.

This procedure is defined as follows:

```lisp
(define (call-with-output-bytevector procedure)
  (let ((port (open-output-bytevector)))
    (procedure port)
    (get-output-bytevector port)))
```

### 23.7 URL ports

(open-input-url url)
Takes a url referring to an existing resource and returns a textual input port that is capable of reading data from the resource (e.g. via HTTP).
**LispPad Library Reference**

(open-binary-input-url url)
Takes a url referring to an existing resource and returns a binary input port that is capable of reading data from the resource (e.g. via HTTP).

(with-input-from-url url thunk)
The given url is opened for input as if by open-input-url, and the new input port is made to be the value returned by current-input-port. The thunk is then called with no arguments. When the thunk returns, the port is closed and the previous default is restored. It is an error if thunk does not accept zero arguments. The procedure returns the values yielded by thunk. If an escape procedure is used to escape from the continuation of this procedure, they behave exactly as if current-input-port had been bound dynamically with parameterize.

(call-with-input-url obj)
call-with-input-url creates a textual input port by opening the resource at url for input as if by open-input-url. This port and proc are then passed to a procedure equivalent to call-with-port. It is an error if proc does not accept one argument.

23.8 Asset ports

(open-input-asset name type)
(open-input-asset name type dir)
This function can be used to open a textual LispKit asset file located in one of LispKit’s asset paths. An asset is identified via a file name, a file type, and an optional directory path dir. name, type, and dir are all strings. open-input-asset constructs a relative file path in the following way (assuming name does not have a suffix already):

dir/name.type

It then searches the asset paths in their given order for a file matching this relative file path. Once the first matching file is found, the file is opened as a text file and a corresponding textual input port that is capable of reading data from the file is returned. It is an error if no matching asset is found.

(open-binary-input-asset name type)
(open-binary-input-asset name type dir)
This function can be used to open a binary LispKit asset file located in one of LispKit’s asset paths. An asset is identified via a file name, a file type, and an optional directory path dir. name, type, and dir are all strings. open-input-asset constructs a relative file path in the following way (assuming name does not have a suffix already):

dir/name.type

It then searches the asset paths in their given order for a file matching this relative file path. Once the first matching file is found, the file is opened as a binary file and a corresponding binary input port that is capable of reading data from the file is returned. It is an error if no matching asset is found.

23.9 Reading from ports

If port is omitted from any input procedure, it defaults to the value returned by (current-input-port) . It is an error to attempt an input operation on a closed port.

(read)
(read port)
The `read` procedure converts external representations of Scheme objects into the objects themselves by parsing the input. `read` returns the next object parsable from the given textual input `port`, updating `port` to point to the first character past the end of the external representation of the object.

If an end of file is encountered in the input before any characters are found that can begin an object, then an end-of-file object is returned. The port remains open, and further attempts to read will also return an end-of-file object. If an end of file is encountered after the beginning of an object's external representation, but the external representation is incomplete and therefore not parsable, an error that satisfies `read-error?` is signaled.

**(read-char)**

**(read-char port)**

Returns the next character available from the textual input `port`, updating `port` to point to the following character. If no more characters are available, an end-of-file object is returned.

**(peek-char)**

**(peek-char port)**

Returns the next character available from the textual input `port`, but without updating `port` to point to the following character. If no more characters are available, an end-of-file object is returned.

Note: The value returned by a call to `peek-char` is the same as the value that would have been returned by a call to `read-char` with the same port. The only difference is that the very next call to `read-char` or `peek-char` on that port will return the value returned by the preceding call to `peek-char`. In particular, a call to `peek-char` on an interactive port will hang waiting for input whenever a call to `read-char` would have hung.

**(char-ready?)**

**(char-ready? port)**

Returns `#t` if a character is ready on the textual input `port` and returns `#f` otherwise. If `char-ready?` returns `#t` then the next `read-char` operation on the given `port` is guaranteed not to hang. If the `port` is at end of file, then `char-ready?` returns `#t`.

Rationale: The `char-ready?` procedure exists to make it possible for a program to accept characters from interactive ports without getting stuck waiting for input. Any input editors associated with such ports must ensure that characters whose existence has been asserted by `char-ready?` cannot be removed from the input. If `char-ready?` were to return `#f` at end of file, a port at end of file would be indistinguishable from an interactive port that has no ready characters.

**(read-token)**

**(read-token port)**

**(read-token port charset)**

Returns the next token of text available from the textual input `port`, updating `port` to point to the following character. A token is a non-empty sequence of characters delimited by characters from character set `charset`. Tokens never contain characters from `charset`. `charset` defaults to the set of all whitespace and newline characters.

**(read-line obj)**

**(read-line port)**

Returns the next line of text available from the textual input `port`, updating `port` to point to the following character. If an end of line is read, a string containing all of the text up to (but not including) the end of line is returned, and `port` is updated to point just past the end of line. If an end of file is encountered before any end of line is read, but some characters have been read, a string containing those characters is returned. If an end of file is encountered before any characters are read, an end-of-file object is returned.
For the purpose of this procedure, an end of line consists of either a linefeed character, a carriage return character, or a sequence of a carriage return character followed by a linefeed character.

\[(\text{read-string } k)\]
\[(\text{read-string } k \text{ port})\]

Reads the next \(k\) characters, or as many as are available before the end of file, from the textual input \(\text{port}\) into a newly allocated string in left-to-right order and returns the string. If no characters are available before the end of file, an end-of-file object is returned.

\[(\text{read-u8})\]
\[(\text{read-u8 } \text{port})\]

Returns the next byte available from the binary input \(\text{port}\), updating \(\text{port}\) to point to the following byte. If no more bytes are available, an end-of-file object is returned.

\[(\text{peek-u8 } \text{obj})\]

Returns the next byte available from the binary input \(\text{port}\), but without updating \(\text{port}\) to point to the following byte. If no more bytes are available, an end-of-file object is returned.

\[(\text{u8-ready?})\]
\[(\text{u8-ready? } \text{port})\]

Returns \#t if a byte is ready on the binary input \(\text{port}\) and returns \#f otherwise. If \(\text{u8-ready?}\) returns \#t then the next \text{read-u8} operation on the given \(\text{port}\) is guaranteed not to hang. If the \(\text{port}\) is at end of file then \(\text{u8-ready?}\) returns \#t.

\[(\text{read-bytevector } k)\]
\[(\text{read-bytevector } k \text{ port})\]

Reads the next \(k\) bytes, or as many as are available before the end of file, from the binary input \(\text{port}\) into a newly allocated bytevector in left-to-right order and returns the bytevector. If no bytes are available before the end of file, an end-of-file object is returned.

\[(\text{read-bytevector! } \text{bvector})\]
\[(\text{read-bytevector! } \text{bvector } \text{port})\]
\[(\text{read-bytevector! } \text{bvector } \text{port } \text{start})\]
\[(\text{read-bytevector! } \text{bvector } \text{port } \text{start } \text{end})\]

Reads the next \(end - start\) bytes, or as many as are available before the end of file, from the binary input \(\text{port}\) into bytevector \(\text{bvector}\) in left-to-right order beginning at the \(\text{start}\) position. If \(\text{end}\) is not supplied, reads until the end of bytevector \(\text{bvector}\) has been reached. If \(\text{start}\) is not supplied, reads beginning at position 0. Returns the number of bytes read. If no bytes are available, an end-of-file object is returned.

### 23.10 Writing to ports

If \(\text{port}\) is omitted from any output procedure, it defaults to the value returned by \((\text{current-output-port})\). It is an error to attempt an output operation on a closed port.

\[(\text{write } \text{obj})\]
\[(\text{write } \text{obj } \text{port})\]

Writes a representation of \(\text{obj}\) to the given textual output \(\text{port}\). Strings that appear in the written representation are enclosed in quotation marks, and within those strings backslash and quotation mark characters are escaped by backslashes. Symbols that contain non-ASCII characters are escaped with vertical lines. Character objects are written using the \#\(\backslash\) notation.
If obj contains cycles which would cause an infinite loop using the normal written representation, then at least the objects that form part of the cycle will be represented using datum labels. Datum labels will not be used if there are no cycles.

**(write-shared obj)**
**(write-shared obj port)**

The **write-shared** procedure is the same as **write**, except that shared structures will be represented using datum labels for all pairs and vectors that appear more than once in the output.

**(write-simple obj)**
**(write-simple obj port)**

The **write-simple** procedure is the same as **write**, except that shared structures will never be represented using datum labels. This can cause **write-simple** not to terminate if obj contains circular structures.

**(display obj)**
**(display obj port)**

Writes a representation of obj to the given textual output port. Strings that appear in the written representation are output as if by **write-string** instead of by **write**. Symbols are not escaped. Character objects appear in the representation as if written by **write-char** instead of by **write**. **display** will not loop forever on self-referencing pairs, vectors, or records.

The **write** procedure is intended for producing machine-readable output and **display** for producing human-readable output.

**(newline)**
**(newline port)**

Writes an end of line to textual output port.

**(write-char char)**
**(write-char char port)**

Writes the character char (not an external representation of the character) to the given textual output port.

**(write-string str)**
**(write-string str port)**
**(write-string str port start)**
**(write-string str port start end)**

Writes the characters of string str from index start to end (exclusive) in left-to-right order to the textual output port. The default of start is 0, the default of end is the length of str.

**(write-u8 byte)**
**(write-u8 byte)**

Writes the byte to the given binary output port.

**(write-bytevector bvector)**
**(write-bytevector bvector port)**
**(write-bytevector bvector port start)**
**(write-bytevector bvector port start end)**

Writes the bytes of bytevector bvector from start to end (exclusive) in left-to-right order to the binary output port. The default of start is 0, the default of end is the length of bvector.
Flushes any buffered output from the buffer of the given output port to the underlying file or device.

(eof-object)  
Returns an end-of-file object.
24 LispKit Record

Library (lispkit record) implements record types for LispKit. A record provides a simple and flexible mechanism for building structures with named components wrapped in distinct types.

24.1 Declarative API

record-type syntax is used to introduce new record types in a declarative fashion. Like other definitions, record-type can either appear at the outermost level or locally within a body. The values of a record type are called records and are aggregations of zero or more fields, each of which holds a single location. A predicate, a constructor, and field accessors and mutators are defined for each record type.

(define-record-type <name> <constr> <pred> <field> ...) syntax

<name> and <pred> are identifiers. The <constructor> is of the form:

(\<constructor name> \<field name> ...)

and each <field> is either of the form:

(\<field name> \<accessor name>), or
(\<field name> \<accessor name> \<modifier name>).

It is an error for the same identifier to occur more than once as a field name. It is also an error for the same identifier to occur more than once as an accessor or mutator name.

The define-record-type construct is generative: each use creates a new record type that is distinct from all existing types, including the predefined types and other record types - even record types of the same name or structure.

An instance of define-record-type is equivalent to the following definitions:

- <name> is bound to a representation of the record type itself.
- <constructor name> is bound to a procedure that takes as many arguments as there are <field name> elements in the (<constructor name> ...) subexpression and returns a new record of type <name>. Fields whose names are listed with <constructor name> have the corresponding argument as their initial value. The initial values of all other fields are unspecified. It is an error for a field name to appear in <constructor> but not as a <field name>.
- <pred> is bound to a predicate that returns #t when given a value returned by the procedure bound to <constructor name> and #f for everything else.
- Each <accessor name> is bound to a procedure that takes a record of type <name> and returns the current value of the corresponding field. It is an error to pass an accessor a value which is not a record of the appropriate type.
- Each <modifier name> is bound to a procedure that takes a record of type <name> and a value which becomes the new value of the corresponding field. It is an error to pass a modifier a first argument which is not a record of the appropriate type.

