ShenScript

Robert Koeninger

Jul 28, 2021

INTRODUCTION

An implementation of the Shen Language by Mark Tarver for JavaScript. Built for modern browsers and recent versions of Node, requiring the latest features of the ECMAScript standard.

CHAPTER

ONE

MOTIVATION

JavaScript is one of the most commonly used languages in the world, running in every browser and on almost every platform, as well as being the basis for the prolific Node.js platform. It has some built-in capabilities like dynamic expression evaluation, async/await syntax and highly optimised runtimes that make it a preferred target. It's not a perfect match, however and the details of how the gaps between JavaScript and Shen are bridged is detailed in this documentation.

The purpose of building ShenScript is to bring the powerful functionality inherent in the Shen language to web development. And with the way ShenScript is implemented, asynchronous code is handled transparently so the Shen developer doesn't have to think about the distinction between a synchronous call and an async one. This is very helpful in the JavaScript ecosystem where asynchronous operations are everywhere.

CHAPTER

TWO

PRIOR ART

Before going into detail on ShenScript, I wanted to highlight and say thank you for pre-existing work on Shen and on the porting of Shen to JavaScript that both this port and I personally have benefitted from studying.

2.1 shen-js

Shen has previously been brought to JavaScript by way of the shen-js project by Ramil Farkhshatov. shen-js implements its own KLVM on top of JS, allowing it to handle deep recursion without stack overflow and make asynchronous I/O transparent to Shen code. I wanted to make a JavaScript port that is built more directly on JavaScript and makes use on newer features. ShenScript is also intended to produce a smaller deployable (<1MB vs ~12MB for shen-cl) that is retrievable from npm.

2.2 shen-cl

ShenScript also takes inspiration from the original Shen port, shen-cl. Shen for Common Lisp offers a good demonstration of how to embed Shen's semantics in another dynamic language. shen-cl also has good native interop which this port has attempted to replicate.

CHAPTER

THREE

ENCODING OF SEMANTICS

Many of Shen's semantics are translated directly to matching concepts in JavaScript. Where there are incongruities, they are described here.

3.1 Data Types

numbers, strings, exceptions, absvectors Shen numbers, strings, exceptions (Error) and absvectors (array) are just their related JavaScript types with no special handling.

symbols Shen symbols are interned JavaScript symbols fetched with `Symbol.for` so two Shen symbols with the same name will be equivalent.

empty list JavaScript `null` is used for the empty list.

conses ShenScript declares a `Cons` class specifically to represent conses and cons lists.

functions, lambdas, continuations All function-like types are JavaScript functions that have been wrapped in logic to automatically perform partial and curried application.

streams Streams are implemented in a custom way by the host. Typically, they are objects with a `close` function and a `readByte` or `writeByte` function.

3.2 Special Forms

if, and, or, cond Conditional code translates directly into matching JavaScript conditionals. `cond` expressions are turned into `if - else` chains.

simple-error, trap-error Error handling works just like JavaScript, using the `throw` and `try-catch` constructs.

Both `throw` and `try` are statement syntax in JavaScript, so to make them expressions, there is the `raise` function for throwing errors and `try-catch` is wrapped in an iife so it can be embedded in expressions. That iife is async and its immediate invocation is awaited.

defun, lambda, freeze All function forms build and return JavaScript functions.

let Local variable bindings are translated into immediately-invoked lambda expressions. This is both for brevity and it because it's the most natural way to translate Lisp-style let-as-an-expression since variable declarations are statements in JavaScript.

Consider that the form `(let X Y Z)` can be transformed into the behaviorally equivalent `((lambda X Z) Y)`. ShenScript does not do it precisely this way because of involved optimizations, but that's the idea.

do The do form isn't officially a form in Shen, just a function that returns it's second argument, but it is handled as a form in ShenScript in order to take advantage of an additional opportunity for tail-call optimisation.

3.3 Shen Booleans vs JavaScript Booleans

Shen uses the symbols `true` and `false` for booleans and does not have truthy or falsy semantics like JavaScript or other Lisps. This can make things tricky since Shen's `true` and `false` will always be considered `true` by JavaScript, and in JavaScript, anything not falsy will count as `true` and `false`.

The KLambda-to-JavaScript transpiler does not actually consider booleans to be their own datatype, it treats them as any other symbol.

When doing interop between Shen and JavaScript, it will necessary to carefully convert between the two boolean representations as with so much JavaScript code's dependence on truthy/falsy semantics, there is no general way of doing so.

3.4 Equality

Equality sematics are implemented by the `equate` function in the `backend` module.

Values are considered equivalent in ShenScript if they are equal according to the JavaScript `===` operator or in the following cases:

- Both values are `Cons` and both their `head` and `tail` are equal according to `equate`.
- Both values are JavaScript arrays of the same length and all of their values are equal according to `equate`.
- Both values are JavaScript objects with the same constructor, same set of keys and the values for each key are equal according to `equate`.

3.5 Partial Function Application

In Shen, functions have precise arities, and when a function is applied to fewer arguments than it takes, it returns a function that takes the remaining arguments. So since `+` takes two arguments, if it is applied to a single one, as in `(+ 1)`, the result is a function that takes one number and returns `1` plus that number.

ShenScript also supports curried application, where there are more arguments than the function actually takes. The function is applied to the first `N` arguments equal to the function's arity, the result is asserted to be a function, and then the resulting function is applied to the remaining arguments. Repeat this process until there are no remaining arguments or until a non-function is returned and an error is raised.

> **Warning:** Curried application is not a part of the Shen standard, is not supported by shen-cl, and might be removed from ShenScript.

This is implemented in ShenScript for primitives, kernel functions and user-defined functions by wrapping them in a function that takes a variable number of arguments, checks the number passed in, and then returns another function to take the remaining arguments, performs curried application, or simply returns the result.

3.6 Tail-Call Optimization

Tail-call optimization is required by the Shen standard and Shen code make prolific use of recursion making TCO a necessity.

In ShenScript, tail calls are handled dynamically using trampolines. When a function is built by the transpiler, the lexical position of expressions are tracked as being in head or tail position. Function calls in head position are a simple invocation and the result is settled; calls in tail position are bounced.

settle Settling is the process of taking a value that might be a `Trampoline`, checking if it's a tramoline, and if it is, running it. The result of running the trampoline is checked if it's a trampoline, and if so, that is run and this process is repeated until the final result is a non-trampoline value, which is returned.

bounce Bouncing a function call means making a trampoline from a reference to the function and the list of arguments and returning. The function will actually be invoked when the trampoline is settled at some point later.

CHAPTER

FOUR

GENERATION OF SYNTAX

4.1 Generating JavaScript

ShenScript code generation is built on rendering objects rendering a JavaScript abstract syntax tree conforming to the informal ESTree standard. These ASTs are then rendered to strings using the astring library and then evaluated with an immediately-applied Function constructor.

The Function constructor acts as a kind of isolated-scope version of eval. When a Function is created this way, it does not capture local variables like eval does. This prevents odd behavior from cropping up where Shen code uses a variable or function name that matches some local variable.

4.2 Hoisting Globals and Idle Symbols

Lookups of global functions, global symbol values (`set`/`value`) and idle symbols (`Symbol.for`) don't take up much time, but when done repeatedly are wasteful. To prevent repeated lookups, references to the aforementioned are hoisted to the top of each expression tree evaluated by `eval-kl` and to the top of the pre-rendered kernel. This way, they only get fetched once and are enclosed over so each time a function is called that depends on one of these, it only needs to access a local variable in scope.

Global functions and global symbol values with the same name are both attached to the same lookup Cells and those Cells get assigned to escaped local variables with `$c` appended to the end. Idle symbols get assigned to escaped local variables with `$s` appended to the end.

4.3 Fabrications

Just as a `Context` carries information downward while building an AST, a `Fabrication` carries resulting context back upward. A fabrication contains the subsection of the AST that was built, along with the results of decisions that were made down in that tree so it doesn't have to be scanned again.

Fabrications are useful because they are easily composible. There is a composition function for fabrications called `assemble` which takes a function to combine ASTs and a list of fabrications as arguments. The combining function gets us a single AST and the rest of the metadata being carried by the fabrications has a known means of combination. The result is a single AST and a single body of metadata out of which a single fabrication is made.

At the moment, the only additional information fabrications carry is a substitution map of variable names and the expressions that they will need to be initialized with. The substitution map is used to "hoist" global references to the top of the scope being constructed. When an AST is constructed that depends on one of these substitutions, it refers to a variable by the name specified as a key in the map.

4.4 Escaping Special Variable Names

There are still some variables that need to be accessible to generated code, but not to the source code - referenceable in JavaScript, but not Shen. The main example is the environment object, conventionally named `$`. Dollar signs and other special characters get escaped by replacing them with a dollar sign followed by the two-digit hex code for that character. Since dollar signs are valid identifier characters in JavaScript, hidden environment variables can be named ending with dollar sign, because if a Shen variable ends with `$`, the escaped JavaScript name will have a trailing `$24` instead.

4.5 Dynamic Type-Checking

Many JavaScript operators, like the + operator, are not limited to working with specific types like they are in Shen. In JavaScript, + can do numberic addition, string concatenation and offers a variety of strange behaviors are argument types are mixed. In Shen, the + function only works on numbers and passing a non-number is an error.

So in order to make sure these primitive functions are being applied to the correct types, there are a series of `is` functions like `isNumber`, `isString` and `isCons`, which determine if a value is of that type. There are also a series of `as` functions for the same types which check if the argument is of that type and returns it if it is, but raises an error if it is not.

