The μ_ Specification — r0.3

Preface

mu_ (stylized μ_) is an extremely simple programming language. This document contains a complete specification of the language followed by a series of appendices elucidating motivation, design decisions, implementation strategies, and so on. These appendices are not part of the formal language specification and in the event of a conflict between the specification and the content of an appendix the specification is authoritative.

This is revision `r0.3` of the specification.

1 Source

1.1 Encoding

A mu_ program (recommended extensions `.mu_`, `.muu`) consists of a series of bytes. The behaviors of bytes with the high bit set is implementation defined. As a consequence a conforming implementation MAY choose to interpret mu_ code as text in ASCII, UTF-8, or another similar encoding. This document will adopt such an interpretation when displaying mu_ source.

1.2 Syntax

1.2.1 Grammar

```
<program>  ::= <ws> <expr> <ws>

<expr>     ::= <atom> | <list>

<atom>     ::= <ordinary> { <ordinary> }

<list>     ::= <lparen>
               { <ws> <expr> }
               [ <ws> <dot> <ws> <expr> ] <ws>
               <rparen>

<ws>       ::= { <tab> | <lf> | <cr> | <sp> }

<lparen>   ::= 0x28 ;; LEFT PARENTHESIS  '('
<rparen>   ::= 0x29 ;; RIGHT PARENTHESIS ')'
<dot>      ::= 0x2e ;; FULL STOP         '.'

<number>   ::= 0x23 ;; NUMBER SIGN       '#'
<quote>    ::= 0x22 ;; QUOTATION MARK    '"'
```

```
<tab>      ::= 0x09 ;; HORIZONTAL TAB    '\t'
<lf>       ::= 0x0a ;; LINE FEED         '\n'
<cr>       ::= 0x0d ;; CARRIAGE RETURN   '\r'
<sp>       ::= 0x20 ;; SPACE             ' '

<ordinary> ::= <any> - <special>

<any>      ::= 0x20-0x7e ;; ALL PRINTABLE ASCII
<special>  ::= <lparen> | <rparen> | <dot>
             | <number> | <quote>
             | <tab> | <lf> | <cr> | <sp>
```

The characters # and " are reserved.

1.2.2 Interpretation

Each distinct <**atom**> in the source is an *atom name*, a sequence of bytes, and is given a distinct positive *atom number*.[1][2] How different byte sequences are mapped to numbers is completely implementation defined with the only restrictions being that:

- Atom numbers are between 1 and $2^{31} - 1$.
- Two atoms with the same name within the same execution of a program get the same atom number.
- Two atoms with different names within the same execution of a program get different atom numbers.

For example, all instances of the atom with name `hello` (hex `68 65 6c 6c 6f`) are to be assigned the same atom number (say, 3) which must be different from the atom number for `world` (hex `77 6f 72 6c 64`) (say, 7) but what exactly the atom numbers are can vary between implementations, programs, or even different executions of the same program at the implementer's discretion.

An implementation MAY assume that there are no more than $2^{11} - 1$ distinct atom names referred to in a given source text, that each atom name has a length of at most $2^7 - 1$ bytes, and that the combined length of these names (excluding duplicates) is no more than $2^{15} - 2^{11}$ bytes.[3]

[1] See 2.1 Atoms.

[2] See TODO B Implementation Strategies.

[3] These restrictions permit an implementation to allocate its atom names into a singular region of memory of size i32::MAX separated by NULLs, for instance.

A list in the source code is constructed of cons cells[4] according to the following algorithm:

```
def parse_list(tokens):
    eat a lparen from tokens
    list = parse_list_inner(tokens)
    eat a rparen from tokens
    return list

def parse_list_inner(tokens):
    if ( the next token in tokens is dot ):
        eat a dot from tokens
        return parse an expression from tokens
    elif ( the next token in tokens is rparen ):
        return the 0 atom
    else:
        head = parse an expression from tokens
        tail = parse_list_inner(tokens)
        return a new cons cell of head and tail
```

A few things to note about this algorithm:

- () becomes the 0 atom. For this reason the 0 atom is also called nil and is usually denoted ().
- A list like (i_1 i_2 i_3) denotes an ordinary linked list $i_1 \to i_2 \to i_3 \to$ **NIL**.
- A dotted pair like (l . r) denotes a simple cons cell (l, r).
- A dotted list like (i_1 i_2 i_3 . r) denotes a modified linked list (with non-nil tail) $i_1 \to i_2 \to i_3 \to r$.
- An ordinary list like (i_1 i_2 i_3 $\cdots$ i_n) is equivalent to the dotted (i_1 . (i_2 . (i_3 . ($\cdots$ (i_n . ()))))).

2 Types

μ_ is dynamically typed with exactly two types: atoms and cons cells.

2.1 Atoms

An atom is fundamentally an *unsigned 31 bit integer*.[5]

The 0 atom represents the empty list in list-related contexts.

