Scopes Documentation

Release 0.16

Leonard Ritter

Feb 25, 2020

Contents

This manual aims to provide an introduction to programming with Scopes, a description of Scopes' 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 Scopes

Scopes is a general purpose programming language and compiler infrastructure specifically suited for short turnaround prototyping and development of high performance applications in need of multi-stage compilation at runtime.

The project was started as an alternative to C++ for programming computer games and related tools, specifically for solutions that depend heavily on live procedural generation or aim to provide good modding support.

The compiler is written in about 30k lines of C++ code, supports a LLVM as well as a SPIR-V backend (targeting both CPU and GPU with a single codebase), and exports a minimal runtime environment. The remaining features are bootstrapped from within the language.

The language is expression based, but primarily imperative. The syntactical style marries concepts from Scheme and Python, describing source code with S-expressions but delimiting blocks by indentation rather than braces. Closures are supported as a zero-cost abstraction. The type system is strongly statically typed but fully inferred, therefore every function is a template. Both nominal and structural typing are supported. Type primitives roughly match C level, but are aimed to be expandable without limitations. The memory model is compatible to C/C++ and supports simple unmanaged stack and heap memory, as well as "view propagation", a declaration free variant of borrow checking specifically designed for Scopes. A custom lightweight exception protocol that is fully compatible with C is supported.

Scopes provides many metaprogramming facilities such as programmable syntax sugar and AST macros, metadata necessary for basic step-by-step debugging as well as inspection of types, constants, intermediate code, optimized output and disassembly. The environment is suitable for development of domain specific languages to describe configuration files, user interfaces, state machines or processing graphs.

Scopes embeds the clang compiler infrastructure and is therefore fully C compatible. C libraries can be imported and executed at compile- and runtime without overhead and without requiring special bindings.

The Scopes Compiler Intermediate Language is suitable for painless translation to SSA forms such as LLVM IR and SPIR-V, of which both are supported. The SPIR-V backend also emits GLSL shader code on the fly.

The Scopes compiler fundamentally differs from C++ and other traditional AOT (ahead of time) compilers, in that the compiler is designed to remain on-line at runtime so that functions can be recompiled when the need arises, and generated machine code can adapt to the instruction set present on the target machine. This also diminishes the need for a build system. Still, Scopes is **not** a JIT compiler. Compilation is always explicitly initiated by the user.

CHAPTER 2

Getting Started

2.1 Installation

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

The main repository for scopes is on bitbucket.

Note: any reference to `scopes-repo` in the instructions refers to wherever you have checked out the Scopes source tree.

2.1.1 Building Scopes on Windows

Scopes only supports the `mingw64` toolchain for the foreseeable future.

- Install MSYS2 and install clang, LLVM 9.0.x, cmake and make for `x86_64`. The packages are named `mingw64/mingw-w64-x86_64-llvm`, `mingw64/mingw-w64-x86_64-clang`, `mingw64/mingw-w64-x86_64-cmake` and `make`.
- Nice to have: `mingw-w64-x86_64-gdb`
- at least `usr/bin` from your MSYS2 installation must be added to the `PATH` variable so that the buildscript can find MSYS2.
- You also need the latest source distributions of SPIRV-Tools and SPIRV-Cross checked out into the workspace folder.
- Lastly, you need a build of GENie (binaries available on the page).
- Check `SPIRV-Tools`' build instructions to verify that its dependency on `SPIRV-Headers` is satisfied, and all other dependencies are up-to-date. Build `SPIRV-Tools` using `mkdir build && cd build && cmake .. -G "MSYS Makefiles" -DCMAKE_BUILD_TYPE=Release && make` in `scopes-repo/SPIRV-Tools/build`.
- `SPIRV-Cross` does not have to be built.
- In the workspace folder, run `genie gmake` once to generate the project Makefiles.

- To build in debug mode, run `make -C build`. For release mode, use `make -C build config=release`. Use the option `-j4` to speed up the build on a multicore machine, where `4` is a sensible number of CPU hardware threads to use.
- There should now be a `scopes.exe` executable in the *bin* folder.
- For the clang bridge to work properly, copy `clang/lib/clang/9.0.x/include` to `scopes-repo/lib/clang/include`.
- For a fresh rebuild, just remove the `build` directory before running `make` again.

2.1.2 Building Scopes on Linux

- You need build-essentials, clang, libclang and LLVM 9.0.x installed - preferably locally:
- Put `llvm-config` in your `$PATH`.
- Alternatively, provide your own clang distribution and symlink it to `scopes-repo/clang`.
- You also need the latest source distributions of SPIRV-Tools and SPIRV-Cross checked out or symlinked into the workspace folder.
- Lastly, you need a build of GENie (binaries available on the page).
- Check `SPIRV-Tools`' build instructions to verify that its dependency on `SPIRV-Headers` is satisfied, and all other dependencies are up-to-date. Build `SPIRV-Tools` using `mkdir build && cd build && cmake .. -DCMAKE_BUILD_TYPE=Release && make` in `scopes-repo/SPIRV-Tools/build`.
- `SPIRV-Cross` does not have to be built.
- In the workspace folder, run `genie gmake` once to generate the project Makefiles.
- To build in debug mode, run `make -C build`. For release mode, use `make -C build config=release`. Use the option `-j4` to speed up the build on a multicore machine, where `4` is a sensible number of CPU hardware threads to use.
- There should now be a `scopes` executable in the *bin* folder.
- For the clang bridge to work properly, copy or symlink `$(llvm-config --prefix)/lib/clang/$(llvm-config --version)/include` to `scopes-repo/lib/clang/include`.
- For a fresh rebuild, just remove the `build` directory before running `make` again.

2.1.3 Building Scopes on macOS

- Scopes builds on macOS Mojave (10.14) using LLVM 9.0.
- You'll need the following packages from brew: `llvm` and `cmake`. Scopes' build system respects `brew`'s standard installation paths.
- Alternatively, provide your own clang distribution and symlink it to `scopes-repo/clang`.
- Put `llvm-config` in your `$PATH`. Find it in `$(brew --prefix llvm)/bin` if using brew.
- You'll also need an installation of the Xcode Command Line Tools: `xcode-select --install`.
- You may also need to force installation of the macOS SDK headers: Open `macOS_SDK_headers_for_macOS_10.14.pkg` found in `/Library/Developer/CommandLineTools/Packages`

- You also need the latest source distributions of SPIRV-Tools and SPIRV-Cross checked out or symlinked into the workspace folder.
- Lastly, you need a build of GENie (binaries available on the page).
- Check `SPIRV-Tools`' build instructions to verify that its dependency on `SPIRV-Headers` is satisfied, and all other dependencies are up-to-date. Build `SPIRV-Tools` using `mkdir build && cd build && cmake .. -DCMAKE_BUILD_TYPE=Release && make` in `scopes-repo/SPIRV-Tools/build`.
- `SPIRV-Cross` does not have to be built.
- In the workspace folder, run `genie gmake` once to generate the project Makefiles.
- To build in debug mode, run `make -C build`. For release mode, use `make -C build config=release`. Use the option `-j4` to speed up the build on a multicore machine, where `4` is a sensible number of CPU hardware threads to use.
- There should now be a `scopes` executable in the *`bin`* folder.
- For the clang bridge to work properly, copy or symlink `$(llvm-config --prefix)/lib/clang/$(llvm-config --version)/include` to `scopes-repo/lib/clang/include`.
- For a fresh rebuild, just remove the `build` directory before running `make` again.

CHAPTER 3

The Scopes Tutorial

This tutorial does not attempt to cover every single feature, but focuses on Scopes' most noteworthy features to give you a good idea of the language's flavor and style. After reading it, you will be able to read and write Scopes modules and programs.

3.1 Using the Scopes Live Compiler

After downloading and unpacking the latest release of Scopes, the easiest way to start it is to simply launch the executable shipped with the archive. It is usually located in the root directory, and on Unix compatible systems it can simply be started from the terminal with:

```
$ ./scopes
```

On Windows, and on systems where Scopes has been installed system-wide, it can be started from the command line without the preceding dot:

```
> scopes
```

3.1.1 Interactive Console

When `scopes` is launched without arguments, it enters an interactive read-eval-print loop (REPL), also called a console. Here's an example:

```
$ ./scopes
  \\\
   \\\
 ///\\\
///  \\\  Scopes 0.14 (Apr 17 2019, 19:43:48)
$0
```

Simple expressions can be written on a single line, followed by hitting the return key:

```
$0  print "hello world"
hello world
$0
```

Multiline expressions can be entered by trailing the first line with a space character, and exited by entering nothing on the last line:

```
$0  print#put a space here
....      "yes"
....      "this"
....      "is"
....      "dog"
....
yes this is dog
$0
```

Entering a value binds it to the name indicated by the prompt, and can then be reused:

```
$0  3
$0 = 3
$1  print $0
3
$1
```

A special keyboard shortcut (`Control+D`) at the prompt exits the program. You can also exit the program by typing `exit;` followed by hitting the return key.

3.1.2 Launcher

Most of the time you would like to use Scopes to compile and execute your own written Scopes programs. This is simply done by appending the name of the Scopes file you would like to launch to the executable:

```
$ scopes path/to/my/program.sc
```

3.2 A Fistful of Scopes

Many of the examples in this tutorial include comments, even those entered at the console. Comments in Scopes start with a hash character `#` and extend to the first line starting with a character at a lower or equal indentation.

Some examples:

```
# this is the first comment
print "hey!" # and this is a second comment
               and a third, continuing on the same indentation
let str = "# hash characters inside string quotes don't count as comments"
```

3.2.1 Using Scopes as a Calculator

Scopes is not only a fully-fledged compiler infrastructure, but also works nicely as a comfy calculator:

```
$0  1 + 2 + 3
$0 = 6
$1  23 + 2 * 21
$1 = 65
$2  (23 + 2 * 21) / 5
$2 = 13.0
$3  8 / 5 # all divisions return a floating point number
$3 = 1.6
```

Integer numbers like `6` or `65` have type *i32*, real numbers with a fractional part like `13.0` or `1.6` have type *f32*.

Division always returns a real number. On the off-chance that you want an integer result without the fractional part, use the floor division operator *//*:

```
$0  23 / 3 # regular division returns a real
$0 = 7.666667
$1  23 // 3 # floor division returns an integer
$1 = 7
$2  23 % 3 # modulo returns the remainder
$2 = 2
$3  $1 * 3 + $2 # result * divisor + remainder
$3 = 23
```

3.2.2 Binding Names

Notice how the last example leveraged the auto-memorization function of the console to bind any result to a name for reuse. But we can also make use of *let* to bind values to specific names:

```
$0  let width = 23
23
$0  let height = 42
42
$0  width * height
$0 = 966
```

If a name isn't bound to anything, using it will give you an error, which is useful when you've just mistyped it:

```
$0  let color = "red"
$0  colour
<string>:1:1: while expanding
    colour
error: syntax: identifier 'colour' is not declared in scope. Did you mean 'color'?
```

3.2.3 Strings

Life can be tedious and boring at times. Why not perform some string operations to pass the time? We start with some light declarations of string literals:

```
$0  "make it so" # every string is wrapped in double quotes
$0 = "make it so"
$1  "\"make it so!\", he said" # nested quotes need to be escaped
$1 = "\"make it so!\", he said"
$2  "'make it so!', he said" # single quotes are no problem though
$2 = "'make it so!', he said"
```

(continues on next page)

(continued from previous page)

```
$3  """"1. make it so
          2. ???
          3. profit!
....
$3 = "1. make it so\n2. ???\n3. profit!\n"
```

In the interactive console output, the output string is enclosed in quotes and special characters are escaped with backslashes, to match the way the string has been declared. Sometimes this might look a little different from the input, but the strings are equivalent. The *print* function produces a more readable output that produces the intended look:

```
$0  print "make it so"
make it so
$0  print "\"make it so!\", he said"
"make it so!", he said
$0  print """"1. "make it so!", he said
               2. ???
               3. profit!"
....
1. "make it so!", he said
2. ???
3. profit!
```

Sometimes it's necessary to join several strings into one. Strings can be joined with the `..` operator:

```
$0  "Sco" .. "pes" .. "!" # joining three strings together
$0 = "Scopes!"
$1  .. "Sco" "pes" "!" # using prefix notation
$1 = "Scopes!"
```

The inverse operation, slicing strings, can be performed with the *lslice*, *rslice* and *slice* operations:

```
$0  "scopes" # bind the string we're working on to $0
$0 = "scopes"
$1  rslice $0 1 # slice right side starting at the second character
$1 = "copes"
$2  slice $0 1 5 # slice four letters from the center
$2 = "cope"
$3  lslice $0 ((countof $0) - 1) # a negative index selects from the back
$3 = "scope"
$4  rslice $0 ((countof $0) - 2) # get the last two characters
$4 = "es"
$5  slice $0 2 3 # get the center character
$5 = "o"
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
+---+---+---+---+---+---+
| S | c | o | p | e | s |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
```

If we're interested in the byte value of a single character from a string, we can use the `@` operator, also called the at-operator, to extract it:

```
$0  "abc" @ 0
$0 = 97:i8
$1  "abc" @ 1
$1 = 98:i8
$2  "abc" @ 2
$2 = 99:i8
$3  "abc" @ ((countof "abc") - 1) # get the last character
$3 = 99:i8
```

The *countof* operation returns the byte length of a string:

```
$2  countof "six"
$2 = 3:usize
$3  countof "three"
$3 = 5:usize
$4  countof "five"
$4 = 4:usize
```

3.2.4 A Mild Breeze of Programming

Many calculations require repeating an operation several times, and of course Scopes can also do that. For instance, here is one of the typical examples for such a task, computing the first few numbers of the fibonacci sequence:

```
$0  loop (a b = 0 1)
....     if (b < 10)
....         print b
....         repeat b (a + b)
....     else
....         break b
....
1
1
2
3
5
8
$0 = 13
```

This example introduces several new features.

- The first line declares the entry point of a loop so we can jump back (see the fourth line), bind new values to `a` and `b`, and perform the same operations again.
- The first line also performs multiple assignments at the same time. `a` is initially bound to `0`, while `b` is initialized to `1`. When we jump to this assignment again in line four, `a` will be bound to `b`, while `b` will be bound to the result of calculating `(a + b)`.
- In the second line, we perform a *conditional operation*. That is, the indented block formed by lines three and four is only executed if the expression `(b < 10)` evaluates to *true*. In other words: we are going to be performing the loop as long as `b` is smaller than `10`.
- In line 5, we introduce the alternative block to be executed when `b` is greater or equal to `10`.
- In line 6, we break from the loop, returning the final value of `b`.
- Scopes offers a set of comparison operators for all basic types. You can compare any two numbers using `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to) and `!=` (not equal to).

- The body of the conditional block is indented: indentation is Scopes' way of grouping statements. At the console, you have to type a tab or four spaces for each indented line. In practice you will prepare more complicated input for Scopes with a text editor; all decent text editors have an auto-indent facility. Note that each line within a basic block must be indented by the same amount.

3.3 Controlling Flow

Let's get a little deeper into ways you can structure control flow in Scopes.

3.3.1 `if` Expressions

You have seen a small bit of *if* in that fibonacci example. *if* is your go-to solution for any task that requires the program to make decisions. Another example:

```
$0  __prompt "please enter a word: " ""
please enter a word: bang
$0 $1 = true "bang"
$2  if ($1 < "n")
....     print "early in the dictionary, good choice!"
.... elseif ($1 == "scopes")
....     print "oh, a very good word!"
.... elseif ($1 == "")
....     print "that's no word at all!"
.... else
....     print "late in the dictionary, nice!"
....
early in the dictionary, good choice!
```

You can also use *if* to decide on an expression:

```
$0  print "you chose"
....     if true
....         "poorly"
....     else
....         "wisely"
....
you chose poorly
```

3.3.2 Defining Functions

Let's generalize the fibonacci example from earlier to a function that can write numbers from the fibonacci sequence up to an arbitrary boundary:

```
$0  fn fib (n) # write Fibonacci series up to n
....     loop (a b = 0 1)
....         if (a < n)
....             io-write! (repr a)
....             io-write! " "
....             repeat b (a + b)
....         else
....             io-write! "\n"
....             break b
```

(continues on next page)

(continued from previous page)

```
....
fib:Closure
$0  fib 2000 # call the function we just defined
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
$0 = 4181
```

The keyword *fn* introduces a function definition. It must be followed by an optional name and a list of formal parameters. All expressions that follow form the body of the function and it's good taste to indent them.

Executing (also called *applying*) a function binds the passed arguments to its formal parameters and performs the actions within the function with that argument standing in.

In this example, `n` is bound to `2000`, all instances of `n` in the body of `fib` are replaced with `2000`, and therefore the loop is executed until the condition `a < 2000` is *true*.

CHAPTER 4

Scopes for C/C++ Users

This is an introduction of Scopes for C/C++ users. We are going to highlight commonalities and differences between both languages. At the end of this introduction, you will have a better understanding of how C/C++ concepts translate to Scopes, and which idiosyncrasies to expect.

4.1 Execution

C/C++ requires programs to be built and linked before they can be executed. For this, a build system is typically employed.

In contrast, Scopes' primary execution mode is live, which means that the compiler remains on-line while the program is running, permitting to compile additional functions on demand, and to use other services provided by the Scopes runtime library. Programs are compiled transparently and cached in the background to optimize loading times. This mechanism is designed to be maintenance-free.

However, Scopes can also build object files compatible with GCC and Clang at runtime using `compile-object`, which makes it both suitable for classic offline compilation and as foundation for a build system.

When building objects with Scopes, certain restrictions apply. The generated object file has no ties to the Scopes runtime, and so you must not use any first class objects as constants in your program. You may use `sc_*` functions exported by the Scopes runtime, but must link with `libscopesrt` to make those symbols available.

4.2 Compiler Errors

At compile time, C/C++ compilers attempt to catch and report as many errors encountered as possible by default, with minimal contextual information provided to prevent bloating the error report further.

In contrast, Scopes terminates compilation at the first error encountered, providing a compiler stack trace highlighting the individual steps in chronological order that have led to this error.

4.3 Runtime Debugging

C/C++ programs are debugged at runtime by using a step debugger like GDB or WinDbg, or by what is commonly referred to as "printf debugging".

Scopes generates DWARF/COFF debug information for its programs and therefore supports step debuggers like GDB which understand the format. Due to a limitation in LLVM, debug information generated at runtime is currently not available on Windows. Object files are not affected by this limitation.

Two ways of "printf debugging" are provided, the function *report*, which prints the runtime value of the arguments provided, and the builtin *dump*, which prints, at compile time, the instruction and type of the arguments provided, allowing to inspect constants and types of variables with inferred type. Both functions prefix their output by the source location of where they were called, so they are easy to remove when the session is over.

4.4 Indentation

As indentation has no significance in C/C++, many indentation styles are known, and they constitute a cherished subject of much discussion and many pointless arguments.

In Scopes, scoping is controlled either by parentheses or indentation. To permit users to freely exchange code without friction, the indentation level is fixed at four spaces outside of parenthesed expressions, and the use of tab characters for indentation is not permitted.

4.5 Symbols

For symbolic tokens that can be used to bind names to values and types, C accepts the character set of `0-9A-Za-z_`. This allows users to concatenate many tokens without separating whitespace, such as in `x+y`.

Instead of a whitelist of permitted characters, Scopes only maintains a blacklist of characters that terminate a symbolic sequence. These characters are the whitespace characters and characters from the set `()[]{}"';#,`, where `,` is in itself a context free symbol.

This means that `x+y` would be read as a single token. A semantically equivalent expression in Scopes would have to be written as `x + y`.

4.6 Keywords

C/C++ uses many different declarative forms which are recognized and translated during parsing using specific keywords. Like many other Scheme-likes, Scopes only parses programs as symbolic lists and postpones the interpretation of declarations until the last possible moment, expecting all expressions to follow just one basic form:

```
# classic braced expression
(head argument1 ... argumentN)
# naked syntax
head argument1 ... argumentN
# naked paragraph form
head argument1 ...
    ...
    argumentN
```

The value and type of the head controls whether the expression is dispatched to

- A syntax macro (called a *sugar*), called at expansion time, which has complete control over which, when and how remaining arguments will be expanded and evaluated, and can either return new symbolic lists to be expanded further, or a template IL.
- An IL macro (called a *spice*), called at compile time, which receives typed arguments and can generate new template IL to be specialized.
- A call expression, equivalent to the `head(argument1, ..., argumentN);` syntax in C/C++, called at runtime.

As a result of this principle, there are no keywords in Scopes. Every single symbol can be rebound or deleted, globally or just for one scope.