This allows concise definition of primitive functions. The + primitve is defined like this:

```
(x, y) => asNumber(x) + asNumber(y)
```

and the `cn` primitive is defined like this:

```
(x, y) => asString(x) + asString(y)
```

4.6 Code Inlining and Optimisation

To reduce the volume of generated code, and to improve performance, most primitive operations are inlined when fully applied. Since the type checks described in the previous section are still necessary, they get inlined as well, but can be excluded on certain circumstances. For instance, when generating code for the expression `(+ 1 X)`, it is certain that the argument expression 1 is of a numeric type as it is a numeric constant. So instead of generating the code `asNumber(1) + asNumber(X)`, we can just render `1 + asNumber(X)`.

The transpiler does this simple type inference following a few rules:

- Literal numeric, string and idle symbol values are inferred to be of those types.
  - `123` is `Number`.
  - "`hello`" is `String`.
  - `thing` is `Symbol`.
- The value returned by a primitive function is inferred to be of that function's known return type, regardless of the type of the arguments. If the arguments are of an unexpected type, an error will be raised anyway.
  - `(+ X Y)` is `Number`.
  - `(tlstr X)` is `String`.
  - `(cons X Y)` is `Cons`.
- Local variables in `let` bindings are inferred to be of the type their bound value was inferred to be.

  - The `X` in `(+ X Y)` in `(let X 1 (+ X Y))` is `Number`. `X` would not need an `asNumber` cast in `(+ X Y)`.

- The parameter to a lambda expression used as an error handler in a `trap-error` form is inferred to be `Error`.
  - `(trap-error (whatever) (/. E (error-to-string E)))` does not generate an `asError` check for `E`.

More sophisticated analysis could be done, but with dimishing returns in the number of cases it actually catches. And consider that user-defined functions can be re-defined, either in a REPL session or in code loaded from a file, meaning assumptions made by optimised functions could be invalidated. When a function was re-defined, all dependent functions would have to be re-visited and potentially all functions dependent on those functions. That's why these return type assumptions are only made for primitives.

4.7 Pervasive Asynchronocity

I/O functions, including primitives like `read-byte` and `write-byte` are idiomatically synchronous in Shen. This poses a problem when porting Shen to a platform like JavaScript which pervasively uses asynchronous I/O.

Functions like `read-byte` and `open` are especially problematic, because where `write-byte` and `close` can be fire-and-forget even if they do need to be async, Shen code will expect `read-byte` and `open` to be blocking and the kernel isn't designed to await a promise or even pass a continuation.

In order to make the translation process fairly direct, generated JavaScript uses async/await syntax so that code structured like synchronous code can actually be asynchronous. This allows use of async functions to look just like the use of sync functions in Shen, but be realized however necessary in the host language.

`async` code generation is controlled by a flag on the `Context` object that the compiler disables for functions that it is sure can be synchronous.

CHAPTER

FIVE

FUTURE DESIGN OPTIONS

Some designs are still up in the air. They either solve remaining problems like performance or provide additional capabilites.

5.1 Polychronous Functions

Currently, it is difficult to be sure an execution path will be entirely synchronous so the compiler plays it safe and only renders functions as sync when it is sure that function and its referents are sync.

Instead, the compiler could render both sync and async versions of most functions and choose which to call on a case-by-case basis. And when a referent is re-defined from sync to async or vice-versa, the environment could quickly switch the referring function from preferring one mode or the other.

For example, if we have Shen code like `(map F Xs)` and `F` is known to be sync, we can call the sync version of `map` which is tail-recursive or is a simple for-loop by way of a pinhole optimisation. This way, we won't have to evaluate the long chain of promises and trampolines the async version would result in for any list of decent length.

The compiler would have to keep track of additional information like which functions are always sync, which are always async and which can be one or the other and based on what criteria.

5.2 KLambda-Expression Interpreter

Some JavaScript environments will have a Content Security Policy enabled that forbids the use of `eval` and `Function`. This would completely break the current design of the ShenScript evaluator. The transpiler would continue to work, and could produce JavaScript ASTs, but they could not be evaluated.

A scratch ESTree interpreter could be written, but as it might need to support most of the capabilities of JavaScript itself, it would easier to write an interpreter that acts on the incoming KLambda expression trees themselves and forgo the transpiler entirely.

The obvious downside is that the interpreter would be much slower that generated JavaScript which would enjoy all the optimisations built into a modern runtime like V8. The interpreter would only be used when it was absolutely necessary.

5.3 Expression Type Tracking

Right now, the compiler only tracks the known types of local variables and nested expressions. It could also retain information like `(tl X)` is a cons and not just that `X` is a cons and then have to check again later when evaluating `(hd (tl X))`. This would remove plenty of `asCons` calls that are not strictly necessary.

CHAPTER

SIX

HISTORICAL AND ABANDONED DESIGN

6.1 String Concatenation

In a much earlier version, code generation was done with JavaScript template strings and string concatenation. This was replaced with the use of the astring library since it is cleaner, more reliable and more flexible to have an AST that can undergo further manipulation as opposed to a final code string that can only be concatenated.

6.2 Using Fabrications for Statement-oriented Syntax

Shen's code style is very expression-oriented and is most easily translated to another expression-oriented language. Most of Shen's forms translate directly to expression syntax in JavaScript without a problem. All function calls are expressions in JavaScript, conditions can use the `? :` ternary operator, etc. Two forms in particular pose a problem: `let` and `trap-error` would most naturally be represented with statements in JavaScript.

We could just emit `VariableDeclaration` and `TryStatement` AST nodes for these forms, but that causes a compilication when either a statement or an expression might be emitted by the transpiler at any point in the code. And while it's easy to embed an expression in a statement construct - just by wrapping in an `ExpressionStatement` - it's harder to embed a statement in an expression.

An expression like `(+ 1 (trap-error (whatever) (/. _ 0)))` would normally be rendered like `1 + asNumber(trap(() => whatever(), _ => 0))`. How would it be rendered if we wanted to inline a `try-catch` instead of using the `trap` function?

The concept of a Fabrication (aka "fabr") was introduced to represent the composition of the two forms of syntax. Whenever a child form is built, it would return a fabr, consisting of a list of prerequiste statements and a resulting expression. Fabrs are typically composed by making a new fabr with all the statements from the first fabr, followed by the statements from the second fabr and then the result expressions are combined as they would be in the current design.

Since every fabr needs a result expression, for statement syntax, an additional variable declaration is added for a result variable and the result of the fabr is just the identifier expression for that variable.

An example like `(+ (let X 3 (* X 2)) (trap-error (whatever) (/. _ 0)))` would get rendered like this. The `(let X 3 (* X 2))` expression becomes:

```
{
  "statements": [
    << const X = 3; >>
  ],
  "result": << X * 2 >>
}
```

The `(trap-error (whatever) (/. _ 0))` becomes:

```
{
  "statements": [
    << let R123$; >>,
    <<
      try {
        R123$ = settle(whatever());
      } catch (e$) {
        R123$ = 0;
      }
    >>
  ],
  "result": << R123$ >>
}
```

And composed together, they are:

```
{
  "statements": [
    << const X = 3; >>
    << let R123$; >>,
    <<
      try {
        R123$ = settle(whatever());
      } catch (e$) {
        R123$ = 0;
      }
    >>
  ],
  "result": << (X * 2) + asNumber(R123$) >>
}
```

This whole approach was attempted on the premise that using more idiomatic JavaScript syntax would give the runtime more opportunities to identify optimisations vs using `trap` and immediately-invoked lambdas. Turns out using fabrs produced about twice the code volume and benchmarks took 3-4 times as long to run. I guess V8 is really good at optimising IIFEs. So fabrs were reverted. The design is documented here for historical reasons.

This approach could be brought back in order to better handle `trap-error` at the root of a lambda, avoiding an additional iife.

Another possibility is this would allow the introduction of `js.while`, `js.for`, etc. types of special forms where Shen code could directly invoke imperative syntax.

CHAPTER

SEVEN

BUILDING AN ENVIRONMENT

module shen

This is the top-level module. The `exports` of this module is a function that constructs a full populated ShenScript environment object.

`(options) => $`

> **Parameters** `options` (*object*) – Can have all of the same properties as the options object accepted by the `backend` function.
>
> **Returns** A complete ShenScript environment.

The default configuration options for this environment are specified in the `config` module and environment-derived properties are in the `config.node` and `config.web` modules. Any of these can be overwritten by specifying them in the `options` to the `shen` function.

CHAPTER

EIGHT

THE KERNEL SANDWICH

A full ShenScript environment is created by initialising a new backend with the options passed into the top-level, running that through the pre-rendered kernel and then applying the frontend decorator for whichever node or web environment is specified in the options. The composition looks like `frontend(kernel(backend(options)))`. I call this "The Kernel Sandwich".

8.1 The Backend

module backend

The backend module contains the KLambda-to-JavaScript transpiler, global function and symbol indexes and proto-primitives for conses, trampolines, equality and partial application.

The `exports` of this module is just a function that constructs a new ShenScript environment object, which is conventionally named `$`.

`(options = {}) => $`

Parameters

- **options** (*Object*) – Environment config and overrides.
- **options.clock** (*function*) – Provides current time in fractional seconds from the Unix epoch. Defaults to `() => Date.now`.
- **options.homeDirectory** (*string*) – Initial working directory in file system. Defaults to "/".
- **options.implementation** (*string*) – Name of JavaScript platform in use. Defaults to "Unknown".
- **options.InStream** (*class*) – Class used for input streams. Not required if `isInStream` and `openRead` are specified.
- **options.OutStream** (*class*) – Class used for output streams. Not required if `isInStream` and `openRead` are specified.
- **options.isInStream** (*function*) – Returns true if argument is an `InStream`. Defaults to a function that returns false.
- **options.isOutStream** (*function*) – Returns true if argument is an `OutStream`. Defaults to a function that returns false.
- **options.openRead** (*function*) – Opens an `InStream` for the given file path. Defaults to a function that raises an error.

