# Bangra Documentation

***Release 0.5***

**Leonard Ritter**

February 28, 2017

Contents

This manual aims to provide an introduction to programming with Bangra, a description of Bangra's architecture and its two language modes, and a reference of all special forms, macros, functions and modules provided in the language repository.

Contents:

CHAPTER 1

# About Bangra

Bangra is a minimalist, interpreted, mixed functional/imperative general purpose programming language inspired by Scheme, Python and C.

The core interpreter is written in under 6k lines of C++ code, and exports the fewest features necessary to allow the language to expand itself to a comfortable level. These features are:

- A polymorphic value of type `Any` implemented as a 16-byte sized fat pointer.
- A `Symbol` type that maps strings to cached identifiers.
- An **annotation system** that permits to attach a file name, line number and column index in the form of an `Anchor` to any value.
- A flat, C-compatible **type system** that implements overloadable operations for booleans, all signed and unsigned integer types, floats, doubles, arrays, vectors, tuples, pointers, structs, unions, enums, function pointers, slices, tables, lists and strings.
- A **lexer and parser** for Bangra-style symbolic expressions, which use whitespace indentation to delimit code blocks, but also support a restricted form of traditional Lisp/Scheme syntax. The parser outputs a tree of cons cells annotated with file, line number and column.
- A **foreign function interface** (FFI) based on libffi that permits calling into C libraries without requiring any conversions, thanks to the C-compatible type system.
- A **clang-based C importer** that generates type libraries from include files on the fly.
- A **fully embedded** build of **LLVM** that can be accessed via FFI.
- A **programmable macro preprocessor** that expands symbolic expressions and requires only two builtin macros: `fn/cc` and `syntax-extend`. These are sufficient to bootstrap the rest of the language.

  The macro preprocessor supports hooks to preprocessing arbitrary lists and symbols. This feature is used to support special syntax without having to expand the parser.
- A **translator** that generates flow nodes from fully expanded symbolic expressions.

  Flow nodes are functions stored in **control flow form** (CFF), a novel intermediate format which requires continuations as its only primitive and eases specialization and optimization considerably. For more information see A Graph-Based Higher-Order Intermediate Representation by Leissa et al.
- A small `eval-apply` style **interpreter loop** that executes flow nodes.
- A small set of built-in functions exported as globals.
- A loader that permits attaching a payload script to the main executable.
- Initialization routines for the root environment.

With these features, the runtime environment is loaded which bootstraps the remaining elements of the language in several stages:

- Various utility functions, types and macros.
- A global hook to recognize and translate **infix notation**, **dot notation** and **symbol prefixes**.
- A read-eval-print (REPL) **console** for casual use.

As Bangra is in alpha stage, some essential features are still missing. These features will be added in future releases:

- Better error reporting.
- An on-demand **specializer** that translates flow nodes to fast machine code using LLVM. The specializer will feature closure elimination and memory tracking.
- A **garbage collection** mechanism for the interpreter. Right now Bangra doesn't free any memory at all.
- Debugging support for gdb.

As designer of Bangra, I hope that, over time, you will find Bangra use- and delightful, instructive, sometimes even entertaining, as you experiment with, toy around with, learn about, and engineer new programs, utilities, languages, infrastructures, games and toys written with and for Bangra.

CHAPTER 2

# Getting Started

## Installation

You can either download a binary distribution of Bangra from the website or build Bangra from source.