Some positive atoms correspond to particular byte sequences (atom names) in the source code.[6] Not all atom values have a corresponding name nor is there a standard mechanism for converting between atoms and names during the execution of a program. This correspondence is simply a mechanism to allow source code to provide human-readable names to arbitrary symbols rather than a property of the language's runtime.

2.2 Cons Cells

A cons cell is simply an ordered pair of items (each of which may be either an atom or another cons cell).

The first item in the pair is called the *head* and the second is called the *tail*.

A list refers to a particular structure formed of either the 0 atom or a cons cell whose tail is itself a list.[7]

3 Environments

At the core of μ_'s scoping rules is the environment, a mapping from atoms to arbitrary values. This mapping is defined as a list of pairs of positive atoms and values. The first element in each pair is a variable to bind and the second is an value to bind to that variable.

For example:

```
( (hello . hello)
  (somelist . (a b c))
  (a_variable . a_value) )
```

is an environment mapping the atom `hello` to itself, the atom `somelist` to the list (`a b c`), and the atom `a_variable` to the atom `a_value`.

An environment may contain multiple bindings for the same atom in which case the earlier binding *shadows* the later one, effectively overriding it[8].

() is a legal environment containing no mappings.

3.1 Lookup

Looking up the value corresponding to an atom in an environment is defined according to the following algorithm:

```
def lookup(sym, env):
    if ( env is empty ):
        return sym
    else:
        binding = the head of env
        if ( the head of binding is sym):
            return the tail of binding
        else:
            return lookup(sym, the tail of env)
```

Note that duplicate bindings are resolved in favor of the first, and missing bindings resolve to themselves.

[4] See 2.2 Cons Cells.

[5] See B Implementation Strategies for a rational for this unusual choice.

[6] See 1.2.2 Interpretation.

[7] See 1.2.2 Interpretation for how these are denoted.

[8] The shadowed binding is still accessible via ~env, though.

4 Pattern Matching

A pattern is a value which can be matched against another value (called the scrutinee) in some base environment to produce a new environment which contains everything in the base environment in addition to new bindings representing components of the value.

Every value which does not contain duplicate positive atoms is a valid pattern.

Matching a pattern against a value follows the following rules:

- ( ) matches the value ( ) and introduces no bindings. Attempting to match this pattern against any other object is undefined behavior.[9]

- A positive atom matches any value and introduces a binding from the pattern atom to the value.

- A cons cell matches a cons cell value, recursively pattern matching the value's head and tail against its head and tail. The order in which the bindings from the head and tail are included in the final environment is implementation defined and may even be non-deterministic. Attempting to match a cons cell pattern against any other object is undefined behavior.

- Introduced bindings shadow bindings in the base environment (i.e. are placed before) but do not remove them. This allows careful environment manipulation to recover the base environment which is necessary for writing fully hygienic macros.

4.1 Examples

- Matching the expression ( ) against the pattern ( ) in the base environment `((a . b))` produces the environment `((a . b))`.

- Matching the expression `(a b c d)` against the pattern `(x y . z)` in the base environment `()` produces some permutation of the environment `((x . a) (y . b) (z . (c d)))`

- Matching the expression `(a b c)` against the pattern `(x y)` is undefined behavior.

- Matching the expression `(a b)` against the pattern `(x y)` in the base environment `((x . xx))` may produce either the environment `((x . a) (y . b) (x . xx))` or `((y . b) (x . a) (x . xx))` but MUST NOT produce the environment `((x . a) (y . b))`.

5 Evaluation

The heart of the language is evaluation. In fact, executing a program simply consists of parsing it then evaluating it in an empty environment.

An *expression* is any value to be evaluated.

Evaluation takes place in an environment e.[10]

Evaluating an expression behaves differently depending on whether the expression is the 0 atom, a positive atom, or a non-empty list.

5.1 Zero Atom

The 0 atom ( ) evaluates to itself.

5.2 Positive Atoms

A positive atom a evaluates to `lookup(`a`, `e`)`.[11]

5.3 Lists

If a cons cell is evaluated it should be a valid list. An implementation MAY reject attempting to evaluate a non-list cons cell or MAY try to interpret such an expression cohesively.

A list is evaluated as a call by first evaluating its head and then, based on the result (referred to as the *receiver*), performing an operation on the remaining elements.

We will use the meta syntax of angle brackets $\langle expr \rangle_{env}$ to denote an expression which evaluates to $expr$ in the environment env and the syntax of an arrow with environment above ($\xrightarrow{env}$) to denote the evaluation of an expression to a value in an environment.

5.3.1 Builtins

If the receiver is an atom it should belong to the following list of atoms with builtin behavior. Notice that with the exception of ( ) which has no corresponding name these builtin receivers all have names starting ~ (hex `7e 7e`).