Scopes also supports wildcard syntax macros (wildcard sugars), which can be applied to either any expression or symbol before the expansion begins, and which can be used to implement exceptions to the head dispatch rule. One of these exceptions for instance is infix notation support.

4.7 Infix Expressions

C/C++ offers a fixed set of infix operators which are recognized during parsing and translated to AST nodes right then. They can be used within expressions but must be put in parentheses or separated by semicolon or comma from neighboring expressions in argument or statement lists.

Scopes also offers a set of commonly used infix operators not unlike the ones provided by C, utilizing the same associativity and nearly the same precedence (the `<<` and `>>` operators use a different precedence), but renaming and adding operators where it improves clarity.

Unlike C/C++, Scopes allows to define new scoped infix operators at the users convenience using *define-infix<* and *define-infix>*, as a simple alias to an existing sugar, spice or function.

An infix expression is recognized by looking for an infix definition for the second token of an expression. If a matching definition is found, all tokens within that expression are treated as left/right-hand arguments and operators.

The infix expression must follow the pattern `(x0 op x1 op ... op xN)` to be recognized. If an odd argument is not a valid infix token, a syntax error will be raised.

Scopes does deliberately not implement any concept of mixed infix, prefix or postfix expressions to keep confusion to a minimum. Even infix expressions can be entirely disabled by replacing the default wildcard sugar.

A special symbol sugar exists which aims to simplify trivial container lookups. An expression like `(object . attribute . attribute)` can also be written as a single symbol, `object.attribute.attribute`, which will be expanded to the former form, provided no value is already bound to this symbol in the current scope.

4.8 Declarations, Statements and Expressions

C/C++ distinguishes between three major lexical contexts: declaration level, statement level and expression level.

```
// declaration level
typedef int MyInt;

// illegal at this level
// printf("hello!\n");

MyInt test (MyInt x) {
    // statement level
    int k =
```

(continues on next page)

(continued from previous page)

```
        // expression level
        x * x
    ;
    int m = ({
        // statement expressions, a GCC extension
        // usage of statements here is legal.
        printf("hello again!\n");
        k * k;
    });
    return m;
}
```

Scopes does not make such a distinction, and instead treats every declaration as an expression with a result type. The top level of a program is equivalent to its main function:

```
# the right hand side is not limited to constant expressions.
let MyInt = int

# legal at this level.
print "hello!"

fn test (x)
    let k = (x * x)

    let m =
        do
            # equivalent to statement expressions in GCC.
            print "hello again!"
            k * k
    # even `return` declares an expression of type `noreturn`.
    return m
```

4.9 Constants and Variables

C/C++ expects named values to be declared with a type. Each value is mutable by default unless qualified by `const`. Outside of a function it represents a globally accessible value, within a function it represents a stack value.

```
const int constant = 100;

// a global value mapped to data segment
int variable1 = 0;
int variable2 = 0;

// conceptually a copy operation
const int variable1_copy = variable1;

void test () {
    // conceptually declared on the stack
    // initialization from value conceptually a copy operation
    int variable3 = constant;

    // mutable by default
    variable3 = variable2;
```

(continues on next page)

(continued from previous page)

```
    printf("%i\n", constant);
    printf("%i\n", variable1);
    printf("%i\n", variable2);
    printf("%i\n", variable3);
}
```

In Scopes, expressions are bound to names using *let*. *let* does only perform the binding. The type of the value and where it is stored depends entirely on the expression that produces it. The *local* and *global* forms must be used to explicitly allocate a stack or data segment value:

```
# a compile time constant integer
let constant = 100

# not a global value, but allocated on the main function's stack
local variable1 = 0
# a global value mapped to data segment
global variable2 = 0

# variable1 is bound to another name - not a copy operation.
  variable1_copy remains mutable.
let variable1_copy = variable1

fn test ()
    # just a rebind - not a copy operation
    let variable3 = constant
    # variable3 is not a reference, so can not be mutated.
      we should have declared it as local for that.
    # variable3 = variable2

    print constant
    # illegal: variable1 is a stack variable outside of function scope
    # print variable1
    # legal: variable2 is a global
    print variable2
    # variable3 is a constant
    print variable3
```

Unlike in C/C++, declarations of the same name within the same scope are also permitted, and the previous binding is still accessible during evaluation of the right-hand side:

```
let x = 1
# x is now bound to the value 3
let x = (x + 2)
# x is now bound to a string
let x = "test"
```

4.10 Lexical Scope

Both C/C++ and Scopes employ lexical scope to control visibility of bound names.

Unlike in C/C++, lexical scope is a first order object in Scopes and can be used by new declarative forms as well as to export symbols from modules:

```
let scope =
    do
        let x = 1
        let y = "test"

        # build a new scope from locally bound names
        locals;

# scope is constant if all values in it are constant
static-assert (constant? scope)
# prints 1 "test"
print scope.x scope.y
```

4.11 Macros

The C preprocessor provides the only means of using macros in C/C++ code. The latest edition permits variadic arguments, but reflection and conditional behavior can only be achieved through tricks. C macros are also unable to bind names in a way that prevents collision with existing names in scope, which is called "unhygienic" in the Scheme community. Macros are able to transparently override call expressions and symbolic tokens, and do not have to respect semantic structure.

As a Scheme-like, Scopes' macro facilities are extensive.

Sugars are functions able to rewrite expressions at the syntactical level during syntax expansion. Wildcard sugars can rewrite symbols or even just parts of symbols. They are evaluated top-down and can produce hygienic and unhygienic expressions.

Spices are evaluated bottom-up during typechecking and receive eagerly evaluated arguments. Both forms can generate new code in the form of untyped IL.

Hygienic macro functions are provided by the *inline* form, which is expanded during typechecking. Inlines must respect semantic structure and are not programmable, but can make use of spices to perform reflection and conditional code generation, as well as generate new functions.

4.12 Templates

In C, every function must be typed as it is (forward) declared. C++ introduces the concept of templates, which are functions that can be lazily typed. As of C++14, templates can now also deduct their return type. Templates can be forward declared, but forward declared templates with automatic return type can not be instantiated.

```
// forward declaration of typed function
int typed_forward_decl (int x, const char *text, bool toggle);

// declaration of typed function
int typed_decl (int x, const char *text, bool toggle) {
    return 0;
}

// forward declaration of template
template<typename A, typename B, typename C>
void lazy_typed_decl_returns_void (A a, B b, C c);

void test1 () {
```

(continues on next page)

(continued from previous page)

```
    // forward declaration can be used
    lazy_typed_decl_returns_void(1,2,3);
}

// forward declaration of template with auto return type
template<typename A, typename B, typename C>
auto lazy_typed_decl_returns_auto (A a, B b, C c);

void test2 () {
    // error: use before deduction of 'auto'
    lazy_typed_decl_returns_auto(1,2,3);
}

// implementation of template with auto return type
template<typename A, typename B, typename C>
auto lazy_typed_decl_returns_auto (A a, B b, C c) {
    return 0;
}
```

In Scopes, all function declarations are lazily typed, and *static-typify* can be used to instantiate concrete functions at compile time. Forward declarations are possible but must be completed within the same scope:

```
# forward declarations can not be typed
#fn typed_forward_decl

fn typed_decl (x text toggle)
    return 0

# create typed function
let typed_decl = (static-typify typed_decl i32 rawstring bool)

# forward declaration of template has no parameter list
fn lazy_typed_decl_returns_void

# test1 is another template
fn test1 ()
    # legal because test1 is not instantiated yet
    # note: lazy_typed_decl_returns_void must be implemented before
            test1 is instantiated.
    lazy_typed_decl_returns_void 1 2 3

# forward declaration of template with auto return type
  note that all our forward declarations have no return type
fn lazy_typed_decl_returns_auto

fn test2 ()
    # legal because test2 is not instantiated yet
    lazy_typed_decl_returns_auto 1 2 3

# implementation of template with auto return type
fn lazy_typed_decl_returns_auto (a b c)
    return 0

# instantiate test2
let test2 = (static-typify test2)
```

4.13 Variadic Arguments

C introduced variadic arguments at runtime using the `va_list` type, in order to support variadic functions like `printf()`. C++ improved upon this concept by introducing variadic template arguments. It remains difficult to perform reflection on variadic arguments, such as iteration or targeted capturing.

Functions in Scopes do not support runtime variadic functions (although calling variadic C functions is supported), but support compile time variadic arguments. See the following example:

```
# any trailing parameter ending in '...' is interpreted to be variadic.
fn takes-varargs (x y rest...)
    # count the number of arguments in rest...
    let numargs = (va-countof rest...)

    # use let's support for variadic values to split arguments into
      first argument and remainder.
    let z rest... = rest...

    # get the 5th argument from rest...
    let fifth_arg = (va@ 5 rest...)

    # iterate through all arguments, perform an action on each one
      and store the result in a new variadic value.
    let processed... =
        va-map
            inline (value)
                print value
                value + 1
            rest...

    # return variadic result as multiple return values
    return processed...
```

4.14 Overloading

C++ allows overloading of functions by specifying multiple functions with the same name but different type signatures. On call, the call arguments types are used to deduce the correct function to use.

```
int overloaded (int a, int b) { return 0; }
int overloaded (int a, float b) { return 1; }
int overloaded (float a, int b) { return 2; }

// this new form of overloaded could be specified in a different file
int overloaded (float a, float b) { return 3; }
```

Scopes offers a similar mechanism as a library form, but requires that overloads must be grouped at the time of declaration. The first form that matches argument types implicitly is selected, in order of declaration:

```
fn... overloaded
case (a : i32, b : i32)
    return 0
case (a : i32, b : f32)
    return 1
case (a : f32, b : i32)
```

(continues on next page)

(continued from previous page)

```
    return 2

# expanding overloaded in a different file, limited to local scope

# overwrites the previous declaration
fn... overloaded
case using overloaded # chains the previous declaration
case (a : f32, b : f32) # will be tried last
    return 3
```

4.15 Code Generation

In C/C++, files are interpreted either as translation units (the root file of a compiler invocation) or as header files, which are type and forward declarations that typically do not generate code on their own, embedded into translation units. Fully declared functions are guaranteed to generate code, and will only be optimized out at linking stage.

In Scopes, every invocation of `sc_compile`, typically through `compile` or *`import`*, opens a new translation unit. Function declarations are template declarations, so they do not generate any code, nor does any other compile time construct. Instantiating a function through *`static-typify`* does also not guarantee that code will be generated. Only actual first time use will generate code for whatever translation unit is currently active, and make that code available to every future translation unit. This guarantees that a particular template is only instantiated once.

When objects are compiled through `compile-object`, only functions exported through the scope argument are guaranteed to be included, and all functions they depend on. Objects are complete. Previously generated functions will not be externally defined, but will be redefined as private functions within the objects translation unit. The same rules apply to global variables.

4.16 Using Third Party Libraries

With C/C++, third party libraries are typically built in a separate build process provided by the libraries developer, either as static or shared libraries. Their definitions are made available through include files that one can embed into one's own translation units using the `#include` preprocessing command. The libraries' precompiled symbols are merged into the executable during linking or at runtime, either when the process is mapped into memory or function pointers are loaded manually from the library.

Scopes provides a module system which allows shipping libraries as sources. Any scopes source file can be imported as a module. When a module is first imported using *`import`* or `using import`, its main body is compiled and executed. The returned scope which contains the modules' exported functions and types is cached under the modules name and returned to the importing program. The modules' functions and types are now available to the program and can be embedded directly into its translation unit. No code is generated until the libraries' functions are actually used.

Scopes also supports embedding existing third party C libraries in the classical way, using *`include`*, `load-library` and `load-object`:

```
# how to create trivial bindings for a C library

# include thirdparty.h and make its declarations available as a scope object
let thirdparty =
    include "thirdparty.h"
        options "-I" (module-dir .. "/../include") # specify options for clang

# access a define from thirdparty.h
```

(continues on next page)

(continued from previous page)

```
if thirdparty.define.USE_SHARED_LIBRARY
    # load thirdparty as a shared library from system search paths
    if (operating-system == 'windows)
        load-library "thirdparty.dll"
    else
        load-library "libthirdparty.so"
else
    # load thirdparty as a static library from an object file
    load-object (module-dir .. "/../lib/thirdparty.o")

# assemble a ready-to-use scope object for this module
do
    # import only symbols beginning with thirdparty
    using thirdparty.define filter "^THIRDPARTY_.*$"
    using thirdparty.typedef filter "^thirdparty_.*$"
    using thirdparty.const filter "^thirdparty_.*$"
    using thirdparty.extern filter "^thirdparty_.*$"

    locals;
```

Externals using C signatures can also be defined and used directly:

```
let puts = (extern 'puts (function i32 rawstring))
# definition becomes immediately available
puts "hello\n"
```

4.17 Type Primitives

C/C++'s type primitives map to Scopes in the following way:

C++	Scopes
bool	*bool*
int8_t	*i8*
int16_t	*i16*
int32_t	*i32*
int64_t	*i64*
uint8_t	*u8*
uint16_t	*u16*
uint32_t	*u32*
uint64_t	*u64*
float	*f32*
double	*f64*
typedef U V	let V = U
using V = U	let V = U
const T *	@ T
T *	mutable (@ T)
const T &	& T
T &	mutable (& T)
std::array<T, N>	array T N
std::tuple<T0, ..., Tn>	tuple T0 ... Tn
const T [N] __attribute__((aligned (A)))	vector T N

4.18 Initializer Lists

Scopes provides convenience constructors for arrays and tuple types, as well as a special initializer type called *typeinit*, which can be used to initialize fields without knowing their type. *typeinit* stores passed arguments in a temporary closure which is turned into a constructor call as soon as the typeinit instance is cast to its target type during assignment.

C++	Scopes
`std::array<T> _ = { arg0, ..., argN };`	`arrayof T arg0 ... argN`
`std::tuple<auto> _ = { arg0, ..., argN };`	`tupleof arg0 ... argN`
`x.member = { arg0, ..., argN };`	`x.member = (typeinit arg0 ... argN)`

4.19 Structs

Scopes supports structs in a format not unlike the one C/C++ provides, but does not permit composition by inheritance. Composition must be strictly explicit.

Compare this C++ example, which makes use of recently introduced default initializers and designated initializers:

```
struct Example {
    int value;
    // default initializers only supported in C++11 and up
    bool choice = false;
    const char *text = "";
};

// designated initializers only supported in C99
Example example = { .value = 100, .text = "test" };
```

to this equivalent declaration in Scopes:

```
using import struct

struct Example plain
    value : i32
    # type can be deduced from initializer
    choice = false
    text : rawstring = ""

global example : Example
    value = 100
    text = "test"
```

4.20 Methods

C++ introduced methods as a way to associate functions directly with structs and classes. In C++, the argument referencing the object argument is hidden and implicitly bound to the `this` symbol. Members and other methods of the struct are in the lexical scope of the method.

```
// this example is a little contrived for illustrational purposes
struct Example {
    int value;

    // a method declaration
    int get_add_value (int n) {
        return this->value + n;
    }

    // another method declaration
    void print_value_plus_one () {
        printf("%i\n", get_add_value(1));
    }
};

void use_example (Example example) {
    example.print_value_plus_one();
}
```

Scopes supports methods in a more explicit way that makes refactorings from function to method and back easier, both in declaration and in usage:

```
struct Example plain
    value : i32

    # note the explicit presence of the object parameter
    fn get_add_value (self n)
        self.value + n

    fn print_value_plus_one (self)
        print ('get_add_value self 1)

fn use_example (example)
    'print_value_plus_one example
```

What happens here is that we call a quoted symbol with arguments. The call handler for the *Symbol* type rewrites `'methodname object arg0 ... argN` as `(getattr (typeof object) 'methodname) object arg0 ... argN`.

4.21 Virtual Methods

As Scopes doesn't provide a native abstraction for composition by inheritance, virtual methods are not supported out of the box, but can be implemented through its extensive domain specific language support.

4.22 Classes

C++'s concept of classes is only indirectly supported through structs in Scopes. Access modifiers are not available, but methods can be made "private" by keeping their definition local. Fields can not be hidden, but they can be visibly marked as private by convention:

```
struct Example plain
    # an underscore indicates that the attribute is not meant to be
```

(continues on next page)

(continued from previous page)

```
      accessed directly.
    _value : i32

    fn get_add_value (self n)
        self._value + n

    fn print_value_plus_one (self)
        # use get_add_value directly
        print (get_add_value self 1)

    # unbind get_add_value from local scope to prevent it
      from being added as an attribute to Example.
    unlet get_add_value

fn use_example (example)
    # this operation is not possible from here:
    # 'get_add_value example 1
    'print_value_plus_one example
```

4.23 Template Classes

C++ supports generics in the form of template classes, which are lazily typed structs.

```
template<typename T, int x>
struct Example {
    T value;

    bool compare () {
        value == x;
    }
};
```

Scopes leverages constant expression folding and compile time closures to trivially provide this feature via *inline* functions:

```
# a function decorator memoizes the result so we get the same type for
  the same arguments
@@ memo
inline Example (T x)
    # construct type name from string
    struct ("Example<" .. (tostring T) .. ">")
        value : T

        fn compare ()
            value == x
```

Partial template specialization allows to choose different implementations depending on instantiation arguments. The same mechanism is also used to do type based dispatch. Here is an example:

```
#include <stdlib.h>

template<typename T> struct to_int {
    // linker complains: missing symbol
    int operator()(T x);
```

(continues on next page)

(continued from previous page)

```
};

template<> struct to_int<int> {
    int operator()(int x) {
        return x;
    }
};

template<> struct to_int<const char *> {
    int operator()(const char *x) {
        return atoi(x);
    }
};
```

In Scopes, it is not necessary to create types in order to build single type based dispatch operators. Here are three ways to supply the same functionality:

```
include "stdlib.h"

# a function that generates a function
@@ memo
inline to_int1 (T)
    static-match T
    case i32 _
    case rawstring atoi
    default
        static-error "unsupported type"

# a function that performs the operation directly
inline to_int2 (x)
    let T = (typeof x)
    static-if (T == i32) x
    elseif (T == rawstring) (atoi x)
    else
        static-error "unsupported type"

# using the overloaded function abstraction
fn... to_int3
case (x : i32,) x
case (x : rawstring,) (atoi x)
```

4.24 Constructors

Scopes supports construction from type through the *__typecall* special method. A type implementing a method under this name becomes callable. By convention, it is used to construct both specialized types and to instantiate a type. Its first argument is the name of the type that has been called.

Here is an example that changes the default constructor of a struct:

```
struct Example plain
    _value : i32

    inline __typecall (cls n)
        # within the context of a struct definition, super-type is bound
```

(continues on next page)

(continued from previous page)

```
            to the super type of the struct we are defining. In this case
            the supertype is `CStruct`.
        super-type.__typecall cls
            _value = (n * n)
```

4.25 Destructors

C++ provides so-called destructors which permit to execute code when a value goes out of scope. Destructors typically free resources, but can also be used to switch contexts.

```
struct Handle {
    void *_handle;

    // constructor
    Handle(void *handle) : _handle(handle) {}
    // destructor
    ~Handle() {
        printf("destroying handle\n");
        free(_handle);
    }
};
```

Values of non-plain type, so-called unique types, are guaranteed to be referenced only at a single point within a program. Because of this guarantee, a unique type is able to supply a destructor through the *__drop* special method that is automatically called when the value goes out of scope:

```
struct Handle
    _handle : voidstar

    # constructor
    inline __typecall (cls handle)
        super-type.__typecall cls handle

    # destructor
    inline __drop (self)
        print "destroying handle"
        free self._handle
        return;
```

4.26 Operator Overloading

C++ allows overloading type operators through special methods defined either in a struct, class or namespace.

```
class Accumulable {
public:
    // overload the addition operator for class + class
    Accumulable operator +(Accumulable x) {
        return Accumulable(this->value + x.value);
    }
    // another overload for class + int
    Accumulable operator +(int x) {
```

(continues on next page)

(continued from previous page)

```
        return Accumulable(this->value + x);
    }

    Accumulable (int _value) : value(_value) {}

    int value;
};

// a third overload for supporting int + class
Accumulable operator +(Accumulable a, int b) {
    return Accumulable(a.value + b.value);
}
```

Scopes supports operator overloading through informally specified operator protocols that that any type can support by exposing dispatch methods bound to special attributes. See this equivalent example, which applies not only to structs, but any type definition:

```
struct Accumulable
    # one compile time function for all left-hand side variants receives
        left-hand and right-hand types and returns a function which can
        perform the operation or void.
    inline __+ (cls T)
        # test for type + type
        static-if (T == this-type)
            # return new closure
            inline (self other)
                this-type (self.value + other.value)
        # if T can be implicitly cast to i32, support it
        elseif (imply? T i32)
            inline (self other)
                this-type (self.value + other)

    # another function covers all right-hand side variants
    inline __r+ (T cls)
        static-if (imply? T i32)
            inline (self other)
                this-type (self + other.value)

    value : i32

    inline __repr (self)
        tostring self.value
```

4.27 Standard Library

C and C++ support an extensive standard library, covering many system functions and algorithmic container types.