- **options.openWrite** (*function*) – Opens an OutStream for the given file path. Defaults to a function that raises an error.
- **options.os** (*string*) – Name of operating system in use. Defaults to "Unknown".
- **options.port** (*string*) – Current version of ShenScript. Defaults to "Unknown".
- **options.porters** (*string*) – Author(s) of ShenScript. Defaults to "Unknown".
- **options.release** (*string*) – Current version of JavaScript platform in use. Defaults to "Unknown".
- **options.sterror** (*string*) – OutStream for error messages. Defaults to stdoutput.
- **options.stinput** (*string*) – InStream for standard input. Defaults to an object that raises an error.
- **options.stoutput** (*string*) – OutStream for standard output. Defaults to an object that raises an error.

Returns An object conforming to the Backend class description.

class Backend

This class is a description of object returned by the backend function and does not actually exist. It contains an initial ShenScript environment, without the Shen kernel loaded.

Parameters

- **assemble** (*function*) – Composes a sequence of JavaScript ASTs and Fabrications into a single Fabrication.
- **assign** (*function*) – Initialize or set a global symbol.
- **bounce** (*function*) – Creates a trampoline from function and rest arguments.
- **compile** (*function*) – Turns KLambda expression array tree into JavaScript AST.
- **construct** (*function*) – Turns KLambda expression array tree into Fabrication.
- **cons** (*function*) – Creates a Cons from a head and tail.
- **defun** (*function*) – Adds function to the global function registry.
- **equate** (*function*) – Determines if two values are equal according to the semantics of Shen.
- **evalJs** (*function*) – Evalutes a JavaScript AST in isolated scope with access to $.
- **evalKl** (*function*) – Builds and evaluates a KLambda expression tree in isolated scope with access to $.
- **globals** (*Map*) – Map of symbol names to lookup Cells.
- **inline** (*function*) – Registers an inlining rule.
- **lookup** (*function*) – Looks up Cell in globals, adding one if it doesn't exist yet.
- **settle** (*function*) – If value is a Trampoline, runs Trampoline and repeats.
- **show** (*function*) – toString function. Returns string representation of any value.
- **valueOf** (*function*) – Returns the value of the given global symbol. Raises an error if it is not defined.

8.2 The Kernel

module `kernel`

The `kernel` module contains a JavaScript rendering of the Shen kernel that can be installed into a ShenScript environment.

The `exports` of this module is just a function that augments an environment and returns it.

`($) => $`

> **Parameters** `$` (*object*) – A ShenScript environment to add functions to.
>
> **Returns** Same `$` that was passed in, conforming to the `Kernel` class.

`Kernel extends Backend`

> This class is a description of object returned by the `kernel` module and does not actually exist. It contains a primitive ShenScript environment along with the Shen kernel and it adequate to run standard Shen programs.
>
> The `Kernel` virtual class adds no members, but does imply additional entries in the `globals` map.

8.3 The Frontend

module `frontend`

The frontend module augments a ShenScript environment with JavaScript- and ShenScript-specific functionality.

Functionality provided includes:

- `js` package functions that allow access to common JavaScript types, objects and functions.
- `js.ast` package functions that allow generation, rendering and evaluation of JavaScript code.
- `shen-script` package functions that allow access to ShenScript environment internals.

The `exports` of this module is just a function that augments an environment and returns it.

`($) => $`

> **Parameters** `$` (*object*) – A ShenScript environment to add functions to.
>
> **Returns** Same `$` that was passed in, conforming to the `Frontend` class.

`Frontend extends Kernel`

> This class is a description of object returned by the `frontend` function and does not actually exist. It contains a complete ShenScript environment.
>
> **Parameters**
>
> - **`caller`** (*function*) – Returns a function that invokes the function by the given name, settling returned Trampolines.
> - **`define`** (*function*) – Defines Shen function that defers to given JavaScript function.
> - **`defineTyped`** (*function*) – Defines Shen function that defers to given JavaScript function and declares with the specified Shen type signature.
> - **`defmacro`** (*function*) – Defines a Shen macro in terms of the given JavaScript function.
> - **`evalShen`** (*function*) – Evaluates Shen expression tree in isolated environment.
> - **`exec`** (*function*) – Parses string as Shen source, evaluates each expression and returns last result.

- **execEach** (*function*) – Parses string as Shen source, evaluates each expression and returns an array of the results.
- **load** (*function*) – Loads Shen code from the given file path.
- **parse** (*function*) – Returns parsed Shen source code as a cons tree.
- **pre** (*function*) – Registers a preprocessor function.
- **symbol** (*function*) – Declares a global symbol with the given value and a function by the same name that retrieves the value.

Returns Same `$` that was passed in.

8.3.1 The Node Frontend

module `frontend.node`

Further adds `node` package helpers for interacting with the capabilites of the Node.js runtime.

Functions are described here.

8.3.2 The Web Frontend

module `frontend.web`

Further adds `web` package helpers for interacting with the capabilites of a web browser or electron instance.

Functions are described here.

CHAPTER

NINE

INTEROP FROM JAVASCRIPT TO SHEN

The environment object, `$`, comes with additional functions to make JavaScript functions callable from Shen, setting global symbols, declaring types and macros, etc.

9.1 Exported Functions

Important: Some of these will return a promise since they invoke kernel functions. Recommended practice would be to await any calls made on the environment object.

`assign`(*name*, *value*)
Assigns a value to a global symbol, creating that global symbol if necessary.

Parameters

- **name** (*string*) – Global symbol name.
- **value** (*any*) – The value to assign.

Returns The assigned value.

`caller`(*name*)
Returns a handle to a function in the Shen environment which automatically performs trampoline settling.

Parameters **name** (*string*) – Function name.

Returns A function to call Shen function by given name. Returned function will be async.

`cons`(*x*, *y*)
Creates a new `Cons` cell with the given `head` and `tail` values.

Parameters

- **x** (*any*) – Any Shen or JavaScript value.
- **y** (*any*) – Any Shen or JavaScript value.

Returns A new `Cons`.

`define`(*name*, *f*)
Defines a new global function in the Shen environment by the given name. Function gets wrapped so it automatically handles partial application.

Parameters

- **name** (*string*) – Name which function will be accessible by, including package prefix(es).
- **f** (*function*) – JavaScript function to defer fully-applied invocation to.

Returns Name as a symbol.

defineTyped(*name*, *type*, *f*)

Defines a new global function in the Shen environment by the given name and declared Shen type signature. Function gets wrapped so it automatically handles partial application.

Parameters

- **name** (*string*) – Name which function will be accessible by, including package prefix(es).
- **type** (*any*) – Shen type signature in array tree form, gets recursively converted to Shen lists.
- **f** (*function*) – JavaScript function to defer fully-applied invocation to.

Returns Name as a symbol.

defmacro(*name*, *f*)

Defines a macro in the Shen environment by the given name. Syntax gets pre-processed with `toArrayTree` before being passed into wrapped function. Result returned from `f` gets post-processed with `toListTree`. If `f` returns `undefined`, then it causes the macro to have no effect.

Parameters

- **name** (*string*) – Name which macro function will be accessible by, including package prefix(es).
- **f** (*function*) – JavaScript function to defer invocation to.

Returns Name as a symbol.

equate(*x*, *y*)

Determines if `x` and `y` are equal using Shen-specific semantics.

Parameters

- **x** (*any*) – Any Shen or JavaScript value.
- **y** (*any*) – Any Shen or JavaScript value.

Returns A JavaScript boolean.

evalJs(*ast*)

Converts JavaScript AST to JavaScript syntax string and evaluates in isolated context.

Parameters **ast** (*ast*) – JavaScript AST Node.

Returns Evaluation result.

evalKl(*expr*)

Invokes the backend `eval-kl` function which will evaluate the expression tree in the Shen environment.

Parameters **expr** (*expr*) – Parsed KLambda expression tree.

Returns Evaluation result.

evalShen(*expr*)

Invokes the Shen `eval` function which will evaluate the expression tree in the Shen environment.

Parameters **expr** (*expr*) – Parsed Shen expression tree.

Returns Evaluation result.

exec(*syntax*)

Parses and evaluates all Shen syntax forms passed in. Returns final evaluation result.

Parameters **syntax** (*string*) – Shen syntax.

Returns Final evaluation result.

execEach(*syntax*)

Parses and evaluates all Shen syntax forms passed in. Returns array of evaluation results.

Parameters **syntax** (*string*) – Shen syntax.

Returns Array of evaluation results.

inline(*name*, *dataType*, *paramTypes*, *f*)

Registers a new inline rule for JavaScript code generation. When the KLambda-to-JavaScript transpilier encounters a form starting with a symbol named `name`, it will build child forms and then pass the rendered ASTs into `f`. Whatever `f` returns gets inserted into the greater JavaScript AST at the relative point where the form was encountered.

Example:

```
// Renders a `(not X)` call with the JavaScript `!` operator
// and inserts type conversions only as needed
inline('not', 'JsBool', ['JsBool'], x => Unary('!', x));
```

Parameters

- **name** (*string*) – Name of symbol that triggers this rule to be applied.
- **dataType** (*string*) – The type of the whole expression, or `null` if it is unknown.
- **paramTypes** (*array*) – The types of argument expressions, each can be `null` if they are unknown.
- **f** (*function*) – Function that handles the JavaScript AST transformation for this rule.