For instance, the following record-type definition:
(define-record-type <pare>
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))

defines \texttt{kons} to be a constructor, \texttt{kar} and \texttt{kdr} to be accessors, \texttt{set-kar!} to be a modifier, and \texttt{pare?} to be a type predicate for instances of \texttt{<pare>}.

\begin{verbatim}
(pare? (kons 1 2)) ⇒ #t
(pare? (cons 1 2)) ⇒ #f
(kar (kons 1 2)) ⇒ 1
(kdr (kons 1 2)) ⇒ 2
(let ((k (kons 1 2)))
  (set-kar! k 3) (kar k)) ⇒ 3
\end{verbatim}

24.2 Procedural API

Besides the syntactical \texttt{define-record-type} abstraction for defining record types in a declarative fashion, LispKit provides a low-level, procedural API for creating and instantiating records and record types. Record types are represented in form of record type descriptor objects which itself are records.

\begin{itemize}
  \item \texttt{(record? obj)} \quad procedure
  \begin{itemize}
    \item Returns \#t if \texttt{obj} is a record of any type; returns \#f otherwise.
  \end{itemize}
  \item \texttt{(record-type? obj)} \quad procedure
  \begin{itemize}
    \item Returns \#t if \texttt{obj} is a record type descriptor; returns \#f otherwise.
  \end{itemize}
  \item \texttt{(record-type obj)} \quad procedure
  \begin{itemize}
    \item Returns the record type descriptor for objects \texttt{obj} which are records; returns \#f for all non-record values.
  \end{itemize}
  \item \texttt{(make-record-type name fields)} \quad procedure
  \begin{itemize}
    \item Returns a record type descriptor which represents a new data type that is disjoint from all other types. \texttt{name} is a string which is only used for debugging purposes, such as the printed representation of a record of the new type. \texttt{fields} is a list of symbols naming the fields of a record of the new type. It is an error if the list contains duplicate symbols.
  \end{itemize}
  \item \texttt{(record-type-name rtd)} \quad procedure
  \begin{itemize}
    \item Returns the type name (a string) associated with the type represented by the record type descriptor \texttt{rtd}. The returned value is \texttt{eqv?} to the \texttt{name} argument given in the call to \texttt{make-record-type} that created the type represented by \texttt{rtd}.
  \end{itemize}
  \item \texttt{(record-type-field-names rtd)} \quad procedure
  \begin{itemize}
    \item Returns a list of the symbols naming the fields in members of the type represented by the record type descriptor \texttt{rtd}. The returned value is \texttt{equal?} to the \texttt{fields} argument given in the call to \texttt{make-record-type} that created the type represented by \texttt{rtd}.
  \end{itemize}
  \item \texttt{(make-record rtd)} \quad procedure
  \begin{itemize}
    \item Returns an uninitialized instance of the record type for which \texttt{rtd} is the record type descriptor.
  \end{itemize}
  \item \texttt{(record-constructor rtd fields)} \quad procedure
  \begin{itemize}
    \item Returns a procedure for constructing new members of the type represented by the record type descriptor \texttt{rtd}. The returned procedure accepts exactly as many arguments as there are symbols in the given \texttt{fields} list; these are used, in order, as the initial values of those fields in a new record, which is returned by the
  \end{itemize}
\end{itemize}
constructor procedure. The values of any fields not named in fields are unspecified. It is an error if fields contain any duplicates or any symbols not in the fields list of the record type descriptor rtd.

(record-predicate rtd)

Returns a procedure for testing membership in the type represented by the record type descriptor rtd. The returned procedure accepts exactly one argument and returns #t if the argument is a member of the indicated record type; it returns #f otherwise.

(record-field-accessor rtd field)

Returns a procedure for reading the value of a particular field of a member of the type represented by the record type descriptor rtd. The returned procedure accepts exactly one argument which must be a record of the appropriate type; it returns the current value of the field named by the symbol field in that record. The symbol field must be a member of the list of field names in the call to make-record-type that created the type represented by rtd.

(record-field-mutator rtd field)

Returns a procedure for writing the value of a particular field of a member of the type represented by the record type descriptor rtd. The returned procedure accepts exactly two arguments: first, a record of the appropriate type, and second, an arbitrary Scheme value; it modifies the field named by the symbol field in that record to contain the given value. The returned value of the modifier procedure is unspecified. The symbol field must be a member of the list of field names in the call to make-record-type that created the type represented by rtd.
25 LispKit Regexp

Library (lispkit regexp) provides an API for defining regular expressions and applying them to strings. Supported are both matching as well as search/replace.

25.1 Regular expressions

The regular expression syntax supported by this library corresponds to the one of NSRegularExpression of Apple's Foundation framework. This is also the origin of the documentation of this section.

25.1.1 Meta-characters

\a : Match a bell ( ^A )
\A : Match at the beginning of the input. Differs from ^ in that \A will not match after a new line within the input.
\b : Outside of a [Set], match if the current position is a word boundary. Boundaries occur at the transitions between word ( \w ) and non-word ( \W ) characters, with combining marks ignored. Inside of a [Set], match a backspace ( \b ).
\B : Match if the current position is not a word boundary.
\cX : Match a control-X character.
\d : Match any character with the unicode general category of Nd, i.e. numbers and decimal digits.
\D : Match any character that is not a decimal digit.
\e : Match an escape ( \e ).
\E : Terminates a \Q ... \E quoted sequence.
\f : Match a form feed ( \f ).
\G : Match if the current position is at the end of the previous match.
\n : Match a line feed ( \n ).
\N{unicode character} : Match the named character.
\p{unicode property} : Match any character with the specified unicode property.
\P{unicode property} : Match any character not having the specified unicode property.
\Q : Quotes all following characters until \E.
\r : Match a carriage return ( \r ).
\s : Match a whitespace character. Whitestace is defined as [\t\n\f\r\p{Z}].
\S : Match a non-whitespace character.
\t : Match a horizontal tabulation ( \t ).
\u{hhhh} : Match the character with the hex value hhhh.
\u{hhhhhhhh} : Match the character with the hex value hhhhhhhhh. Exactly eight hex digits must be provided, even though the largest Unicode code point is \U0010ffff.
\w : Match a word character. Word characters are [p{\p{Lu}\p{Ll}\p{Lt}\p{Lo}\p{Nd}}].
\W : Match a non-word character.
\x{hhhh} : Match the character with hex value hhhh. From one to six hex digits may be supplied. \x{hh} : Match the character with two digit hex value hh.
\X : Match a grapheme cluster.
\Z: Match if the current position is at the end of input, but before the final line terminator, if one exists.
\z: Match if the current position is at the end of input.  \ n: Back Reference. Match whatever the \-th capturing group matched.  \ must be a number ≥ 1 and ≤ total number of capture groups in the pattern.
\|oooo: Match an octal character. ooo is from one to three octal digits.  0377 is the largest allowed octal character. The leading zero is required and distinguishes octal constants from back references.
[\pattern] : Match any one character from the pattern.
\ : Match any character.
^: Match at the beginning of a line.
$: Match at the end of a line.
\ : Quotes the following character. Characters that must be quoted to be treated as literals are * ? + [ ( ) { } ^ $ | \ . /.

25.2 Regular expression operators

| : Alternation.  A|B matches either A or B.
* : Match 0 or more times, as many times as possible.
+ : Match 1 or more times, as many times as possible.
? : Match zero or one times, preferring one time if possible.
{n} : Match exactly \times.
{n,} : Match at least \times, as many times as possible.
{n,m} : Match between \times and \times, as many times as possible, but not more than \times.
*? : Match zero or more times, as few times as possible.
+? : Match one or more times, as few times as possible.
?? : Match zero or one times, preferring zero.
{n}? : Match exactly \times.
{n,}? : Match at least \times, but no more than required for an overall pattern match.
{n,m}? : Match between \times and \times, as few times as possible, but not less than \times.
**: Match zero or more times, as many times as possible when first encountered, do not retry with fewer even if overall match fails (possessive match).  ++ : Match one or more times (possessive match).
?? : Match zero or one times (possessive match).
{n}+: Match exactly \times.
{n,}+: Match at least \times (possessive match).
{n,m}+: Match between \times and \times (possessive match).
( ... ) : Capturing parentheses; the range of input that matched the parenthesized subexpression is available after the match.
(??: ... ) : Non-capturing parentheses; groups the included pattern, but does not provide capturing of matching text (more efficient than capturing parentheses).
(?>... ) : Atomic-match parentheses; first match of the parenthesized subexpression is the only one tried. If it does not lead to an overall pattern match, back up the search for a match to a position before the "(?>".
(?# ... ) : Free-format comment (?# comment).
( ?= ... ) : Look-ahead assertion. True, if the parenthesized pattern matches at the current input position, but does not advance the input position.
(?! ... ) : Negative look-ahead assertion. True, if the parenthesized pattern does not match at the current input position. Does not advance the input position.
( ?< ... ) : Look-behind assertion. True, if the parenthesized pattern matches text preceding the current input position, with the last character of the match being the input character just before the current position. Does not alter the input position. The length of possible strings matched by the look-behind pattern must not be unbounded (no * or + operators).  (?! ... ) : Negative look-behind assertion. True, if the parenthesized pattern does not match text preceding the current input position,
with the last character of the match being the input character just before the current position. Does not alter the input position. The length of possible strings matched by the look-behind pattern must not be unbounded (no \* or + operators).

\(?ismw\-iswx\: ... \) : Flag settings. Evaluate the parenthesized expression with the specified flags enabled or disabled.

\(?ismw\-iswx\) : Flag settings. Change the flag settings. Changes apply to the portion of the pattern following the setting. For example, \(?i\) changes to a case insensitive match.

25.2.1 Template Matching

$n$ : The text of capture group $n$ will be substituted for $\$n$. $n$ must be $\geq 0$ and not greater than the number of capture groups. A $\$ not followed by a digit has no special meaning, and will appear in the substitution text as itself, i.e. $\$.

\:\ : Treat the following character as a literal, suppressing any special meaning. Backslash escaping in substitution text is only required for $\$ and \\, but may be used on any other character.

25.2.2 Flag options

The following flags control various aspects of regular expression matching. These flags get specified within the pattern using the (\(?ismx\-isnx\)) pattern options.

i : If set, matching will take place in a case-insensitive manner.
x : If set, allow use of white space and #comments within patterns.
s : If set, a “.” in a pattern will match a line terminator in the input text. By default, it will not. Note that a carriage-return/line-feed pair in text behave as a single line terminator, and will match a single “.” in a regular expression pattern.
m : Control the behavior of ^ and $ in a pattern. By default these will only match at the start and end, respectively, of the input text. If this flag is set, ^ and $ will also match at the start and end of each line within the input text.
w : Controls the behavior of \b in a pattern. If set, word boundaries are found according to the definitions of word found in Unicode UAX 29, Text Boundaries. By default, word boundaries are identified by means of a simple classification of characters as either word or non-word, which approximates traditional regular expression behavior.

25.3 API

(regexp? obj) procedure

Returns #t if obj is a regular expression object; otherwise #f is returned.

(regexp str) (regexp str opt ...) procedure

Returns a new regular expression object from the given regular expression pattern str and matching options opt, ... . str is a string, matching options opt are symbols. The following matching options are supported:

• case-insensitive: Match letters in the regular expression independent of their case.
• allow-comments: Ignore whitespace and # -prefixed comments in the regular expression pattern.
• ignore-meta: Treat the entire regular expression pattern as a literal string.
• dot-matches-line-separator: Allow . to match any character, including line separators.
• anchors-match-lines: Allow ^ and $ to match the start and end of lines.
• unix-only-line-separators: Treat only \n as a line separator; otherwise, all standard line
separators are used.
• unicode-words: Use Unicode TR#29 to specify word boundaries; otherwise, all traditional regular
expression word boundaries are used.

(regexp-pattern regexp) procedure
Returns the regular expression pattern for the given regular expression object regexp. A regular expression
pattern is a string matching the regular expression syntax supported by library (lispkit regexp).

(regexp-capture-groups regexp) procedure
Returns the number of capture groups of the given regular expression object regexp.

(escape-regexp-pattern str) procedure
Returns a regular expression pattern string by adding backslash escapes to pattern str as necessary to
protect any characters that would match as pattern meta-characters.

(escape-regexp-template str) procedure
Returns a regular expression pattern template string by adding backslash escapes to pattern template str
as necessary to protect any characters that would match as pattern meta-characters.

(regexp-matches regexp str) procedure
(regexp-matches regexp str start) procedure
(regexp-matches regexp str start end) procedure
Returns a matching spec if the regular expression object regexp successfully matches the entire string str
from position start (inclusive) to end (exclusive); otherwise, #f is returned. The default for start is 0; the
default for end is the length of the string.

A matching spec returned by regexp-matches consists of pairs of fixnum positions (startpos . endpos) in
str. The first pair is always representing the full match (i.e. startpos is 0 and endpos is the length of str),
all other pairs represent the positions of the matching capture groups of regexp.

(define email
  (regexp "[A-Z0-9a-z_.%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")
(regexp-matches email "matthias@objecthub.net")
⇒ ((0 . 22))
(define series
  (regexp "Season\\s+\\d+\\sEpisode\\s+\\d+")
(regexp-matches series "Season 3 Episode 12")
⇒ ((0 . 20) (7 . 8) (18 . 20))

(regexp-matches? regexp str) procedure
(regexp-matches? regexp str start) procedure
(regexp-matches? regexp str start end) procedure
Returns #t if the regular expression object regexp successfully matches the entire string str from position
start (inclusive) to end (exclusive); otherwise, #f is returned. The default for start is 0; the default for end
is the length of the string.

(regexp-search regexp str) procedure
(regexp-search regexp str start) procedure
(regexp-search regexp str start end) procedure
Returns a matching spec for the first match of the regular expression regexp with a part of string str between
position start (inclusive) and end (exclusive). If regexp does not match any part of str between start and
end, #f is returned. The default for start is 0; the default for end is the length of the string.
A matching spec returned by `regexp-search` consists of pairs of fixnum positions (startpos, endpos) in str. The first pair is always representing the full match of the pattern, all other pairs represent the positions of the matching capture groups of `regexp`.

```
(define email
  (regexp "([A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4})")
(regexp-search email "Contact matthias@objecthub.net or foo@bar.org")
⇒ ((8 . 39))
(define series
  (regexp "Season\s+(\d+)\s+Episode\s+(\d+)")
(regexp-search series "New Season 3 Episode 12: Pilot")
⇒ ((4 . 23) (11 . 12) (21 . 23))
```

```
(regexp-search-all regexp str) probe
(regexp-search-all regexp str start)
(regexp-search-all regexp str start end)
```

Returns a list of all matching specs for matches of the regular expression `regexp` with parts of string `str` between position `start` (inclusive) and `end` (exclusive). If `regexp` does not match any part of `str` between `start` and `end`, the empty list is returned. The default for `start` is 0; the default for `end` is the length of the string.