Scopes supports the C standard library through the clang bridge accessible by the `include` mechanism.

Only a few containers from the C++ standard library have functional equivalents in Scopes yet. Here is a comparison table:

C++	Scopes
`std::array`	*array*
`std::tuple`	*tuple*
`std::vector`	`Array.GrowingArray`
`std::unordered_set`	`Map.Set`
`std::unordered_map`	`Map.Map`
`std::unique_ptr`	`Box.Box`
`std::function`	`Capture.capture`

4.28 Memory Handling & Management

C and C++ use a stack based machine model that is compatible with native targets. Within this model, mutable memory can be pre-allocated globally, on the function's stack and in main memory, called the "heap".

Scopes uses an identical model. No garbage collection is employed at runtime.

A comparison of concepts by example:

C/C++	Scopes
`T value;` (globally)	`global value : T`
`T value;` (locally)	`local value : T`
`T values[size];` (static size)	`local value : (array T size)`
`T values[size];` (dynamic size)	`let value = (alloca-array T size)`
`alloca(sizeof(T))`	`alloca T`
`alloca(sizeof(T) * size)`	`alloca-array T size`
`malloc(sizeof(T))`	`malloc T`
`malloc(sizeof(T) * size)`	`malloc-array T size`

In addition, C++ allows to manage memory by recursively invoking a type-defined destructor on stack values when exiting a bracketed scope, as well as on globals when the program is exited or a library unloaded. Based on this mechanism and intricate elision rules, various smart pointer types are implemented, of which the most useful is `std::unique_ptr`, which is a type that manages the lifetime of a single heap value until it goes out of scope.

In Scopes, types can be defined as unique, which instructs scope to manage the lifetime of values in a similar fashion as C++ does. In addition, weak references to these values (direct or indirect) can be borrowed as "views", which become inaccessible as soon as the viewed unique value goes out of scope. This mechanism incurs no performance cost at runtime. See *Destructors* for an example.

4.29 Closures

C++ supports runtime closures through the `std::function` type, which allows to implicitly bind values to a function, so that when the function is called, the bound values become available to the function without having to pass them as arguments. In functional programming, this process can be used to implement currying.

```
void print_bound_constant () {
    const int y = 42;
    // capture `y` along with the function
    auto f = [y](int x) -> int { return x + y; }
    // prints 65
    std::cout << f(23) << std::endl;
```

(continues on next page)

(continued from previous page)

```
}

void print_bound_value (int y) {
    // capture `y` along with the function
    auto f = [y](int x) -> int { return x + y; }
    // prints 23+y
    std::cout << f(23) << std::endl;
}
```

Scopes supports compile time closures natively, as demonstrated previously, but runtime closures are also supported through so-called captures. Above example would be translated as follows:

```
fn print_bound_constant ()
    let y = 42
    # capture constant `y` along with the function
    fn f (x)
        x + y
    # prints 65
    print (f 23)

fn print_bound_value (y)
    # capture variable `y` along with the function
    capture f (x) {y}
        x + y
    # prints 65
    print (f 23)
```

4.30 Loops

C offers two structured control flows for loops which depend on mutation of an exit variable, namely `while` and `for`. In addition, C++11 introduced the range-based `for` loop, which provides syntactical sugar for iterating elements of a collection.

```
// C-style for-loop implementing a counter
for (int i = 0; i < 10; ++i) {
    printf("%i\n", i);
}

// C-style for-loop implementing an iterator
for (iter_t it = first(container), int k = 0; is_valid(it); it = next(it), k++) {
    process(k, at(it));
}

// while-loop implementing a counter
int i = 0;
while (i < 10) {
    printf("%i\n", i);
    ++i;
}

// range-based for-loop implementing an iterator (C++17 form)
for (auto &&[first,second] : map) {
    process(first, second);
}
```

Scopes defines a single builtin primitive for loops which leverages backpropagation of immutable values, upon which various other library forms are implemented:

```
# implementing a counter using the range-based form
for i in (range 10)
    print i

# implementing an iterator using the loop primitive and immutable values
loop (it k = (first container) 0)
    if (is_valid it)
        process k (at it)
        repeat (next it) (k + 1)
    else
        # break can return values
        break it k

# implementing a counter using a while loop and mutation
local i = 0
while (i < 10)
    print i
    i += 1

# range-based form implementing an iterator
for key value in map
    process key value
```

In addition, with the `fold .. for .. in` form, Scopes combines both immutable loop and range-based form.

4.31 Targeting Shader Programs

C/C++ do not offer a native way to compile functions to shader code. However, there exist various third party solutions to provide equivalent features. The GLSL (GL shader language) offers a C-like domain specific language to write shaders that has enough overlap with C/C++ in order to allow users to share definitions.

Scopes is able to natively compile functions to SPIR-V as well as GLSL at compile time using the builtins `compile-spirv` and `compile-glsl` respectively, allowing the CPU and GPU side to share all definitions. To aid in this task, Scopes provides the *glm* and *glsl* modules, which implement native GLSL types and functions.

See the following example implementing and compiling a pixel shader:

```
using import glm
using import glsl

in uv : vec2 (location = 0)
out color : vec4 (location = 1)
fn main ()
    color = (vec4 (uv * 0.5 + 0.5) 0 1)

print
    compile-glsl 330 'fragment
        static-typify main
```

The program output is as follows:

```
#version 330
#ifdef GL_ARB_shading_language_420pack
```

(continues on next page)

(continued from previous page)

```
#extension GL_ARB_shading_language_420pack : require
#endif

in vec2 uv;
layout(location = 1) out vec4 color;

void main()
{
    vec2 _14 = (uv * vec2(0.5)) + vec2(0.5);
    vec4 _19 = vec4(0.0);
    _19.x = _14.x;
    vec4 _21 = _19;
    _21.y = _14.y;
    vec4 _22 = _21;
    _22.z = 0.0;
    vec4 _24 = _22;
    _24.w = 1.0;
    color = _24;
}
```

4.32 Exceptions

C provides only a primitive kind of unstructured exception handling via the `setjmp()` and `longjmp()` functions provided by `setjmp.h`.

C++ provides structured and polymorphic exception handling at runtime. Any value can be thrown as an exception using the `throw` keyword, and caught using the `try .. catch` form.

```
struct myexception {
    const char *what;
};

void main () {
    try {
        // throw value of type myexception
        myexception exc = { "an error occurred" };
        throw exc;
    } catch (myexception& e) {
        // print content to screen
        std::cout << e.what << std::endl;
    }
}
```

Scopes supports a form of structured exception handling that is monomorphic, light weight and C compatible. A value of any type can be raised using the `raise` form, and handled using the `try .. except` form:

```
using import struct

struct myexception
    what : string

try
    # raise value of type myexception
    raise (myexception "an error occurred")
```

(continues on next page)

(continued from previous page)

```
except (e)
    # print content to screen
    print e.what
```

The presence of an exception modifies the return type of a function to a hidden tagged union type which returns which path the function returned on, and both return and exception value, of which only the appropriate value has been set.

Monomorphic means that in contrast to C++, Scopes does not allow more than one exception type per expression to be backpropagated. If you wish to support a polymorphic type, you can use `enum` to define a tagged union type which can be dispatched to the correct exception type.

4.33 ABI Compliance

Scopes aims to achieve full compliance with the C ABI used on x64 platforms for Linux, MacOS X and Windows, defaulting to the `cdecl` calling convention. Other calling conventions are not yet supported. Any Scopes function can be passed as a callback to a C library, and C functions can be called from Scopes without any additional hinting required.

All types aim to follow the same alignment and size conventions as C types, including plain unions.

On Windows, Scopes is built for and communicates with system resources through mingw64. Operating with WINAPI functions directly has not been extensively tested yet.

CHAPTER 5

Notation

Scopes source code is written in a notation that introduces syntactic rules even before the first function is even written: *Scopes List Notation*, abbreviated **SLN**.

Closely related to S-Expressions, SLN can be seen as a human-readable serialization format comparable to YAML, XML or JSON. It has been optimized for simplicity and terseness.

SLN files do not have to contain code on their own. They're more likely to store configuration or metadata. Therefore, the examples in this document are schema free and do only contain arbitrary data. They're not necessarily valid Scopes source code.

5.1 At a Glance

In case you don't have time to read the full documentation, here's an example that gives you an overview of all notation aspects:

```
# below is some random data without any schema

# a naked list of five 32-bit signed integers
1 2 3 4 5

# a list that begins with a symbol 'float-values:' and contains a braced
# sublist of floats.
float-values: (1.0 2.0 3.1 4.2 5.5:f64 inf nan)

# we can also nest the sublist using indentation
# note the extravagant heading, another context-free symbol.
==string-values==
    "A" "B" "NCC-1701\n" "\xFFD\xFF" "\"E\""

# a single top-level element, a single-line string
"I am Locutus of Borg."
```

(continues on next page)

(continued from previous page)

```
# a raw block string
"""
    Ma'am is acceptable in a crunch, but I prefer Captain.
                                    -- Kathryn Janeway

# a list of pairs (also lists), arranged horizontally
(1 x) (2 y) (3 z)
# same list, with last two entries arranged vertically
(1 x)
    (2 y)
    (3 z)
# we can line up all entries by using a semicolon to indicate an empty head
;
    (1 x)
    (2 y)
    (3 z)
# parentheses can also be removed for each line entry
;
    1 x
    2 y
    3 z

# appending values to the parent list in the next line
symbol-values one two three four five \
    six seven-of-nine ten

# line continuation can also begin at the start of the next line
::typed-integers:: 0:u8 1:i8 2:i16 3:u16
    \ 4:u32 5:i32 6:u64 7:i64

# which comes in handy when we want to continue the parent list
people like
    jim kirk
    commander spock
    hikari sulu
    \ and many more

# a list with a symbol header and two entries
address-list
    # a list with a header and three more lists of two values each
    entry
        name: "Jean-Luc Picard"
        age: 59
        address: picard@enterprise.org
    entry
        # the semicolon acts as list separator
        name: "Worf, Son of Mogh"; age: 24; address: worf@house-of-mogh.co.klingon
    # line comments double as block comments
    #entry
        name: "Natasha Yar"
        age: 27
        address: natasha.yar@enterprise.org

# the same list with braced notation; within braced lists,
  indentation is meaningless.
(address-list
    # a list with a header and three more lists of two values each
```

(continues on next page)

(continued from previous page)

```
    (entry
        (name: "Jean-Luc Picard")
        (age: 59)
        (address: picard@enterprise.org))
    (entry (name: "Worf, Son of Mogh") (age: 24)
        (address: worf@house-of-mogh.co.klingon)))

# a list of comma separated values - a comma is always recorded as
  a separate symbol, so the list has nine entries
1, 2, 3,4, 5

# a list of options beginning with a symbol in a list with
  square brace style
[task]
    cmd = "bash"
    # the last element is a symbol in a list with curly brace style
    working-dir = {project-base}
```

5.2 Formatting Rules

SLN files are always assumed to be encoded as UTF-8.

Whitespace controls scoping in the SLN format. Therefore, to avoid possible ambiguities, SLN files must always use spaces, and one indentation level equals four spaces.

5.3 Element Types

SLN recognizes only five kinds of elements:

- **Numbers**
- **Strings**
- **Symbols**
- **Lists**

In addition, users can specify comments which are not part of the data structure.

5.3.1 Comments

Both line and block comments are initiated with a single token, `#`. A comment lasts from its beginning token to the first non-whitespace character with equal or lower indentation. Some examples for valid comments:

```
# a line comment
not a comment
# a block comment that continues
  in the next line because the line has
  a higher indentation level. Note, that
        comments do not need to respect
    indentation rules
but this line is not a comment
```

5.3.2 Strings

Strings describe sequences of unsigned 8-bit characters in the range of 0-255. 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). Other unprintable characters can be encoded via `\\xNN`, where `NN` is the character's hexadecimal code. Strings are parsed as-is, so UTF-8 encoded strings will be copied over verbatim.

Here are some examples for valid strings:

```
"a single-line string in double quotations"
"return: \n, tab: \t, backslash: \\, double quote: \", nbsp: \xFF."
```

5.3.3 Raw Block Strings

Raw block strings provide a way to quote multiple lines of text with characters that should not be escaped. A raw block string begins with `"""` (four double quotes). A raw block string ends at the first newline before a printable character that has a lower indentation.

Here are some examples for valid raw block strings:

```
"""a single-line string as a block string
# commented line inbetween
"""// a multi-line string that describes a valid C function
    #include <stdio.h>
    void a_function_in_c() {
        printf("hello world\n");
    }
```

5.3.4 Symbols

Like strings, a symbol describes a sequence of 8-bit characters, but acts as a label or bindable name. Symbols may contain any character from the UTF-8 character set and terminate when encountering 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. Two symbols sharing the same sequence of characters always map to the same value.

As a special case, `,` is always parsed as a single character.

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
```

5.3.5 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 32-bit values unless the value is too big, in which case it will be extended to 64-bit signed, then

64-bit unsigned. Reals are floating point numbers parsed and stored as IEEE 754 binary32 values.

Numbers can be explicitly specified to be of a certain type by appending a `:` to the number as well as a numerical typename that is either `i8`, `i16`, `i32`, `i64`, `u8`, `u16`, `u32`, `u64`, `f32` and `f64`.

Here are some examples for valid numbers:

```
# positive and negative integers in decimal and hexadecimal notation
0 +23 42 -303 12 -1 -0x20 0xAFFE
# 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
# zero as unsigned 64-bit integer and as signed 8-bit integer
0:u64 0:i8
# a floating-point number with double precision
1.0:f64
```

5.3.6 Lists

Lists are the only nesting type, and can be either scoped by braces or indentation. For braces, `()`, `[]` and `{}` are accepted.

Lists can be empty or contain a virtually unlimited number of elements, only separated by whitespace. They typically describe expressions in Scopes.

Here are some examples for valid lists:

```
# a list of numbers in naked format
1 2 3 4 5
# three empty braced lists within a naked list
() () ()
# a list containing a symbol, a string, an integer, a real, and an empty list
(print (.. "hello world") 303 606 909)
# three nesting lists
(((())))
```

5.4 Naked & Braced Lists

Every Scopes source file is parsed as a tree of expresion lists.

The classic notation (what we will call *braced notation*) uses a syntax close to what Lisp and Scheme users know as *restricted* S-expressions:

```
(print
    (.. "Hello" "World")
    303 606 909)
```

As a modern alternative, Scopes 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 previous, braced example:

```
# The same list as above, but in naked format.
    A sub-paragraph continues the list.
print
    # elements on a single line with or without sub-paragraph are wrapped
      in a list.
    .. "Hello" "World"

    # values that should not be wrapped have to be prefixed with an
      escape token which causes a continuation of the parent list
    \ 303 606 909
```

5.4.1 Mixing Modes

Naked lists can contain braced lists, and braced 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 appended to the parent list:
print
    + 1 2
        3 * 4

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

# and a braced 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 braced notation sparingly and in good taste. Purists and Scheme enthusiasts may however prefer to work with braced lists almost exclusively.

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

5.5 Brace Styles

In addition to regular curvy braces `()`, SLN parses curly `{}` and square `[]` brace styles. They are merely meant for providing variety for writing SLN based formats, and are expanded to simple lists during parsing. Some examples:

```
[a b c d]
# expands to
(\[\] a b c d)

{1 2 3 4}
```

(continues on next page)

(continued from previous page)

```
# expands to
(\{\} 1 2 3 4)
```

5.6 List Separators

Both naked and braced lists support a special control character, the list separator `;` (semicolon). Known as statement separator in other languages, it groups atoms into separate lists, and permits to reduce the amount of required parentheses or lines in complex trees.

In addition, it is possible to list-wrap the first element of a list in naked mode by starting the head of the block with `;`.

Here are some examples:

```
# in braced 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 with semicolons in braced mode tho though: if trailing elements aren't terminated with `;`, they're not going to be wrapped:

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

5.7 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.

5.7.1 Single Elements

Special care must be taken when single elements are defined which the user wishes to wrap in a list.

Here is a braced 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 not wrapped:

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

What if we want to just print a newline, passing no arguments?:

```
print            # print
```

The statement above will be ignored because a symbol is resolved but not called. One can make use of the `;` (split-statement) control character, which ends the current list:

```
print;           # (print)
```

5.7.2 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 braced 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

# braced 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))
```

This can be fixed by using the `splice-line` control character, `\`:

```
# 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, and as previously demonstrated, `\` 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
```

5.7.3 Tail Splicing

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

```
# braced
(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)
```

Once again, we can reuse the splice-line control character to get what we want:

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

5.7.4 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)
```

The equivalent naked mode version makes extensive use of list separator and splice-line characters to describe the same tree:

```
# 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 list separator and splice-line:

```
# braced
(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, expressing similar trees in partially braced notation might often be easier on the eyes.

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

CHAPTER 6

Module Reference

6.1 globals

These names are bound in every fresh module and main program by default. Essential symbols are created by the compiler, and subsequent utility functions, macros and types are defined and documented in `core.sc`.

The core module implements the remaining standard functions and macros, parses the command-line and optionally enters the REPL.

definebackslash-char
A constant of type *i8*.

definebarrier-kind-control
A constant of type *i32*.

definebarrier-kind-memory
A constant of type *i32*.

definebarrier-kind-memory-buffer
A constant of type *i32*.

definebarrier-kind-memory-group
A constant of type *i32*.

definebarrier-kind-memory-image
A constant of type *i32*.

definebarrier-kind-memory-shared
A constant of type *i32*.

definecache-dir
A constant of type `String`.

definecompile-flag-O1
A constant of type *u64*.

definecompile-flag-O2
A constant of type *u64*.

definecompile-flag-O3
A constant of type *u64*.

definecompile-flag-cache
A constant of type *u64*.

definecompile-flag-dump-disassembly
A constant of type *u64*.

definecompile-flag-dump-function
A constant of type *u64*.

definecompile-flag-dump-module
A constant of type *u64*.

definecompile-flag-dump-time
A constant of type *u64*.

definecompile-flag-no-debug-info
A constant of type *u64*.

definecompiler-dir
A string containing the folder path to the compiler environment. Typically the compiler environment is the folder that contains the `bin` folder containing the compiler executable.

definecompiler-file-kind-asm
A constant of type *i32*.

definecompiler-file-kind-bc
A constant of type *i32*.

definecompiler-file-kind-llvm
A constant of type *i32*.

definecompiler-file-kind-object
A constant of type *i32*.

definecompiler-path
A string constant containing the file path to the compiler executable.

definecompiler-timestamp
A string constant indicating the time and date the compiler was built.

definedebug-build?
A boolean constant indicating if the compiler was built in debug mode.

definedefault-target-triple
A constant of type `String`.

definee
Euler's number, also known as Napier's constant. Explicitly type-annotated versions of the constant are available as *e:f32* and *e:f64*

definee:f32
See *e*.

definee:f64
See *e*.

definefalse
A constant of type *bool*.

defineglobal-flag-block
A constant of type *u32*.

defineglobal-flag-buffer-block
A constant of type *u32*.

defineglobal-flag-coherent
A constant of type *u32*.

defineglobal-flag-flat
A constant of type *u32*.

defineglobal-flag-non-readable
A constant of type *u32*.

defineglobal-flag-non-writable
A constant of type *u32*.

defineglobal-flag-restrict
A constant of type *u32*.

defineglobal-flag-volatile
A constant of type *u32*.

defineinfinite-range
A *Generator* that iterates through all 32-bit signed integer values starting at 0. This generator does never terminate; when it exceeds the maximum positive integer value of 2147483647, it overflows and continues with the minimum negative integer value of -2147483648.

definelist-handler-symbol
A constant of type *Symbol*.

definenone
A constant of type *Nothing*.

definenull
A constant of type *NullType*.

defineoperating-system
A string constant indicating the operating system the compiler was built for. It equals to `"linux"` for Linux builds, `"windows"` for Windows builds, `"macos"` for macOS builds and `"unknown"` otherwise.

definepi
The number π, the ratio of a circle's circumference C to its diameter d. Explicitly type-annotated versions of the constant are available as *pi:f32* and *pi:f64*.

definepi:f32
See *pi*.

definepi:f64
See *pi*.

definepointer-flag-non-readable
A constant of type *u64*.

definepointer-flag-non-writable
A constant of type *u64*.

definequestion-mark-char
A constant of type *i8*.

defineslash-char
A constant of type *i8*.

definestyle-comment
A constant of type *Symbol*.