Returns `f`.

load(*path*)

Invokes the Shen `load` function which will read the file at the given path and evaluates its contents.

Parameters **path** (*string*) – Local file system path relative to `shen.*home-directory*`

Returns The `loaded` symbol.

parse(*syntax*)

Parses Shen syntax using the `read-from-string` function from the Shen kernel.

Parameters **syntax** (*string*) – Shen syntax in string form.

Returns A Shen list of syntax forms, wrapped in a promise.

pre(*name*, *f*)

Registers a new pre-processor rule for JavaScript code generation. Similar to `inline`, but child forms are not rendered and are passed into `f` as KLambda expression trees. `f` should then return a JavaScript AST which will get inserted into the greater JavaScript AST at the relative point where the form was encountered.

Example:

```
// Evaluates math expression at build time and inserts result
// in place of JavaScript AST that would have been built
pre('build-time-math', x => evalShen(x));
```

Parameters

- **name** (*string*) – Name of symbol that triggers this rule to be applied.

- **f** (*function*) – Function that handles the KLambda expression tree to JavaScript AST conversion.

Returns f.

show(*value*)
Builds Shen-specific string representation of value.

Parameters **value** (*any*) – Value to show.

Returns String representation of value.

symbol(*name*, *value*)
Declares Shen global symbol by the given name (with added earmuffs per convention), setting it to given initial value and declaring a function by the given name that accesses it.

Example: symbol('package.thing', 0) declares global symbol package.*thing*, sets it to 0 and declares function package.thing which takes no arguments and returns (value package.*thing*).

Parameters

- **name** (*string*) – Name of accessor function and basis for name of global symbol.
- **value** (*any*) – Value to initialise global symbol with.

Returns value.

toArray(*x*)
Builds a JavaScript array from a Shen list. Elements are not transformed. x is passed through if not a Shen list.

Parameters **x** (*any*) – A Shen list or any other value.

Returns A JavaScript array or whatever was passed in.

toArrayTree(*x*)
Recursively builds a JavaScript array tree from a Shen list tree. Non-list children are not transformed. x is passed through if not a Shen list.

Parameters **x** (*any*) – A tree of Shen lists or any other value.

Returns A tree of JavaScript arrays or whatever was passed in.

toList(*x*[, *tail* = *null*])
Builds a Shen list from a JavaScript array. Elements are not transformed. x is passed through if not a JavaScript array.

Aliased as r for brevity in generated code.

Parameters

- **x** (*any*) – A JavaScript array or any other value.
- **tail** (*any*) – Optional tail value for the final Cons cell, instead of null. Ignored if x is not an array.

Returns A Shen list or whatever was passed in.

toListTree(*x*)
Recursively builds a Shen list tree from a JavaScript array tree. Non-list children are not transformed. x is passed through if not a JavaScript array.

Parameters **x** (*any*) – A tree of nested JavaScript arrays or any other value.

Returns A tree of nested Shen lists or whatever was passed in.

`valueOf`(*name*)

Returns the value of the global symbol with the given name.

Parameters `name` (*string*) – Name of global symbol.

Returns Global symbol's value.

Throws Error if symbol is not bound.

CHAPTER

TEN

INTEROP FROM SHEN TO JAVASCRIPT

ShenScript provides functions in the `js` namespace to access JavaScript standard classes and functionality.

10.1 Raw Operators

Functions starting with `js.raw` allow access to underlying JavaScript operators without any additional typechecks or conversions. Operations are inlined when fully applied but can still be partially applied or passed as arguments like any other function.

(js.raw.== X Y)
Applies the JavaScript == operator to arguments without additional typechecks or conversions.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns A JavaScript boolean.

(js.raw.=== X Y)
Applies the JavaScript === operator to arguments without additional typechecks or conversions.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns A JavaScript boolean.

(js.raw.!= X Y)
Applies the JavaScript != operator to arguments without additional typechecks or conversions.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns A JavaScript boolean.

(js.raw.!== X Y)
Applies the JavaScript !== operator to arguments without additional typechecks or conversions.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns A JavaScript boolean.

`(js.raw.and X Y)`
Applies the JavaScript && operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Operator is inlined when fully applied.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever && does based on JavaScript-specific behavior.

`(js.raw.or X Y)`
Applies the JavaScript || operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Operator is inlined when fully applied.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever || does based on JavaScript-specific behavior.

`(js.raw.not X)`
Performs JavaScript boolean inversion.

Parameters **X** (*any*) – Value to invert.

Returns A JavaScript boolean.

`(js.raw.+ X Y)`
Applies the JavaScript + operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever + does based on JavaScript-specific behavior.

`(js.raw.- X Y)`
Applies the JavaScript - operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever - does based on JavaScript-specific behavior.

`(js.raw.* X Y)`
Applies the JavaScript * operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.

- **Y** (*any*) – Whatever.

Returns Whatever * does based on JavaScript-specific behavior.

(js.raw./ X Y)

Applies the JavaScript / operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever / does based on JavaScript-specific behavior.

(js.raw. X Y)**

Applies the JavaScript ** operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever ** does based on JavaScript-specific behavior.

(js.raw.< X Y)

Applies the JavaScript < operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever < does based on JavaScript-specific behavior.

(js.raw.> X Y)

Applies the JavaScript > operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever > does based on JavaScript-specific behavior.

(js.raw.<= X Y)

Applies the JavaScript <= operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever <= does based on JavaScript-specific behavior.

(js.raw.>= X Y)

Applies the JavaScript >= operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever >= does based on JavaScript-specific behavior.

(js.raw.bitwise.not X)

Applies the JavaScript ! operator to argument without additional typechecks, perserving JavaScript coercion behavior.

Parameters **X** (*any*) – Whatever.

Returns Whatever ! does based on JavaScript-specific behavior.

(js.raw.bitwise.and X Y)

Applies the JavaScript & operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever & does based on JavaScript-specific behavior.

(js.raw.bitwise.or X Y)

Applies the JavaScript | operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever | does based on JavaScript-specific behavior.

(js.raw.bitwise.xor X Y)

Applies the JavaScript ^ operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns Whatever ^ does based on JavaScript-specific behavior.

(js.raw.<< X Y)

Applies the JavaScript << operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Value to shift.
- **Y** (*any*) – Amount to shift by.

Returns Whatever << does based on JavaScript-specific behavior.

(js.raw.>> X Y)

Applies the JavaScript >> operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Value to shift.
- **Y** (*any*) – Amount to shift by.

Returns Whatever >> does based on JavaScript-specific behavior.

`(js.raw.>>> X Y)`
Applies the JavaScript >>> operator to arguments without additional typechecks, perserving JavaScript coercion behavior.

Parameters

- **X** (*any*) – Value to shift.
- **Y** (*any*) – Amount to shift by.

Returns Whatever >>> does based on JavaScript-specific behavior.

`(js.raw.delete Object Key)`
Removes a key from an object.

Parameters

- **Object** (*object*) – Object to remove key from.
- **Key** (*any*) – String or symbol name of key to remove.

Returns JavaScript `true` if the delete was successful.

`(js.raw.eval Code)`

> **Warning:** Using `eval` is even more dangerous than usual in ShenScript because it will be difficult to know what indentifiers will be in scope and how their names might have been aliased when code is evaluated.

Note: With `js.raw.eval`, the call does get inlined when fully applied, which might help a bit with the scoping issues.

So, for instance, `(let X 2 (js.raw.eval "1 + X"))` would successfully evaluate to `3` with `js.raw.eval` where it would fail with `js.eval`.

Calls the built-in JavaScript `eval` function.

Parameters **Code** (*string*) – JavaScript code in string form.

Returns The result of evaluating the code.

`(js.raw.in Key Object)`
Determines if value is a key in an object.

Parameters

- **Key** (*any*) – String or symbol name of a property.
- **Object** (*object*) – Object that might contain a property by that key.

Returns A JavaScript boolean.

`(js.raw.instanceof X Class)`
Determines if value is the product of a constructor, class or anything higher up its prototype chain.

Parameters

- **X** (*any*) – The value to inspect.
- **Class** (*class*) – A class or constructor function.

Returns A JavaScript boolean.

(js.raw.typeof X)

Applies the JavaScript `typeof` operator to a value.

Parameters **X** (*any*) – Anything.

Returns A string identifying the basic type of the value: object, number, string, symbol, undefined, boolean.

(js.raw.void X)

Applies the JavaScript `void` operator to argument, which will always return `undefined`.

Parameters **X** (*any*) – Anything.

Returns `undefined`.

10.2 Typed Operators

js.== : A --> B --> boolean

Applies the JavaScript == operator to arguments without additional typechecks, and converts result to a Shen boolean.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns A Shen boolean.

js.=== : A --> B --> boolean

Applies the JavaScript === operator to arguments without additional typechecks, and converts result to a Shen boolean.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns A Shen boolean.

js.!= : A --> B --> boolean

Applies the JavaScript != operator to arguments without additional typechecks, and converts result to a Shen boolean.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns A Shen boolean.

js.!== : A --> B --> boolean

Applies the JavaScript !== operator to arguments without additional typechecks, and converts result to a Shen boolean.

Parameters

- **X** (*any*) – Whatever.
- **Y** (*any*) – Whatever.

Returns A Shen boolean.

js.% : number --> number --> number

Checks arguments are numbers and then applies the JavaScript % operator.

Parameters

- **X** (*number*) – A Shen number.
- **Y** (*number*) – A Shen number.