How to build Bangra on Linux:

- You need build-essentials, clang 3.9, libclang 3.9 and LLVM 3.9 installed
- put `clang++` and `llvm-config` in your path OR extract the clang distro into the repo folder and rename it to `clang`.
- You also need the latest source distribution of libffi.
- build libffi using `./configure --enable-shared=no --enable-static=yes && make` and softlink the generated build folder (e.g. `x86_64-unknown-linux-gnu`) as `libffi` in the repo folder.
- execute `./makebangra`

How to build Bangra on Windows:

- Install MSYS2 and install both llvm and clang 3.9 for `x86_64`. The packages are named `mingw64/mingw-w64-x86_64-llvm` and `mingw64/mingw-w64-x86_64-clang`.
- You also need to install the `mingw64/mingw-w64-x86_64-libffi` package.
- put `clang++` in your path OR make sure msys2 resides in `C:\msys64` OR edit `makebangra.bat` and change the path accordingly.
- copy `libstdc++-6.dll`, `libgcc_s_seh-1.dll`, `libwinpthread-1.dll` and `libffi-6.dll` from the msys2 installation into the repo folder. `bangra.exe` will depend on them.
- run `makebangra.bat`

There should now be a `bangra` executable in your root folder.

You can verify that everything works by running:

```
bangra testing/test_all.b
```

## Running

Bangra has a built-in console that can be launched from the command-line by passing no arguments to the `bangra` interpreter:

```
bangra
```

To execute a Bangra program, pass the source file as first argument to the `bangra` interpreter:

```
bangra <path-to-file.b>
```

## Hello World

A simple "Hello World" program in Bangra looks as follows:

```
print "Hello world!"
```

*Note that in order to be valid, a Bangra program must not contain any tabs, and each sub-block must be indented by four spaces.*

TODO

CHAPTER 3

# The Interchange Format

*Please excuse the at times misleading syntax highlighting, there isn't a working syntax highlighter available yet, so I had to pick the next best one available.*

This chapter outlines the syntax of Bangra source code at the data interchange format level from the perspective of nesting arbitrary lists. No programming happens at this stage, it's just data formatting, which means that the format can also act as a replacement for other interchange formats like XML and JSON.

## Element Types

The Bangra parser has been designed for minimalism and recognizes only five element types:

- Symbols
- Strings
- Integers
- Reals
- Lists

All further understanding of types is done with additional parsing at later stages that depend on language context and evaluation scope.

### Comments

Line comments are skipped by the lexer. A comment token is recognized by its `#` prefix and scanned until the next newline character. Here are some examples for valid comments:

```
#mostly harmless
# 123
#"test"
#(an unresolved tension
## #(another comment)# ##
```

### Strings

Strings describe sequences of unsigned 8-bit characters in the range of 0-255 and are stored as zero-terminated string arrays. A string begins and ends with `"` (double quotes). The `\` escape character can be used to include quotes in a string and describe unprintable control characters such as `\n` (return) and `\t` (tab). The parser parses strings as-is, so

UTF-8 encoded strings will be copied over verbatim, and return characters will be preserved, allowing strings to span multiple lines.

Here are some examples for valid strings:

```
"single-line string, 'double' quotations"
"multi-
line
string"
"return: \n, tab: \t, backslash: \\, double quote: \"."
```

## Symbols

Like strings, a symbol describes a sequence of 8-bit characters, but uses whitespace as delimiter and is often understood on a language level as a label or bindable name. Symbols may contain any character from the UTF-8 character set except whitespace and any character from the set `#;()[]{},`. A symbol always terminates when one of these characters is encountered. Any symbol that parses as a number is also excluded. Like strings, symbols are stored as zero-terminated string arrays and share almost all string API functions.

Here are some examples for valid symbols:

```
# classic underscore notation
some_identifier _some_identifier
# hyphenated
some-identifier
# mixed case
SomeIdentifier
# fantasy operators
&+ >~ >>= and= str+str
# numbered
_42 =303
```

## Numbers

Numbers come in two forms: integers and reals. The parser understands integers in the range -(2^63) to 2^64-1 and records them as signed 64-bit values. Reals are floating point numbers parsed and stored as IEEE 754 binary64 values, which guarantees integer precision up to 52-bit.