A matching spec returned by `regexp-search` consists of pairs of fixnum positions (startpos, endpos) in str. The first pair is always representing the full match of the pattern, all other pairs represent the positions of the matching capture groups of `regexp`.

```
(define email
  (regexp "([A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4})")
(regexp-search-all email "Contact matthias@objecthub.net or foo@bar.org")
⇒ (((8 . 30)) ((34 . 45)))
(define series
  (regexp "Season\s+(\d+)\s+Episode\s+(\d+)")
(regexp-search-all series "New Season 3 Episode 12: Pilot")
⇒ (((4 . 23) (11 . 12) (21 . 23)))
```

```
(regexp-extract regexp str)
(regexp-extract regexp str start)
(regexp-extract regexp str start end)
```

Returns a list of substrings from `str` which all represent full matches of the regular expression `regexp` with parts of string `str` between position `start` (inclusive) and `end` (exclusive). If `regexp` does not match any part of `str` between `start` and `end`, the empty list is returned. The default for `start` is 0; the default for `end` is the length of the string.

```
(define email
  (regexp "([A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4})")
(regexp-extract email "Contact matthias@objecthub.net or foo@bar.org" 10)
⇒ ("matthias@objecthub.net" "foo@bar.org")
(define series
  (regexp "Season\s+(\d+)\s+Episode\s+(\d+)")
(regexp-extract series "New Season 3 Episode 12: Pilot")
⇒ ("Season 3 Episode 12")
```

```
(regexp-split regexp str)
(regexp-split regexp str start)
(regexp-split regexp str start end)
```

Splits string \texttt{str} into a list of possibly empty substrings separated by non-empty matches of regular expression \texttt{regexp} within position \texttt{start} (inclusive) and \texttt{end} (exclusive). If \texttt{regexp} does not match any part of \texttt{str} between \texttt{start} and \texttt{end}, a list with \texttt{str} as its only element is returned. The default for \texttt{start} is 0; the default for \texttt{end} is the length of the string.

\begin{verbatim}
(define email
  (regexp 
    "[A-Z0-9a-z.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}\")

(regexp-split email "Contact matthias@objecthub.net or foo@bar.org" 10)
⇒ ("Contact ma" " or " "")

(define series
  (regexp
    "Season\s+([^\d]+)\s+Episode\s+([^\d]+)"

(regexp-split series "New Season 3 Episode 12: Pilot")
⇒ ("New " "Season 3 Episode 12: Pilot")
\end{verbatim}

\begin{verbatim}
(regexp-partition \texttt{regexp} \texttt{str})
(regexp-partition \texttt{regexp} \texttt{str} \texttt{start})
(regexp-partition \texttt{regexp} \texttt{str} \texttt{start} \texttt{end})
\end{verbatim}

Partitions string \texttt{str} into a list of non-empty strings matching regular expression \texttt{regexp} within position \texttt{start} (inclusive) and \texttt{end} (exclusive), interspersed with the unmatched portions of the whole string. The first and every odd element is an unmatched substring, which will be the empty string if \texttt{regexp} matches at the beginning of the string or end of the previous match. The second and every even element will be a substring fully matching \texttt{regexp}. If \texttt{str} is the empty string or if there is no match at all, the result is a list with \texttt{str} as its only element.

\begin{verbatim}
(define email
  (regexp 
    "[A-Z0-9a-z.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}\")

(regexp-partition email "Contact matthias@objecthub.net or foo@bar.org" 10)
⇒ ("Contact ma<omitted> or <omitted>

(define series
  (regexp
    "Season\s+([^\d]+)\s+Episode\s+([^\d]+)"

(regexp-partition series "New Season 3 Episode 12: Pilot")
⇒ ("New " "Season 3 Episode 12: Pilot")
\end{verbatim}

\begin{verbatim}
(regexp-replace \texttt{regexp} \texttt{str} \texttt{subst})
(regexp-replace \texttt{regexp} \texttt{str} \texttt{subst} \texttt{start})
(regexp-replace \texttt{regexp} \texttt{str} \texttt{subst} \texttt{start} \texttt{end})
\end{verbatim}

Returns a new string replacing all matches of regular expression \texttt{regexp} in string \texttt{str} within position \texttt{start} (inclusive) and \texttt{end} (exclusive) with string \texttt{subst}. \texttt{regexp-replace} will always return a new string, even if there are no matches and replacements.

The optional parameters \texttt{start} and \texttt{end} restrict both the matching and the substitution, to the given positions, such that the result is equivalent to omitting these parameters and replacing on \texttt{(substring str start end)}.

\begin{verbatim}
(define email
  (regexp 
    "[A-Z0-9a-z.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}\")

(regexp-replace email "Contact matthias@objecthub.net or foo@bar.org" "<omitted>" 10)
⇒ "Contact ma<omitted> or <omitted>

(define series
  (regexp
    "Season\s+([^\d]+)\s+Episode\s+([^\d]+)"

(regexp-replace series "New Season 3 Episode 12: Pilot" "Series")
⇒ "New Series: Pilot"
\end{verbatim}

\begin{verbatim}
(regexp-replace! \texttt{x})
\end{verbatim}

Mutates string \texttt{str} by replacing all matches of regular expression \texttt{regexp} within position \texttt{start} (inclusive)
and end (exclusive) with string subst. The optional parameters start and end restrict both the matching and the substitution. regexp-replace! returns the number of replacements that were applied.

```
(define email
  (regexp "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")
(define str "Contact mathias@objecthub.net or foo@bar.org")
(regexp-replace! email str "<omitted>" 10) ⇒ 2
str ⇒ "Contact ma<omitted> or <omitted>"
```

(regexp-fold regexp kons knil str)
(regexp-fold regexp kons knil str finish)
(regexp-fold regexp kons knil str finish start)
(regexp-fold regexp kons knil str finish start end)

regexp-fold is the most fundamental and generic regular expression matching iterator. It repeatedly searches string str for the regular expression regexp so long as a match can be found. On each successful match, it applies (kons i regexp-match str acc) where \( i \) is the index since the last match (beginning with start), regexp-match is the resulting matching spec, and acc is the result of the previous kons application, beginning with knil. When no more matches can be found, regexp-fold calls finish with the same arguments, except that regexp-match is #f. By default, finish just returns acc.

```
(regexp-fold (regexp "(\w+)")
    (lambda (i m str acc)
        (let ((s (substring str (caar m) (cdar m))))
            (if (zero? i) s
                (string-append acc "-" s))))
    "" "to be or not to be"
⇒ "to-be-or-not-to-be"
```
26 LispKit Set

Library (lispkit set) provides a generic implementation for sets of objects. Its API design is compatible to the R6RS-style API of library (lispkit hashtable).

A set is a data structure for representing collections of objects. Any object can be used as element, provided a hash function and a suitable equivalence function is available. A hash function is a procedure that maps elements to exact integer objects. It is the programmer’s responsibility to ensure that the hash function is compatible with the equivalence function, which is a procedure that accepts two objects and returns true if they are equivalent and #f otherwise. Standard sets for arbitrary objects based on the eq?, eqv?, and equal? predicates are provided.

26.1 Constructors

(make-eq-set)
Create a new empty set using eq? as equivalence function.

(make-eqv-set)
Create a new empty set using eqv? as equivalence function.

(make-equal-set)
Create a new empty set using equal? as equivalence function.

(make-set hash equiv)
Create a new empty set using the given hash function hash and equivalence function equiv. An initial capacity k can be provided optionally.

(make-set hash equiv k)
Create a new set using the given hash function hash and equivalence function equiv. An initial capacity k can be provided optionally.

26.2 Inspection

(set-equivalence-function s)
Returns the equivalence function used by set s.

(set-hash-function s)
Returns the hash function used by set s.

(set-mutable? s)
Returns #t if set s is mutable.
26.3 Predicates

(set? obj)
Returns #t if obj is a set.

(set-empty? obj)
Returns #t if obj is an empty set.

(set=? s1 s2)
Returns #t if set s1 and set s2 are using the same equivalence function and contain the same elements.

(disjoint? s1 s2)
Returns #t if set s1 and set s2 are disjoint sets.

(subset? s1 s2)
Returns #t if set s1 is a subset of set s2.

(proper-subset? s1 s2)
Returns #t if set s1 is a proper subset of set s2, i.e. s1 is a subset of s2 and s1 is not equivalent to s2.

(set-contains? s element)
Returns #t if set s contains element.

(set-any? s proc)
Returns true if there is at least one element in set s for which procedure proc returns true (i.e. not #f).

(set-every? s proc)
Returns true if procedure proc returns true (i.e. not #f) for all elements of set s.

26.4 Procedures

(set-size s)
Returns the number of elements in set s.

(set-elements s)
Returns the elements of set s as a vector.

(set-copy s)

(set-copy s mutable)
Copies set s creating an immutable copy if mutable is set to #f or if mutable is not provided.

(set-for-each s proc)
Applies procedure proc to all elements of set s in an undefined order.

(set-filter s pred)
Creates a new set containing the elements of set s for which the procedure pred returns true.

(set-union s s1 …)
Creates a new set containing the union of s with s1 ….

(set-intersection s s1 …)
Creates a new set containing the intersection of s with s1 ….

(set-difference s s1 …)
Creates a new set containing the difference of s and the sets in s1 ….

(set->list s)
Returns the elements of set s as a list.
(list->eq-set elements)
Creates a new set using the equivalence function eq? from the values in list elements.

(list->eqv-set elements)
Creates a new set using the equivalence function eqv? from the values in list elements.

(list->equal-set elements)
Creates a new set using the equivalence function equal? from the values in list elements.

26.5 Mutators

(set-adjoin! s element …)
Adds element … to the set s.

(set-delete! s element …)
Deletes element … from the set s.

(set-clear! s)
(set-clear! s k)
Clears set s and reserves a capacity of k elements if k is provided.

(list->set! s elements)
Adds the values of list elements to set s.

(set-filter! s pred)
Removes all elements from set s for which procedure pred returns #f.

(set-union! s s1 …)
Stores the union of set s and sets s1 … in s.

(set-intersection! s s1 …)
Stores the intersection of set s and the sets s1 … in s.

(set-difference! s s1 …)
Stores the difference of set s and the sets s1 … in s.
27 LispKit String

Strings are sequences of characters. In LispKit, characters are UTF-16 code units. Strings are written as sequences of characters enclosed within quotation marks (" "). Within a string literal, various escape sequences represent characters other than themselves. Escape sequences always start with a backslash \ :

- \a: alarm (U+0007)
- \b: backspace (U+0008)
- \t: character tabulation (U+0009)
- \n: linefeed (U+000A)
- \r: return (U+000D)
- \": double quote (U+0022)
- \\: backslash (U+005C)
- \|: vertical line (U+007C)
- \line-end: used for encoding multi-line string literals
- \hex-scalar-value ;: specified character

The result is unspecified if any other character in a string occurs after a backslash.

Except for a line ending, any character outside of an escape sequence stands for itself in the string literal. A line ending which is preceded by a backslash expands to nothing and can be used to encode multi-line string literals.

```lisp
(display "The word \"recursion\" has many meanings.") ⇒
The word "recursion" has many meanings.
(display "Another example:\n\ntwo lines of text.\n") ⇒
Another example:
two lines of text.
(display "\x03B1; is named GREEK SMALL LETTER ALPHA." ) ⇒
α is named GREEK SMALL LETTER ALPHA.
```

The length of a string is the number of characters, i.e. UTF-16 code units, that it contains. This number is an exact, non-negative integer that is fixed when the string is created. The valid indexes of a string are the exact non-negative integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

Some of the procedures that operate on strings ignore the difference between upper and lower case. The names of the versions that ignore case end with -ci (for “case insensitive”).

LispKit only supports mutable strings.

27.1 Basic constructors and procedures

(make-string k)  
(make-string k char)

The make-string procedure returns a newly allocated string of length k. If char is given, then all the characters of the string are initialized to char, otherwise the contents of the string are unspecified.
Returns a newly allocated string composed of the arguments. It is analogous to procedure \texttt{list}.

Returns a newly allocated string composed of the characters contained in \texttt{list}.

The \texttt{string-ref} procedure returns character \texttt{k} of string \texttt{str} using zero-origin indexing. It is an error if \texttt{k} is not a valid index of string \texttt{str}.

The \texttt{string-set!} procedure stores \texttt{char} in element \texttt{k} of string \texttt{str}. It is an error if \texttt{k} is not a valid index of string \texttt{str}.

Returns the number of characters in the given string \texttt{str}.

### 27.2 Predicates

\texttt{(string? obj)}

Returns \texttt{#t} if \texttt{obj} is a string; otherwise returns \texttt{#f}.

\texttt{(string-empty? str)}

Returns \texttt{#t} if \texttt{str} is an empty string, i.e. a string of length 0. Otherwise, \texttt{string-empty?} returns \texttt{#f}.

\texttt{(string=? str ...)}

Returns \texttt{#t} if all the strings have the same length and contain exactly the same characters in the same positions; otherwise \texttt{string=?} returns \texttt{#f}.