Returns A Shen number.

js. : number --> number --> number**

Checks arguments are numbers and then applies the JavaScript ** operator.

Parameters

- **X** (*number*) – A Shen number.
- **Y** (*number*) – A Shen number.

Returns A Shen number.

js.bitwise.not : number --> number

Checks argument is a number and then applies the JavaScript ~ operator.

Parameters **X** (*number*) – A Shen number.

Returns A Shen number.

js.bitwise.and : number --> number --> number

Checks arguments are numbers and then applies the JavaScript & operator.

Parameters

- **X** (*number*) – A Shen number.
- **Y** (*number*) – A Shen number.

Returns A Shen number.

js.bitwise.or : number --> number --> number

Checks arguments are numbers and then applies the JavaScript | operator.

Parameters

- **X** (*number*) – A Shen number.
- **Y** (*number*) – A Shen number.

Returns A Shen number.

js.bitwise.xor : number --> number --> number

Checks arguments are numbers and then applies the JavaScript ^ operator.

Parameters

- **X** (*number*) – A Shen number.
- **Y** (*number*) – A Shen number.

Returns A Shen number.

js.<< : number --> number --> number
Checks arguments are numbers and then applies the JavaScript << operator.

Parameters

- **X** (*number*) – Value to shift.
- **Y** (*number*) – Amount to shift by.

Returns A Shen number.

js.>> : number --> number --> number
Checks arguments are numbers and then applies the JavaScript >> operator.

Parameters

- **X** (*number*) – Value to shift.
- **Y** (*number*) – Amount to shift by.

Returns A Shen number.

js.>>> : number --> number --> number
Checks arguments are numbers and then applies the JavaScript >>> operator.

Parameters

- **X** (*number*) – Value to shift.
- **Y** (*number*) – Amount to shift by.

Returns A Shen number.

10.3 Typed Standard Functions

js.clear : js.timeout --> unit
Cancels or discontinues a task scheduled by `js.delay` or `js.repeat`.

Parameters **Timeout** (*js.timeout*) – A timeout handle returned by `js.delay` or `js.repeat`.

js.decode-uri : string --> string
Decodes a URI by un-escaping special characters.

Parameters **Uri** (*string*) – URI to decode.

Returns Decoded URI.

js.decode-uri-component : string --> string
Decodes a URI component by un-escaping special characters.

Parameters **Uri** (*string*) – URI component to decode.

Returns Decoded URI.

js.delay : number --> (lazy A) --> js.timeout
Calls standard JavaScript `setTimeout` function, but with arguments reversed.

Parameters

- **Duration** (*number*) – Time in milliseconds to delay running continuation.
- **Continuation** (*function*) – Function to run. Expected to take 0 arguments.

Returns A timeout handle.

`js.encode-uri : string --> string`
Encodes a URI by escaping special characters.

Parameters `Uri` (*string*) – URI to encode.

Returns Encoded URI.

`js.encode-uri-component : string --> string`
Encodes a URI component by escaping special characters.

Parameters `Uri` (*string*) – URI component to encode.

Returns Encoded URI.

`js.eval : string --> A`

> **Warning:** Using `eval` is even more dangerous than usual in ShenScript because it will be difficult to know what indentifiers will be in scope and how their names might have been aliased when code is evaluated.

Note: Unlike `js.raw.eval`, the never gets inlined, so the code gets evaluated in another scope.

So, for instance, `(let X 2 (js.eval "1 + X"))` would fail with `X is not defined` where with `js.raw.eval`, it would work.

Calls the built-in JavaScript `eval` function, asserting argument is a string.

Parameters `Code` (*string*) – JavaScript code in string form.

Returns The result of evaluating the code.

`js.log : A --> unit`
Logs given value using `console.log`.

Parameters `X` (*any*) – Value to log

Returns Empty list.

`js.parse-float : string --> number`
Parses a floating-point number.

Parameters `String` (*string*) – Numeric string to parse.

Returns Parsed number.

Throws If string does not represent a valid number.

`js.parse-int : string --> number`
Parses an integral number with radix specified to be 10 to avoid unusual parsing behavior.

Parameters `String` (*string*) – Numeric string to parse.

Returns Parsed number.

Throws If string does not represent a valid number.

`js.parse-int-with-radix : string --> number --> number`
Parses an integral number with the given radix.

Parameters

- `String` (*string*) – Numeric string to parse.

- **Radix** (*number*) – Radix to parse the number with, an integer 2 or greater.

Returns Parsed number.

Throws If string does not represent a valid number or invalid radix is passed.

`js.repeat : number --> (lazy A) --> js.timeout`
Calls standard JavaScript `setInterval` function, but with arguments reversed.

Parameters

- **Duration** (*number*) – Time in milliseconds to between calls of continuation.
- **Continuation** (*function*) – Function to run. Expected to take 0 arguments.

Returns A timeout handle.

`js.sleep : number --> unit`
Simulates a blocking `Thread.sleep` by awaiting a promise resolved with `setTimeout`.

Returns a promise.

Parameters **Duration** (*number*) – Time in milliseconds to sleep.

10.4 JSON Functions

`(json.parse String)`
`JSON.parse`.

Parameters **String** (*string*) – Serialized JavaScript value.

Returns Object parsed from `String`.

`(json.str Value)`
`JSON.stringify`.

Parameters **Value** (*any*) – JavaScript value to serialize.

Returns `Value` serialized to string.

10.5 Object Construction, Member Access

Remember that properties on JavaScript object are named with strings, so using Shen strings for property names is recommended for the function below. For example, the JavaScript code `x.y` would get written like `(js.get X "y")`.

Idle symbols can be used for property names, but they will represented with interned JavaScript symbols.

`(js.get Object Property)`
Retrieves a property's value from a JavaScript object.

Parameters

- **Object** (*object*) – Object to read from.
- **Property** (*any*) – Property name to get.

Returns Property value.

`js.get-macro`
Macro that converts variable-depth accessor syntax like `(. X Y Z)` to `(js.get (js.get X Y) Z)`.

`(js.new Class Args)`
Creates new instance of class by calling given constructor on argument list.

Parameters

- **Class** (*constructor*) – Constructor to call.
- **Args** (*list*) – Constructor arguments in a Shen list.

Returns New instance of `Class`.

`(js.obj Values)`
Creates new `Object` with properties of given names and values.

Parameters **Values** (*list*) – A flat Shen list of property names and values, like `["name1" Val1 "name2" Val2]`.

Returns New `Object`.

`js.obj-macro`
Macro that converts syntax like `({ A B C D })` to `(js.obj [A B C D])`.

`(js.set Object Property Value)`
Assigns property on a JavaScript object.

Parameters

- **Object** (*object*) – Object to write to.
- **Property** (*any*) – Property name to set.
- **Value** (*any*) – Value to assign.

Returns `Value`, just like the JavaScript assignment operator.

10.6 Recognisor Functions

`js.array? : A --> boolean`
Determines if value is a JavaScript array.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.async? : A --> boolean`
Determines if value is an asynchronous function.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.boolean? : A --> boolean`
Determines if value is a JavaScript boolean.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.defined? : A --> boolean`
Determines if value is *not* `undefined`.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.false? : A --> boolean`
Determines if value is the JavaScript `false` value.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.falsy? : A --> boolean`
Determines if value is coercible to `false` by JavaScript standards.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.finite? : A --> boolean`
Determines if value is a finite number.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.function? : A --> boolean`
Determines if value is a function. This test will also work for Shen functions.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.generator? : A --> boolean`
Determines if value is a generator function.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.infinite? : A --> boolean`
Determines if value is positive or negative infinity.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.+infinity? : A --> boolean`
Determines if value is positive infinity.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.-infinity? : A --> boolean`
Determines if value is negative infinity.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.integer? : A --> boolean`
Determines if value is an integer.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.+integer? : A --> boolean`
Determines if value is a positive integer.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.-integer? : A --> boolean`
Determines if value is a negative integer.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.nan? : A --> boolean`
Determines if value is `NaN` (not-a-number) which will normally not be equal to itself according to the === operator.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.null? : A --> boolean`
Determines if value is `null`.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.object? : A --> boolean`
Determines if value is an object with the direct protoype `Object` which means it is probably the product of object literal syntax.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.symbol? : A --> boolean`
Determines if a value is a JavaScript symbol. Shen symbols are represented with JS symbols, so this test will pass for idle symbols as well.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.true? : A --> boolean`
Determines if value is the JavaScript `true` value.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.truthy? : A --> boolean`
Determines if value is coercible to `true` by JavaScript standards.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

`js.undefined? : A --> boolean`
Determines if value is `undefined`.

Parameters **X** (*any*) – Value to inspect.

Returns A Shen boolean.

10.7 Global Classes, Objects and Values

Functions to retrieve common JavaScript globals. All take zero arguments and return what they're called. They Shen global symbols name with additional earmuffs (e.g. `js.*Array*`) where the value is actually held.

Hint: Some of these are not available in all JavaScript environments. You can check if a symbol is defined with `(bound? js.*Array*)`.

(js.Array)
Returns the global JavaScript `Array` class.

(js.ArrayBuffer)
Returns the global JavaScript `ArrayBuffer` class.

(js.AsyncFunction)
Returns the global JavaScript `AsyncFunction` class.

(js.Atomics)
Returns the global JavaScript `Atomics` class. This is not available in Firefox.

(js.Boolean)
Returns the global JavaScript `Boolean` class.

(js.console)
Returns the global JavaScript `console` object.

(js.DataView)
Returns the global JavaScript `DataView` class.