Here are some examples for valid numbers:

```
# positive and negative integers
0 +23 42 -303 12 -1 -0x20
# positive and negative reals
0.0 1.0 3.14159 -2.0 0.000003 0xa400.a400
# reals in scientific notation
1.234e+24 -1e-12
# special reals
+inf -inf nan
```

## Lists

Lists are the only nesting type, scoped by the bracket pairs `()`, `[]` and `{}`. A list is stored as a pointer to its first element, and elements are chained in a single link that terminates with a null pointer. Empty lists are stored as null pointers.

Lists can be empty or contain an unlimited number of elements, separated by whitespace. They typically describe expressions in Bangra.

Here are some examples for valid lists:

```
# empty list
()
# list containing a symbol, a string, an integer, a real, and an empty list
(print "hello world" 303 3.14 ())
# three nesting lists
(((())))
```

## Naked & Coated Lists

Every Bangra source file is parsed as a top level list, where the head element is the language name.

The classic notation (what we will call *coated notation*) uses the syntax known to Lisp and Scheme users as *restricted* S-expressions:

```
# there must not be any tokens outside the parentheses guarding the
# top level list.

# nested lists as nested expressions:
(print (.. "Hello" "World") 303)
```

As a modern alternative, Bangra offers a *naked notation* where the scope of lists is implicitly balanced by indentation, an approach used by Python, Haskell, YAML, Sass and many other languages.

This source parses as the same list in the coated example:

```
# nesting is implied by indentation.
# a sub paragraph continues the list.
print
    # nested arguments must be indendent by four spaces.
    # tabs are not permitted.

    # multiple elements on a single line without sub-paragraph are wrapped
    # in a list.
    .. "Hello" "World"

    # single arguments are appended to the parent list
    303
```

### Mixing Modes

Naked lists can contain coated lists, and coated lists can contain naked lists:

```
# compute the value of (1 + 2 + (3 * 4)) and print the result
(print
    (+ 1 2
        (3 * 4)))

# the same list in naked notation.
# indented lists are spliced into the parent list:
print
    + 1 2
```

```
        3 * 4

# any part of a naked list can be coated
print
    + 1 2 (3 * 4)

# and a coated list can contain naked parts.
# the escape character \ enters naked mode at its indentation level.
print
    (+ 1 2
        \ 3 * 4) # parsed as (+ 1 2 (3 * 4))
```

Because it is more convenient for users without specialized editors to write in naked notation, and balancing parentheses can be challenging for beginners, the author suggests to use coated notation sparingly and in good taste. Purists and Scheme enthusiasts may however prefer to keep only the top level naked, as in most Lisp-like languages, and work exclusively with coated lists otherwise.

Therefore Bangra's reference documentation describes all available symbols in coated notation, while code examples make ample use of naked notation.

## Block Comments

In addition to `# single line comments`, Bangra recognizes and strips a special kind of multi-line comment.

A list beginning with a symbol that starts with a `///` (triple slash) describes a block comment. Block comments have to remain syntactically consistent. Here are some examples for valid block comments:

```
# block comments in coated notation
(///this comment
    will be removed)
(///
    # commenting out whole sections
    (function ()
        true)
    (function ()
        false))

# block comments in naked notation
///this comment
    will be removed

///
    # commenting out whole sections
    function ()
        true
    function ()
        false
```

## List Separators

Both naked and coated lists support a special control character, the list separator `;` (semicolon). Known as statement separator in other languages, it wraps all elements from the previous `;` or beginning of list in a new list, and allows to reduce the amount of required parentheses in complex trees.

Here are some examples:

```
# in coated notation
(print a; print (a;b;); print c;)
# parses as
((print a) (print ((a) (b))) (print c))

# in naked notation
print a;
    print b; print c;
        print d;
# parses as
((print a) (print b) ((print c) (print d)))
```

There's a caveat though: if trailing elements aren't terminated with `;`, they're not going to be wrapped:

```
# in coated notation
print a; print (a;b;); print c
# parses as
((print a) (print ((a) (b))) print c)
```

## Pitfalls of Naked Notation

As naked notation giveth the user the freedom to care less about parentheses, it also taketh away. In the following section we will discuss the few small difficulties that can arise and how to solve them efficiently.