definestyle-error
A constant of type *Symbol*.

definestyle-function
A constant of type *Symbol*.

definestyle-instruction
A constant of type *Symbol*.

definestyle-keyword
A constant of type *Symbol*.

definestyle-location
A constant of type *Symbol*.

definestyle-none
A constant of type *Symbol*.

definestyle-number
A constant of type *Symbol*.

definestyle-operator
A constant of type *Symbol*.

definestyle-sfxfunction
A constant of type *Symbol*.

definestyle-string
A constant of type *Symbol*.

definestyle-symbol
A constant of type *Symbol*.

definestyle-type
A constant of type *Symbol*.

definestyle-warning
A constant of type *Symbol*.

definesymbol-handler-symbol
A constant of type *Symbol*.

definetrue
A constant of type *bool*.

definetype-kind-arguments
A constant of type *i32*.

definetype-kind-array
A constant of type *i32*.

definetype-kind-function
A constant of type *i32*.

definetype-kind-image
A constant of type *i32*.

definetype-kind-integer
A constant of type *i32*.

definetype-kind-pointer
A constant of type *i32*.

definetype-kind-qualify
A constant of type *i32*.

definetype-kind-real
A constant of type *i32*.

definetype-kind-sampled-image
A constant of type *i32*.

definetype-kind-tuple
A constant of type *i32*.

definetype-kind-typename
A constant of type *i32*.

definetype-kind-vector
A constant of type *i32*.

definetyped-symbol-handler-symbol
A constant of type *Symbol*.

definetypename-flag-plain
A constant of type *u32*.

defineunknown-anchor
A constant of type *Anchor*.

defineunnamed
A constant of type *Symbol*.

defineunroll-limit
A constant of type *i32* indicating the maximum number of recursions permitted for an inline. When this number is exceeded, an error is raised during typechecking. Currently, the limit is set at 64 recursions. This restriction has been put in place to prevent the compiler from overflowing its stack memory.

definevalue-kind-alloca
A constant of type *i32*.

definevalue-kind-annotate
A constant of type *i32*.

definevalue-kind-argument-list
A constant of type *i32*.

definevalue-kind-argument-list-template
A constant of type *i32*.

definevalue-kind-atomicrmw
A constant of type *i32*.

definevalue-kind-barrier
A constant of type *i32*.

definevalue-kind-binop
A constant of type *i32*.

definevalue-kind-call
A constant of type *i32*.

definevalue-kind-call-template
A constant of type *i32*.

definevalue-kind-cast
A constant of type *i32*.

definevalue-kind-cmpxchg
A constant of type *i32*.

definevalue-kind-compile-stage
A constant of type *i32*.

definevalue-kind-condbr
A constant of type *i32*.

definevalue-kind-const-aggregate
A constant of type *i32*.

definevalue-kind-const-int
A constant of type *i32*.

definevalue-kind-const-pointer
A constant of type *i32*.

definevalue-kind-const-real
A constant of type *i32*.

definevalue-kind-discard
A constant of type *i32*.

definevalue-kind-exception
A constant of type *i32*.

definevalue-kind-execution-mode
A constant of type *i32*.

definevalue-kind-expression
A constant of type *i32*.

definevalue-kind-extract-argument
A constant of type *i32*.

definevalue-kind-extract-argument-template
A constant of type *i32*.

definevalue-kind-extract-element
A constant of type *i32*.

definevalue-kind-extract-value
A constant of type *i32*.

definevalue-kind-fcmp
A constant of type *i32*.

definevalue-kind-free
A constant of type *i32*.

definevalue-kind-function
A constant of type *i32*.

definevalue-kind-get-element-ptr
A constant of type *i32*.

definevalue-kind-global
A constant of type *i32*.

definevalue-kind-icmp
A constant of type *i32*.

definevalue-kind-if
A constant of type *i32*.

definevalue-kind-image-query-levels
A constant of type *i32*.

definevalue-kind-image-query-lod
A constant of type *i32*.

definevalue-kind-image-query-samples
A constant of type *i32*.

definevalue-kind-image-query-size
A constant of type *i32*.

definevalue-kind-image-read
A constant of type *i32*.

definevalue-kind-image-write
A constant of type *i32*.

definevalue-kind-insert-element
A constant of type *i32*.

definevalue-kind-insert-value
A constant of type *i32*.

definevalue-kind-keyed
A constant of type *i32*.

definevalue-kind-keyed-template
A constant of type *i32*.

definevalue-kind-label
A constant of type *i32*.

definevalue-kind-label-template
A constant of type *i32*.

definevalue-kind-load
A constant of type *i32*.

definevalue-kind-loop
A constant of type *i32*.

definevalue-kind-loop-arguments
A constant of type *i32*.

definevalue-kind-loop-label
A constant of type *i32*.

definevalue-kind-loop-label-arguments
A constant of type *i32*.

definevalue-kind-malloc
A constant of type *i32*.

definevalue-kind-merge
A constant of type *i32*.

definevalue-kind-merge-template
A constant of type *i32*.

definevalue-kind-parameter
A constant of type *i32*.

definevalue-kind-parameter-template
A constant of type *i32*.

definevalue-kind-pure-cast
A constant of type *i32*.

definevalue-kind-quote
A constant of type *i32*.

definevalue-kind-raise
A constant of type *i32*.

definevalue-kind-repeat
A constant of type *i32*.

definevalue-kind-return
A constant of type *i32*.

definevalue-kind-sample
A constant of type *i32*.

definevalue-kind-select
A constant of type *i32*.

definevalue-kind-shuffle-vector
A constant of type *i32*.

definevalue-kind-store
A constant of type *i32*.

definevalue-kind-switch
A constant of type *i32*.

definevalue-kind-switch-template
A constant of type *i32*.

definevalue-kind-template
A constant of type *i32*.

definevalue-kind-triop
A constant of type *i32*.

definevalue-kind-undef
A constant of type *i32*.

definevalue-kind-unop
A constant of type *i32*.

definevalue-kind-unquote
A constant of type *i32*.

definevalue-kind-unreachable
A constant of type *i32*.

typeAnchor
A plain type of storage type *_Anchor*.

typeArguments
An opaque type.

spice__typecall (...)

typeBuiltin
A plain type of storage type *u64*.

spice__hash (...)

typeCEnum
An opaque type of supertype *immutable*.

spice__!= (...)

spice__& (...)

spice__* (...)

spice__+ (...)

spice__- (...)

spice__/ (...)

spice__// (...)

spice__< (...)

spice__<= (...)

spice__== (...)

spice__> (...)

spice__>= (...)

spice__^ (...)

spice__imply (...)

inline__neg (*self*)

spice__rimply (...)

spice__| (...)

inline__~ (*self*)

typeCStruct
An opaque type.

spice__drop (...)

spice__getattr (...)

spice__typecall (...)

typeCUnion
An opaque type.

spice__getattr (...)

inline__typecall (*cls* , *value...*)

typeClosure
A plain type of storage type *_Closure*.

spice__!= (...)

spice__== (...)

spice__hash (...)

spice__imply (...)

compiledfndocstring (...)
An external function of type `String<-(Closure).`

typeCollector
A plain type of storage type *_Closure*.

spice__call (...)

inline__typecall (*cls* , *init* , *valid?* , *at* , *collect*)

typeCompileStage
A plain type of storage type `{_Value Anchor}`.

typeError
A plain type of storage type *_Error*.

inlineappend (*self* , *anchor* , *traceback-msg*)

compiledfndump (...)
An external function of type `void<-(Error).`

compiledfnformat (...)
An external function of type `String<-(Error).`

typeGenerator
Generators provide a protocol for iterating the contents of containers and enumerating sequences. They are primarily used by *for* and *fold*, but can also be used separately.

Each generator instance is equivalent to a closure that when called returns four functions:

- A function `state... <- fn start ()` which returns the initial state of the generator as an arbitrary number of arbitrarily typed values. The initially returned state defines the format of the generators internal state.
- A function `bool <- fn valid? (state...)` which takes the current generator state and returns *true* when the generator can resolve the state to a collection item, otherwise *false*, indicating that the generator has been depleted.
- A function `value... <- fn at (state...)` which takes the current generator state and returns the collection item this state maps to. The function may not be called for a state for which `valid?` has reported to be depleted.
- A function `state... <- fn next (state...)` which takes the current generator state and returns the state mapping to the next item in the collection. The new state must have the same type signature as the previous state. The function may not be called for a state for which `valid?` has reported to be depleted.

It is allowed to call any of these functions multiple times with any valid state, effectively restarting the Generator at an arbitrary point, as Generators are not expected to have side effects. In controlled circumstances a Generator may choose to be impure, but should be documented accordingly.

Here is a typical pattern for constructing a generator:

```
inline make-generator (container)
    Generator
        inline "start" ()
            # return the first iterator of sequence (might not be valid)
            'start container
        inline "valid?" (it...)
            # return true if the iterator is still valid
            'valid-iterator? container it...
        inline "at" (it...)
```

(continues on next page)

(continued from previous page)

```
            # return variadic result at iterator
            '@ container it...
        inline "next" (it...)
            # return the next iterator in sequence
            'next container it...
```

The generator can then be subsequently used like this:

```
# this example prints up to two elements returned by a generator
# generate a new instance bound to container
let gen = (make-generator container)
# extract all methods
let start valid? at next = (gen)
# get the init state
let state... = (start)
# check if the state is valid
if (valid? state...)
    # container has at least one item; print it
    print (at state...)
    # advance to the next state
    let state... = (next state...)
    if (valid? state...)
        # container has one more item; print it
        print (at state...)
# we are done; no cleanup necessary
```

spice__call (*self*)
Returns, in this order, the four functions `start`, `valid?`, `init` and `next` which are required to enumerate generator `self`.

inline__typecall (*cls* , *start* , *valid?* , *at* , *next*)
Takes four functions `start`, `valid?`, `at` and `next` and returns a new generator ready for use.

typeImage
An opaque type.

spice__typecall (...)

compiledfntype (...)
An external function of type `type<-(type Symbol i32 i32 i32 i32 Symbol Symbol)`.

typeNothing
A plain type of storage type `{}`.

inline__tobool ()

typeNullType
A plain type of storage type *void*.

spice__== (...)

spice__imply (...)

spice__r== (...)

inline__repr (*self*)

typeOverloadedFunction
An opaque type.

spice__typecall (...)

spiceappend (...)

typeQualify
An opaque type.

typeRaises
An opaque type.

typeSampledImage
An opaque type.

spice__typecall (...)

compiledfntype (...)
An external function of type `type<-(type)`.

typeSampler
An opaque type.

typeScope
A plain type of storage type *_Scope*.

compiledfn@ (...)
An external function of type `Value<-Error(Scope Value)`.

spice__.. (...)

spice__== (...)

spice__as (...)

spice__getattr (...)

spice__hash (...)

spice__typecall (...)

spicebind (...)

inlinebind-symbols (*self*, *values...*)

compiledfnbind-with-docstring (...)
An external function of type `Scope<-(Scope Value Value String)`.

spicedefine (...)

inlinedefine-symbols (*self*, *values...*)

inlinedeleted (*self*)

compiledfndocstring (...)
An external function of type `String<-(Scope Value)`.

inlinelineage (*self*)

compiledfnlocal@ (...)
An external function of type `Value<-Error(Scope Value)`.

compiledfnmodule-docstring (...)
An external function of type `String<-(Scope)`.

compiledfnnext (...)
An external function of type `λ(Value Value i32)<-(Scope i32)`.

compiledfnnext-deleted (...)
An external function of type `λ(Value i32)<-(Scope i32)`.

compiledfnparent (...)
An external function of type `Scope<-(Scope)`.

compiledfnreparent (...)
An external function of type `Scope<-(Scope Scope)`.

compiledfnunbind (...)
An external function of type `Scope<-(Scope Value)`.

compiledfnunparent (...)
An external function of type `Scope<-(Scope)`.

typeSourceFile
A plain type of storage type *_SourceFile*.

typeSpiceMacro
A plain type of storage type `Value`.

spice__rimply (...)

typeSpiceMacroFunction
A plain type labeled `Value<->Error(Value)<*>` of supertype *pointer* and of storage type `Value`.

typeStruct
An opaque type.

spice__drop (...)

builtin__getattr (...)

spice__typecall (...)

typeSugarMacro
A plain type of storage type λ`(List Scope)`.

spice__call (...)

typeSugarMacroFunction
A plain type labeled λ`(List Scope)<->Error(List Scope)<*>` of supertype *pointer* and of storage type λ`(List Scope)`.

typeSymbol
A plain type of supertype *immutable* and of storage type *u64*.

spice__!= (...)

spice__== (...)

spice__as (...)

spice__call (...)

spice__hash (...)

inline__typecall (*cls* , *str*)

inlineunique (*cls* , *name*)

compiledfnvariadic? (...)
An external function of type `bool<-(Symbol)`.

typeTypeArrayPointer
A plain type labeled `type(*)` of supertype *pointer* and of storage type `type(*)`.

typeTypeInitializer
An opaque type.

inline__static-imply (*cls* , *T*)

typeUnknown
A plain type of storage type *_type*.

typeValue
A plain type of storage type `{_Value Anchor}`.

spice__== (...)

inline__as (*vT* , *T*)

compiledfn__repr (...)
An external function of type `String<-(Value).`

inline__rimply (*vT* , *T*)

spice__typecall (...)

compiledfnanchor (...)
An external function of type `Anchor<-(Value).`

compiledfnargcount (...)
An external function of type `i32<-(Value).`

inlinearglist-sink (*N*)

inlineargs (*self*)

compiledfnconstant? (...)
An external function of type `bool<-(Value).`

fndekey (*self*)

inlinedump (*self*)

compiledfngetarg (...)
An external function of type `Value<-(Value i32).`

compiledfngetarglist (...)
An external function of type `Value<-(Value i32).`

compiledfnkind (...)
An external function of type `i32<-(Value).`

compiledfnnone? (...)
A compiled function of type `bool<-(Value).`

compiledfnpure? (...)
An external function of type `bool<-(Value).`

compiledfnqualified-typeof (...)
An external function of type `type<-(Value).`

inlinereverse-args (*self*)

compiledfnspice-repr (...)
An external function of type `String<-(Value).`

inlinetag (*self* , *anchor*)

compiledfntypeof (...)
An external function of type `type<-(Value).`

typeValueArrayPointer
A plain type labeled `Value(*)` of supertype *pointer* and of storage type `Value(*)`.

typeVariadic
An opaque type labeled `...`.

typeaggregate
An opaque type.

spice__drop (...)

typearray
An opaque type of supertype *aggregate*.

inline__@ (*self* , *index*)

spice__as (...)

spice__countof (...)

spice__typecall (...)

spice__unpack (...)

inlinetype (*element-type* , *size*)

typebool
A plain type of supertype *integer* and of storage type *bool*.

typeconstant
An opaque type.

typef128
A plain type of supertype *real* and of storage type *f128*.

typef16
A plain type of supertype *real* and of storage type *f16*.

typef32
A plain type of supertype *real* and of storage type *f32*.

typef64
A plain type of supertype *real* and of storage type *f64*.

typef80
A plain type of supertype *real* and of storage type *f80*.

typefunction
An opaque type.

spice__typecall (...)

spicetype (...)

typehash
A plain type of storage type *u64*.

spice__!= (...)

spice__== (...)

spice__as (...)

inline__hash (*self*)

spice__ras (...)

spice__typecall (...)

inlinefrom-bytes (*data* , *size*)

typei16
A plain type of supertype *integer* and of storage type *i16*.

typei32
A plain type of supertype *integer* and of storage type *i32*.

typei64
A plain type of supertype *integer* and of storage type *i64*.

typei8
A plain type of supertype *integer* and of storage type *i8*.

typeimmutable
An opaque type.

typeincomplete
An opaque type.

typeinteger
An opaque type of supertype *immutable*.

spice__!= (...)

spice__% (...)

spice__& (...)

spice__* (...)

spice__** (...)

spice__+ (...)

spice__- (...)

spice__/ (...)

spice__// (...)

spice__< (...)

spice__<< (...)

spice__<= (...)

spice__== (...)

spice__> (...)

spice__>= (...)

spice__>> (...)

spice__^ (...)

spice__as (...)

spice__hash (...)

spice__imply (...)

inline__neg (*self*)

inline__rcp (*self*)

spice__static-imply (...)

spice__tobool (...)

spice__typecall (...)

builtin__vector!= (...)

spice__vector% (...)

builtin__vector& (...)

builtin__vector* (...)

builtin__vector+ (...)

builtin__vector- (...)

spice__vector// (...)

spice__vector< (...)

builtin__vector<< (...)

spice__vector<= (...)

builtin__vector== (...)

spice__vector> (...)

spice__vector>= (...)

spice__vector>> (...)

builtin__vector^ (...)

builtin__vector| (...)

spice__| (...)

inline__~ (*self*)

typeintptr
A plain type labeled `u64` of supertype *integer* and of storage type *u64*.

typelist
A plain type labeled `List` of storage type *_List*.

compiledfn@ (...)
An external function of type `Value<-(List)`.

spice__.. (...)

spice__== (...)

spice__as (...)

spice__countof (...)

inline__repr (*self*)

spice__typecall (...)

spice__unpack (...)

inlinecons-sink (*self*)

spicedecons (...)

compiledfndump (...)
An external function of type `List<-(List)`.

compiledfnjoin (...)
An external function of type `List<-(List List)`.

compiledfnnext (...)
An external function of type `List<-(List)`.

compiledfnreverse (...)
An external function of type `List<-(List)`.

fnrjoin (*lside* , *rside*)

fntoken-split (*expr* , *token* , *errmsg*)

typenodefault
An opaque type.

typenoreturn
An opaque type.

typeopaquepointer
An opaque type.

typepackage
A symbol table of type *Scope* which holds configuration options and module contents. It is managed by the module import system.

`package.path` holds a list of all search paths in the form of simple string patterns. Changing it alters the way modules are searched for in the next run stage.