(js.Date)
Returns the global JavaScript `Date` class.

(js.Function)
Returns the global JavaScript `Function` class.

(js.GeneratorFunction)
Returns the global JavaScript `GeneratorFunction` class.

(js.globalThis)
Returns the global JavaScript `globalThis` object. If `globalThis` is not defined in this JavaScript environment, then `window`, `global`, etc is used as appropriate.

(js.Infinity)
Returns the JavaScript `Infinity` value.

(js.JSON)
Returns the global JavaScript `JSON` object.

(js.Map)
Returns the global JavaScript `Map` class.

(js.NaN)
Returns the JavaScript `NaN` value.

(js.Number)
Returns the global JavaScript `Number` class.

(js.null)
Returns the JavaScript `null` value.

(js.Object)
Returns the global JavaScript `Object` class.

(js.Promise)
Returns the global JavaScript `Promise` class.

(js.Proxy)
Returns the global JavaScript `Proxy` class.

(js.Reflect)
Returns the global JavaScript `Reflect` class.

(js.RegExp)
Returns the global JavaScript `RegExp` class.

(js.Set)
Returns the global JavaScript `Set` class.

(js.SharedArrayBuffer)
Returns the global JavaScript `SharedArrayBuffer` class. This is not available in Firefox.

(js.String)
Returns the global JavaScript `String` class.

(js.Symbol)
Returns the global JavaScript `Symbol` class.

(js.undefined)
Returns the JavaScript `undefined` value.

(js.WeakMap)
Returns the global JavaScript `WeakMap` class.

(js.WeakSet)
Returns the global JavaScript `WeakSet` class.

(js.WebAssembly)
Returns the global JavaScript `WebAssembly` class.

10.8 Web-specific Interop

Only available when running in a browser or browser-based environment like Electron.

(web.atob String)
Converts a string to a base64-encoded string.

> **Parameters** `String` (*string*) – Any string.

(web.btoa Base64)
Converts a base64-encoded string to a string.

> **Parameters** `String` (*string*) – Any base64 string.

(web.confirm? Message)
Shows a synchronous web confirm pop-up with the given message. Returns Shen `true` or `false` depending on whether the user hit "OK" or "Cancel".

> **Parameters** `Message` (*string*) – Message to show on the pop-up.

(web.document)
Returns the global `document`.

(web.fetch-json Url)
Same as `web.fetch-text`, but parses received value as JSON.

Parameters **Url** (*string*) – URL to GET from.

Returns A JSON object wrapped in a `Promise`.

(web.fetch-json* Url)
Same as `web.fetch-text*`, but parses received values as JSON.

Parameters **Urls** (*list*) – A Shen list of URLs to GET from.

Returns A Shen list of JSON objects wrapped in :js:`Promise`s.

(web.fetch-text Url)
Does an HTTP GET on the given url and returns the result as a string.

Parameters **Url** (*string*) – URL to GET from.

Returns A string wrapped in a promise.

(web.fetch-text* Urls)
Does concurrent HTTP GETs on the given URLs and returns the results as a list of strings.

Parameters **Urls** (*list*) – A Shen list of URLs to GET from.

Returns A Shen list of strings wrapped in a promise.

(web.navigator)
Returns the global `navigator`.

(web.self)
Returns the value of the `self` keyword.

(web.window)
Returns the global `window`.

10.8.1 DOM-specific Interop

Functions for building elements and interacting with the DOM.

(dom.append Parent Child)
Adds `Child` as the last child node of `Parent`.

Parameters

- **Parent** (*node*) – DOM Node to append `Child` to.
- **Child** (*node*) – DOM Node to append to `Parent`.

Returns Empty list.

(dom.build Tree)
Builds a DOM Node out of a Shen list tree. Each node in the tree is represented by a list starting with a key symbol. That symbol is the name of the HTML element to be built (e.g. `div`, `span`), with a couple of exceptions:

If the key symbol starts with a `@` or `!`, it is interpreted as an attribute (e.g. `@id`, `@class`) or an event listener (e.g. `!click`, `!mouseover`), respectively. The 2nd element in that list should be the attribute value or the event handler function. An event handler function should take a single argument, the event object.

String children get built into text nodes.

Child elements, text nodes, attributes and event handlers are all appended or set on enclosing parents in respective order.

```
[div [@id "example"]
     [p "Click Me!"
        [!click (/. _ (js.log "hi!"))]]]
```

gets built to:

```
<div id="example">
  <p onclick="console.log('hi!')">Click Me!</p>
</div>
```

Parameters **Tree** (*any*) – Tree of Shen lists.

Returns DOM Node built from `Tree`.

(dom.onready F)

Calls the function `F` when the DOM is loaded and ready. Callback takes zero arguments (is a `freeze`). Value returned by callback is ignored.

Parameters **F** (*function*) – Function that performs operations requiring the DOM.

Returns Empty list.

(dom.prepend Parent Child)

Adds `Child` as the first child node of `Parent`.

Parameters

- **Parent** (*node*) – DOM Node to prepend `Child` to.
- **Child** (*node*) – DOM Node to prepend to `Parent`.

Returns Empty list.

(dom.query Selector)

Finds an element matching the given CSS selector in the `document`.

Parameters **Selector** (*string*) – CSS selector to match with.

Returns Returns matching node or empty list if not found.

(dom.query* Selector)

Finds all elements matching the given CSS selector in the `document`.

Parameters **Selector** (*string*) – CSS selector to match with.

Returns A Shen list of matching elements.

(dom.remove Node)

Removes `Node` from its parent. Does nothing if `Node` has no parent.

Parameters **Node** (*node*) – DOM Node to remove.

Returns Empty list.

(dom.replace Target Child)

Removes `Target` from its parent and appends `Child` in its place. Does nothing if `Target` has no parent.

Parameters

- **Target** (*node*) – DOM Node to replace with `Child`.
- **Child** (*node*) – DOM Node to replace `Target` with.

Returns Empty list.

10.8.2 Local Storage Functions

Functions for getting and setting keys in Local Storage.

`(local-storage.clear)`
Removes all entries for the current domain.

Returns Empty list.

`(local-storage.get Key)`
Gets value for given string key. Can check `local-storage.has?` first to make sure value exists.

Parameters `Key` (*string*) – Key to fetch value for.

Returns Value for key, or empty list if it does not exist.

`(local-storage.has? Key)`
Returns `true` if value is set for given key.

Parameters `Key` (*string*) – Key to check.

Returns A Shen boolean.

`(local-storage.remove Key)`
Deletes value for given key.

Parameters `Key` (*string*) – Key to delete.

Returns Empty list.

`(local-storage.set Key Value)`
Sets value for given key.

Parameters

- `Key` (*string*) – Key to save value under.
- `Value` (*any*) – Value to assign.

Returns The assigned value.

10.9 Node-specific Interop

Only available when running in a Node environment.

`(node.exit Code)`
Takes an exit code and terminates the runtime with the process returning that exit code.

Parameters `Code` (*number*) – Exit code for the process to return.

Returns Doesn't.

`node.exit-macro`
Macro to convert `node.exit` called with no arguments by inserting `0` as the default exit code.

`(node.global)`
Called with no arguments, returns the value of the Node `global` object.

`(node.require Id)`
Calls Node's `require` function with the given module identifier.

Parameters `Id` (*string*) – Name or path to a Node module.

Returns `exports` object returned by the module.

CHAPTER

ELEVEN

ACCESSING SHENSCRIPT INTERNALS FROM JAVASCRIPT

Note: Some of these functions and classes are exported, but they are primarily referred to internally. They are exported so generated JavaScript code will have access to them. Or, in rare cases, they are exported in case your Shen code needs to work around something.

11.1 Data

Warning: These objects are not meant to be tampered with by user or client code. Tinker with them if you must, but it will void the warranty.

`globals`
A `Map` of `Cell` objects, indexed by string name. Used to hold references to both global functions and global symbol values. If a function and a global symbol have the same name, they will be referred to by the same entry in this map.

11.2 Classes

`class Cell`(*name*)
Contains mutable pointers to the function and/or the global symbol value by the given name. Should only be created by the `lookup` function.

Parameters `name` (*string*) – The name of the global.

`class Cons`(*head*, *tail*)
The classic Lisp cons cell. `head` and `tail` are akin to `car` and `cdr`.

When a chain of `Cons` is used to build a list, the last `Cons` in the chain has a `tail` of `null`.

Parameters

- `head` (*any*) – Named `head` because it's typically the head of a list.
- `tail` (*any*) – Named `tail` because it's typically the tail of a list.

`class Context`(*options*)
`Context` objects are passed between calls to the `build` function to track syntax context and rendering options.

Parameters

- **options** (*object*) – Collection of code generation options.
- **options.async** (*boolean*) – `true` if code should be generated in async mode.
- **options.head** (*boolean*) – `true` if current expression is in head position.
- **options.locals** (*Map*) – Used as an immutable map of local variables and their known types.
- **options.inlines** (*object*) – Map containing code inlining rules.

class Fabrication(*ast*, *subs*)

Returned by the `build` function, containing the resulting AST and a substitution map of hoisted references.

Parameters

- **ast** (*ast*) – The AST result of building a KL expression.
- **subs** (*object*) – Has string keys and AST values.

class Trampoline(*f*, *args*)

A `Trampoline` represents a deferred tail call.

Parameters

- **f** (*function*) – A JavaScript function.
- **args** (*array*) – A JavaScript array of arguments that `f` will get applied to.

11.3 Functions

as___(*x*)

There are several functions following this naming pattern which first check if their argument passes the related `is___` function and returns it if it does. If it does not pass the type check, an error is raised.