### Single Elements

Special care must be taken when single elements are defined, which are not to be wrapped in lists.

Here is a coated list describing an expression printing the number 42:

```
(print 42)
```

The naked equivalent declares two elements in a single line, which are implicitly wrapped in a single list:

```
print 42
```

A single element on its own line is always taken as-is:

```
print           # (print
    42          #       42)
```

What happens when we want to call functions without arguments? Consider this example:

```
# a coated list in a new scope, describing an expression
# printing a new line, followed by the number 42
(do
    (print)
    (print 42))
```

A naive naked transcription would probably look like this, and be very wrong:

```
do
    # suprisingly, the new line is never printed, why?
    print
    print 42
```

Translating the naked list back to coated reveals what is going on:

```
(do
    # interpreted as a symbol, not as an expression
    print
    (print 42))
```

The straightforward fix to this problem would be to explicitly wrap the single element in parentheses:

```
do
    (print)
    print 42
```

Nudists might however want to use the list separator `;` (semicolon) which forces the line to be wrapped in a list and therefore has the same effect:

```
do
    print;
    print 42
```

## Wrap-Around Lines

There are often situations when a high number of elements in a list interferes with best practices of formatting source code and exceeds the line column limit (typically 80 or 100).

In coated lists, the problem is easily corrected:

```
# import many symbols from an external module into the active namespace
(import-from "OpenGL"
    glBindBuffer GL_UNIFORM_BUFFER glClear GL_COLOR_BUFFER_BIT
    GL_STENCIL_BUFFER_BIT GL_DEPTH_BUFFER_BIT glViewport glUseProgram
    glDrawArrays glEnable glDisable GL_TRIANGLE_STRIP)
```

The naked approach interprets each new line as a nested list:

```
# produces runtime errors
import-from "OpenGL"
    glBindBuffer GL_UNIFORM_BUFFER glClear GL_COLOR_BUFFER_BIT
    GL_STENCIL_BUFFER_BIT GL_DEPTH_BUFFER_BIT glViewport glUseProgram
    glDrawArrays glEnable glDisable GL_TRIANGLE_STRIP

# coated equivalent of the term above; each line is interpreted
# as a function call and fails.
(import-from "OpenGL"
    (glBindBuffer GL_UNIFORM_BUFFER glClear GL_COLOR_BUFFER_BIT)
    (GL_STENCIL_BUFFER_BIT GL_DEPTH_BUFFER_BIT glViewport glUseProgram)
    (glDrawArrays glEnable glDisable GL_TRIANGLE_STRIP))
```

It comes easy to just fix this issue by putting each element on a separate line, which is not the worst solution:

```
# correct solution using single element lines
import-from "OpenGL"
    glBindBuffer
    GL_UNIFORM_BUFFER
    glClear
    GL_COLOR_BUFFER_BIT
    GL_STENCIL_BUFFER_BIT
    GL_DEPTH_BUFFER_BIT
    glViewport
```

```
    glUseProgram
    glDrawArrays
    # comments should go on a separate line
    glEnable
    glDisable
    GL_TRIANGLE_STRIP
```

A terse approach would be to make use of the \ (splice-line) control character, which is only available in naked notation and splices the line starting at the next token into the active list:

```
# correct solution using splice-line, postfix style
import-from "OpenGL" \
    glBindBuffer GL_UNIFORM_BUFFER glClear GL_COLOR_BUFFER_BIT \
    GL_STENCIL_BUFFER_BIT GL_DEPTH_BUFFER_BIT glViewport glUseProgram \
    glDrawArrays glEnable glDisable GL_TRIANGLE_STRIP
```