`package.modules` is another scope symbol table mapping full module paths to their contents. When a module is first imported, its contents are cached in this table. Subsequent imports of the same module will be resolved to these cached contents.

typepointer
An opaque type.

spice__== (...)

inline__@ (*self* , *index*)

spice__as (...)

spice__call (...)

inline__getattr (*self* , *key*)

spice__hash (...)

spice__imply (...)

inline__toref (*self*)

spice__typecall (...)

inlinetype (*T*)

typerawstring
A plain type labeled `i8(*)` of supertype *pointer* and of storage type `i8(*)`.

typereal
An opaque type of supertype *immutable*.

spice__!= (...)

spice__% (...)

spice__* (...)

spice__** (...)

spice__+ (...)

spice__- (...)

spice__/ (...)

spice__// (...)

spice__< (...)

spice__<= (...)

spice__== (...)

spice__> (...)

spice__>= (...)

spice__as (...)

spice__hash (...)

spice__imply (...)

inline__neg (*self*)

inline__rcp (*self*)

inline__tobool (*self*)

inline__typecall (*cls* , *value*)

builtin__vector!= (...)

builtin__vector% (...)

builtin__vector* (...)

builtin__vector** (...)

builtin__vector+ (...)

builtin__vector- (...)

builtin__vector/ (...)

builtin__vector< (...)

builtin__vector<= (...)

builtin__vector== (...)

builtin__vector> (...)

builtin__vector>= (...)

typestring

A plain type labeled `String` of supertype *opaquepointer* and of storage type *_String*.

spice__!= (...)

spice__.. (...)

spice__< (...)

spice__<= (...)

spice__== (...)

spice__> (...)

spice__>= (...)

fn__@ (*self* , *i*)

spice__as (...)

compiledfn__countof (...)
An external function of type `usize<-(String)`.

inline__hash (*self*)

spice__imply (...)

compiledfn__lslice (...)
An external function of type `String<-(String usize)`.

spice__ras (...)

compiledfn__rslice (...)
An external function of type `String<-(String usize)`.

compiledfnbuffer (...)
An external function of type `λ(i8(*) usize)<-(String)`.

inlinecollector (*maxsize*)

compiledfnjoin (...)
An external function of type `String<-(String String)`.

compiledfnmatch? (...)
An external function of type `λ(bool i32 i32)<-Error(String String)`.

inlinerange (*self* , *start* , *end*)

typetuple
An opaque type of supertype *aggregate*.

builtin__@ (...)

spice__countof (...)

builtin__getattr (...)

spice__typecall (...)

spice__unpack (...)

spicepacked (...)

spicepacked-type (...)

spicetype (...)

typetype
A plain type of supertype *opaquepointer* and of storage type *_type*.

compiledfn@ (...)
An external function of type `Value<-Error(type Symbol)`.

spice__!= (...)

spice__< (...)

spice__<= (...)

spice__== (...)

spice__> (...)

spice__>= (...)

compiledfn__@ (...)
An external function of type `type<->Error(type i32)`.

spice__call (...)

spice__countof (...)

spice__getattr (...)

spice__hash (...)

spice__toptr (...)

inline__toref (*self*)

compiledfnalignof (...)
An external function of type `usize<->Error(type)`.

compiledfnbitcount (...)
An external function of type `i32<-(type)`.

fnchange-element-type (*cls* , *ET*)

fnchange-storage-class (*cls* , *storage-class*)

spicedefine-symbol (...)

inlinedefine-symbols (*self* , *values...*)

spicedispatch-attr (...)

compiledfndocstring (...)
An external function of type `String<-(type Symbol)`.

compiledfnelement-count (...)
An external function of type `i32<->Error(type)`.

compiledfnelement@ (...)
An external function of type `type<->Error(type i32)`.

inlineelements (*self*)

fnfunction-pointer? (*cls*)

fnfunction? (*cls*)

fnimmutable (*cls*)

inlinekey-type (*self* , *key*)

compiledfnkeyof (...)
An external function of type `λ(Symbol type)<-(type)`.

compiledfnkind (...)
An external function of type `i32<-(type)`.

compiledfnlocal@ (...)
An external function of type `Value<->Error(type Symbol)`.

fnmutable (*cls*)

fnmutable& (*cls*)

compiledfnoffsetof (...)
An external function of type `usize<->Error(type i32)`.

compiledfnopaque? (...)
An external function of type `bool<-(type)`.

compiledfnplain? (...)
An external function of type `bool<-(type)`.

fnpointer->refer-type (*cls*)

fnpointer-storage-class (*cls*)

fnpointer? (*cls*)

spiceraises (...)

fnreadable? (*cls*)

fnrefer->pointer-type (*cls*)

compiledfnrefer? (...)
An external function of type `bool<-(type)`.

compiledfnreturn-type (...)
An external function of type λ`(type type)<-(type)`.

compiledfnset-docstring (...)
An external function of type `void<-(type Symbol String)`.

inlineset-opaque (*type*)

inlineset-plain-storage (*type* , *storage-type*)

inlineset-storage (*type* , *storage-type*)

spiceset-symbol (...)

inlineset-symbols (*self* , *values...*)

compiledfnsigned? (...)
An external function of type `bool<-(type)`.

compiledfnsizeof (...)
An external function of type `usize<->Error(type)`.

compiledfnstorageof (...)
An external function of type `type<->Error(type)`.

compiledfnstring (...)
An external function of type `String<-(type)`.

fnstrip-pointer-storage-class (*cls*)

compiledfnstrip-qualifiers (...)
An external function of type `type<-(type)`.

compiledfnsuperof (...)
An external function of type `type<-(type)`.

inlinesymbols (*self*)

compiledfnunique-type (...)
An external function of type `type<-(type i32)`.

compiledfnvariadic? (...)
An external function of type `bool<-(type)`.

inlineview-type (*self* , *id*)

fnwritable? (*cls*)

typetypename
An opaque type.

spice__!= (...)

spice__= (...)

spice__methodcall (...)

spice__toptr (...)

spice__typecall (...)

compiledfntype (...)
An external function of type `type<->Error(String type)`.

typeu16
A plain type of supertype *integer* and of storage type *u16*.

typeu32
A plain type of supertype *integer* and of storage type *u32*.

typeu64
A plain type of supertype *integer* and of storage type *u64*.

typeu8
A plain type of supertype *integer* and of storage type *u8*.

typeunion
An opaque type.

typeusize
A plain type of supertype *integer* and of storage type *u64*.

typevector
An opaque type of supertype *immutable*.

spice__!= (...)

spice__% (...)

spice__& (...)

spice__* (...)

spice__** (...)

spice__+ (...)

spice__- (...)

spice__/ (...)

spice__// (...)

spice__< (...)

spice__<< (...)

spice__<= (...)

spice__== (...)

spice__> (...)

spice__>= (...)

spice__>> (...)

inline__@ (*self* , *index*)

spice__^ (...)

spice__countof (...)

spice__lslice (...)

spice__rslice (...)

spice__typecall (...)

spice__unpack (...)

spice__| (...)

spicesmear (...)

inlinetype (*element-type* , *size*)

typevoid
An opaque type of supertype *Arguments*.

typevoidstar
A plain type labeled `void<*>` of supertype *pointer* and of storage type *void*.

inline%= (*lhs* , *rhs*)

inline&= (*lhs* , *rhs*)

inline*= (*lhs* , *rhs*)

inline+= (*lhs* , *rhs*)

inline-= (*lhs* , *rhs*)

inline..= (*lhs* , *rhs*)

inline//= (*lhs* , *rhs*)

inline/= (*lhs* , *rhs*)

inline<<= (*lhs* , *rhs*)

inline>>= (*lhs* , *rhs*)

inline^= (*lhs* , *rhs*)

inline|= (*lhs* , *rhs*)

fnValue-none? (*value*)

inlineaggregate-type-constructor (*start* , *f*)

fnall? (*v*)

fnany? (*v*)

fnas-converter (*vT* , *T* , *static?*)

fnautoboxer (*T* , *x*)

inlinebalanced-binary-op-dispatch (*symbol* , *rsymbol* , *friendly-op-name*)

fnbalanced-binary-operation (*args* , *symbol* , *rsymbol* , *friendly-op-name*)

fnbalanced-binary-operator (*symbol* , *rsymbol* , *lhsT* , *rhsT* , *lhs-static?* , *rhs-static?*)
for an operation performed on two argument types, of which either type can provide a suitable candidate, return a matching operator. This function only works inside a spice macro.

inlinebalanced-lvalue-binary-op-dispatch (*symbol* , *friendly-op-name*)

fnbalanced-lvalue-binary-operation (*args* , *symbol* , *friendly-op-name*)

fnbalanced-lvalue-binary-operator (*symbol* , *lhsT* , *rhsT* , *rhs-static?*)
for an operation performed on two argument types, of which only the left type type can provide a suitable candidate, return a matching operator. This function only works inside a spice macro.

fnbin (*value*)

fnbinary-op-error (*friendly-op-name* , *lhsT* , *rhsT*)

fnbinary-operator (*symbol* , *lhsT* , *rhsT*)
for an operation performed on two argument types, of which only the left type can provide a suitable candidate, find a matching operator function. This function only works inside a spice macro.

fnbinary-operator-r (*rsymbol* , *lhsT* , *rhsT*)
for an operation performed on two argument types, of which only the right type can provide a suitable candidate, find a matching operator function. This function only works inside a spice macro.

fnbox-integer (*value*)

fnbox-pointer (*value*)

inlinebox-spice-macro (*l*)

fnbox-symbol (*value*)

fnbuild-typify-function (*f*)

fncast-converter (*symbol* , *rsymbol* , *vT* , *T*)
for two given types, find a matching conversion function this function only works inside a spice macro

inlinecast-error (*intro-string* , *vT* , *T*)

fncheck-count (*count* , *mincount* , *maxcount*)

inlineclamp (*x* , *mn* , *mx*)

fnclone-scope-contents (*a* , *b*)
Join two scopes `a` and `b` into a new scope so that the root of `a` descends from `b`.

fncompare-type (*args* , *f*)

inlineconvert-assert-args (*args* , *cond* , *msg*)

fndec (*value*)

inlinedefer-type (...)

fndispatch-and-or (*args* , *flip*)

fndots-to-slashes (*pattern*)

fndotted-symbol? (*env* , *head*)

inlineempty? (*value*)

inlineenumerate (*x*)

fnerror (*msg*)

fnerror@ (*anchor* , *traceback-msg* , *error-msg*)

usage example:: error@ (‘anchor value) “while checking parameter” “error in value”

fnerror@+ (*error* , *anchor* , *traceback-msg*)

usage example::

except (err) error@+ err (‘anchor value) “while processing stream”

fnexec-module (*expr* , *eval-scope*)

fnexpand-and-or (*expr* , *f*)

fnexpand-apply (*expr*)

fnexpand-define (*expr*)

fnexpand-define-infix (*args* , *scope* , *order*)

fnexpand-infix-let (*expr*)

inlineextern-new (*name* , *T* , *attrs...*)

fnextract-integer (*value*)

fnextract-name-params-body (*expr*)

fnextract-single-arg (*args*)

fnextract-single-type-arg (*args*)

inlinefloordiv (*a* , *b*)

inlinefunction->SpiceMacro (*f*)

inlinegen-allocator-sugar (*name* , *copyf* , *newf*)

inlinegen-cast-op (*f* , *str*)

inlinegen-cast? (*converterf*)

inlinegen-match-block-parser (*handle-case*)

fngen-match-matcher (*failfunc* , *expr* , *scope* , *cond*)

features: <constant> -> (input == <constant>) (or <expr_a> <expr_b>) -> (or <expr_a> <expr_b>)

TODO: (: x T) -> ((typeof input) == T), let x = input <unknown symbol> -> unpack as symbol

fngen-or-matcher (*failfunc* , *expr* , *scope* , *params*)

fngen-sugar-matcher (*failfunc* , *expr* , *scope* , *params*)

fngen-vector-reduction (*f* , *v* , *sz*)

fnget-ifx-op (*env* , *op*)

fnget-ifx-symbol (*name*)

fnhas-infix-ops? (*infix-table* , *expr*)

fnhex (*value*)

fnimply-converter (*vT* , *T* , *static?*)

inlineinfix-op (*pred*)

fninfix-op-ge (*infix-table* , *token* , *prec*)

fninfix-op-gt (*infix-table* , *token* , *prec*)

fninteger->string (*value* , *base*)

fninteger-as (*vT* , *T*)

fninteger-imply (*vT* , *T*)

fninteger-static-imply (*vT* , *T*)

fninteger-tobool (*args*)

fnload-module (*module-name* , *module-path* , *opts...*)

fnltr-multiop (*args* , *target* , *mincount*)

inlinemake-const-type-property-function (*func*)

inlinemake-const-value-property-function (*func*)

inlinemake-expand-and-or (*f*)

inlinemake-expand-define-infix (*order*)

inlinemake-inplace-let-op (*op*)

inlinemake-inplace-op (*op*)

fnmake-module-path (*pattern* , *name*)

inlinemake-unpack-function (*extractf*)

inlinememo (*f*)

inlinememoize (*f*)

fnmerge-scope-symbols (*source* , *target* , *filter*)

fnnext-head? (*next*)

fnnodefault? (*x*)

fnoct (*value*)

fnoperator-valid? (*value*)

fnpatterns-from-namestr (*base-dir* , *namestr*)

fnpointer-as (*vT* , *T*)

fnpointer-imply (*vT* , *T*)

fnpointer-type-imply? (*src* , *dest*)

fnpowi (*base* , *exponent*)

inlineprint (*values...*)

fnptrcmp!= (*t1* , *t2*)

fnptrcmp== (*t1* , *t2*)

inlinequasiquote-any (*x*)

fnquasiquote-list (*x*)

inlinerange (*a* , *b* , *c*)

fnreal-as (*vT* , *T*)

fnreal-imply (*vT* , *T*)

fnrequire-from (*base-dir* , *name*)

inlinerrange (*a* , *b* , *c*)
same as range, but iterates range in reverse; arguments are passed in the same format, so rrange can act as a drop-in replacement for range.

fnrtl-infix-op-eq (*infix-table* , *token* , *prec*)

fnrtl-multiop (*args* , *target* , *mincount*)

inlineruntime-aggregate-type-constructor (*f*)

inlinesafe-integer-cast (*self* , *T*)

fnsc_argument_list_join (*a* , *b*)

fnsc_argument_list_join_values (*a* , *b...*)

inlinesc_argument_list_map_filter_new (*maxN* , *mapf*)

inlinesc_argument_list_map_new (*N* , *mapf*)

inlineselect-op-macro (*sop* , *uop* , *fop* , *numargs*)

inlinesigned-vector-binary-op (*sf* , *uf*)

inlinesimple-binary-op (*f*)
for cases where the type only interacts with itself

inlinesimple-folding-autotype-binary-op (*f* , *unboxer*)

inlinesimple-folding-autotype-signed-binary-op (*sf* , *uf* , *unboxer*)

inlinesimple-folding-binary-op (*f* , *unboxer* , *boxer*)

inlinesimple-folding-signed-binary-op (*sf* , *uf* , *unboxer* , *boxer*)

inlinesimple-signed-binary-op (*sf* , *uf*)

inlineslice (*value* , *start* , *end*)

inlinespice-binary-op-macro (*f*)
to be used for binary operators of which either type can provide an operation. returns a callable operator (f lhs rhs) that performs the operation or no arguments if the operation can not be performed.

inlinespice-cast-macro (*f*)
to be used for __as, __ras, __imply and __rimply returns a callable converter (f value) that performs the cast or no arguments if the cast can not be performed.

inlinespice-converter-macro (*f*)
to be used for converter that need to do additional dispatch, e.g. do something else when the value is a constant returns a quote that performs the cast (f value T)

inlinespice-macro (*l*)

fnsplit-dotted-symbol (*name*)

fnstring@ (*self* , *i*)

inlinesugar-block-scope-macro (*f*)

inlinesugar-macro (*f*)

inlinesugar-scope-macro (*f*)

fnswap (*a* , *b*)
safely exchanges the contents of two references

inlinetype-comparison-func (*f*)

inlinetype-factory (*f*)

inlinetypeinit (*...*)

inlineunary-op-dispatch (*symbol* , *friendly-op-name*)

fnunary-op-error (*friendly-op-name* , *T*)

fnunary-operation (*args* , *symbol* , *friendly-op-name*)

fnunary-operator (*symbol* , *T*)
for an operation performed on one variable argument type, find a matching operator function. This function only works inside a spice macro.

inlineunary-or-balanced-binary-op-dispatch (*usymbol* , *ufriendly-op-name* , *symbol* , *rsymbol* , *friendly-op-name*)

fnunary-or-balanced-binary-operation (*args* , *usymbol* , *ufriendly-op-name* , *symbol* , *rsymbol* , *friendly-op-name*)

inlineunary-or-unbalanced-binary-op-dispatch (*usymbol* , *ufriendly-op-name* , *symbol* , *rtype* , *friendly-op-name*)

fnunary-or-unbalanced-binary-operation (*args* , *usymbol* , *ufriendly-op-name* , *symbol* , *rtype* , *friendly-op-name*)

inlineunbalanced-binary-op-dispatch (*symbol* , *rtype* , *friendly-op-name*)

fnunbalanced-binary-operation (*args* , *symbol* , *rtype* , *friendly-op-name*)

inlineunbox (*value* , *T*)

inlineunbox-integer (*value* , *T*)

inlineunbox-pointer (*value* , *T*)

inlineunbox-symbol (*value* , *T*)

fnunbox-verify (*value* , *wantT*)

fnuncomma (*l*)
uncomma list l, wrapping all comma separated symbols as new lists example:

```
(uncomma '(a , b c d , e f , g h)) -> '(a (b c d) (e f) (g h))
```

fnunpack-infix-op (*op*)

fnunpack2 (*args*)

inlineva-join (*a...*)

fnvalue-as (*vT* , *T* , *expr*)

inlinevector-binary-op-dispatch (*symbol*)

fnvector-binary-operator (*symbol* , *lhsT* , *rhsT*)

fnverify-count (*count* , *mincount* , *maxcount*)

sugar(. ...)

sugar(:: ...)

sugar(:= ...)

sugar(<- ...)

sugar(@@ ...)

sugar(**and** ...)

sugar(**as:=** ...)

sugar(**assert** ...)

sugar(**bind** ...)

sugar(**chain-typed-symbol-handler** ...)

sugar(**decorate-fn** ...)

sugar(**decorate-inline** ...)

sugar(**decorate-let** ...)

sugar(**decorate-struct** ...)

sugar(**decorate-typedef** ...)

sugar(**decorate-vvv** ...)

sugar(**define** ...)

sugar(**define-infix<** ...)

sugar(**define-infix>** ...)

sugar(**define-sugar-block-scope-macro** ...)

sugar(**define-sugar-macro** ...)

sugar(**define-sugar-scope-macro** ...)

sugar(**fn...** ...)

sugar(**fold**) *state* , ... **=** , *init...*
for , *name* , ... **in** , *gen* , *body...* This is a combination of the *loop* and *for* forms. It enumerates all elements in collection or sequence `gen`, unpacking each element and binding its arguments to the names defined by `name ...`, while the loop state `state ...` is initialized from `init...`.

Similar to *loop*, the body expression must return the next state of the loop. The state of `gen` is transparently maintained and does not have to be managed.

Unlike *for*, *fold* requires both calls to `break` and `continue` to pass a state compatible with `state ...`. Otherwise they serve the same function.

Usage example:

```
# add numbers from 0 to 9, skipping number 5, and print the result
print
    fold (sum = 0) for i in (range 100)
        if (i == 10)
            # abort the loop
            break sum
        if (i == 5)
            # skip this index
            continue sum
        # continue with the next state for sum
        sum + i
```

sugar(**fold-locals** ...)

sugar(**for** *name* , ... **in** , *gen* , *body...*)

Defines a loop that enumerates all elements in collection or sequence `gen`, unpacking each element and binding its arguments to the names defined by `name ...`.

`gen` must either be of type *Generator* or provide a cast to *Generator*.

Within the loop body, special forms `break` and `continue` can be used to abort the loop early or skip ahead to the next element. The loop will always evaluate to no arguments.

For a loop form that permits you to maintain additional state and break with a value, see *fold*.

Usage example:

```
# print numbers from 0 to 9, skipping number 5
for i in (range 100)
    if (i == 10)
        # abort the loop
        break;
    if (i == 5)
        # skip this index
        continue;
    print i
```

sugar(**from** ...)

sugar(**global** ...)

sugar(**import** ...)

sugar(**include** ...)

sugar(**inline...** ...)

sugar(**local** ...)

sugar(**locals** ...)
Export locals as a chain of up to two new scopes: a scope that contains all the constant values in the immediate scope, and a scope that contains the runtime values. If all values in the scope are constant, then the resulting scope will also be constant.

sugar(**match** ...)

sugar(**or** ...)

sugar(**qq** ...)

sugar(**spice** ...)

sugar(**static-assert** ...)

sugar(**static-if** ...)

sugar(**static-match** ...)

sugar(**sugar** ...)

sugar(**sugar-eval** ...)

sugar(**sugar-if** ...)

sugar(**sugar-match** ...)

sugar(**sugar-set-scope!** ...)

sugar(**typedef** ...)
a type declaration syntax; when the name is a string, the type is declared at runtime.

sugar(**typedef+** ...)

sugar(**unlet** ...)

sugar(**using** ...)

sugar(**va-option** ...)

sugar(**vvv** ...)

sugar(**while** ...)

builtin? (...)

builtin_ (...)

builtinImage-query-levels (...)

builtinImage-query-lod (...)

builtinImage-query-samples (...)

builtinImage-query-size (...)

builtinImage-read (...)

builtinImage-texel-pointer (...)

builtinImage-write (...)

builtinacos (...)

builtinacosh (...)

builtinadd (...)

builtinadd-nsw (...)

builtinadd-nuw (...)

builtinalloca (...)

builtinalloca-array (...)

builtinashr (...)

builtinasin (...)

builtinasinh (...)

builtinassign (...)

builtinatan (...)

builtinatan2 (...)

builtinatanh (...)

builtinatomic (...)

builtinatomicrmw (...)

builtinband (...)

builtinbitcast (...)

builtinbnand (...)

builtinbor (...)

builtinbranch (...)

builtinbreak (...)

builtinbxor (...)

builtincall (...)

builtinceil (...)

builtincmpxchg (...)

builtincopy (...)

builtincos (...)

builtincosh (...)

builtincross (...)

builtindegrees (...)

builtinderef (...)

builtindiscard (...)

builtindistance (...)

builtindo (...)

builtindropped? (...)

builtindump (...)

builtindump-debug (...)

builtindump-spice (...)

builtindump-template (...)

builtindump-uniques (...)

builtindupe (...)

builtinembed (...)

builtinexp (...)

builtinexp2 (...)

builtinextractelement (...)

builtinextractvalue (...)

builtinfabs (...)

builtinfadd (...)

builtinfcmp!=o (...)

builtinfcmp!=u (...)

builtinfcmp-ord (...)

builtinfcmp-uno (...)

builtinfcmp<=o (...)

builtinfcmp<=u (...)

builtinfcmp<o (...)

builtinfcmp<u (...)

builtinfcmp==o (...)

builtinfcmp==u (...)

builtinfcmp>=o (...)

builtinfcmp>=u (...)

builtinfcmp>o (...)

builtinfcmp>u (...)

builtinfdiv (...)

builtinfloor (...)

builtinfma (...)