\texttt{(string<? str ...)}\texttt{(string>? str ...)}\texttt{(string<=? str ...)}\texttt{(string=>? str ...)}

These procedures return \texttt{#t} if their arguments are (respectively): monotonically increasing, monotonically decreasing, monotonically non-decreasing, or monotonically non-increasing. These predicates are transitive.

These procedures compare strings in a lexicographic fashion; i.e. \texttt{string<?} implements a the lexicographic ordering on strings induced by the ordering \texttt{char<?} on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string would be considered to be lexicographically less than the longer string.

A pair of strings satisfies exactly one of \texttt{string<?}, \texttt{string=?}, and \texttt{string>?}. A pair of strings satisfies \texttt{string<?} if and only if they do not satisfy \texttt{string>?}. A pair of strings satisfies \texttt{string>=?} if and only if they do not satisfy \texttt{string<?}.


These procedures compare strings in a case-insensitive fashion. The \texttt{"-ci"} procedures behave as if they applied \texttt{string-foldcase} to their arguments before invoking the corresponding procedures without \texttt{"-ci"}. 
(string-contains? str sub)
Returns #t if string str contains string sub; returns #f otherwise.

(string-prefix? str sub)
Returns #t if string str has string sub as a prefix; returns #f otherwise.

(string-suffix? str sub)
Returns #t if string str has string sub as a suffix; returns #f otherwise.

27.3 Composing and extracting strings

Many of the following procedures accept an optional start and end argument as their last two arguments. If both or one of these optional arguments are not provided, start defaults to 0 and end defaults to the length of the corresponding string.

(string-contains str sub)
(string-contains str sub start)
(string-contains str sub start end)

This procedure checks whether string sub is contained in string str within the index range start to end. It returns the first index into str at which sub is fully contained within start and end. If sub is not contained in the substring of str, then #f is returned.

(substring str start end)
The substring procedure returns a newly allocated string formed from the characters of string str beginning with index start and ending with index end. This is equivalent to calling string-copy with the same arguments, but is provided for backward compatibility and stylistic flexibility.

(string-append str ...
Returns a newly allocated string whose characters are the concatenation of the characters in the given strings str ....

(string-concatenate list)
(string-concatenate list sep)

Returns a newly allocated string whose characters are the concatenation of the characters in the strings contained in list. sep is either a character or string, which, if provided, is used as a separator between two strings that get concatenated. It is an error if list is not a proper list containing only strings as elements.

(string-upcase str)
(string-downcase str)
(string-titlecase str)
(string-foldcase str)

These procedures apply the Unicode full string uppercasing, lowercasing, titlecasing, and case-folding algorithms to their argument string str and return the result as a newly allocated string. It is not guaranteed that the resulting string has the same length like str. Language-sensitive string mappings and foldings are not used.

(string-copy str)
(string-copy str start)
(string-copy str start end)

Returns a newly allocated copy of the part of the given string str between start and end.

(string-split str sep allow-empty?)
Procedure string-split splits string str using the separator sep and returns a list of the component
strings, in order. `sep` is either a string or a character. Boolean argument `allow-empty?` determines whether empty component strings are dropped. `allow-empty?` is #t by default.

```lisp
(string-split "name-|-street-|-zip-|-city-|-" "-|-") ⇒ ("name" "street" "zip" "city" ")
(string-split "name-|-street-|-zip-|-city-|-" "-|-" #f) ⇒ ("name" "street" "zip" "city")
```

**(string-trim str)**  
**(string-trim str chars)**

Returns a newly allocated string by removing all characters from the beginning and end of string `str` that are contained in `chars`. `chars` is either a string or it is a character set. If `chars` is not provided, whitespaces and newlines are being removed.

```lisp
(string-trim " lispkit is fun ") ⇒ "lispkit is fun"
(string-trim "________" ") ⇒ ""
(string-trim "712+72=784" (char-set->string char-set:digit)) ⇒ "+72="
(string-trim "712+72=784" char-set:digit) ⇒ "+72="
```

**(string-pad-right str char k)**  
**(string-pad-right str char k force-length?)**

Procedure `string-pad-right` returns a newly allocated string created by padding string `str` at the beginning of the string with character `char` until it is of length `k`. If `k` is less than the length of string `str`, the resulting string gets truncated at length `k` if boolean argument `force-length?` is #t; otherwise, the string `str` gets returned as is.

```lisp
(string-pad-right "scheme" 
  
  ) ⇒ "scheme 
(string-pad-right "scheme" 
  " " 
  ) ⇒ "scheme"
(string-pad-right "scheme" 
  
  ) ⇒ "sche"
(string-pad-right "scheme" 
  "_" 
  ) ⇒ "scheme___"
```

**(string-pad-left str char k)**  
**(string-pad-left str char k force-length?)**

Procedure `string-pad-left` returns a newly allocated string created by padding string `str` at the beginning of the string with character `char` until it is of length `k`. If `k` is less than the length of string `str`, the resulting string gets truncated at length `k` if boolean argument `force-length?` is #t; otherwise, the string `str` gets returned as is.

```lisp
(string-pad-left "scheme" 
  " " 
  ) ⇒ " scheme
(string-pad-left "scheme" 
  " " 
  ) ⇒ "scheme"
(string-pad-left "scheme" 
  "_" 
  ) ⇒ "heme"
(string-pad-left "scheme" 
  "_" 
  ) ⇒ "___scheme"
```

**(string-pad-center str char k)**  
**(string-pad-center str char k force-length?)**

Procedure `string-pad-center` returns a newly allocated string created by padding string `str` at the beginning and end with character `char` until it is of length `k`, such that `str` is centered in the middle. If `k` is less than the length of string `str`, the resulting string gets truncated at length `k` if boolean argument `force-length?` is #t; otherwise, the string `str` gets returned as is.

```lisp
(string-pad-center "scheme" 
  " " 
  ) ⇒ " scheme"
(string-pad-center "scheme" 
  " " 
  ) ⇒ "scheme"
(string-pad-center "scheme" 
  "_" 
  ) ⇒ "heme"
(string-pad-center "scheme" 
  "_" 
  ) ⇒ "___scheme"
```
27.4 Manipulating strings

\[(\text{string-replace!} \ \text{str} \ \text{sub} \ \text{repl})\]  
\[(\text{string-replace!} \ \text{str} \ \text{sub} \ \text{repl} \ \text{start})\]  
\[(\text{string-replace!} \ \text{str} \ \text{sub} \ \text{repl} \ \text{start} \ \text{end})\]

Replaces all occurrences of string \(\text{sub}\) in string \(\text{str}\) between indices \(\text{start}\) and \(\text{end}\) with string \(\text{repl}\) and returns the number of occurrences of \(\text{sub}\) that were replaced.

\[(\text{string-replace-first!} \ \text{str} \ \text{sub} \ \text{repl})\]  
\[(\text{string-replace-first!} \ \text{str} \ \text{sub} \ \text{repl} \ \text{start})\]  
\[(\text{string-replace-first!} \ \text{str} \ \text{sub} \ \text{repl} \ \text{start} \ \text{end})\]

Replaces the first occurrence of string \(\text{sub}\) in string \(\text{str}\) between indices \(\text{start}\) and \(\text{end}\) with string \(\text{repl}\) and returns the index at which the first occurrence of \(\text{sub}\) was replaced.

\[(\text{string-insert!} \ \text{str} \ \text{repl})\]  
\[(\text{string-insert!} \ \text{str} \ \text{repl} \ \text{start})\]  
\[(\text{string-insert!} \ \text{str} \ \text{repl} \ \text{start} \ \text{end})\]

Replaces the part of string \(\text{str}\) between index \(\text{start}\) and \(\text{end}\) with string \(\text{repl}\). The default for \(\text{start}\) is 0, for \(\text{end}\) it is \(+ (\text{string-length} \ \text{str}) 1\).

\[(\text{string-append!} \ \text{str} \ \text{other} \ \ldots)\]

Appends the strings \(\text{other}, \ldots\) to mutable string \(\text{str}\) in the given order.

\[(\text{string-copy!} \ \text{to} \ \text{at} \ \text{from})\]  
\[(\text{string-copy!} \ \text{to} \ \text{at} \ \text{from} \ \text{start})\]  
\[(\text{string-copy!} \ \text{to} \ \text{at} \ \text{from} \ \text{start} \ \text{end})\]

Copies the characters of string \(\text{from}\) between index \(\text{start}\) and \(\text{end}\) to string \(\text{to}\), starting at index \(\text{at}\). If the source and destination overlap, copying takes place as if the source is first copied into a temporary string and then into the destination. It is an error if \(\text{at}\) is less than zero or greater than the length of string \(\text{to}\). It is also an error if \((- (\text{string-length} \ \text{to}) \ \text{at})\) is less than \((- \text{end} \ \text{start})\).

\[(\text{string-fill!} \ \text{str} \ \text{fill})\]  
\[(\text{string-fill!} \ \text{str} \ \text{fill} \ \text{start})\]  
\[(\text{string-fill!} \ \text{str} \ \text{fill} \ \text{start} \ \text{end})\]

The \(\text{string-fill!}\) procedure stores \(\text{fill}\) in the elements of string \(\text{str}\) between index \(\text{start}\) and \(\text{end}\). It is an error if \(\text{fill}\) is not a character.

27.5 Iterating over strings

\[(\text{string-map} \ \text{proc} \ \text{str} \ \ldots)\]

The \(\text{string-map}\) procedure applies procedure \(\text{proc}\) element-wise to the characters of the strings \(\text{str} \ \ldots\) and returns a string of the results, in order. If more than one string \(\text{str}\) is given and not all strings have the same length, \(\text{string-map}\) terminates when the shortest string runs out. It is an error if \(\text{proc}\) does not accept as many arguments as there are strings and returns a single character.

\[(\text{string-map} \ \text{char-foldcase} \ "AbdEgH") \Rightarrow \ "abdegh"\]
\[(\text{string-map} \ (\text{lambda} (\text{c}) (\text{integer->char} (+ 1 (\text{char->integer} \ \text{c})))) \ "HAL") \Rightarrow \ "IBM"\]

\[(\text{string-for-each} \ \text{proc} \ \text{str} \ \ldots)\]

The arguments to \(\text{string-for-each}\) are like the arguments to \(\text{string-map}\), but \(\text{string-for-each}\)
calls proc for its side effects rather than for its values. Unlike string-map, string-for-each is guaranteed to call proc on the characters of the strings in order from the first character to the last. If more than one string str is given and not all strings have the same length, string-for-each terminates when the shortest string runs out. It is an error for proc to mutate any of the strings. It is an error if proc does not accept as many arguments as there are strings.

### 27.6 Converting strings

(string->list str)  
(string->list str start)  
(string->list str start end)

The string->list procedure returns a list of the characters of string str between start and end preserving the order of the characters.

### 27.7 Input/Output

(read-file path)  

Reads the text file at path and stores its content in a newly allocated string which gets returned by read-file.

(write-file path str)  

Writes the characters of string str into a new text file at path. write-file returns #t if the file could be written successfully; otherwise #f is returned.
28 LispKit System

28.1 Source files

(load filename)
(load filename environment)

(load) reads a source file specified by filename and executes it in the given environment. If no environment is specified, the current interaction environment is used, which can be accessed via (interaction-environment). Execution of the file consists of reading expressions and definitions from the file, compiling them, and evaluating them sequentially in the environment. load returns the result of evaluating the last expression or definition from the file. During compilation, the special form source-directory can be used to access the directory in which the executed file is located.

It is an error if filename is not a string. If filename is not an absolute file path, LispKit will try to find the file in a predefined set of directories, such as the default libraries search path. If no file suffixes are provided, the system is trying to determine the right suffix. For instance, (load "Prelude") will find the prelude file, determine its suffix and load and execute the file.

28.2 File operations

Files and directories are referenced by paths. Paths are strings consisting of directory names separated by character '/' optionally followed by a filename (if the path refers to a file). Paths are either absolute, if they start with character '/', or they are relative to some unspecified directory.

If a relative path is used to refer to a concrete directory or file, e.g. in the API provided by library (lisp-kit port), typically the path is interpreted as relative to the current-directory, unless specified otherwise.

LispKit supports ways to read and write files, inspect directories, get metadata about files (e.g. file sizes), etc.

current-directory

Defines the path referring to the current directory. Each LispKit virtual machine has its own current directory.

source-directory

Returns the directory in which the source file is located which is currently being compiled and executed. Typically, such source files are executed via procedure load.

(path path comp …)

Constructs a new relative file or directory path consisting of a relative (or absolute) base path base and a number of path components comp …. If it is not possible to construct a valid path, this procedure returns #f.

(path "one" "two" "three.png") ⇒ "one/two/three.png"
(parent-path path)
Returns the parent path of path. The result is either a relative path if path is relative, or the result is an absolute path. parent-path returns #f if path is not a valid path.

((parent-path "one/two/three.png")  ⇒ "one/two"
(parent-path "three.png")  ⇒ ".")

(path-components path)
Returns the individual components of a (relative or absolute) path as a list of strings. Returns #f if path is not a valid path.

((path-components "one/two/three.png")  ⇒ ("one" "two" "three.png")

(file-path path)
(file-path path base)
Constructs a new absolute file or directory path consisting of a base path base and a relative file path path.