Unlike in other languages, \ splices at the token level rather than the character level, and can therefore also be placed at the beginning of nested lines, where the parent is still the active list:

```
# correct solution using splice-line, prefix style
import-from "OpenGL"
    \ glBindBuffer GL_UNIFORM_BUFFER glClear GL_COLOR_BUFFER_BIT
    \ GL_STENCIL_BUFFER_BIT GL_DEPTH_BUFFER_BIT glViewport glUseProgram
    \ glDrawArrays glEnable glDisable GL_TRIANGLE_STRIP
```

## Tail Splicing

While naked notation is ideal for writing nested lists that accumulate at the tail:

```
# coated
(a b c
    (d e f
        (g h i))
    (j k l))

# naked
a b c
    d e f
        g h i
    j k l
```

...there are complications when additional elements need to be spliced back into the parent list:

```
(a b c
    (d e f
        (g h i))
    j k l)
```

A simple but valid approach would be to make use of the single-element rule again and put each tail element on a separate line:

```
a b c
    d e f
        g h i
    j
    k
    l
```

But we can also reuse the splice-line control character to this end:

```
a b c
    d e f
        g h i
    \ j k l
```

## Left-Hand Nesting

When using infix notation, conditional blocks or functions producing functions, lists occur that nest at the head level rather than the tail:

```
((((a b)
    c d)
        e f)
            g h)
```

While this list tree is easy to describe in coated notation, it can not be described in pure naked notation. Though these situations seldom occur, a mix of multiple techniques may act as a compromise:

```
# equivalent structure
(a b; c d) e f;
    \ g h
```

A more complex tree which also requires splicing elements back into the parent list can be realized with the same combo of line comment and splice-line control character:

```
# coated
(a
    ((b
        (c d)) e)
    f g
    (h i))

# naked
a
    b (c d);
        e
    \ f g
    h i
```

While this example demonstrates the versatile usefulness of splice-line and list separator, less seasoned users may prefer to express similar trees in partial coated notation instead.

As so often, the best format is the one that fits the context.

CHAPTER 4

# Using Bangra

This chapter will provide an overview and usage information for Bangra in a future release.

## Functions

### Variable Arguments

## Bindings

## Blocks

## Conditionals

## Loops

## Iterators

## Stream Processing

## Modules

## Scopes

## Working with Numbers

## Working with Strings

## Working with Tables

## Working with Lists

## Infix Notation

## Dot Notation

## Number Formats

## Splicing

## Exception Handling

## Types

## Classes

CHAPTER 5

# C API Reference

The Bangra C API can be used from within Bangra programs and as a C library, and is defined in `bangra.h`.

This chapter will provide usage and reference information in a future release.