Parameters **x** (*any*) – Whatever.

Returns The same value.

Throws If argument does not pass the type check.

assemble(*f*, *...xs*)

Composes a series of fabrications into a single fabrication.

Parameters

- **f** (*function*) – A function that combines a sequence of ASTs into an AST (or fabrication).
- **xs** (*array*) – A sequence of fabrications/ASTs.

Returns The composed fabrication.

bounce(*f*, *args*)

Creates a `Trampoline`.

Aliased as `b` for brevity in generated code.

Parameters

- **f** (*function*) – A JavaScript function.
- **args** – A variadic parameter containing any values.

Returns A `Trampoline`.

`compile`(*expr*)
Builds a KLambda expression tree in the root context.

Parameters **expr** (*expr*) – Expression to build.

Returns Rendered JavaScript AST.

`construct`(*expr*)
Like `compile`, but returns a fabrication, not an AST.

Parameters **expr** (*expr*) – Expression to build.

Returns Rendered fabrication.

`lookup`(*name*)
Retrieves the `Cell` for the given name, creating it and adding it to `globals` if it did not already exist.

Aliased as `c` for brevity in generated code.

Parameters **name** (*string*) – Name of a global function or symbol.

Returns A `Cell` for that name.

`fun`(*f*)
Takes a function that takes a precise number of arguments and returns a wrapper that automatically applies partial and curried application.

Aliased as `l` for brevity in generated code.

Parameters **f** (*function*) – Function wrap with partial application logic.

Returns Wrapper function.

`is___`(*x*)
There are several functions following this naming pattern which checks if the argument qualifies as the type it's named for.

Parameters **x** (*any*) – Whatever.

Returns A JavaScript boolean.

`nameOf`(*symbol*)
Returns string name of given symbol. Symbol does not have to have been declared.

Parameters **symbol** (*symbol*) – A symbol.

Returns Symbol name.

`raise`(*message*)
Throws an Error with the given message.

Parameters **message** (*string*) – Error message.

Returns Doesn't.

Throws Error with the given message.

`settle`(*x*)
If given a trampoline, runs trampoline and checks if result is a trampoline, in which case that is then run. Process repeats until result is not a trampoline. Never returns a trampoline. Potentially any function in `functions` will need to be settled after being called to get a useful value.

Handles async trampolines by branching off to unexported `future` function when encountering a promise. `future` will automatically thread trampoline settling through promise chains.

Aliased as `t` for brevity in generated code.

Parameters **x** (*any*) – May be a `Trampoline` (or a `Promise`), which will be run, or any other value, which will be returned immediately.

Returns Final non-trampoline result.

symbolOf(*name*)

Returns the interned symbol by the given name.

Aliased as `s` for brevity in generated code.

Parameters **name** (*string*) – Symbol name.

Returns Symbol by that name.

CHAPTER

TWELVE

ACCESSING SHENSCRIPT INTERNALS FROM SHEN

Functions in the `shen-script` namespace are for directly accessing ShenScript internals.

12.1 Functions

These functions are callable from Shen to give access to the implementation details of ShenScript.

(shen-script.$)
Provides access to the ShenScript environment object, which when combined with `js` interop functions, allows arbitrary manipulation of the port's implementation details from Shen.

Returns ShenScript environment object.

(shen-script.ast)
Returns a JavaScript object with all the JavaScript AST constructor functions.

Returns The `exports` of the `ast` module.

(shen-script.lookup-function Name)
Allows lookup of global function by name instead of building wrapper lambdas or the like.

Parameters **Name** (*symbol*) – Name of function to lookup.

Returns Shen function by that name, or `[]` when function does not exist.

(shen-script.boolean.js->shen X)
Converts a JavaScript boolean to a Shen boolean. Any truthy value counts as JavaScript `true` and any falsy value counts as JavaScript `false`.

Parameters **X** (*any*) – Accepts any value as an argument.

Returns A Shen boolean.

(shen-script.boolean.shen->js X)
Converts a Shen boolean to a JavaScript boolean.

Parameters **X** (*boolean*) – A Shen boolean.

Returns A JavaScript boolean.

Throws Error if argument is not a Shen boolean.

(shen-script.array->list X)
Converts a JavaScript array to a cons list.

Parameters **X** (*array*) – A JavaScript array.

Returns A Shen list.

`(shen-script.array->list+ X Tail)`
Converts a JavaScript array to a cons list with a specified value for the tail of the last cons.

Parameters

- **X** (*array*) – A JavaScript array.
- **Tail** (*any*) – Specific final tail value.

Returns A Shen list.

`(shen-script.array->list-tree X)`
Converts a tree of nested JavaScript arrays to a tree of nested cons lists.

Parameters **X** (*array*) – A JavaScript array with elements that may be arrays.

Returns A Shen list with elements that will be lists.

`(shen-script.list->array X)`
Converts cons list to JavaScript array.

Parameters **X** (*array*) – A Shen list.

Returns A JavaScript array.

`(shen-script.list->array-tree X)`
Converts tree of nested cons lists to tree of nested JavaScript arrays.

Parameters **X** (*array*) – A Shen list with elements that may be lists.

Returns A JavaScript array with elements that will be arrays.

12.2 AST Construction Functions

Functions in the *js.ast* namespace are used to construct, emit and evaluate arbitrary JavaScript code. All of the AST builder functions return JavaScript objects conforming to the informal ESTree standard ESTree.

`(js.ast.arguments)`
Constructs a reference to the `arguments` object.

Returns An `Identifier` AST node.

`(js.ast.array Values)`
Constructs array literal syntax.

Example: `[x, y, z]`.

Parameters **Values** (*list*) – A Shen list of value AST's to initialise a JavaScript array with.

Returns An `ArrayExpression` AST node.

`(js.ast.arrow Parameters Body)`
Constructs a lambda expression.

Example: `x => ...`

Parameters

- **Parameters** (*list*) – A Shen list of parameter identifiers.
- **Body** (*ast*) – A body expression.

Returns A `ArrowFunctionExpression` AST Node.

(js.ast.assign Target Value)
Constructs an assignment expression.

Example `x = y`

Parameters

- **Target** (*ast*) – The variable to assign to.
- **Value** (*ast*) – The value to assign.

Returns An `AssignmentExpression` AST Node.

(js.ast.async Ast)
Makes function or class member async.

Examples: `async (x, y) => ...`, `async function(x, y) { ... }`, `async method(x, y) { ... }`

Parameters **Ast** (*ast*) – Ast to make async.

Returns The same AST after setting the `async` property to `true`.

(js.ast.await Argument)
Constructs an await expression for use in an async function.

Example: `await x`

Parameters **Argument** (*ast*) – Expression to await.

Returns An `AwaitExpression` AST Node.

(js.ast.binary Operator Left Right)
Constructs a binary operator application.

Examples: `x && y`, `x + y`

Parameters

- **Operator** (*string*) – Name of operator to apply.
- **Left** (*ast*) – Expression on the left side.
- **Right** (*ast*) – Expression on the right side.

Returns A `BinaryExpression` AST Node.

(js.ast.block Statements)
Constructs a block that groups statements into a single statement and provides isolated scope for `const` and `let` bindings.

Example: `{ x; y; z; }`

Parameters **Statements** (*list*) – A Shen list of statement AST's.

Returns A `BlockStatement` AST Node.

(js.ast.call Function Args)
Constructs a function call expression.

Example: `f(x, y)`

Parameters

- **Function** (*ast*) – An expression AST that evaluates to a function.
- **Args** (*list*) – A Shen list of argument AST's.

Returns A `CallExpression` AST Node.

(js.ast.catch Parameter Body)
Constructs a catch clause.

Example: `catch (e) { ... }`

Parameters

- **Parameter** (*ast*) – An identifer for the error that was caught.
- **Body** (*ast*) – A block of statements that get run when the preceeding try has failed.

Returns A `CatchClause` AST Node.

(js.ast.class Name SuperClass Slots)
Constructs ES6 class syntax. Members are constructed using `js.ast.constructor`, `js.ast.method`, `js.ast.getter`, `js.ast.setter` or the more general function `js.ast.slot`.

Example:

```
class Class extends SuperClass {
  constructor(...) {
    ...
  }
  method(...) {
    ...
  }
}
```

Parameters

- **Name** (*ast*) – Identifier naming the class.
- **SuperClass** (*ast*) – Identifier node of super class, can be undefined or null.
- **Slots** (*list*) – A Shen list of slot AST's.

Returns A `ClassExpression` AST Node.

(js.ast.const Id Value)
Constructs `const` variable declaration.

Parameters

- **Id** (*ast*) – Variable name.
- **Value** (*ast*) – Value to initialise variable with.

Returns A `VariableDeclaration` AST node.

(js.ast.constructor Body)
Specialisation of `js.ast.slot` for class constructors.

Example: `constructor(...) { ... }`

(js.ast.debugger)
Constructs a `debugger;` statement.

Returns A `DebuggerStatement` AST node.

(js.ast.do-while Test Body)
Constructs a do-while loop.

Example: `do { ... } while (condition);`

Parameters

- **Test** (*ast*) – Conditional expression that determines if the loop will run again.
- **Body** (*ast*) – Block of statements to run each time the loop repeats or the first time.

Returns A `DoWhileStatement` AST Node.

(js.ast.empty)
Constructs an empty statement.

Example: `;`

Returns An `EmptyStatement` AST Node.

(js.ast.for Init Test Update Body)
Constructs a for loop.