((file-path "Photos/img01.jpg" "/Users/objecthub")  ⇒ "/Users/objecthub/Photos/img01.jpg"

(asset-file-path name type)
(asset-file-path name type dir)
Returns a new absolute file or directory path to a LispKit asset. An asset is identified via a file name, a file type, and an optional directory path dir. name, type, and dir are all strings. An asset is a file which is located directly or indirectly in one of the asset directories part of the LispKit installation. An asset has a type, which is the default suffix of the file (e.g. "png" for PNG images). If dir is provided, it is a relative path to a sub-directory within a matching asset directory.

asset-file-path constructs a relative file path in the following way (assuming there is no existing file suffix already):

dir/name.type

It then searches the asset paths in their given order for a file matching this relative file path. Once the first matching file is found, an absolute file path for this file is returned by asset-file-path. If no valid (and existing) file is found, asset-file-path returns #f.

(parent-file-path path)
If path refers to a file, then parent-file-path returns the directory in which this file is contained. If path refers to a directory, then parent-file-path returns the directory in which this directory is contained. The result of parent-file-path is always an absolute path.

(file-path-root? path)
Returns #t if path exists and corresponds to the root of the directory hierarchy. The root is typically equivalent to “/”. It is an error if path is not a string.

(file-exists? filepath)
The file-exists? procedure returns #t if the named file exists at the time the procedure is called, and #f otherwise. It is an error if filename is not a string.

(directory-exists? dirpath)
The directory-exists? procedure returns #t if the named directory exists at the time the procedure is called, and #f otherwise. It is an error if filename is not a string.
(file-or-directory-exists? path)
The file-or-directory-exists? procedure returns #t if the named file or directory exists at the time the procedure is called, and #f otherwise. It is an error if filename is not a string.

(delete-file filepath)
The delete-file procedure deletes the file specified by filepath if it exists and can be deleted. If the file does not exist or cannot be deleted, an error that satisfies file-error? is signaled. It is an error if filepath is not a string.

(delete-directory dirpath)
The delete-directory procedure deletes the directory specified by dirpath if it exists and can be deleted. If the directory does not exist or cannot be deleted, an error that satisfies file-error? is signaled. It is an error if dirpath is not a string.

(delete-file-or-directory path)
The delete-file-or-directory procedure deletes the directory or file specified by path if it exists and can be deleted. If path neither leads to a file nor a directory or the file or directory cannot be deleted, an error that satisfies file-error? is signaled. It is an error if path is not a string.

(copy-file filepath targetpath)
The copy-file procedure copies the file specified by filepath to the file specified by targetpath. An error satisfying file-error? is signaled if filepath does not lead to an existing file or if a file at targetpath cannot be written. It is an error if either filepath or targetpath are not strings.

(copy-directory dirpath targetpath)
The copy-directory procedure copies the directory specified by dirpath to the directory specified by targetpath. An error satisfying file-error? is signaled if dirpath does not lead to an existing directory or if a directory at targetpath cannot be written. It is an error if either dirpath or targetpath are not strings.

(copy-file-or-directory sourcepath targetpath)
The copy-file-or-directory procedure copies the file or directory specified by sourcepath to the file or directory specified by targetpath. An error satisfying file-error? is signaled if sourcepath does not lead to an existing file or directory, or if a file or directory at targetpath cannot be written. It is an error if either sourcepath or targetpath are not strings.

(move-file filepath targetpath)
Moves the file at filepath to targetpath. This procedure fails if filepath does not reference an existing file, or if the file cannot be moved to targetpath. It is an error if either filepath or targetpath are not strings.

(move-directory dirpath targetpath)
Moves the directory at dirpath to targetpath. This procedure fails if dirpath does not reference an existing directory, or if the directory cannot be moved to targetpath. It is an error if either dirpath or targetpath are not strings.

(move-file-or-directory sourcepath targetpath)
Moves the file or directory at sourcepath to targetpath. This procedure fails if sourcepath does not reference an existing file or directory, or if the file or directory cannot be moved to targetpath. It is an error if either sourcepath or targetpath are not strings.

(file-size filepath)
Returns the size of the file specified by filepath in bytes. It is an error if filepath is not a string or if filepath does not reference an existing file.

(directory-list dirpath)
Returns a list of names of files and directories contained in the directory specified by dirpath. It is an error if dirpath is not a string or if dirpath does not reference an existing directory.
(make-directory dirpath)  
Creates a directory with path dirpath. If the directory exists already or if it is not possible to create a directory with path dirpath, make-directory fails with an error. It is an error if dirpath is not a string.

(open-file filepath)  
(open-file filepath app)  
(open-file filepath app activate)

Opens the file specified by filepath with the application app. activate is a boolean argument. If it is #t, it will make app the frontmost application after invoking it. If app is not specified, the default application for the type of the file specified by filepath is used. If activate is not specified, it is assumed it is #t. open-file returns #t if it was possible to open the file, #f otherwise. Example: (open-file "~/Users/objecthub/main.swift" "TextEdit").

28.3 Network operations

(open-url url)  
Opens the given url in the default browser and makes the browser the frontmost application.

(http-get url)  
(http-get url timeout)

http-get performs an http get request for the given URL. timeout is a floating point number defining the time in seconds it should take at most for receiving a response. http-get returns two values: the HTTP header in form of an association list, and the content in form of a bytevector. It is an error if the http get request fails. Example:

```
(http-get "http://github.com/objecthub")
⇒
(("Date" . "Sat, 17 Nov 2018 22:47:19 GMT")
 ("Referrer-Policy" . "origin-when-cross-origin, strict-origin-when-cross-origin")
 ("X-XSS-Protection" . "1; mode=block")
 ("Status" . "200 OK")
 ("Transfer-Encoding" . "Identity")
 ...
 ("Content-Type" . "text/html; charset=utf-8")
 ("Server" . "GitHub.com"))
```

28.4 Time operations

(time expr)  
time compiles expr and executes it. The form displays the time it took to execute expr as a side-effect. It returns the result of executing expr.

(current-second)  
Returns a floating-point number representing the current time on the International Atomic Time (TAI) scale. The value 0.0 represents midnight on January 1, 1970 TAI (equivalent to ten seconds before midnight UTC) and the value 1.0 represents one TAI second later. Note: The current implementation returns the same number like current-seconds. This is not conforming to the R7RS spec requiring TAI scale.
(current-jiffy) procedure
Returns the number of jiffies as a fixnum that have elapsed since an arbitrary epoch. A jiffy is a fraction of a second which is defined by the return value of the jiffies-per-second procedure. The starting epoch is guaranteed to be constant during a run of the program, but may vary between runs.

(jiffies-per-second) procedure
Returns a fixnum representing the number of jiffies per SI second. Here is an example for how to use jiffies-per-second:

```
(define (time-length)
  (let ((list (make-list 100000))
         (start (current-jiffy)))
    (length list)
    (/ (- (current-jiffy) start) (jiffies-per-second))))
```

28.5 Locales

For handling locale-specific behavior, e.g. for formatting numbers and dates, library (lispkit system) defines a framework in which

- regions/countries are identified via ISO 3166-1 Alpha 2-code strings,
- languages are identified via ISO 639-1 2-letter strings, and
- locales (i.e. combinations of regions and languages) are identified as symbols.

Library (lispkit system) provides functions for returning all available regions, languages, and locales. It also defines functions to map identifiers to human-readable names and to construct identifiers out of other identifiers.

(available-regions) procedure
Returns a list of 2-letter region code identifiers (strings) for all available regions.

(region-name ident) (region-name ident locale) procedure
Returns the name of the region identified by the 2-letter region code string ident for the given locale locale. If locale is not provided, the current (system-provided) locale is used.

(available-languages) procedure
Returns a list of 2-letter language code identifiers (strings) for all available languages.

(language-name ident) (language-name ident locale) procedure
Returns the name of the language identified by the 2-letter language code string ident for the given locale locale. If locale is not provided, the current (system-configured) locale is used.

(available-locales) procedure
Returns a list of all available locale identifiers (symbols).

(available-locale? locale) procedure
Returns #t if the symbol locale is identifying a locale supported by the operating system; returns #f otherwise.

(locale) (locale lang) (locale lang country)
If no argument is provided, `locale` returns the current locale (symbol) which got configured by the user for the operation system. If the string argument `lang` is provided, a locale representing `lang` (and all countries for which `lang` is supported) is returned. If both `lang` and string `country` are provided, `locale` will return a symbol identifying the corresponding locale.

This function never fails if both `lang` and `country` are strings. It can be used for constructing locales that are not supported by the underlying operating system. This can be checked with function `available-locale?`.

```
(locale) ⇒ en_US
(locale "de") ⇒ de
(locale "en" "GB") ⇒ en_GB
```

### (locale-region locale)

Returns the 2-letter region code string for the region targeted by the locale identifier `locale`. If `locale` does not target a region, `locale-region` returns `#f`.

### (locale-language locale)

Returns the 2-letter language code string for the language targeted by the locale identifier `locale`. If `locale` does not target a language, `locale-language` returns `#f`.

### (locale-currency locale)

Returns the 3-letter currency code string for the currency associated with the country targeted by `locale`. If `locale` does not target a country, `locale-currency` returns `#f`.

## 28.6 Execution environment

### (get-environment-variable name)

Many operating systems provide each running process with an environment consisting of `environment variables`. Both the name and value of an environment variable are represented as strings. The procedure `get-environment-variable` returns the value of the environment variable `name`, or `#f` if the named environment variable is not found. Example: `(get-environment-variable "PATH") `/

```
/usr/local/bin:/usr/bin:/bin
```

### (get-environment-variables)

Returns the names and values of all the environment variables as an association list, where the car of each entry is the name of an environment variable and the cdr is its value, both as strings. Example: `(("USER" . "root") ("HOME" . "/"))`

### (command-line)

Returns the command line passed to the process as a list of strings. The first string corresponds to the command name.

### (features)

Returns a list of the feature identifiers which `cond-expand` treats as true. Here is an example of what `features` might return: `(modules x86-64 lispkit macOS syntax-rules complex 64bit macOS little-endian dynamic-loading ratios r7rs)`. LispKit supports at least the following feature identifiers:

- lispkit
- r7rs
- ratios
- complex
- syntax-rules
• little-endian
• big-endian
• dynamic-loading
• modules
• 32bit
• 64bit
• macos
• macosx
• ios
• linux
• i386
• x86-64
• arm64
• arm

**(implementation-name)**

Returns the name of the Scheme implementation. For LispKit, this function returns the string “LispKit”.

**(implementation-version)**

Returns the version of the Scheme implementation as a string.

**(cpu-architecture)**

Returns the CPU architecture on which this Scheme implementation is executing as a string.

**(machine-name)**

Returns a name for the particular machine on which the Scheme implementation is currently running.

**(machine-model)**

Returns an identifier for the machine on which the Scheme implementation is currently running.

**(os-type)**

Returns the type of the operating system on which the Scheme implementation is running as a string. For macOS, this procedure returns “Darwin”.

**(os-name)**

Returns the name of the operating system on which the Scheme implementation is running as a string. For macOS, this procedure returns “macOS”.

**(os-version)**

Returns the build number of the operating system on which the Scheme implementation is running as a string. For macOS 10.14.1, this procedure returns “18B75”.

**(os-release)**

Returns the (major) release version of the operating system on which the Scheme implementation is running as a string. For macOS 10.14.1, this procedure returns “10.14”.

**(current-user-name)**

Returns the username of the user running the Scheme implementation as a string.

**(user-data username)**

Returns information about the user specified via `username` in form of a list. The list provides the following information in the given order:

1. User id (fixnum)
2. Group id (fixnum)
3. Username (string)
4. Full name (string)
5. Home directory (string)
6. Default shell (string)

Here is an example showing the result for invocation `(user-data "objecthub")`: (501 20 "objecthub" "Max Mustermann" "/Users/objecthub/" "/bin/bash")

### 28.7 Scheme environment

**gc**

Force garbage collection to be performed.