CHAPTER 6

# Bangra Language Reference

## Built-in Constants

**define** `none`

**define** `true`

**define** `false`

## Runtime Constants

**define** `empty-list`

**define** `empty-tuple`

## Built-in Types

**type** `closure`

**type** `flow`

**type** `frame`

**type** `half`

**type** `list`

**type** `parameter`

**type** `rawstring`

**type** `ssize_t`

**type** `string`

**type** `table`

**type** `usize_t`

**type** `void`

## Runtime Types

**type** `bool`

**type** `double`

**type** `float`

**type** `int`

**type** `int8`

**type** `int16`

**type** `int32`

**type** `int64`

**type** `real16`

**type** `real32`

**type** `real64`

**type** `size_t`

**type** `uint`

**type** `uint8`

**type** `uint16`

**type** `uint32`

**type** `uint64`

## Built-in Supertypes

**type** `array`

**type** `cfunction`

**type** `enum`

**type** `integer`

**type** `pointer`

**type** `qualifier`

**type** `real`

**type** `tag`

**type** `tuple`

**type** `struct`

**type** `symbol`

**type** `vector`

## Runtime Supertypes

**type** `iterator`

## Built-in Type Factories

**type-factory** ( **`array`** *type* , *count* )

**type-factory** ( **`cfunction`** *return-type* , *parameter-tuple-type* , *varargs?* )

**type-factory** ( **`integer`** *bit-width* , *signed?* )

**type-factory** ( **`pointer`** *type* )

**type-factory** ( **`real`** *bit-width* )

**type-factory** ( **`struct`** *name-symbol* )

**type-factory** ( **`symbol`** *name-string* )

**type-factory** ( **`tag`** *name-symbol* )

**type-factory** ( **`tuple`** [ *element-type* , ... ])

**type-factory** ( **`vector`** *type* , *count* )

## Builtin-in Special Forms

**special** ( **`call`** *callable* [, *expression* , ... ])

**special** ( **`cc/call`** *return-callable* , *callable* [, *expression* , ... ])

**special** ( **`do`** [ *expression* , ... ], *return-expression* )

**special** ( **`form:fn/cc`** [ *name* ]) [ *parameter-name* , ... ]
[, *expression* , ... ], *return-expression*

**special** ( **`splice`** *expression* )

## Built-in Macros

**macro** ( **`fn/cc`** [ *name* ]) [ *parameter-name* , ... ]
[, *expression* , ... ], *return-expression*

**macro** ( **`syntax-extend`** [ *name* ]) *return-callable* , *scope-table*
[, *expression* , ... ], *scope-expression*

## Runtime Macros

**macro** ( **`::@`** *expression* )

**macro** ( **`::*`** *expression* )

**macro** ( **`:`** *name* [, *value* ])

**macro** ( **.** *value* , *name* )

**macro** ( **?** *condition-bool* , *then-expression* , *else-expression* )

**macro** ( **and** *first-expression* , *second-expression* , ... )

**macro** ( **assert** *expression* [, *error-message* ])

**macro** ( **define** *name* , *compile-time-expression* , ... )

**macro** ( **dump-syntax** *expression* [, ... ])

**macro** ( **fn** [ *name* ]) [ *parameter-name* , ... ]
[, *expression* , ... ], *return-expression*

**macro** ( **if** *condition-bool* , ... , *expression* )

**macro** ( **elseif** *condition-bool* , ... , *expression* )

**macro** ( **else** *expression* , ... )

**macro** ( **for** *name* , ... **in** , *iterable* [] **with** *name* **=** , *value* , ... , ... , *body-expression* , ... )