Example: `for (let x = 0; x < i; ++x) { ... }`

Parameters

- **Init** (*ast*) – Declarations and initial statements. This can be a sequence expression.
- **Test** (*ast*) – Conditional expression that determines if the loop will run again or for the first time.
- **Update** (*ast*) – Update expressions to evaluate at the end of each iteration. This can be a sequence expression.
- **Body** (*ast*) – Block of statements to run each time the loop repeats.

Returns A `ForStatement` AST Node.

(js.ast.for-await-of Left Right Body)
Constructs an asynchronous for-of loop. Each value iterated from `Right` gets awaited before being bound to the variable or pattern declared in `Left`.

Example: `for await (let x of xs) { ... }`

Parameters

- **Left** (*ast*) – Declaration of local variable that each value from the iterable on the right side gets assigned to.
- **Right** (*ast*) – Expression that evaluates to an iterable value.
- **Body** (*ast*) – Block of statements to run each time the loop repeats.

Returns A `ForOfStatement` AST Node.

(js.ast.for-in Left Right Body)
Constructs a for-in loop.

Example: `for (let x in xs) { ... }`

Parameters

- **Left** (*ast*) – Declaration of local variable that each key from the object on the right side gets assigned to.
- **Right** (*ast*) – Expression that evaluates to some object.
- **Body** (*ast*) – Block of statements to run each time the loop repeats.

Returns A `ForInStatement` AST Node.

(js.ast.for-of Left Right Body)
Constructs a for-of loop.

Example: `for (let x of xs) { ... }`

Parameters

- **Left** (*ast*) – Declaration of local variable that each value from the iterable on the right side gets assigned to.
- **Right** (*ast*) – Expression that evaluates to an iterable value.
- **Body** (*ast*) – Block of statements to run each time the loop repeats.

Returns A `ForOfStatement` AST Node.

(js.ast.function Name Parameters Body)

Constructs a function expression.

Example: `function name(x, y) { ... }`

Parameters

- **Name** (*ast*) – Optional identifier naming the function.
- **Parameters** (*list*) – A Shen list of parameter expression.
- **Body** (*ast*) – A block of statements that make of the body of the function.

Returns A `FunctionExpression` AST Node.

(js.ast.function* Name Parameters Body)

Constructs a generator function expression.

Example: `function* name(x, y) { ... }`

Parameters

- **Name** (*ast*) – Optional identifier naming the function.
- **Parameters** (*list*) – A Shen list of parameter expression.
- **Body** (*ast*) – A block of statements that make of the body of the function.

Returns A `FunctionExpression` AST Node.

(js.ast.getter Name Body)

Specialisation of `js.ast.slot` for class getters.

Example: `get thing(...) { ... }`

(js.ast.id Name)

Constructs an identifier - the name of a function or variable. Identifier is named exactly as the given argument.

Example: `x`

Parameters **Name** (*string*) – Name of identifier.

Returns An `Identifier` AST node.

(js.ast.if Condition Consequent Alternate)

Constructs an if statement with optional else clause.

Examples:

```
if (condition) {
  ...
} else {
  ...
}
```

```
if (condition) {
  ...
}
```

Parameters

- **Condition** (*ast*) – Conditional expression that determines which clause to step into.
- **Consequent** (*ast*) – The then clause.
- **Alternate** (*ast*) – Optional else clause.

Returns An `IfStatement` AST Node.

(js.ast.let Id Value)

Constructs `let` variable declaration.

Parameters

- **Id** (*ast*) – Variable name.
- **Value** (*ast*) – Value to initialise variable with.

Returns A `VariableDeclaration` AST node.

(js.ast.literal Value)

Constructs literal value syntax.

Parameters **Value** – JavaScript value that can be a literal (number, string, boolean, null).

Returns A `Literal` AST node.

(js.ast.member Object Member)

Constructs a member access expression with the dot operator.

Examples: `x.y`, `x[y]`

Parameters

- **Object** (*ast*) – Expression AST to access member of.
- **Member** (*ast*) – Expression that computes member name to access. If non-string, will automatically be wrapped in square brackets.

Returns A `MemberExpression` AST Node.

(js.ast.method Name Body)

Specialisation of `js.ast.slot` for class methods.

Example: `method(...) { ... }`

(js.ast.new-target)

Constructs a reference to the `new.target` meta-property.

Returns A `MetaProperty` AST node.

(js.ast.object Properties)

Constructs object literal syntax.

Example: `{ a: b, c: d }`

Parameters **Properties** (*list*) – A flat Shen list of property names and values.

Returns An `ObjectExpression` AST Node.

(js.ast.return Argument)
Constructs a return statement.

Example: `return x;`

Parameters **Argument** (*ast*) – Expression to return.

Returns A `ReturnStatement` AST Node.

(js.ast.safe-id Name)
Constructs an identifier where the name is escaped to make it valid in JavaScript and to not collide with reserved names in ShenScript.

Parameters **Name** (*string*) – Name of identifier.

Returns An `Identifier` AST node.

(js.ast.setter Name Body)
Specialisation of `js.ast.slot` for class setters.

Example: `set thing(...) { ... }`

(js.ast.sequence Expressions)
Constructs a compound expression using the comma operator.

Example: `(x, y, z)`

Parameters **Expressions** (*list*) – A Shen list of expression AST's.

Returns A `SequenceExpresssion` AST Node.

(js.ast.slot Kind Name Body)
Constructs a class property of the given kind.

Parameters

- **Kind** (*string*) – "constructor", "method", "get" or "set".
- **Name** (*ast*) – Identifier naming the property.
- **Body** (*ast*) – Expression representing the function or value assigned to the property.

Returns A `MethodDefinition` AST Node.

(js.ast.statement Expression)
Constructs a wrapper that allows an expression to be a statement.

Parameters **Expression** (*ast*) – The expression in question.

Returns An `ExpressionStatement` AST Node.

(js.ast.static Ast)
Makes class member static.

Example: `static method(x, y) { ... }`

Parameters **Ast** (*ast*) – Ast to make static.

Returns The same AST after setting the `static` property to `true`.

(js.ast.spread Argument)
Constructs spread operator/pattern syntax.

Example: `...x`

Parameters **Argument** (*ast*) – Identifier or pattern that are gathered or spread.

Returns A `SpreadElement` AST Node.

(js.ast.super Arguments)
Constructs a call to the super (prototype) constructor.

Example: `super(x, y);`

Parameters **Arguments** (*list*) – A Shen list of argument AST's.

Returns A `Super` AST Node.

(js.ast.ternary Condition Consequent Alternate)
Constructs an application of the ternary operator.

Example `x ? y :  z`

Parameters

- **Condition** (*ast*) – True/false expression on the left of the `?`.
- **Consequent** (*ast*) – Expression that gets evaluated if the condition is true.
- **Alternate** (*ast*) – Expression that gets evaluated if the condition is false.

Returns A `ConditionalExpression` AST Node.

(js.ast.this)
Constructs a reference to the `this` keyword.

Returns A `ThisExpression` AST node.

(js.ast.try Body Handler)
Constructs a try statement.

Example:

```
try {
  ...
} catch (e) {
  ...
}
```

Parameters

- **Body** (*ast*) – A block of statements that get tried.
- **Handler** (*ast*) – A catch clause as constructed by `js.ast.catch`.

Returns A `TryStatement` AST Node.

(js.ast.unary Operator Argument)
Construts a unary operator application.

Examples: `!x`, `-x`

Parameters

- **Operator** (*string*) – Name of operator to apply.
- **Argument** (*ast*) – Argument to apply operator to.

Returns A `UnaryExpression` AST Node.

(js.ast.update Operator Target Value)
Constructs an assignment expression with a specific operator.

Examples `x += y`, `x *= y`

Parameters

- **Operator** (*string*) – The update operator without the =, so +, -, etc.
- **Target** (*ast*) – The variable to assign to.
- **Value** (*ast*) – The value to assign.

Returns An `AssignmentExpression` AST Node.

(js.ast.var Id Value)
Constructs `var` variable declaration.

Parameters

- **Id** (*ast*) – Variable name.
- **Value** (*ast*) – Value to initialise variable with.

Returns A `VariableDeclaration` AST node.

(js.ast.while Test Body)
Constructs a while loop.

Example: `while (condition) { ... }`

Parameters

- **Test** (*ast*) – Conditional expression that determines if the loop will run again or for the first time.
- **Body** (*ast*) – Block of statements to run each time the loop repeats.

Returns A `WhileStatement` AST Node.

(js.ast.yield Argument)
Constructs a yield expression for use in a generator function.

Example: `yield x`

Parameters **Argument** (*ast*) – Expression to yield.

Returns A `YieldExpression` AST Node.

(js.ast.yield* Argument)
Constructs a yield delegate expression for use in a generator function.

Example: `yield* x`

Parameters **Argument** (*ast*) – Iterable or generator expression to yield.

Returns A `YieldExpression` AST Node.

12.3 AST Evaluation Functions

(js.ast.compile Expr)
Builds a JavaScript AST from given KLambda expression. Any hoisted references will be enclosed per the backend `hoist` function.

Parameters **Ast** (*expr*) – KLambda expression.

Returns JavaScript AST.

(js.ast.eval Ast)
Takes a JavaScript AST as built by the `js.ast` functions, renders it to JavaScript and evaluates it in the current environment.

Parameters `Ast` (*ast*) – JavaScript AST.

Returns Whatever the code the AST represents evaluates to.

`(js.ast.render Ast)`

Renders the string source code representation of a JavaScript AST.

Parameters `Ast` (*ast*) – Code to be rendered.

Returns String representation of that `Ast`.

CHAPTER

THIRTEEN

INDEX

INDEX

F

G

I

L

N

P

R

S

T

V