**compile expr**

Compiles expression `expr` and displays the disassembled code. This is what is being printed when executing `(compile '(do ((i 0 (fx1+ i)))((fx> i 10))(display i)(newline)))`:

<table>
<thead>
<tr>
<th>CONSTANTS:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0: #&lt;procedure display&gt;</td>
<td></td>
</tr>
<tr>
<td>1: #&lt;procedure newline&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INSTRUCTIONS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: alloc 1</td>
</tr>
<tr>
<td>1: push_fixnum 0</td>
</tr>
<tr>
<td>2: make_local_variable 0</td>
</tr>
<tr>
<td>3: push_local_value 0</td>
</tr>
<tr>
<td>4: push_fixnum 10</td>
</tr>
<tr>
<td>5: fx_gt</td>
</tr>
<tr>
<td>6: branch_if 14 ;; jump to 20</td>
</tr>
<tr>
<td>7: make_frame</td>
</tr>
<tr>
<td>8: push_constant 0 ;; #&lt;procedure display&gt;</td>
</tr>
<tr>
<td>9: push_local_value 0</td>
</tr>
<tr>
<td>10: call 1</td>
</tr>
<tr>
<td>11: pop</td>
</tr>
<tr>
<td>12: make_frame</td>
</tr>
<tr>
<td>13: push_constant 1 ;; #&lt;procedure newline&gt;</td>
</tr>
<tr>
<td>14: call 0</td>
</tr>
<tr>
<td>15: pop</td>
</tr>
<tr>
<td>16: push_local_value 0</td>
</tr>
<tr>
<td>17: fx_inc</td>
</tr>
<tr>
<td>18: set_local_value 0</td>
</tr>
<tr>
<td>19: branch -16 ;; jump to 3</td>
</tr>
<tr>
<td>20: push_void</td>
</tr>
<tr>
<td>21: reset 0, 1</td>
</tr>
<tr>
<td>22: return</td>
</tr>
</tbody>
</table>

**disassemble proc**

Disassembles procedure `proc` and prints out the code. This is what is being printed when executing `(disassemble caddr)`:

<table>
<thead>
<tr>
<th>CONSTANTS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: assert_arg_count 1</td>
</tr>
<tr>
<td>1: push_global 426</td>
</tr>
<tr>
<td>2: make_frame</td>
</tr>
<tr>
<td>3: push_global 431</td>
</tr>
<tr>
<td>4: push_local 0</td>
</tr>
<tr>
<td>5: call 1</td>
</tr>
<tr>
<td>6: tail_call 1</td>
</tr>
</tbody>
</table>

**(trace-calls)**

**(trace-calls level)**
This function is used to enable/disable call tracing. When call tracing is enabled, all function calls that are
executed by the virtual machine are being printed to the console. Call tracing operates at three levels:

- **0**: Call tracing is switched off
- **1**: Call tracing is enabled only for procedures for which it is enabled (via function `set-procedure-
  trace!`)
- **2**: Call tracing is switched on for all procedures (independent of procedure-level tracing being
  enabled or disabled)

`(trace-calls n)` will set call tracing to level `n`. If the level is omitted, `trace-calls` will return the
current call tracing level.

For instance, if call tracing is enabled via `(trace-calls 2)`, executing `(fib 3)` will print the following
call trace.

```lisp
> (define (fib n)
  (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
> (trace-calls 2)
> (fib 2)
  ($) (fib 2) in <repl>
  ($) (< 2 2) in fib
  ($) #f from <
  ($) (- 2 1) in fib
  ($) 1 from -
  ($) (fib 1) in fib
  ($) (< 1 2) in fib
  ($) #t from < in fib
  ($) 1 from fib in fib
  ($) (- 2 2) in fib
  ($) 0 from -
  ($) (fib 0) in fib
  ($) (< 0 2) in fib
  ($) #t from < in fib
  ($) #t from fib in fib
  ($) (+ 1 0) in fib
  ($) 1 from fib
1
```

Function invocations are prefixed with `$`, or `#` if it’s a tail call. The value returned by a function call is
prefixed with `$`.

**(procedure-trace? proc)**

Returns `#f` if procedure-level call tracing is disabled for `proc`, `#t` otherwise.

**(set-procedure-trace! proc trace?)**

Enables procedure-level call tracing for procedure `proc` if `trace?` is set to `#t`. It disables call tracing for
`proc` if `trace?` is `#f`.

**(available-symbols)**

Returns a list of all symbols that have been used so far.

**(loaded-libraries)**

Returns a list of all libraries that have been loaded so far.
(loaded-sources)
Returns a list of all sources that have been loaded.

(environment-info)
Prints out debug information about the current execution environment (mostly relevant for developing LispKit).
29 LispKit Type

Library (lispkit type) provides a simple, lightweight type abstraction mechanism. It allows for creating new types at runtime that are disjoint from all existing types. The library provides two different types of APIs: a purely procedural API for type creation and management, as well as a declarative API. The procedural API does not have an explicit representation of types. The declarative API introduces extensible types which do have a runtime representation.

29.1 Usage of the procedural API

New types are created with function make-type. make-type accepts one argument, which is a type label. The type label is an arbitrary value that is only used for debugging purposes. Typically, symbols are used as type labels.

The following line introduces a new type for intervals:

```
(define-values (new-interval interval? interval-ref make-interval-subtype)  
  (make-type 'interval))
```

(make-type 'interval) returns four functions:

- `new-interval` is a procedure which takes one argument, the internal representation of the interval, and returns a new object of the new interval type
- `interval?` is a type test predicate which accepts one argument and returns `#t` if the argument is of the new interval type, and `#f` otherwise.
- `interval-ref` takes one object of the new interval type and returns its internal representation. interval-ref is the inverse operation of new-interval.
- `make-interval-subtype` is a type generator (similar to make-type), a function that takes a type label and returns four functions representing a new subtype of the interval type.

Now it is possible to implement a constructor make-interval for intervals:

```
(define (make-interval lo hi)  
  (if (and (real? lo) (real? hi) (<= lo hi))  
    (new-interval (cons (inexact lo) (inexact hi)))  
    (error "make-interval: illegal arguments" lo hi)))
```

make-interval first checks that the constructor arguments are valid and then calls new-interval to create a new interval object. Interval objects are represented via pairs whose car is the lower bound, and cdr is the upper bound. Nevertheless, pairs and interval objects are distinct values as the following code shows:

```
(define interval-obj (make-interval 1.0 9.5))  
(define pair-obj (cons 1.0 9.5))

(interval? interval-obj) ⇒ #t  
(interval? pair-obj) ⇒ #f  
(equal? interval-obj pair-obj) ⇒ #f
```
The type is displayed along with the representation in the textual representation of interval objects: 

```
#interval:(1.0 . 9.5)
```

Below are a few functions for interval objects. They all use `interval-ref` to extract the internal representation from an interval object and then operate on the internal representation.

```lisp
(define (interval-length interval)
  (let ((bounds (interval-ref interval)))
    (- (cdr bounds) (car bounds))))

(define (interval-empty? interval)
  (zero? (interval-length interval)))
```

The following function calls show that `interval-ref` fails with a type error if its argument is not an interval object.

```
(interval-length interval-obj) ⇒ 8.5
(interval-empty? '(1.0 . 1.0)) ⇒
  [error] not an instance of type interval: (1.0 . 1.0)
```

## 29.2 Usage of the declarative API

The procedural API provides the most flexible way to define a new type in LispKit. On the other hand, this approach comes with two problems:

1. a lot of boilerplate needs to be written, and
2. programmers need to be experienced to correctly encapsulate new data types and to provide means to extend them.

These problems are addressed by the declarative API of `(lispkit type)`. At the core, this API defines a syntax `define-type` for declaring new types of data. `define-type` supports defining simple, encapsulated types as well as provides a means to make types extensible.

The syntax for defining a simple, non-extensible type has the following form:

```
(define-type name name?
  ((make-name x ... ) expr ... )
  name-ref
  functions)
```

`name` is a symbol and defines the name of the new type. `name?` is a predicate for testing whether a given object is of type `name`. `make-name` defines a constructor which returns a value representing the data of the new type. `name-ref` is a function to unwrap values of type `name`. It is optional and normally not needed since `functions` can be declared such that the unwrapping happens implicitly. All functions defined via `define-type` take an object (usually called `self`) of the defined type as their first argument.

There are two forms to declare a function as part of `define-type`: one providing access to `self` directly, and one only providing access to the unwrapped data value:

```
(( name-func self y ... ) expr ... )
```

provides access directly to `self` (which is a value of type `name`), and

```
(( name-func (repr) y ... ) expr ... )
```

which provides access only to the unwrapped data `repr`.

With this new syntax, type `interval` from the section describing the procedural API, can now be re-written like this:
(define-type interval
  interval?
  ((make-interval lo hi)
    (if (and (real? lo) (real? hi) (<= lo hi))
        (cons (inexact lo) (inexact hi))
        (error "make-interval: illegal arguments" lo hi)))
  ((interval-length (bounds))
    (- (cdr bounds) (car bounds)))
  ((interval-empty? self)
    (zero? (interval-length self))))

interval is a standalone type which cannot be extended. define-type provides a simple means to make types extensible such that subtypes can be created reusing the base type definition. This is done with a small variation of the define-type syntax:

(define-type (name super) name?
  ((make-name x ...) expr ...)
  name-ref
  functions )

In this syntax, super refers to the type extended by name. All extensible types extend another extensible type and there is one supertype called object provided by library (lispkit type) as a primitive.

With this syntactic facility, interval can be easily re-defined to be extensible:

(define-type (interval object)
  interval?
  ((make-interval lo hi)
    (if (and (real? lo) (real? hi) (<= lo hi))
        (cons (inexact lo) (inexact hi))
        (error "make-interval: illegal arguments" lo hi)))
  ((interval-length (bounds))
    (- (cdr bounds) (car bounds)))
  ((interval-empty? self)
    (zero? (interval-length self))))

It is now possible to define a tagged-interval data structure which inherits all functions from interval and encapsulates a tag with the interval:

(define-type (tagged-interval interval)
  tagged-interval?
  ((make-tagged-interval lo hi tag)
    (values lo hi tag))
  ((interval-tag (bounds tag))
    tag))

tagged-interval is a subtype of interval; i.e. values of type tagged-interval are also considered to be of type interval. Thus, tagged-interval inherits all function definitions from interval and defines a new function interval-tag just for tagged-interval values. Here is some code explaining the usage of tagged-interval:

(define ti (make-tagged-interval 4.0 9.0 'inclusive))
(tagged-interval? ti) ⇒ #t
(interval? ti) ⇒ #t
(interval-length ti) ⇒ 5.0
(interval-tag ti) ⇒ inclusive
(interval-tag interval-obj) ⇒ [error] not an instance of type tagged-interval: #interval:((1.0 ≤ . 9.5))
Constructors of extended types, such as `make-tagged-interval` return multiple values: all the parameters for a super-constructor call and one additional value (the last value) representing the data provided by the extended type. In the example above, `make-tagged-interval` returns three values: `lo`, `hi`, and `tag`. After the constructor `make-tagged-interval` is called, the super-constructor is invoked with arguments `lo` and `hi`. The result of `make-tagged-interval` is a tagged-interval object consisting of two state values contained in a list: one for the supertype `interval` (consisting of the bounds `(lo . hi)`) and one for the subtype `tagged-interval` (consisting of the tag). This can also be seen when displaying a tagged-interval value:

```
ti ⇒ #tagged-interval:((4.0 . 9.0) inclusive)
```

This is also the reason why function `interval-tag` gets access to two unwrapped values, `bounds` and `tag`: one (`bounds`) corresponds to the value associated with type `interval`, and the other one (`tag`) corresponds to the value associated with type `tagged-interval`.

### 29.3 API

**`make-type`**

```
(make-type type-label)
```

*procedure*

Creates a new, unique type, and returns four procedures dealing with this new type:

1. The first procedure takes one argument returning a new object of the new type wrapping the argument.
2. The second procedure is a type test predicate which accepts one argument and returns `#t` if the argument is of the new type, and `#f` otherwise.
3. The third procedure takes one object of the new type and returns its internal representation (what was passed to the first procedure).
4. The fourth procedure is a type generator (similar to `make-type`), a function that takes a type label and returns four functions representing a new subtype of the new type.

**`define-type`**

```
(define-type name name? ((make-name x ... e ...) func ...))
```

*syntax*

Defines a new standalone type `name` consisting of a type test predicate `name?`, a constructor `make-name`, and an optional function `ref` used to unwrap values of type `name`. `ref` is optional and normally not needed since functions `func` can be declared such that the unwrapping happens implicitly. All functions `func` defined via `define-type` take an object (usually called `self`) of the defined type as their first argument.

There are two ways to declare a function as part of `define-type`: one providing access to `self` directly, and one only providing access to the unwrapped data value:

- `((name-func self y ...) expr ...)` provides access directly to `self` (which is a value of type `name`), and
- `((name-func (repr) y ...) expr ...)` provides access only to the unwrapped data `repr`.

This variant of `define-type` defines a new extensible type `name` extending supertype `super`, which also needs to be an extensible type. A new extensible type `name` comes with a type test predicate `name?`, a constructor `make-name`, and an optional function `ref` used to unwrap values of type `name`. `ref` is optional.
and normally not needed since functions `func` can be declared such that the unwrapping happens implicitly. All functions `func` defined via `define-type` take an object (usually called `self`) of the defined type as their first argument.

There are two ways to declare a function as part of `define-type`: one providing access to `self` directly, and one providing access to the unwrapped data values (one for each type in the supertype chain):

- `((name-func self y ...) expr ...)` provides access directly to `self` (which is a value of type `name`), and
- `((name-func (repr ...) y ...) expr ...)` provides access only to the unwrapped data values `repr`.

Constructors of extended types return multiple values: all the parameters for a super-constructor call and one additional value (the last value) representing the data provided by the extended type.

**object**  [value]

The supertype of all extensible types defined via `define-type`.

**(extensible-type? obj)**  [*procedure*]

Returns `#t` if `obj` is a value representing an extensible type. For instance, `(extensible-type? object)` returns `#t`.
30 LispKit Vector

Vectors are heterogeneous data structures whose elements are indexed by a range of integers. A vector typically occupies less space than a list of the same length, and a randomly chosen element can be accessed in constant time vs. linear time for lists.

The length of a vector is the number of elements that it contains. This number is a non-negative integer that is fixed when the vector is created. The valid indexes of a vector are the exact, non-negative integers less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector.