builtinfmix (...)

builtinfmul (...)

builtinfn (...)

builtinfpext (...)

builtinfptosi (...)

builtinfptoui (...)

builtinfptrunc (...)

builtinfract (...)

builtinfree (...)

builtinfrem (...)

builtinfrexp (...)

builtinfsign (...)

builtinfsub (...)

builtingetelementptr (...)

builtingetelementref (...)

builtinhide-traceback (...)

builtinicmp!= (...)

builtinicmp<=s (...)

builtinicmp<=u (...)

builtinicmp<s (...)

builtinicmp<u (...)

builtinicmp== (...)

builtinicmp>=s (...)

builtinicmp>=u (...)

builtinicmp>s (...)

builtinicmp>u (...)

builtinif (...)

builtinindirect-let (...)

builtininline (...)

builtininsertelement (...)

builtininsertvalue (...)

builtininttoptr (...)

builtininversesqrt (...)

builtinitrunc (...)

builtinlabel (...)

builtinldexp (...)

builtinlength (...)

builtinlet (...)

builtinload (...)

builtinlog (...)

builtinlog2 (...)

builtinloop (...)

builtinlose (...)

builtinlshr (...)

builtinmalloc (...)

builtinmalloc-array (...)

builtinmerge (...)

builtinmove (...)

builtinmul (...)

builtinmul-nsw (...)

builtinmul-nuw (...)

builtinnormalize (...)

builtinnullof (...)

builtinpowf (...)

builtinptrtoint (...)

builtinptrtoref (...)

builtinradians (...)

builtinraise (...)

builtinraising (...)

builtinrawcall (...)

builtinreftoptr (...)

builtinrepeat (...)

builtinreturn (...)

builtinreturning (...)

builtinround (...)

builtinroundeven (...)

builtinrun-stage (...)

builtinsample (...)

builtinsdiv (...)

builtinset-execution-mode (...)

builtinsext (...)

builtinshl (...)

builtinshufflevector (...)

builtinsin (...)

builtinsinh (...)

builtinsitofp (...)

builtinsmax (...)

builtinsmin (...)

builtinsmoothstep (...)

builtinspice-quote (...)

builtinspice-unquote (...)

builtinspice-unquote-arguments (...)

builtinsqrt (...)

builtinsquare-list (...)

builtinsrem (...)

builtinssign (...)

builtinstep (...)

builtinstore (...)

builtinsub (...)

builtinsub-nsw (...)

builtinsub-nuw (...)

builtinsugar-log (...)

builtinsugar-quote (...)

builtinswapvalue (...)

builtinswitch (...)

builtintan (...)

builtintanh (...)

builtintrunc (...)

builtintry (...)

builtintypeof (...)

builtinudiv (...)

builtinuitofp (...)

builtinumax (...)

builtinumin (...)

builtinundef (...)

builtinunique-visible? (...)

builtinunreachable (...)

builtinurem (...)

builtinva-countof (...)

builtinview (...)

builtinviewing (...)

builtinvolatile (...)

builtinvolatile-load (...)

builtinvolatile-store (...)

builtinxchg (...)

builtinzext (...)

spice% (...)

spice& (...)

spice* (...)

spice+ (...)

spice- (...)

spice/ (...)

spice< (...)

spice= (...)

spice> (...)

spice@ (...)

spice^ (...)

spice| (...)

spice~ (...)

spice!= (...)

spice&? (*value*)
Returns *true* if *value* is a reference, otherwise *false*.

spice** (...)

spice.. (...)

spice// (...)

spice<< (...)

spice<= (...)

spice== (...)

spice>= (...)

spice>> (...)

spice_static-compile (...)

spice_static-compile-glsl (...)

spice_static-compile-spirv (...)

spiceClosure->Collector (...)

spiceClosure->Generator (...)

spiceabs (...)

spicealignof (...)

spiceand-branch (...)
The type of the *null* constant. This type is uninstantiable.

spiceappend-to-scope (...)

spiceappend-to-type (...)

spiceargumentsof (...)

spicearrayof (...)

spiceas (...)

spiceas? (...)

spicebindingof (...)

spicebitcountof (...)

spicecoerce-call-arguments (...)

spicecons (...)

spiceconst.add.i32.i32 (...)

spiceconst.icmp<=.i32.i32 (...)

spiceconstant? (...)

spicecountof (...)

spicedecons (...)

spicedefer (...)

spicedrop (...)

spiceelementof (...)

spiceelementsof (...)

spiceextern (...)

spiceforward-repr (...)

spicegen-union-extractvalue (...)

spicegetattr (...)

spicehash-storage (...)

spicehash1 (...)

spiceimply (...)

spiceimply? (...)

spiceinteger->integer (...)

spiceinteger->real (...)

spicekey (...)

spicekeyof (...)

spicelist-constructor (...)

spicelocationof (...)

spicelslice (...)

spicemax (...)

spicememocall (...)

spicemin (...)

spicemutable (...)

spicenone? (...)

spicenot (...)

spiceoffsetof (...)

spiceopaque (...)

spiceor-branch (...)

spiceoverloaded-fn-append (...)

spicepackedtupleof (...)

spiceparse-compile-flags (...)

spicepow (...)

spiceprivate (...)

spiceprotect (...)

spicequalifiersof (...)

spiceraises (...)

spicereal->integer (...)

spicereal->real (...)

spicereport (...)

spicerepr (...)

spicereturnof (...)

spicerslice (...)

spicesabs (...)

spicesafe-shl (...)

spicesign (...)

spicesigned? (...)

spicesizeof (...)

spicestatic-branch (...)

spicestatic-error (...)

spicestatic-integer->integer (...)

spicestatic-integer->real (...)

spicestatic-typify (...)

spicestoragecast (...)

spicestorageof (...)

spicesuperof (...)

spicetostring (...)

spicetupleof (...)

spicetype!= (...)

spicetype< (...)

spicetype<= (...)

spicetype== (...)

spicetype> (...)

spicetype>= (...)

spicetypify (...)

spiceunion-storage-type (...)

spiceunion-storageof (...)

spiceuniqueof (...)

spiceunpack (...)

spiceunqualified (...)

spiceva-append-va (...)
(va-append-va (inline () (_ b . . .)) a. . .) -> a. . . b. . .

spiceva-empty? (...)

spiceva-lfold (...)

spiceva-lifold (...)

spiceva-map (*f*, ...)
Filter each argument in `...` through `f` and return the resulting list of arguments. Arguments where `f` returns void are filtered from the result.

spiceva-option-branch (...)

spiceva-range (*a* [, *b*])
If `b` is not specified, returns a sequence of integers from zero to `b`, otherwise a sequence of integers from `a` to `b`.

spiceva-rfold (...)

spiceva-rifold (...)

spiceva-split (...)
(va-split n a...) -> (inline () a...[n .. (va-countof a...)-1]) a...[0 .. n-1]

spiceva-unnamed (...)
filter all keyed values

spiceva@ (...)

spicevector-reduce (...)

spicevectorof (...)

spiceviewof (...)

spicewrap-if-not-run-stage (...)

spicezip (...)

compiledfncompiler-version (...)
An external function of type λ(i32 i32 i32)<-().

compiledfndefault-styler (...)
An external function of type String<-(Symbol String).

compiledfnexit (...)
An external function of type noreturn<-(i32).

compiledfnfunction->SugarMacro (...)
A compiled function of type SugarMacro<-(λ(List Scope)<->Error(List Scope)<*>).

compiledfnglobals (...)
An external function of type Scope<-().

compiledfnio-write! (...)
An external function of type void<-(String).

compiledfnlaunch-args (...)
An external function of type λ(i32 i8(*)(*))<-().

compiledfnlist-handler (...)
A compiled function of type λ(List Scope)<->Error(List Scope).

compiledfnlist-load (...)
An external function of type Value<->Error(String).

compiledfnlist-parse (...)
An external function of type Value<->Error(String).

compiledfnload-library (...)
An external function of type void<->Error(String).

compiledfnload-object (...)
An external function of type void<->Error(String).

compiledfnparse-infix-expr (...)
A compiled function of type λ(Value List)<->Error(Scope Value List i32).

compiledfnrealpath (...)
An external function of type String<-(String).

compiledfnsc_abort (...)
An external function of type noreturn<-().

compiledfnsc_anchor_column (...)
An external function of type `i32<-(Anchor)`.

compiledfnsc_anchor_lineno (...)
An external function of type `i32<-(Anchor)`.

compiledfnsc_anchor_offset (...)
An external function of type `Anchor<-(Anchor i32)`.

compiledfnsc_anchor_path (...)
An external function of type `Symbol<-(Anchor)`.

compiledfnsc_argcount (...)
An external function of type `i32<-(Value)`.

compiledfnsc_argument_list_new (...)
An external function of type `Value<-(i32 Value(*))`.

compiledfnsc_arguments_type (...)
An external function of type `type<-(i32 type(*))`.

compiledfnsc_arguments_type_argcount (...)
An external function of type `i32<-(type)`.

compiledfnsc_arguments_type_getarg (...)
An external function of type `type<-(type i32)`.

compiledfnsc_arguments_type_join (...)
An external function of type `type<-(type type)`.

compiledfnsc_array_type (...)
An external function of type `type<->Error(type usize)`.

compiledfnsc_basename (...)
An external function of type `String<-(String)`.

compiledfnsc_cache_misses (...)
An external function of type `i32<-()`.

compiledfnsc_call_append_argument (...)
An external function of type `void<-(Value Value)`.

compiledfnsc_call_is_rawcall (...)
An external function of type `bool<-(Value)`.

compiledfnsc_call_new (...)
An external function of type `Value<-(Value)`.

compiledfnsc_call_set_rawcall (...)
An external function of type `void<-(Value bool)`.

compiledfnsc_closure_get_context (...)
An external function of type `Value<-(Closure)`.

compiledfnsc_closure_get_docstring (...)
An external function of type `String<-(Closure)`.

compiledfnsc_closure_get_template (...)
An external function of type `Value<-(Closure)`.

compiledfnsc_compile (...)
An external function of type `Value<->Error(Value u64)`.

compiledfnsc_compile_glsl (...)
An external function of type `String<-Error(i32 Symbol Value u64)`.

compiledfnsc_compile_object (...)
An external function of type `void<-Error(String i32 String Scope u64)`.

compiledfnsc_compile_spirv (...)
An external function of type `String<-Error(Symbol Value u64)`.

compiledfnsc_compiler_version (...)
An external function of type `λ(i32 i32 i32)<-()`.

compiledfnsc_const_aggregate_new (...)
An external function of type `Value<-(type i32 Value(*))`.

compiledfnsc_const_extract_at (...)
An external function of type `Value<-(Value i32)`.

compiledfnsc_const_int_extract (...)
An external function of type `u64<-(Value)`.

compiledfnsc_const_int_extract_word (...)
An external function of type `u64<-(Value i32)`.

compiledfnsc_const_int_new (...)
An external function of type `Value<-(type u64)`.

compiledfnsc_const_int_words_new (...)
An external function of type `Value<-(type i32 u64(*))`.

compiledfnsc_const_null_new (...)
An external function of type `Value<-Error(type)`.

compiledfnsc_const_pointer_extract (...)
An external function of type `void<*><-(Value)`.

compiledfnsc_const_pointer_new (...)
An external function of type `Value<-(type void<*>)`.

compiledfnsc_const_real_extract (...)
An external function of type `f64<-(Value)`.

compiledfnsc_const_real_new (...)
An external function of type `Value<-(type f64)`.

compiledfnsc_default_styler (...)
An external function of type `String<-(Symbol String)`.

compiledfnsc_default_target_triple (...)
An external function of type `String<-()`.

compiledfnsc_dirname (...)
An external function of type `String<-(String)`.

compiledfnsc_dump_error (...)
An external function of type `void<-(Error)`.

compiledfnsc_empty_argument_list (...)
An external function of type `Value<-()`.

compiledfnsc_enter_solver_cli (...)
An external function of type `void<-()`.

compiledfnsc_error_append_calltrace (...)
An external function of type `void<-(Error Value)`.

compiledfnsc_error_new (...)
An external function of type `Error<-(String)`.

compiledfnsc_eval (...)
An external function of type `Value<->Error(Anchor List Scope)`.

compiledfnsc_eval_inline (...)
An external function of type `Anchor<->Error(Value List Scope)`.

compiledfnsc_eval_stage (...)
An external function of type `Value<->Error(Anchor List Scope)`.

compiledfnsc_exit (...)
An external function of type `noreturn<-(i32)`.

compiledfnsc_expand (...)
An external function of type λ`(Value List Scope)<->Error(Value List Scope)`.

compiledfnsc_expression_append (...)
An external function of type `void<-(Value Value)`.

compiledfnsc_expression_new (...)
An external function of type `Value<-()`.

compiledfnsc_expression_set_scoped (...)
An external function of type `void<-(Value)`.

compiledfnsc_extract_argument_list_new (...)
An external function of type `Value<-(Value i32)`.

compiledfnsc_extract_argument_new (...)
An external function of type `Value<-(Value i32)`.

compiledfnsc_format_error (...)
An external function of type `String<-(Error)`.

compiledfnsc_format_message (...)
An external function of type `String<-(Anchor String)`.

compiledfnsc_function_type (...)
An external function of type `type<-(type i32 type(*))`.

compiledfnsc_function_type_is_variadic (...)
An external function of type `bool<-(type)`.

compiledfnsc_function_type_raising (...)
An external function of type `type<-(type type)`.

compiledfnsc_function_type_return_type (...)
An external function of type λ`(type type)<-(type)`.

compiledfnsc_get_globals (...)
An external function of type `Scope<-()`.

compiledfnsc_get_original_globals (...)
An external function of type `Scope<-()`.

compiledfnsc_getarg (...)
An external function of type `Value<-(Value i32)`.

compiledfnsc_getarglist (...)
An external function of type `Value<-(Value i32)`.

compiledfnsc_global_binding (...)
An external function of type `i32<->Error(Value)`.

compiledfnsc_global_descriptor_set (...)
An external function of type `i32<->Error(Value)`.

compiledfnsc_global_location (...)
An external function of type `i32<->Error(Value)`.

compiledfnsc_global_new (...)
An external function of type `Value<-(Symbol type u32 Symbol)`.

compiledfnsc_global_set_binding (...)
An external function of type `void<->Error(Value i32)`.

compiledfnsc_global_set_constructor (...)
An external function of type `void<->Error(Value Value)`.

compiledfnsc_global_set_descriptor_set (...)
An external function of type `void<->Error(Value i32)`.

compiledfnsc_global_set_initializer (...)
An external function of type `void<->Error(Value Value)`.

compiledfnsc_global_set_location (...)
An external function of type `void<->Error(Value i32)`.

compiledfnsc_global_storage_class (...)
An external function of type `Symbol<->Error(Value)`.

compiledfnsc_hash (...)
An external function of type `u64<-(u64 usize)`.

compiledfnsc_hash2x64 (...)
An external function of type `u64<-(u64 u64)`.

compiledfnsc_hashbytes (...)
An external function of type `u64<-(i8(*) usize)`.

compiledfnsc_identity (...)
An external function of type `Value<-(Value)`.

compiledfnsc_if_append_else_clause (...)
An external function of type `void<-(Value Value)`.

compiledfnsc_if_append_then_clause (...)
An external function of type `void<-(Value Value Value)`.

compiledfnsc_if_new (...)
An external function of type `Value<-()`.

compiledfnsc_image_type (...)
An external function of type `type<-(type Symbol i32 i32 i32 i32 Symbol Symbol)`.

compiledfnsc_import_c (...)
An external function of type `Scope<->Error(String String List Scope)`.

compiledfnsc_integer_type (...)
An external function of type `type<-(i32 bool)`.

compiledfnsc_integer_type_is_signed (...)
An external function of type `bool<-(type)`.

compiledfnsc_is_directory (...)
An external function of type `bool<-(String)`.

compiledfnsc_is_file (...)
An external function of type `bool<-(String)`.

compiledfnsc_key_type (...)
An external function of type `type<-(Symbol type)`.

compiledfnsc_keyed_new (...)
An external function of type `Value<-(Symbol Value)`.

compiledfnsc_label_new (...)
An external function of type `Value<-(i32 Symbol)`.

compiledfnsc_label_set_body (...)
An external function of type `void<-(Value Value)`.

compiledfnsc_launch_args (...)
An external function of type `λ(i32 i8(*)(*))<-()`.

compiledfnsc_list_at (...)
An external function of type `Value<-(List)`.

compiledfnsc_list_compare (...)
An external function of type `bool<-(List List)`.

compiledfnsc_list_cons (...)
An external function of type `List<-(Value List)`.

compiledfnsc_list_count (...)
An external function of type `i32<-(List)`.

compiledfnsc_list_decons (...)
An external function of type `λ(Value List)<-(List)`.

compiledfnsc_list_dump (...)
An external function of type `List<-(List)`.

compiledfnsc_list_join (...)
An external function of type `List<-(List List)`.

compiledfnsc_list_next (...)
An external function of type `List<-(List)`.

compiledfnsc_list_repr (...)
An external function of type `String<-(List)`.

compiledfnsc_list_reverse (...)
An external function of type `List<-(List)`.

compiledfnsc_list_serialize (...)
An external function of type `String<-(List)`.

compiledfnsc_load_history (...)
An external function of type `void<-(String)`.

compiledfnsc_load_library (...)
An external function of type `void<->Error(String)`.

compiledfnsc_load_object (...)
An external function of type `void<-(String)`.

compiledfnsc_loop_arguments (...)
An external function of type `Value<-(Value)`.

compiledfnsc_loop_new (...)
An external function of type `Value<-(Value)`.

compiledfnsc_loop_set_body (...)
An external function of type `void<-(Value Value)`.

compiledfnsc_map_get (...)
An external function of type `Value<-(Value)`.

compiledfnsc_map_set (...)
An external function of type `void<-(Value Value)`.

compiledfnsc_merge_new (...)
An external function of type `Value<-(Value Value)`.

compiledfnsc_mutate_type (...)
An external function of type `type<-(type)`.

compiledfnsc_packed_tuple_type (...)
An external function of type `type<-(i32 type(*))`.

compiledfnsc_parameter_is_variadic (...)
An external function of type `bool<-(Value)`.

compiledfnsc_parameter_name (...)
An external function of type `Symbol<-(Value)`.

compiledfnsc_parameter_new (...)
An external function of type `Value<-(Symbol)`.

compiledfnsc_parse_from_path (...)
An external function of type `Value<-(String)`.

compiledfnsc_parse_from_string (...)
An external function of type `Value<-(String)`.

compiledfnsc_pointer_type (...)
An external function of type `type<-(type u64 Symbol)`.

compiledfnsc_pointer_type_get_flags (...)
An external function of type `u64<-(type)`.

compiledfnsc_pointer_type_get_storage_class (...)
An external function of type `Symbol<-(type)`.

compiledfnsc_pointer_type_set_element_type (...)
An external function of type `type<-(type type)`.

compiledfnsc_pointer_type_set_flags (...)
An external function of type `type<-(type u64)`.

compiledfnsc_pointer_type_set_storage_class (...)
An external function of type `type<-(type Symbol)`.

compiledfnsc_prompt (...)
An external function of type `λ(bool String)<-(String String)`.

compiledfnsc_prove (...)
An external function of type `Value<->Error(Value)`.

compiledfnsc_quote_new (...)
An external function of type `Value<-(Value)`.

compiledfnsc_realpath (...)
An external function of type `String<-(String)`.

compiledfnsc_refer_flags (...)
An external function of type `u64<-(type)`.

compiledfnsc_refer_storage_class (...)
An external function of type `Symbol<-(type)`.

compiledfnsc_refer_type (...)
An external function of type `type<-(type u64 Symbol)`.

compiledfnsc_sampled_image_type (...)
An external function of type `type<-(type)`.

compiledfnsc_save_history (...)
An external function of type `void<-(String)`.

compiledfnsc_scope_at (...)
An external function of type `Value<->Error(Scope Value)`.

compiledfnsc_scope_bind (...)
An external function of type `Scope<-(Scope Value Value)`.

compiledfnsc_scope_bind_with_docstring (...)
An external function of type `Scope<-(Scope Value Value String)`.

compiledfnsc_scope_docstring (...)
An external function of type `String<-(Scope Value)`.

compiledfnsc_scope_get_parent (...)
An external function of type `Scope<-(Scope)`.

compiledfnsc_scope_local_at (...)
An external function of type `Value<->Error(Scope Value)`.

compiledfnsc_scope_module_docstring (...)
An external function of type `String<-(Scope)`.

compiledfnsc_scope_new (...)
An external function of type `Scope<-()`.

compiledfnsc_scope_new_subscope (...)
An external function of type `Scope<-(Scope)`.