**macro** ( **else** *expression* , ... )

**macro** ( **let** *name* , ... **=** , *expression* , ... )

**macro** ( **let** ) *name* , ... **=** , *expression* , ...
, ...

**macro** ( **loop** ) [ *name* , ... ]
, | **with** *name* **=** , *value* , ... , ... , *body-expression* , ...

**macro** ( **max** *first-expression* , *second-expression* , ... )

**macro** ( **min** *first-expression* , *second-expression* , ... )

**macro** ( **or** *first-expression* , *second-expression* , ... )

**macro** ( **qquote** *value* [, *value* , ... ])

**macro** ( **quote** *value* [, *value* , ... ])

**macro** ( **set!** *parameter* , *value* )

**macro** ( **try** *expression* , ... )

**macro** ( **except** ) *parameter*
, *expression* , ...

**macro** ( **xlet** *name* **=** , *expression* , ... )

**macro** ( **xlet** ) *name* **=** , *expression* , ...
, ...

## Built-in Functions

**function** ( **bitcast** *type* , *value* )

**function** ( **block-scope-macro** *closure* )

**function** ( **branch** *condition-bool* , *true-continuation* , *false-continuation* )

**function** ( **cons** *head* [, *list* , ... ])

**function** ( **countof** *container-value* )

**function** ( **cstr** *value-rawstring* )

**function** ( **dump** *expression* )

**function** ( **element-type** *type* )

**function** ( **error** *message-string* )

**function** ( **escape** *expression* )

**function** ( **eval** *expression* [, *globals-table* ])

**function** ( **exit** [ *code* ])

**function** ( **expand** *expression-block-list* , *scope-table* )

**function** ( **external** *name-symbol* , *cfunction-type* )

**function** ( **get-exception-handler** )

**function** ( **globals** )

**function** ( **import-c** *path-string* , *option-tuple* )

**function** ( **list-load** *path-string* )

**function** ( **list-parse** *expression-string* )

**function** ( **next-key** *table* , *key* )

**function** ( **print** [ *expression* , ... ])

**function** ( **prompt** *prompt-string* [, *prepend-string* ])

**function** ( **repr** *expression* )

**function** ( **raise** *error-value* )

**function** ( **set-exception-handler!** *closure* )

**function** ( **set-globals!** *globals-table* )

**function** ( **set-key!** *table* , *key* , *value* )

**function** ( **structof** [ *key-value-tuple* , ... ])

**function** ( **tableof** [ *key-value-tuple* , ... ])

**function** ( **tupleof** [ *expression* , ... ])

**function** ( **typeof** *expression* )

**function** ( **va-arg** *index* , *vararg-parameter...* )

**function** ( **va-countof** *vararg-parameter...* )

## Runtime Functions

**function** ( **block-macro** *function* )

**function** ( **disqualify** *tag-type* , *value* )

**function** ( **empty?** *countable-value* )

**function** ( **enumerate** *iterable* [, *from* [, *step* ] ])

**function** ( **iter** *iterable-value* )

**function** ( **iterator?** *value* )

**function** ( **key?** *table* , *key* )

**function** ( **list-atom?** *value* )

**function** ( **list-head?** *list* , *symbol* )

**function** ( **list?** *value* )

**function** ( **load** *table* , *key* )

**function** ( **macro** *function* )

**function** ( **none?** *value* )

**function** ( **qualify** *tag-type* , *value* )

**function** ( **range** *count* )

**function** ( **range** *start* , *stop* [, *step* ])

**function** ( **require** *module-name-string* )

**function** ( **symbol?** *value* )

**function** ( **xpcall** *callable* , *exception-callable* )

**function** ( **zip** *left-iterable* , *right-iterable* )

## Built-in Operators

**function** ( **==** *first-value* , *second-value* )

**function** ( **!=** *first-value* , *second-value* )

**function** ( **>** *first-value* , *second-value* )

**function** ( **>=** *first-value* , *second-value* )

**function** ( **<** *first-value* , *second-value* )

**function** ( **<=** *first-value* , *second-value* )

**function** ( **..** *first-value* , *second-value* )

**function** ( **+** *first-value* , *second-value* [, *...* ])

**function** ( **-** *first-value* [, *second-value* ])

**function** ( ***** *first-value* , *second-value* [, *...* ])

**function** ( ****** *base-number* , *exponent-number* )

**function** ( **/** *first-value* [, *second-value* ])

**function** ( **//** *first-value* , *second-value* )

**function** ( **%** *first-value* , *second-value* )

**function** ( **|** *first-value* , *second-value* [, *...* ])

**function** ( **&** *first-value* , *second-value* )

**function** ( **^** *first-value* , *second-value* )

**function** ( **~** *expression* )

**function** ( **<<** *value* , *offset* )

**function** ( **>>** *value* , *offset* )

**function** ( **@** *container-value* , *index-value* , | , *name-symbol* )

**function** ( **hash** *expression* )

**function** ( **not** *expression* )

**function** ( **slice** *expression* , *start-index* [, *end-index* ])

**function** ( **string** *expression* )

## Runtime Operators

**function** ( **@** *container-value* , *index-value* , | , *name-symbol* , ... )

CHAPTER 7

# Indices and tables

- genindex
- modindex
- search

## Symbols



## A



## B



## C



## D



## E



## F

## U



## V



## X



## Z