Two vectors are equal if they have the same length, and if the values in corresponding slots of the vectors are equal.

A vector can be mutable or immutable. Trying to change the state of an immutable vector, e.g. via vector-set! will result in an error being raised.

Vectors are written using the notation #(obj ...). For example, a vector of length 3 containing the number zero in element 0, the list (1 2 3 4) in element 1, and the string “Lisp” in element 2 can be written as follows: #(0 (1 2 3 4) "Lisp").

Vector constants are self-evaluating, so they do not need to be quoted in programs. Vector constants, i.e. vectors created with a vector literal, are immutable.

LispKit also supports growable vectors via library (lispkit gvector). As opposed to regular vectors, a growable vector does not have a fixed size and supports adding and removing elements. While a growable vector does not satisfy the vector? predicate, this library also accepts growable vectors as parameters whenever a vector is expected. Use predicate mutable-vector? for determining whether a vector is either a regular mutable vector or a growable vector.

30.1 Predicates

(vector? obj)
Procedure vector? is a generic comparator for vectors. Vectors a and b are considered equal by vector? if their lengths are the same, and for each respective elements ai and bi, (eql ai bi) evaluates to true. eql is always applied to two arguments.

If there are only zero or one vector argument, #t is automatically returned. The dynamic order in which comparisons of elements and of vectors are performed is unspecified.
(vector= eq? #(a b c d) #(a b c d)) ⇒ #t
(vector= eq? #(a b c d) #(a b d c)) ⇒ #f
(vector= = #(1 2 3 4 5) #(1 2 3 4)) ⇒ #f
(vector= = #(1 2 3 4) #(1.0 2.0 3.0 4.0)) ⇒ #t
(vector= eq?) ⇒ #t
(vector= eq? '#(a)) ⇒ #t

30.2 Constructors

(make-vector k)
(make-vector k fill)

Returns a newly allocated vector of k elements. If a second argument is given, then each element is initialized to fill. Otherwise the initial contents of each element is unspecified.

(vector obj …)

Returns a newly allocated mutable vector whose elements contain the given arguments. It is analogous to list.

(vector 'a 'b 'c) ⇒ #(a b c)

(immutable-vector obj …)

Returns a newly allocated immutable vector whose elements contain the given arguments in the given order.

(list->vector list)

The list->vector procedure returns a newly created mutable vector initialized to the elements of the list list in the order of the list.

(list->vector '(a b c)) ⇒ #(a b c)

(list->immutable-vector list)

The list->vector procedure returns a newly created immutable vector initialized to the elements of the list list in the order of the list.

(string->vector str)
(string->vector str start)
(string->vector str start end)

The string->vector procedure returns a newly created mutable vector initialized to the elements of the string str between start and end (i.e. including all characters from index start to index end-1).

(string->vector "ABC") ⇒ (#\A #\B #\C)

(vector-copy vector)
(vector-copy vector mutable)
(vector-copy vector start)
(vector-copy vector start end)
(vector-copy vector start end mutable)

Returns a newly allocated copy of the elements of the given vector between start and end, but excluding the element at index end. The elements of the new vector are the same (in the sense of eqv?) as the elements of the old.
mutable is a boolean argument. If it is set to #f, an immutable copy of vector will be created. The type of the second argument of vector-copy is used to disambiguate between the second and third version of the function. An exact integer will always be interpreted as start, a boolean value will always be interpreted as mutable.

(define a #(1 8 2 8)) ; a may be immutable
(define b (vector-copy a)) ; creates a mutable copy of a
(vector-set! b 0 3) ; b is mutable
b ⇒ #(3 8 2 8)
(define c (vector-copy a #f)) ; creates an immutable copy of a
(vector-set! c 0 3) ⇒ error ; error, since c is immutable
(define d (vector-copy b 1 3))
d ⇒ #(8 2)

(vector-append vector ...)
Returns a newly allocated mutable vector whose elements are the concatenation of the elements of the given vectors.

(vector-append #(a b c) #(d e f)) ⇒ #(a b c d e f)

(vector-concatenate vector xs)
Returns a newly allocated mutable vector whose elements are the concatenation of the elements of the vectors in xs. xs is a proper list of vectors.

(vector-concatenate '(#(a b c) #(d) #(e f))) ⇒ #(a b c d e f)

(vector-map f vector1 vector2 ...)
Constructs a new mutable vector of the shortest size of the vector arguments vector1, vector2, etc. Each element at index i of the new vector is mapped from the old vectors by (f (vector-ref vector1 i) (vector-ref vector2 i) ...). The dynamic order of the application of f is unspecified.

(vector-map + #(1 2 3 4 5) #(10 20 30 40)) ⇒ #(11 22 33 44)

(vector-map/index f vector1 vector2 ...)
Constructs a new mutable vector of the shortest size of the vector arguments vector1, vector2, etc. Each element at index i of the new vector is mapped from the old vectors by (f i (vector-ref vector1 i) (vector-ref vector2 i) ...). The dynamic order of the application of f is unspecified.

(vector-map/index (lambda (i x y) (cons i (+ x y))) #(1 2 3) #(10 20 30)) ⇒ #((0 . 11) (1 . 22) (2 . 33))

30.3 Managing vector state

(vector-length vector)
Returns the number of elements in vector as an exact integer.

(vector-ref vector k)
The vector-ref procedure returns the contents of element k of vector. It is an error if k is not a valid index of vector.
(vector-ref '#(1 1 2 3 5 8 13 21) 5) ⇒ 8
(vector-ref '#(1 1 2 3 5 8 13 21) (exact (round (* 2 (acos -1))))) ⇒ 13

(vector-set! vector k obj)
The `vector-set!` procedure stores `obj` in element `k` of `vector`. It is an error if `k` is not a valid index of `vector`.

(let ((vec (vector 0 '(2 2 2 ) "Anna")))
  (vector-set! vec 1 "("Sue" "Sue")
  ⇒ '#(0 "Sue" "Sue") "Anna")
(vector-set! '#(0 1 2) 1 "doe")
⇒ error ;; constant/immutable vector

(vector-swap! vector j k)
The `vector-swap!` procedure swaps the element `j` of `vector` with the element `k` of `vector`.

30.4 Destructive vector operations

Procedures which operate only on a part of a vector specify the applicable range in terms of an index interval `[start; end]; i.e. the end index is always exclusive.

(vector-copy! to at from)
(vector-copy! to at from start)
(vector-copy! to at from start end)

Copies the elements of `vector` from between `start` and `end` to `vector to`, starting at `at`. The order in which elements are copied is unspecified, except that if the source and destination overlap, copying takes place as if the source is first copied into a temporary vector and then into the destination. `start` defaults to 0 and `end` defaults to the length of `vector`.

It is an error if `at` is less than zero or greater than the length of `to`. It is also an error if `(- (vector-length to) at)` is less than `(- end start)`.

(define a (vector 1 2 3 4 5))
(define b (vector 10 20 30 40 50)) (vector-copy! b 1 a 0 2)
b ⇒ #(10 1 2 40 50)

(vector-fill! vector fill)
(vector-fill! vector fill start)
(vector-fill! vector fill start end)

The `vector-fill!` procedure stores `fill` in the elements of `vector` between `start` and `end`. `start` defaults to 0 and `end` defaults to the length of `vector`.

(define a (vector 1 2 3 4 5))
(vector-fill! a 'smash 2 4)
a ⇒ #(1 2 smash smash 5)

(vector-reverse! vector)
(vector-reverse! vector start)
(vector-reverse! vector start end)

Procedure `vector-reverse!` destructively reverses the contents of `vector` between `start` and `end`. `start` defaults to 0 and `end` defaults to the length of `vector`.
(define a (vector 1 2 3 4 5))
(a ⇒ #(5 4 3 2 1))

(vector-sort! pred vector)
Procedure vector-sort! destructively sorts the elements of vector using the “less than” predicate pred.

(define a (vector 7 4 9 1 2 8 5))
(a ⇒ #(1 2 4 5 7 8 9))

(vector-map! f vector1 vector2 ...
Similar to vector-map which maps the various elements into a new vector via function f, procedure vector-map! destructively inserts the mapped elements into vector1. The dynamic order in which f gets applied to the elements is unspecified.

(define a (vector 1 2 3 4))
(vector-map! + a #(10 20 30))
(a ⇒ #(11 22 33 4))

(vector-map/index! f vector1 vector2 ...) Similar to vector-map/index which maps the various elements together with their index into a new vector via function f, procedure vector-map/index! destructively inserts the mapped elements into vector1. The dynamic order in which f gets applied to the elements is unspecified.

(define a (vector 1 2 3 4))
(vector-map/index! (lambda (i x y) (cons i (+ x y))) a #(10 20 30))
(a ⇒ #((0 . 11) (1 . 22) (2 . 33) 4))

30.5 Converting vectors

(vector->list vector)
(vector->list vector start)
(vector->list vector start end)
The vector->list procedure returns a newly allocated list of the objects contained in the elements of vector between start and end in the same order line in vector.

(vector->list 'dah dah dah) ⇒ (dah dah dah)
(vector->list 'dah dah dah) 1 2) ⇒ (dah)

(vector->string vector)
(vector->string vector start)
(vector->string vector start end)
The vector->string procedure returns a newly allocated string of the objects contained in the elements of vector between start and end. This procedure preserves the order of the characters. It is an error if any element of vector between start and end is not a character.
(vector->string #\1 #\2 #\3) ⇒ "123"
31 LispPad Speech

Library (lisppad speech) provides a speech synthesis API which parses text and converts it into audible speech. The conversion is based on factors like the language, the voice, and a range of parameters which are all aggregated by speaker objects.

31.1 Speech synthesis

(speak text)
(speak text speaker)

Speaks the given string text using with the speaker object providing all speech synthesis parameters. If speaker is not provided, the value of parameter object current-speaker is used.

(phonemes text)
(phonemes text speaker)

Converts the given natural language string text into a string of phonemes using the given speaker. If speaker is not provided, the value of parameter object current-speaker is used.

Speakers can be configured to speak phonemes instead of natural language via procedure speaker-interpret-phonemes!.

31.2 Speakers

A speaker is an object defining speech synthesis parameters. There is a current speaker which is used by default, unless a speaker is explicitly specified for the various procedures that require a speaker parameter.

A speaker object has the following components:
- an immutable voice,
- a mutable speaking rate,
- a mutable speaking volume,
- a flag determining whether the speaker interprets text or phonemes,
- a flag determining how numbers are interpreted, as well as
- a speaking pitch.

current-speaker

Defines the current speaker, which is used as a default by all functions for which the speaker argument is optional. If there is no current speaker, this parameter is set to #f.

(speaker? obj)

Returns #t if obj is a speaker object; otherwise #f is returned.

(make-speaker)
(make-speaker voice)
Returns a new speaker for the given voice. If voice is not provided, a default voice, specified at the operating system level, is being used. Speakers are stateful objects which can be configured with a number of procedures: set-speaker-rate!, set-speaker-volume!, set-speaker-interpret-phonemes!, set-speaker-interpret-numbers!, and set-speaker-pitch!.

(speaker-voice)
(speaker-voice speaker)

Returns the voice of speaker. If speaker is not provided, the parameter object current-speaker is used.

(speaker-rate)
(speaker-rate speaker)

Returns the speaking rate of speaker. If speaker is not provided, the parameter object current-speaker is used.

(set-speaker-rate! rate)
(set-speaker-rate! rate speaker)

Sets the speaking rate of speaker to number rate. If speaker is not provided, the parameter object current-speaker is used.

(speaker-volume)
(speaker-volume speaker)

Returns the volume of speaker as a flonum ranging from 0.0 to 1.0. If speaker is not provided, the parameter object current-speaker is used.

(set-speaker-volume! volume)
(set-speaker-volume! volume speaker)

Sets the volume of speaker to number volume which is a flonum between 0.0 and 1.0. If speaker is not provided, the parameter object current-speaker is used.

(speaker-interpret-phonemes)
(speaker-interpret-phonemes speaker)

Returns #t if speaker interprets phonemes instead of natural language text. If speaker is not provided, the parameter object current-speaker is used.

(set-speaker-interpret-phonemes! phoneme?)
(set-speaker-interpret-phonemes! phoneme? speaker)

If boolean argument phoneme? is #f, speaker is configured to interpret natural language. If phoneme? is set to any other value, the speaker is interpreting phonemes instead. If speaker is not provided, the parameter object current-speaker is used.

(speaker-interpret-numbers)
(speaker-interpret-numbers speaker)

Returns #t if speaker interprets numbers as a natural language speaker would do (“100” is spoken as “hundred”). If it returns #f, speaker decomposes numbers into a sequence of digits and speaks them individually (“100” is spoken as “one zero zero”). If speaker is not provided, the parameter object current-speaker is used.

(set-speaker-interpret-numbers! natural?)
(set-speaker-interpret-numbers! natural? speaker)

Sets the number interpretation of speaker to boolean natural?. If natural? is #t speaker will interpret numbers as a natural language speaker would do (“100” is spoken as “hundred”). If natural? is #f, speaker decomposes numbers into a sequence of digits and speaks them individually (“100” is spoken as “one zero zero”). If speaker is not provided, the parameter object current-speaker is used.
(speaker-pitch)
(speaker-pitch speaker)

Returns the pitch of speaker as a pair of two flonums: the car is the base of the pitch, and the cdr is the modulation of the pitch. If speaker is not provided, the parameter object current-speaker is used.