compiledfnsc_scope_new_subscope_with_docstring (...)
An external function of type `Scope<-(Scope String)`.

compiledfnsc_scope_new_with_docstring (...)
An external function of type `Scope<-(String)`.

compiledfnsc_scope_next (...)
An external function of type `λ(Value Value i32)<-(Scope i32)`.

compiledfnsc_scope_next_deleted (...)
An external function of type `λ(Value i32)<-(Scope i32)`.

compiledfnsc_scope_reparent (...)
An external function of type `Scope<-(Scope Scope)`.

compiledfnsc_scope_unbind (...)
An external function of type `Scope<-(Scope Value)`.

compiledfnsc_scope_unparent (...)
An external function of type `Scope<-(Scope)`.

compiledfnsc_set_autocomplete_scope (...)
An external function of type `void<-(Scope)`.

compiledfnsc_set_globals (...)
An external function of type `void<-(Scope)`.

compiledfnsc_set_signal_abort (...)
An external function of type `void<-(bool)`.

compiledfnsc_string_buffer (...)
An external function of type `λ(i8(*) usize)<-(String)`.

compiledfnsc_string_compare (...)
An external function of type `i32<-(String String)`.

compiledfnsc_string_count (...)
An external function of type `usize<-(String)`.

compiledfnsc_string_join (...)
An external function of type `String<-(String String)`.

compiledfnsc_string_lslice (...)
An external function of type `String<-(String usize)`.

compiledfnsc_string_match (...)
An external function of type `λ(bool i32 i32)<->Error(String String)`.

compiledfnsc_string_new (...)
An external function of type `String<-(i8(*) usize)`.

compiledfnsc_string_new_from_cstr (...)
An external function of type `String<-(i8(*))`.

compiledfnsc_string_rslice (...)
An external function of type `String<-(String usize)`.

compiledfnsc_strip_qualifiers (...)
An external function of type `type<-(type)`.

compiledfnsc_switch_append_case (...)
An external function of type `void<-(Value Value Value)`.

compiledfnsc_switch_append_default (...)
An external function of type `void<-(Value Value)`.

compiledfnsc_switch_append_do (...)
An external function of type `void<-(Value Value)`.

compiledfnsc_switch_append_pass (...)
An external function of type `void<-(Value Value Value)`.

compiledfnsc_switch_new (...)
An external function of type `Value<-(Value)`.

compiledfnsc_symbol_count (...)
An external function of type `usize<-()`.

compiledfnsc_symbol_is_variadic (...)
An external function of type `bool<-(Symbol)`.

compiledfnsc_symbol_new (...)
An external function of type `Symbol<-(String)`.

compiledfnsc_symbol_new_unique (...)
An external function of type `Symbol<-(String)`.

compiledfnsc_symbol_style (...)
An external function of type `Symbol<-(Symbol)`.

compiledfnsc_symbol_to_string (...)
An external function of type `String<-(Symbol)`.

compiledfnsc_template_append_parameter (...)
An external function of type `void<-(Value Value)`.

compiledfnsc_template_get_name (...)
An external function of type `Symbol<-(Value)`.

compiledfnsc_template_is_inline (...)
An external function of type `bool<-(Value)`.

compiledfnsc_template_new (...)
An external function of type `Value<-(Symbol)`.

compiledfnsc_template_parameter (...)
An external function of type `Value<-(Value i32)`.

compiledfnsc_template_parameter_count (...)
An external function of type `i32<-(Value)`.

compiledfnsc_template_set_body (...)
An external function of type `void<-(Value Value)`.

compiledfnsc_template_set_inline (...)
An external function of type `void<-(Value)`.

compiledfnsc_template_set_name (...)
An external function of type `void<-(Value Symbol)`.

compiledfnsc_tuple_type (...)
An external function of type `type<->Error(i32 type(*))`.

compiledfnsc_type_alignof (...)
An external function of type `usize<->Error(type)`.

compiledfnsc_type_at (...)
An external function of type `Value<->Error(type Symbol)`.

compiledfnsc_type_bitcountof (...)
An external function of type `i32<-(type)`.

compiledfnsc_type_compatible (...)
An external function of type `bool<-(type type)`.

compiledfnsc_type_countof (...)
An external function of type `i32<->Error(type)`.

compiledfnsc_type_debug_abi (...)
An external function of type `void<-(type)`.

compiledfnsc_type_element_at (...)
An external function of type `type<-Error(type i32)`.

compiledfnsc_type_field_index (...)
An external function of type `i32<-Error(type Symbol)`.

compiledfnsc_type_field_name (...)
An external function of type `Symbol<-Error(type i32)`.

compiledfnsc_type_get_docstring (...)
An external function of type `String<-(type Symbol)`.

compiledfnsc_type_is_default_suffix (...)
An external function of type `bool<-(type)`.

compiledfnsc_type_is_opaque (...)
An external function of type `bool<-(type)`.

compiledfnsc_type_is_plain (...)
An external function of type `bool<-(type)`.

compiledfnsc_type_is_refer (...)
An external function of type `bool<-(type)`.

compiledfnsc_type_is_superof (...)
An external function of type `bool<-(type type)`.

compiledfnsc_type_key (...)
An external function of type `λ(Symbol type)<-(type)`.

compiledfnsc_type_kind (...)
An external function of type `i32<-(type)`.

compiledfnsc_type_local_at (...)
An external function of type `Value<-Error(type Symbol)`.

compiledfnsc_type_next (...)
An external function of type `λ(Symbol Value)<-(type Symbol)`.

compiledfnsc_type_offsetof (...)
An external function of type `usize<-Error(type i32)`.

compiledfnsc_type_set_docstring (...)
An external function of type `void<-(type Symbol String)`.

compiledfnsc_type_set_symbol (...)
An external function of type `void<-(type Symbol Value)`.

compiledfnsc_type_sizeof (...)
An external function of type `usize<-Error(type)`.

compiledfnsc_type_storage (...)
An external function of type `type<-Error(type)`.

compiledfnsc_type_string (...)
An external function of type `String<-(type)`.

compiledfnsc_typename_type (...)
An external function of type `type<-Error(String type)`.

compiledfnsc_typename_type_get_super (...)
An external function of type `type<-(type)`.

compiledfnsc_typename_type_set_opaque (...)
An external function of type `void<->Error(type)`.

compiledfnsc_typename_type_set_storage (...)
An external function of type `void<->Error(type type u32)`.

compiledfnsc_typify (...)
An external function of type `Value<->Error(Closure i32 type(*))`.

compiledfnsc_typify_template (...)
An external function of type `Value<->Error(Value i32 type(*))`.

compiledfnsc_union_storage_type (...)
An external function of type `type<->Error(i32 type(*))`.

compiledfnsc_unique_type (...)
An external function of type `type<-(type i32)`.

compiledfnsc_unquote_new (...)
An external function of type `Value<-(Value)`.

compiledfnsc_value_anchor (...)
An external function of type `Anchor<-(Value)`.

compiledfnsc_value_ast_repr (...)
An external function of type `String<-(Value)`.

compiledfnsc_value_block_depth (...)
An external function of type `i32<-(Value)`.

compiledfnsc_value_compare (...)
An external function of type `bool<-(Value Value)`.

compiledfnsc_value_content_repr (...)
An external function of type `String<-(Value)`.

compiledfnsc_value_is_constant (...)
An external function of type `bool<-(Value)`.

compiledfnsc_value_is_pure (...)
An external function of type `bool<-(Value)`.

compiledfnsc_value_kind (...)
An external function of type `i32<-(Value)`.

compiledfnsc_value_kind_string (...)
An external function of type `String<-(i32)`.

compiledfnsc_value_qualified_type (...)
An external function of type `type<-(Value)`.

compiledfnsc_value_repr (...)
An external function of type `String<-(Value)`.

compiledfnsc_value_tostring (...)
An external function of type `String<-(Value)`.

compiledfnsc_value_type (...)
An external function of type `type<-(Value)`.

compiledfnsc_value_unwrap (...)
An external function of type `Value<-(type Value)`.

compiledfnsc_value_wrap (...)
An external function of type `Value<-(type Value)`.

compiledfnsc_valueref_tag (...)
An external function of type `Value<-(Anchor Value)`.

compiledfnsc_vector_type (...)
An external function of type `type<->Error(type usize)`.

compiledfnsc_view_type (...)
An external function of type `type<-(type i32)`.

compiledfnsc_write (...)
An external function of type `void<-(String)`.

compiledfnset-autocomplete-scope! (...)
An external function of type `void<-(Scope)`.

compiledfnset-globals! (...)
An external function of type `void<-(Scope)`.

compiledfnset-signal-abort! (...)
An external function of type `void<-(bool)`.

compiledfnspice-macro-verify-signature (...)
A compiled function of type `void<-(Value<->Error(Value)<*>)`.

compiledfnsymbol-handler (...)
A compiled function of type `λ(List Scope)<->Error(List Scope)`.

6.2 Array

Provides mutable array types that store their elements on the heap rather than in registers or the stack.

typeArray
An opaque type of supertype *Struct*.

fn__@ (*self* , *index*)
Implements support for the *@* operator. Returns a view reference to the element at `index` of array `self`.

inline__as (*cls* , *T*)
Implements support for the *as* operator. Arrays can be cast to *Generator*, or directly passed to *for*.

inline__countof (*self*)
Implements support for the *countof* operator. Returns the current number of elements stored in `self` as a value of *usize* type.

inline__drop (*self*)
Implements support for freeing the array's memory when it goes out of scope.

inline__imply (*cls* , *T*)
Implements support for pointer casts, to pass the array to C functions for example.

inline__typecall (*cls* , *element-type* , *capacity*)
Construct a mutable array type of `element-type` with a variable or fixed maximum capacity.

If `capacity` is defined, then it specifies the maximum number of array elements permitted. If it is undefined, then an initial capacity of 16 elements is assumed, which is doubled whenever it is exceeded, allowing for an indefinite number of elements.

fnappend (*self* , *value*)
Append *value* as an element to the array `self` and return a reference to the new element. When the *array* is of *GrowingArray* type, this operation will transparently resize the array's storage.

fnclear (*self*)
Clear the array and reset its element count to zero. This will drop all elements that have been previously contained by the array.

inlineemplace-append (*self* , *args...*)
Construct a new element with arguments `args...` directly in a newly assigned slot of array `self`. When the *array* is of *GrowingArray* type, this operation will transparently resize the array's storage.

inlineemplace-append-many (*self* , *size* , *args...*)
Construct a new element with arguments `args...` directly in a newly assigned slot of array `self`. When the *array* is of *GrowingArray* type, this operation will transparently resize the array's storage.

fnresize (*self* , *count* , *args...*)
Resize the array to the specified count. Items are apppend or removed to meet the desired count.

inlinesort (*self* , *key*)
Sort elements of array `self` from smallest to largest, either using the < operator supplied by the element type, or by using the key supplied by the callable *key*, which is expected to return a comparable value for each element value supplied.

fnswap (*self* , *a* , *b*)
Safely swap the contents of two indices.

typeFixedArray
An opaque type of supertype *Array*.

fn__repr (*self*)
Implements support for the *repr* operation.

inline__typecall (*cls* , *opts...*)

inlinecapacity (*self*)
Returns the maximum capacity of array `self`, which is fixed.

fnreserve (*self* , *count*)
Internally used by the type. Ensures that array `self` can hold at least `count` elements. A fixed array will raise an assertion when its capacity has been exceeded.

typeGrowingArray
An opaque type of supertype *Array*.

fn__repr (*self*)
Implements support for the *repr* operation.

inline__typecall (*cls* , *opts...*)

inlinecapacity (*self*)
Returns the current maximum capacity of array `self`.

fnreserve (*self* , *count*)
Internally used by the type. Ensures that array `self` can hold at least `count` elements. A growing array will always attempt to comply.

6.3 Box

Provides a unique reference container for heap allocated values.

typeBox
An opaque type.

spice__= (...)

inline__@ (*self* , *keys...*)

spice__as (...)

inline__countof (*self*)

inline__drop (*self*)

spice__getattr (...)

spice__imply (...)

spice__methodcall (...)

spice__repr (...)

spice__static-imply (...)

inlinemake-cast-op (*f* , *const?*)

inlinenew (*T* , *args...*)

inlineview (*self*)

inlinewrap (*value*)

6.4 Capture

A capture is a runtime closure that transparently captures (hence the name) runtime values outside of the function.

typeCapture
An opaque type.

inline__call (*self* , *args...*)

inline__drop (*self*)

inline__typecall (*cls* , *args...*)

inlinemake-type (...)

typeCaptureTemplate
An opaque type.

spice__imply (...)

inlinebuild-instance (*self* , *f*)

inlineinstance (*self* , *types...*)

inlinetypify-function (*cls* , *types...*)

typeSpiceCapture
An opaque type.

sugar(**capture** ...)

sugar(**decorate-capture** ...)

sugar(**spice-capture** ...)

6.5 console

Implements the read-eval-print loop for Scopes' console.

typeread-eval-print-loop
An opaque type of supertype *OverloadedFunction*.

inlineparameter-defaults ()

fntemplates (*global-scope* , *show-logo* , *history-path*)

6.6 enum

Support for defining tagged unions and classical enums through the *enum* sugar.

typeEnum
An opaque type.

spice__dispatch (...)

inlineOption (...)

sugar(**dispatch** ...)

sugar(**enum** ...)

6.7 FunctionChain

A function chain implements a compile-time observer pattern that allows a module to call back into dependent modules in a decoupled way.

See following example:

```
using import FunctionChain

# declare new function chain
fnchain activate

fn handler (x)
    print "handler activated with argument" x

'append activate handler

'append activate
    fn (x)
        print "last handler activated with argument" x

'prepend activate
    fn (x)
        print "first handler activated with argument" x
```

(continues on next page)

(continued from previous page)

```
activate 1
activate 2
'clear activate
'append activate handler
activate 3
```

Running this program will output:

```
first handler activated with argument 1
handler activated with argument 1
last handler activated with argument 1
first handler activated with argument 2
handler activated with argument 2
last handler activated with argument 2
handler activated with argument 3
```

typeFunctionChain
A plain type of storage type *_type*.

spice__call (...)

inline__repr (*self*)

spice__typecall (...)

inlineappend (*self*, *f*)
Append function `f` to function chain. When the function chain is called, `f` will be called last. The return value of `f` will be ignored.

inlineclear (*self*)
Clear the function chain. When the function chain is applied next, no functions will be called.

inlineon (*self*)
Returns a decorator that appends the provided function to the function chain.

inlineprepend (*self*, *f*)
Prepend function `f` to function chain. When the function chain is called, `f` will be called first. The return value of `f` will be ignored.

sugar(**decorate-fnchain** ...)

sugar(**fnchain** *name*)
Binds a new unique and empty function chain to identifier `name`. The function chain's typename is going to incorporate the name of the module in which it was declared.

6.8 glm

The `glm` module exports the basic vector and matrix types as well as related arithmetic operations which mimic the features available to shaders written in the GL shader language.

typebmat2
A plain type labeled `bmat2x2` of supertype *mat-type* and of storage type `[bvec2 x 2]`.

typebmat2x2
A plain type of supertype *mat-type* and of storage type `[bvec2 x 2]`.

typebmat2x3
A plain type of supertype *mat-type* and of storage type `[bvec3 x 2]`.

typebmat2x4
A plain type of supertype *mat-type* and of storage type `[bvec4 x 2]`.

typebmat3
A plain type labeled `bmat3x3` of supertype *mat-type* and of storage type `[bvec3 x 3]`.

typebmat3x2
A plain type of supertype *mat-type* and of storage type `[bvec2 x 3]`.

typebmat3x3
A plain type of supertype *mat-type* and of storage type `[bvec3 x 3]`.

typebmat3x4
A plain type of supertype *mat-type* and of storage type `[bvec4 x 3]`.

typebmat4
A plain type labeled `bmat4x4` of supertype *mat-type* and of storage type `[bvec4 x 4]`.

typebmat4x2
A plain type of supertype *mat-type* and of storage type `[bvec2 x 4]`.

typebmat4x3
A plain type of supertype *mat-type* and of storage type `[bvec3 x 4]`.

typebmat4x4
A plain type of supertype *mat-type* and of storage type `[bvec4 x 4]`.

typebvec2
A plain type of supertype *gvec2* and of storage type `<bool x 2>`.

typebvec3
A plain type of supertype *gvec3* and of storage type `<bool x 3>`.

typebvec4
A plain type of supertype *gvec4* and of storage type `<bool x 4>`.

typedmat2
A plain type labeled `dmat2x2` of supertype *mat-type* and of storage type `[dvec2 x 2]`.

typedmat2x2
A plain type of supertype *mat-type* and of storage type `[dvec2 x 2]`.

typedmat2x3
A plain type of supertype *mat-type* and of storage type `[dvec3 x 2]`.

typedmat2x4
A plain type of supertype *mat-type* and of storage type `[dvec4 x 2]`.

typedmat3
A plain type labeled `dmat3x3` of supertype *mat-type* and of storage type `[dvec3 x 3]`.

typedmat3x2
A plain type of supertype *mat-type* and of storage type `[dvec2 x 3]`.

typedmat3x3
A plain type of supertype *mat-type* and of storage type `[dvec3 x 3]`.

typedmat3x4
A plain type of supertype *mat-type* and of storage type `[dvec4 x 3]`.

typedmat4
A plain type labeled `dmat4x4` of supertype *mat-type* and of storage type `[dvec4 x 4]`.

typedmat4x2
A plain type of supertype *mat-type* and of storage type `[dvec2 x 4]`.

typedmat4x3
A plain type of supertype *mat-type* and of storage type `[dvec3 x 4]`.

typedmat4x4
A plain type of supertype *mat-type* and of storage type `[dvec4 x 4]`.

typedvec2
A plain type of supertype *gvec2* and of storage type `<f64 x 2>`.

typedvec3
A plain type of supertype *gvec3* and of storage type `<f64 x 3>`.

typedvec4
A plain type of supertype *gvec4* and of storage type `<f64 x 4>`.

typegvec2
An opaque type of supertype *vec-type*.

typegvec3
An opaque type of supertype *vec-type*.

typegvec4
An opaque type of supertype *vec-type*.

typeimat2
A plain type labeled `imat2x2` of supertype *mat-type* and of storage type `[ivec2 x 2]`.

typeimat2x2
A plain type of supertype *mat-type* and of storage type `[ivec2 x 2]`.

typeimat2x3
A plain type of supertype *mat-type* and of storage type `[ivec3 x 2]`.

typeimat2x4
A plain type of supertype *mat-type* and of storage type `[ivec4 x 2]`.

typeimat3
A plain type labeled `imat3x3` of supertype *mat-type* and of storage type `[ivec3 x 3]`.

typeimat3x2
A plain type of supertype *mat-type* and of storage type `[ivec2 x 3]`.

typeimat3x3
A plain type of supertype *mat-type* and of storage type `[ivec3 x 3]`.

typeimat3x4
A plain type of supertype *mat-type* and of storage type `[ivec4 x 3]`.

typeimat4
A plain type labeled `imat4x4` of supertype *mat-type* and of storage type `[ivec4 x 4]`.

typeimat4x2
A plain type of supertype *mat-type* and of storage type `[ivec2 x 4]`.

typeimat4x3
A plain type of supertype *mat-type* and of storage type `[ivec3 x 4]`.

typeimat4x4
A plain type of supertype *mat-type* and of storage type `[ivec4 x 4]`.

typeivec2
A plain type of supertype *gvec2* and of storage type `<i32 x 2>`.

typeivec3
A plain type of supertype *gvec3* and of storage type `<i32 x 3>`.

typeivec4
A plain type of supertype *gvec4* and of storage type `<i32 x 4>`.

typemat-type
An opaque type of supertype *immutable*.

spice__* (...)

spice__== (...)

inline__@ (*self* , *index*)

spice__as (...)

spice__r* (...)

spice__typecall (...)

inline__unpack (*self*)

spicerow (...)

typemat2
A plain type labeled `mat2x2` of supertype *mat-type* and of storage type `[vec2 x 2]`.

typemat2x2
A plain type of supertype *mat-type* and of storage type `[vec2 x 2]`.

typemat2x3
A plain type of supertype *mat-type* and of storage type `[vec3 x 2]`.

typemat2x4
A plain type of supertype *mat-type* and of storage type `[vec4 x 2]`.

typemat3
A plain type labeled `mat3x3` of supertype *mat-type* and of storage type `[vec3 x 3]`.

typemat3x2
A plain type of supertype *mat-type* and of storage type `[vec2 x 3]`.

typemat3x3
A plain type of supertype *mat-type* and of storage type `[vec3 x 3]`.

typemat3x4
A plain type of supertype *mat-type* and of storage type `[vec4 x 3]`.

typemat4
A plain type labeled `mat4x4` of supertype *mat-type* and of storage type `[vec4 x 4]`.

typemat4x2
A plain type of supertype *mat-type* and of storage type `[vec2 x 4]`.

typemat4x3
A plain type of supertype *mat-type* and of storage type `[vec3 x 4]`.

typemat4x4
A plain type of supertype *mat-type* and of storage type `[vec4 x 4]`.

typeumat2
A plain type labeled `umat2x2` of supertype *mat-type* and of storage type `[uvec2 x 2]`.

typeumat2x2
A plain type of supertype *mat-type* and of storage type `[uvec2 x 2]`.

typeumat2x3
A plain type of supertype *mat-type* and of storage type `[uvec3 x 2]`.

typeumat2x4
A plain type of supertype *mat-type* and of storage type `[uvec4 x 2]`.

typeumat3
A plain type labeled `umat3x3` of supertype *mat-type* and of storage type `[uvec3 x 3]`.

typeumat3x2
A plain type of supertype *mat-type* and of storage type `[uvec2 x 3]`.

typeumat3x3
A plain type of supertype *mat-type* and of storage type `[uvec3 x 3]`.

typeumat3x4
A plain type of supertype *mat-type* and of storage type `[uvec4 x 3]`.

typeumat4
A plain type labeled `umat4x4` of supertype *mat-type* and of storage type `[uvec4 x 4]`.

typeumat4x2
A plain type of supertype *mat-type* and of storage type `[uvec2 x 4]`.

typeumat4x3
A plain type of supertype *mat-type* and of storage type `[uvec3 x 4]`.

typeumat4x4
A plain type of supertype *mat-type* and of storage type `[uvec4 x 4]`.

typeuvec2
A plain type of supertype *gvec2* and of storage type `<u32 x 2>`.

typeuvec3
A plain type of supertype *gvec3* and of storage type `<u32 x 3>`.

typeuvec4
A plain type of supertype *gvec4* and of storage type `<u32 x 4>`.

typevec-type
An opaque type of supertype *immutable*.

spice__% (...)

spice__& (...)

spice__* (...)

spice__** (...)

spice__+ (...)

spice__- (...)

spice__/ (...)

spice__// (...)

spice__< (...)

spice__<< (...)

spice__<= (...)

spice__== (...)

spice__> (...)

spice__>= (...)

spice__>> (...)

inline__@ (*self*, *i*)

spice__^ (...)

spice__as (...)

spice__getattr (...)

inline__neg (*self*)

spice__r% (...)

spice__r& (...)

spice__r* (...)

spice__r** (...)

spice__r+ (...)

spice__r- (...)

spice__r/ (...)

spice__r// (...)

spice__r< (...)

spice__r<< (...)

spice__r<= (...)

spice__r> (...)

spice__r>= (...)

spice__r>> (...)

spice__r^ (...)

inline__rcp (*self*)

spice__rimply (...)

spice__r| (...)

spice__static-rimply (...)

spice__typecall (...)

inline__unpack (*self*)

spice__| (...)

typevec2
A plain type of supertype *gvec2* and of storage type `<f32 x 2>`.

typevec3
A plain type of supertype *gvec3* and of storage type `<f32 x 3>`.

typevec4
A plain type of supertype *gvec4* and of storage type `<f32 x 4>`.

inlinedot (*u* , *v*)

spicemix (...)

6.9 glsl

The `glsl` module exports bridge symbols that make it possible to define and access external variables for shader programs.

definegl_ClipDistance
A constant of type `[f32 x ?]`.

definegl_FragCoord
A constant of type *vec4*.

definegl_FragDepth
A constant of type *f32*.

definegl_GlobalInvocationID
A constant of type *uvec3*.

definegl_InstanceID
A constant of type *i32*.

definegl_InstanceIndex
A constant of type *i32*.

definegl_LocalInvocationID
A constant of type *uvec3*.

definegl_LocalInvocationIndex
A constant of type *u32*.

definegl_NumWorkGroups
A constant of type *uvec3*.

definegl_PointSize
A constant of type *f32*.

definegl_Position
A constant of type *vec4*.

definegl_VertexID
A constant of type *i32*.

definegl_VertexIndex
A constant of type *i32*.

definegl_WorkGroupID
A constant of type *uvec3*.

definegl_WorkGroupSize
A constant of type *uvec3*.

definer16
A constant of type *Symbol*.

definer16_snorm
A constant of type *Symbol*.

definer16f
A constant of type *Symbol*.

definer16i
A constant of type *Symbol*.

definer16ui
A constant of type *Symbol*.

definer32
A constant of type *Symbol*.

definer32f
A constant of type *Symbol*.

definer32i
A constant of type *Symbol*.

definer32ui
A constant of type *Symbol*.

definer8
A constant of type *Symbol*.

definer8_snorm
A constant of type *Symbol*.

definer8i
A constant of type *Symbol*.

definer8ui
A constant of type *Symbol*.

definerg16
A constant of type *Symbol*.

definerg16_snorm
A constant of type *Symbol*.

definerg16f
A constant of type *Symbol*.

definerg16i
A constant of type *Symbol*.

definerg16ui
A constant of type *Symbol*.

definerg32
A constant of type *Symbol*.

definerg32f
A constant of type *Symbol*.

definerg32i
A constant of type *Symbol*.

definerg32ui
A constant of type *Symbol*.

definerg8
A constant of type *Symbol*.

definerg8_snorm
A constant of type *Symbol*.

definerg8i
A constant of type *Symbol*.

definerg8ui
A constant of type *Symbol*.

definergba16
A constant of type *Symbol*.

definergba16_snorm
A constant of type *Symbol*.

definergba16f
A constant of type *Symbol*.

definergba16i
A constant of type *Symbol*.

definergba16ui
A constant of type *Symbol*.

definergba32
A constant of type *Symbol*.

definergba32f
A constant of type *Symbol*.

definergba32i
A constant of type *Symbol*.

definergba32ui
A constant of type *Symbol*.

definergba8
A constant of type *Symbol*.

definergba8_snorm
A constant of type *Symbol*.

definergba8i
A constant of type *Symbol*.

definergba8ui
A constant of type *Symbol*.

typeDispatchIndirectCommand
A plain type of supertype *CStruct* and of storage type `{num_groups_x=u32 num_groups_y=u32 num_groups_z=u32}`.

typeDrawArraysIndirectCommand
A plain type of supertype *CStruct* and of storage type `{count=u32 instanceCount=u32 first=u32 baseInstance=u32}`.

typeceil
An opaque type of supertype *OverloadedFunction*.

typedFdx
An opaque type of supertype *OverloadedFunction*.

typedFdy
An opaque type of supertype *OverloadedFunction*.

typefindLSB
An opaque type of supertype *OverloadedFunction*.

typefract
An opaque type of supertype *OverloadedFunction*.

typefwidth
An opaque type of supertype *OverloadedFunction*.

typegsampler
An opaque type.

typegsampler1D
An opaque type of supertype *gsampler*.

builtintexture-levels (...)

builtintexture-samples (...)

typegsampler1DArray
An opaque type of supertype *gsampler*.

builtintexture-levels (...)

builtintexture-samples (...)

typegsampler2D
An opaque type of supertype *gsampler*.

builtintexture-levels (...)

builtintexture-samples (...)

typegsampler2DArray
An opaque type of supertype *gsampler*.

builtintexture-levels (...)

builtintexture-samples (...)

typegsampler2DMS
An opaque type of supertype *gsampler*.

builtintexture-levels (...)

builtintexture-samples (...)

typegsampler2DMSArray
An opaque type of supertype *gsampler*.

builtintexture-levels (...)

builtintexture-samples (...)

typegsampler2DRect
An opaque type of supertype *gsampler*.

builtintexture-levels (...)

builtintexture-samples (...)

typegsampler3D
An opaque type of supertype *gsampler*.

builtintexture-levels (...)

builtintexture-samples (...)

typegsamplerBuffer
An opaque type of supertype *gsampler*.

builtintexture-levels (...)

builtintexture-samples (...)

typegsamplerCube
An opaque type of supertype *gsampler*.

builtintexture-levels (...)

builtintexture-samples (...)

typegsamplerCubeArray
An opaque type of supertype *gsampler*.

builtintexture-levels (...)

builtintexture-samples (...)

typeisampler1D
A plain type of supertype `gsampler1D$3` and of storage type `<SampledImage`.

typeisampler1DArray
A plain type of supertype `gsampler1DArray$3` and of storage type `<SampledImage`.

typeisampler2D
A plain type of supertype `gsampler2D$3` and of storage type `<SampledImage`.

typeisampler2DArray
A plain type of supertype `gsampler2DArray$3` and of storage type `<SampledImage`.

typeisampler2DMS
A plain type of supertype `gsampler2DMS$3` and of storage type `<SampledImage`.

typeisampler2DMSArray
A plain type of supertype `gsampler2DMSArray$3` and of storage type `<SampledImage`.

typeisampler2DRect
A plain type of supertype `gsampler2DRect$3` and of storage type `<SampledImage`.

typeisampler3D
A plain type of supertype `gsampler3D$3` and of storage type `<SampledImage`.

typeisamplerBuffer
A plain type of supertype `gsamplerBuffer$3` and of storage type `<SampledImage`.

typeisamplerCube
A plain type of supertype `gsamplerCube$3` and of storage type `<SampledImage`.

typeisamplerCubeArray
A plain type of supertype `gsamplerCubeArray$3` and of storage type `<SampledImage`.

typesampler1D
A plain type of supertype `gsampler1D$2` and of storage type `<SampledImage`.

typesampler1DArray
A plain type of supertype `gsampler1DArray$2` and of storage type `<SampledImage`.

typesampler2D
A plain type of supertype `gsampler2D$2` and of storage type `<SampledImage`.

typesampler2DArray
A plain type of supertype `gsampler2DArray$2` and of storage type `<SampledImage`.

typesampler2DMS
A plain type of supertype `gsampler2DMS$2` and of storage type `<SampledImage`.

typesampler2DMSArray
A plain type of supertype `gsampler2DMSArray$2` and of storage type `<SampledImage`.

typesampler2DRect
A plain type of supertype `gsampler2DRect$2` and of storage type `<SampledImage`.

typesampler3D
A plain type of supertype `gsampler3D$2` and of storage type `<SampledImage`.

typesamplerBuffer
A plain type of supertype `gsamplerBuffer$2` and of storage type `<SampledImage`.

typesamplerCube
A plain type of supertype `gsamplerCube$2` and of storage type `<SampledImage`.

typesamplerCubeArray
A plain type of supertype `gsamplerCubeArray$2` and of storage type `<SampledImage`.

typesmoothstep
An opaque type of supertype *`OverloadedFunction`*.

typeusampler1D
A plain type of supertype `gsampler1D$4` and of storage type `<SampledImage`.

typeusampler1DArray
A plain type of supertype `gsampler1DArray$4` and of storage type `<SampledImage`.

typeusampler2D
A plain type of supertype `gsampler2D$4` and of storage type `<SampledImage`.

typeusampler2DArray
A plain type of supertype `gsampler2DArray$4` and of storage type `<SampledImage`.

typeusampler2DMS
A plain type of supertype `gsampler2DMS$4` and of storage type `<SampledImage`.

typeusampler2DMSArray
A plain type of supertype `gsampler2DMSArray$4` and of storage type `<SampledImage`.

typeusampler2DRect
A plain type of supertype `gsampler2DRect$4` and of storage type `<SampledImage`.

typeusampler3D
A plain type of supertype `gsampler3D$4` and of storage type `<SampledImage`.

typeusamplerBuffer
A plain type of supertype `gsamplerBuffer$4` and of storage type `<SampledImage`.

typeusamplerCube
A plain type of supertype `gsamplerCube$4` and of storage type `<SampledImage`.

typeusamplerCubeArray
A plain type of supertype `gsamplerCubeArray$4` and of storage type `<SampledImage`.

inlineatomicAdd (*mem* , *data*)

inlineatomicAnd (*mem* , *data*)

inlineatomicCompSwap (*mem* , *compare* , *data*)

inlineatomicExchange (*mem* , *data*)

inlineatomicMax (*mem* , *data*)

inlineatomicMin (*mem* , *data*)

inlineatomicOr (*mem* , *data*)

inlineatomicXor (*mem* , *data*)

inlinebarrier ()

inlinegroupMemoryBarrier ()

inlineiimage1D (*format*)

inlineiimage1DArray (*format*)

inlineiimage2D (*format*)

inlineiimage2DArray (*format*)

inlineiimage2DMS (*format*)

inlineiimage2DMSArray (*format*)

inlineiimage2DRect (*format*)

inlineiimage3D (*format*)

inlineiimageBuffer (*format*)

inlineiimageCube (*format*)

inlineiimageCubeArray (*format*)

inlineimage1D (*format*)

inlineimage1DArray (*format*)

inlineimage2D (*format*)

inlineimage2DArray (*format*)

inlineimage2DMS (*format*)

inlineimage2DMSArray (*format*)

inlineimage2DRect (*format*)

inlineimage3D (*format*)

inlineimageBuffer (*format*)

inlineimageCube (*format*)

inlineimageCubeArray (*format*)

inlineimageLoad (*image* , *coord*)

inlineimageStore (*image* , *coord* , *data*)

inlinelocal_size (*x* , *y* , *z*)

inlinememoryBarrier ()

inlinememoryBarrierBuffer ()

inlinememoryBarrierImage ()

inlinememoryBarrierShared ()

inlinetexelFetch (*sampler* , *P* , *...*)

inlinetexelFetchOffset (*sampler* , *P* , *lod* , *offset*)

inlinetexture (*sampler* , *P* , *...*)

inlinetextureGather (*sampler* , *P* , ...)

inlinetextureLod (*sampler* , *P* , *lod*)

inlinetextureOffset (*sampler* , *P* , *offset* , ...)

inlinetextureProj (*sampler* , *P* , ...)

inlinetextureQueryLevels (*sampler*)

inlinetextureQueryLod (*sampler* , *P*)

inlinetextureSamples (*sampler*)

inlinetextureSize (*sampler* , ...)

inlineuimage1D (*format*)

inlineuimage1DArray (*format*)

inlineuimage2D (*format*)

inlineuimage2DArray (*format*)

inlineuimage2DMS (*format*)

inlineuimage2DMSArray (*format*)

inlineuimage2DRect (*format*)

inlineuimage3D (*format*)

inlineuimageBuffer (*format*)

inlineuimageCube (*format*)

inlineuimageCubeArray (*format*)

sugar(**buffer** ...)

sugar(**fragment_depth** ...)

sugar(**in** ...)

sugar(**inout** ...)

sugar(**inout-geometry** ...)

sugar(**input_primitive** ...)

sugar(**out** ...)

sugar(**output_primitive** ...)

sugar(**shared** ...)

sugar(**uniform** ...)

compiledfnEmitVertex (...)
An external function of type `void<-()`.

compiledfnEndPrimitive (...)
An external function of type `void<-()`.

compiledfnpackHalf2x16 (...)
An external function of type `u32<-(vec2)`.

compiledfnpackSnorm2x16 (...)
An external function of type `u32<-(vec2)`.

compiledfnpackSnorm4x8 (...)
An external function of type `u32<-(vec4)`.

compiledfnpackUnorm2x16 (...)
An external function of type `u32<-(vec2)`.

compiledfnpackUnorm4x8 (...)
An external function of type `u32<-(vec4)`.

compiledfnunpackHalf2x16 (...)
An external function of type `vec2<-(u32)`.

compiledfnunpackSnorm2x16 (...)
An external function of type `vec2<-(u32)`.

compiledfnunpackSnorm4x8 (...)
An external function of type `vec4<-(u32)`.

compiledfnunpackUnorm2x16 (...)
An external function of type `vec2<-(u32)`.

compiledfnunpackUnorm4x8 (...)
An external function of type `vec4<-(u32)`.

6.10 itertools

itertools provides various utilities which simplify the composition of generators and collectors.

definedrain
A constant of type *Collector*.

inline->> (*generator* , *collector...*)

inlinecascade (*collector...*)
two collectors: every time a is full, b collects a and a is reset when b ends, the remainder of a is collected

inlinecat (*coll*)
treat input as a generator and forward its arguments individually

inlinecollect (*coll*)
run collector until full and return the result

inlinedemux (*init-value* , *f* , *collector...*)

inlineeach (*generator* , *collector*)
fold output from generator into collector

inlinefilter (*f* , *coll*)

inlineflatten (*coll*)
collect variadic input as individual single items

inlinegate (*f* , *a* , *b*)
if f is true, collect input in a, otherwise collect in b when both are full, output both until then, new input for full containers is discarded

inlineimap (*gen* , *f*)

inlineipair (*gen* , *N*)
generate one variadic argument from N generated arguments

inlinelimit (*f* , *coll*)

inlinemap (*f* , *coll*)

inlinemux (*collector...*)
send input into multiple collectors which each fork the target collector

inlinereduce (*init* , *f*)

inlinetake (*n* , *coll*)
limit collector to output n items

sugar(**-->** ...)
Expands a processing chain into nested expressions so that each expression is passed as tailing argument to the following expression.

___ can be used as a placeholder token to position the previous expression.

example:

```
--> x
    f
    g
    h 2 ___
    k
```

expands to:

```
k
    h 2
        g
            f x
```

spicecompose (...)

spicejoin (...)

spicespan (...)

spicezip (...)

6.11 Map

This module implements a key -> value store and mathematical sets using hashtables.

typeMap
An opaque type of supertype *Struct*.

inline__as (*cls* , *T*)

inline__countof (*self*)

fn__drop (*self*)

inline__typecall (*cls* , *opts...*)

fnclear (*self*)

fndiscard (*self* , *key*)
erases a key -> value association from the map; if the map does not contain this key, nothing happens.

fndump (*self*)

fnget (*self* , *key*)
returns the value associated with key or raises an error

fngetdefault (*self* , *key* , *value*)
returns the value associated with key or raises an error

fnin? (*self* , *key*)

fnset (*self* , *key* , *value*)
inserts a new key -> value association into map; key can be the output of any custom hash function. If the key already exists, it will be updated.

fnterseness (*self*)
computes the hashmap load as a normal between 0.0 and 1.0

typeMapError
An unique type of supertype *Enum* and of storage type `{u8 {<i8 x 1>}}`.

typeSet
An opaque type of supertype *Struct*.

inline__as (*cls* , *T*)

inline__countof (*self*)

fn__drop (*self*)

inline__typecall (*cls* , *opts...*)

fnclear (*self*)

fndiscard (*self* , *key*)
erases a key -> value association from the map; if the map does not contain this key, nothing happens.

fndump (*self*)

fnin? (*self* , *key*)

fninsert (*self* , *key*)
inserts a new key into set

fnterseness (*self*)
computes the hashmap load as a normal between 0.0 and 1.0

6.12 spicetools

spicetools provides various utilities which aid with the implementation of new spices.

fngen-argument-matcher (*failfunc* , *expr* , *scope* , *params*)

inlineparse-argument-matcher (*failfunc* , *expr* , *scope* , *params* , *cb*)

sugar(**spice-match** ...)

6.13 struct

Support for defining structs through the *struct* sugar.

sugar(**struct** ...)

6.14 testing

The testing module simplifies writing and running tests in an ad-hoc fashion.

typeOne
: this type is used for discovering leaks and double frees. It holds an integer value as well as a pointer to a single reference on the heap which is 1 as long as the object exists, otherwise 0. The refcount is leaked in order to not cause segfaults when a double free occurs.

  In addition, a global refcounter is updated which can be checked for balance.

  inline__!= (*cls* , *T*)

  inline__< (*cls* , *T*)

  inline__<= (*cls* , *T*)

  inline__== (*cls* , *T*)

  inline__> (*cls* , *T*)

  inline__>= (*cls* , *T*)

  fn__drop (*self*)

  fn__repr (*self*)

  inline__typecall (*cls* , *value*)

  fncheck (*self*)

  fnrefcount ()

  fntest-refcount-balanced ()

  fnvalue (*self*)

sugar(**features** ...)

sugar(**test** ...)

sugar(**test-compiler-error** ...)

sugar(**test-error** ...)

sugar(**test-modules** ...)

6.15 UTF-8

This module provides UTF-8 encoder and decoder collectors, as well as an UTF-8 aware *char* function.

inlinedecoder (*coll*)
: convert a i8 character stream as UTF-8 codepoints of type i32. invalid bytes are forwarded as negative numbers; negating the number yields the offending byte character.

inlineencoder (*coll*)
: convert an integer codepoint to i8 byte array chunks of 1 to 4 bytes length the collector forwards two arguments, the number of bytes required and a buffer containing those bytes, which is only valid for this call.

spicechar (...)

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z