(set-speaker-pitch! pitch)
(set-speaker-pitch! pitch speaker)

Sets the pitch of speaker to the pair of flonums pitch whose car is the base of the pitch, and the cdr is the modulation of the pitch. If speaker is not provided, the parameter object current-speaker is used.

31.3 Voices

Voices are provided by the operating system and library (lispkit speech) does not have an explicit representation as objects. Symbols are used as identifiers for voices. For example, com.apple.speech.synthesis.voice.Alex refers to the default US voice.

A voice has the following characteristics:

- Name (string)
- Age (fixnum)
- Gender (male or female)
- Locale (symbol, e.g. en_US)

Library (lispkit system) provides means to handle locales, including language and country codes.

(voice)
(voice name)
(voice id)

Returns a symbol identifying the voice specified by the arguments of voice. If no argument is provided, an identifier for the default voice is returned. If a name string is provided, then an identifier for a voice whose name is name is returned, or #f if no such voice exists. If an id symbol is provided, then an identifier for a voice whose identifier matches id is returned, or #f if no such voice exists.

(available-voices)
(available-voices lang)
(available-voices lang gender)

Returns a list of symbols identifying voices matching the given language filter lang and gender filter gender. Both lang and gender are symbols. lang should either be a language or locale identifier. It can also be set to #f if only a gender filter is needed. gender should either be symbol male or female.

(available-voice? obj)

Returns #t if obj is a symbol identifying an available voice, otherwise #f is returned. This procedure fails if obj is neither a symbol nor the value #f.
(voice-name voice)
Returns the name of the voice identified by symbol voice.

(voice-age voice)
Returns the age of the voice identified by symbol voice.

(voice-gender voice)
Returns the gender of the voice identified by symbol voice.

(voice-locale voice)
Returns the locale of the voice identified by symbol voice.
32 LispPad System

Library (lisppad system) defines an API for scripting the LispPad user interface. This library is specific to LispPad and is not bundled with LispKit.

Library (lisppad system) provides functionality primarily for managing LispPad windows: new windows can be created, properties of existing windows can be changed, and the content of existing windows can be accessed and modified. There is also support for making use of simple dialogs, e.g. for displaying messages, asking the user to make a choice, or for letting the user choose a file or directory in a load or save panel.

32.1 Windows

(lisppad system) does not provide a data structure for modeling references to LispPad windows. Instead, it uses integer ids as references. Two different types of windows can be managed:

- **Edit windows** are used for editing text, and
- **Graphics windows** are used for displaying drawings created via library (lispkit draw).

Other types of windows are currently not accessible via library (lisppad system).

(open-document path) procedure

Opens a document stored in a file at path path. Only documents that LispPad is able to open are supported.

(edit-windows) procedure

Returns an association list containing all open edit windows. Each open window has an entry of the form (window id . window title). For example, the result of invoking (edit-windows) could look like this: `(((106102873393392 . "LispKit Libraries.md") (106377751319520 . "Untitled"))).`

(graphics-windows) procedure

Returns an association list containing all open graphics windows. Each open window has an entry of the form (window id . window title). For example, the result of invoking (graphics-windows) could look like this: `(((106102873393789 . "My Drawing") (106377751899571 . "Untitled Drawing"))).`

(window-position win) procedure

Returns the position of the window with window id win. The position of a window is the upper left corner of its title bar represented as a point.

(set-window-position! win pos) procedure

Sets the position of the window with window id win to point pos. The position of a window is the upper left corner of its title bar.

(window-size win) procedure

Returns the size of the window with window id win. The size of a window consists of its width and height represented as a size.

(set-window-size! win size) procedure

Sets the size of the window with window id win to size size. The size of a window consists of its width and height.
(close-window win)
Closes the window with window id win.

32.2 Edit Windows

(make-edit-window str pos size)
Creates a new edit window containing str as its textual content. The window's initial position is pos and its size is size.

(edit-window-text win)
Returns the textual content of the edit window with the given window id win.

(insert-edit-window-text win str)
(insert-edit-window-text win str start)
(insert-edit-window-text win str start end)
Inserts a string str replacing text between start and end for the edit window with window id win. If start is not provided, start is considered to be 0 (i.e. the text is inserted at the beginning). If end is not provided, it is considered to be the length of the text contained in the edit window win.

(edit-window-text-length win)
Returns the length of the text contained in the edit window with window id win.

32.3 Graphics Windows

(make-graphics-window drawing dsize)
(make-graphics-window drawing dsize title)
(make-graphics-window drawing dsize title pos)
(make-graphics-window drawing dsize title pos size)
Creates a new graphics window for drawing drawing. dsize refers to the size of the drawing. title is the window title of the new window, pos is its initial position, and size corresponds to the initial size of the graphics window.

(use-graphics-window drawing dsize title)
(use-graphics-window drawing dsize title pos)
(use-graphics-window drawing dsize title pos size)
(use-graphics-window drawing dsize title pos size ignore)
This is almost equivalent to function make-graphics-window. The main difference consists in use-graphics-window reusing an existing graphics window if there is one open with the given title. If there is no window whose title matches title, a new graphics window will be created. If a window exists already and boolean argument ignore is set to #t, the existing window's position and size will not be updated.

(update-graphics-window win)
This function forces the graphics window with window id win to redraw its content. Currently, graphics windows are only guaranteed to redraw automatically after executing a command in the session window which was used to create the drawing object.

(graphics-window-drawing win)
Returns the drawing associated with the graphics window with window id win.

(set-graphics-window-drawing! win drawing)
Sets the drawing associated with the graphics window with window id win to drawing.
(graphics-window-label win)
Each graphics window has a label at the bottom of the window. This label can be arbitrarily modified, and e.g. used as a caption. graphics-window-label returns the label of the graphics window with window id win.

(set-graphics-window-label! win str)
Each graphics window has a label at the bottom of the window. The label of graphics window win can be set via function set-graphics-window-label! to string str.

(drawing-size win)
Returns the size of the drawing associated with graphics window win. Please note that this is not the window size of win.

(set-drawing-size! win size)
Sets the size of the drawing associated with graphics window win to size. Please note that this is not setting the window size of win.

32.4 Utilities

(screen-size)
(screen-size win)
Returns the screen size of the screen on which window win is displayed. If argument win is omitted, function screen-size will return the size of the main screen.

(show-message-panel title)
(show-message-panel title str)
(show-message-panel title str button)
Shows a message panel within the current session window. title refers to the panel title, str is the message to be displayed, and button is the label of the confirmation button.

(show-choice-panel title str)
(show-choice-panel title str yes)
(show-choice-panel title str yes no)
Shows a choice panel within the current session window. title refers to the panel title, str is the question to be asked, and yes and no refer to the two labels of the buttons for users to choose. The function returns #t if the user clicked on the “yes button”.

(show-load-panel prompt)
(show-load-panel prompt folders)
(show-load-panel prompt folders filetypes)
Displays a load panel within the current session window together with the given prompt message. folders is a boolean argument; it should be set to #t if the user is required to select a folder. filetypes is a list of suffixes of selectable file types.

(show-save-panel prompt)
(show-save-panel prompt filetypes)
Displays a save panel within the current session window together with the given prompt message. filetypes is a list of suffixes of selectable file types.

(session-id)
Returns a unique fixnum (within LispPad) identifying the session.
(session-name)
Returns the name of the LispPad session which executes this function.

(dark-mode?)
Return #t if the session window of the LispPad session which executes this function is rendered in dark mode; returns #f otherwise.
This is a library implementing a simple graphics window for displaying turtle graphics. The library supports one graphics window per LispPad session which gets initialized by invoking `init-turtle`. `init-turtle` will create a new turtle and display its drawing on a graphics window. If there is already an existing graphics window with the given title, it will be reused.

Once `init-turtle` was called, the following functions can be used to move the turtle across the plane:

- `(pen-up)`: Lifts the turtle
- `(pen-down)`: Drops the turtle
- `(pen-color color)`: Sets the current color of the turtle
- `(pen-size size)`: Sets the size of the turtle pen
- `(home)`: Moves the turtle back to the origin
- `(move x y)`: Moves the turtle to position `(x, y)`
- `(heading angle)`: Sets the angle of the turtle (in radians)
- `(turn angle)`: Turns the turtle by the given angle (in radians)
- `(left angle)`: Turn left by the given angle (in radians)
- `(right angle)`: Turn right by the given angle (in radians)
- `(forward distance)`: Moves forward by `distance` units drawing a line if the pen is down
- `(backward distance)`: Moves backward by `distance` units drawing a line if the pen is down

This library defines a simplified, interactive version of the API provided by library `(lispkit draw turtle)`.

```lisp
(init-turtle) ; procedure
(init-turtle scale) ; procedure
(init-turtle scale title) ; procedure
```

Initializes a new turtle and displays its drawing in a graphics window. `init-turtle` gets two optional arguments: `scale` and `title`. `scale` is a scaling factor which determines the size of the turtle drawing. `title` is a string that defines the window name of the turtle graphics. It also acts as the identify of the turtle graphics window; i.e. it won't be possible to have two sessions with the same name but a different graphics window.

```lisp
(close-turtle-window) ; procedure
```

Closes the turtle window and resets the turtle library.

```lisp
(turtle-drawing) ; procedure
```

Returns the drawing associated with the current turtle.

```lisp
(pen-up) ; procedure
```

Lifts the turtle from the plane. Subsequent `forward` and `backward` operations don't lead to lines being drawn. Only the current coordinates are getting updated.

```lisp
(pen-down) ; procedure
```

Drops the turtle onto the plane. Subsequent `forward` and `backward` operations will lead to lines being drawn.

```lisp
(pen-color color) ; procedure
```

Sets the drawing color of the turtle to `color`. `color` is a color object as defined by library `(lispkit draw)`.
**pen-size** size
Sets the pen size of the turtle to size. The pen size corresponds to the width of lines drawn by **forward** and **backward**.

**home**
Moves the turtle to its home position.

**move** x y
Moves the turtle to the position described by the coordinates x and y.

**heading** angle
Sets the heading of the turtle to angle. angle is expressed in terms of degrees.

**turn** angle
Adjusts the heading of the turtle by angle degrees.

**right** angle
Adjusts the heading of the turtle by angle degrees.

**left** angle
Adjusts the heading of the turtle by -angle degrees.

**forward** distance
Moves the turtle forward by distance units drawing a line if the pen is down.

**backward** distance
Moves the turtle backward by distance units drawing a line if the pen is down.
34  SRFI Libraries

LispPad supports a broad range of libraries standardized and published via the SRFI process. The following libraries come pre-packaged with LispPad:

- SRFI 1: List Library
- SRFI 2: AND-LET* - an AND with local bindings, a guarded LET* special form
- SRFI 6: Basic String Ports
- SRFI 8: receive - Binding to multiple values
- SRFI 11: Syntax for receiving multiple values
- SRFI 14: Character-set library
- SRFI 16: Syntax for procedures of variable arity
- SRFI 17: Generalized set!
- SRFI 19: Time Data Types and Procedures
- SRFI 23: Error reporting mechanism
- SRFI 26: Notation for Specializing Parameters without Currying
- SRFI 27: Sources of Random Bits
- SRFI 28: Basic Format Strings
- SRFI 31: A special form rec for recursive evaluation
- SRFI 33: Integer Bitwise-operation Library
- SRFI 34: Exception Handling for Programs
- SRFI 35: Conditions
- SRFI 39: Parameter objects
- SRFI 41: Streams
- SRFI 46: Basic Syntax-rules Extensions
- SRFI 48: Intermediate Format Strings
- SRFI 51: Handling rest list
- SRFI 54: Formatting
- SRFI 55: require-extension
- SRFI 63: Homogeneous and Heterogeneous Arrays
- SRFI 64: A Scheme API for test suites
- SRFI 69: Basic hash tables
- SRFI 87: => in case clauses
- SRFI 95: Sorting and Merging
- SRFI 98: An interface to access environment variables
- SRFI 101: Purely Functional Random-Access Pairs and Lists
- SRFI 111: Boxes
- SRFI 112: Environment inquiry
- SRFI 113: Sets and bags
- SRFI 121: Generators
- SRFI 125: Intermediate hash tables
- SRFI 128: Comparators
- SRFI 129: Titlecase procedures
- SRFI 132: Sort Libraries
- SRFI 133: Vector Library
- SRFI 134: Immutable Deques
• SRFI 135: Immutable Texts
• SRFI 137: Minimal Unique Types
• SRFI 142: Bitwise Operations
• SRFI 145: Assumptions
• SRFI 146: Mappings
• SRFI 151: Bitwise Operations
• SRFI 152: String Library
• SRFI 158: Generators and Accumulators
• SRFI 161: Unifiable Boxes
• SRFI 162: Comparators sublibrary
• SRFI 165: The Environment Monad
• SRFI 167: Ordered Key Value Store
• SRFI 173: Hooks
• SRFI 174: POSIX Timespecs
• SRFI 175: ASCII Character Library
• SRFI 177: Portable keyword arguments