( ) Called “quote”, but represented by the 0 atom for technical reasons.[12] Evaluates to its first argument *unevaluated*.

$$(\langle (\,) \rangle_e \;\; i) \xrightarrow{e} i$$

[9] See 7 Undefined Behavior.

[10] See 3 Environments.

[11] See 3.1 Lookup for the definition of `lookup()`.

[12] Using the 0 atom as the quote builtin macro ensures that the quote macro is always accessible as the 0 atom is the only atom which cannot be rebound. This, in turn, ensures that any value is always accessible by invoking quote. The ability to access arbitrary content reliably regardless of environment is critical for hygienic macros to be possible.

∼true Takes two arguments and tail evaluates[13] to its first argument. DOES NOT evaluate its second argument.

$$(\langle{\sim}\texttt{true}\rangle_e\ \ \langle t\rangle_e\ \ fe) \xrightarrow{e} t$$

∼false Takes two arguments and tail evaluates to its second argument. DOES NOT evaluate its first argument.

$$(\langle{\sim}\texttt{false}\rangle_e\ \ te\ \ \langle f\rangle_e) \xrightarrow{e} f$$

∼head Takes an argument and evaluates it. The result should be a cons cell. Returns the head of that cons cell.

$$(\langle{\sim}\texttt{head}\rangle_e\ \ \langle(h\ \ .\ \ t)\rangle_e) \xrightarrow{e} h$$

∼tail Takes an argument and evaluates it. The result should be a cons cell. Returns the tail of that cons cell.

$$(\langle{\sim}\texttt{tail}\rangle_e\ \ \langle(h\ \ .\ \ t)\rangle_e) \xrightarrow{e} t$$

∼cons Takes two arguments, evaluates them, and returns a new cons cell constructed from the first and second arguments.

$$(\langle{\sim}\texttt{cons}\rangle_e\ \ \langle h\rangle_e\ \ \langle t\rangle_e) \xrightarrow{e} (h\ \ .\ \ t)$$

∼lte Takes two arguments and evaluates them. Returns ∼true if the first is less than or equal to the second and ∼false otherwise. Atoms are compared according to their numbers. Cons cells are always less than atoms. Two distinguishable cons cells do not compare equal. Two indistinguishable cons cells may or may not compare equal.

$$(\langle{\sim}\texttt{lte}\rangle_e\ \ \langle a\rangle_e\ \ \langle b\rangle_e) \xrightarrow{e} \begin{cases} {\sim}\texttt{true} & \text{if } a \leqslant b \\ {\sim}\texttt{false} & \text{otherwise} \end{cases}$$

∼eq Takes two arguments, evaluates them, and returns ∼true if the first is equal to the second and ∼false otherwise. Atoms are compared according to their numbers. Cons cells are always less than atoms. Two distinguishable cons cells do not compare equal. Two indistinguishable cons cells may or may not compare equal.

$$(\langle{\sim}\texttt{eq}\rangle_e\ \ \langle a\rangle_e\ \ \langle b\rangle_e) \xrightarrow{e} \begin{cases} {\sim}\texttt{true} & \text{if } a = b \\ {\sim}\texttt{false} & \text{otherwise} \end{cases}$$

∼add Takes two arguments and evaluates them. They should both be atoms. Returns the sum of the two atoms modulo 2^{31}.

$$(\langle{\sim}\texttt{add}\rangle_e\ \ \langle a\rangle_e\ \ \langle b\rangle_e) \xrightarrow{e} (a + b) \bmod 2^{31}$$

∼sub Takes two arguments and evaluates them. They should both be atoms. Returns the difference of the two atoms modulo 2^{31}.

$$(\langle{\sim}\texttt{sub}\rangle_e\ \ \langle a\rangle_e\ \ \langle b\rangle_e) \xrightarrow{e} (a - b) \bmod 2^{31}$$

∼and Takes two arguments and evaluates them. They should both be atoms. Returns the bitwise conjunction (bitwise and) of the two atoms

$$(\langle{\sim}\texttt{and}\rangle_e\ \ \langle a\rangle_e\ \ \langle b\rangle_e) \xrightarrow{e} (a \wedge b)$$

∼or Takes two arguments and evaluates them. They should both be atoms. Returns the bitwise disjunction (bitwise or) of the two atoms

$$(\langle{\sim}\texttt{or}\rangle_e\ \ \langle a\rangle_e\ \ \langle b\rangle_e) \xrightarrow{e} (a \vee b)$$

∼not Takes one argument and evaluates it. It should be an atom. Returns the bitwise negation (bitwise not) of the atom

$$(\langle{\sim}\texttt{not}\rangle_e\ \ \langle a\rangle_e) \xrightarrow{e} (\neg a)$$

∼shl Takes two arguments and evaluates them. They should both be atoms. The second argument should be less than 31. Returns the first shifted left by the second number of bits.

$$(\langle{\sim}\texttt{shl}\rangle_e\ \ \langle a\rangle_e\ \ \langle b\rangle_e) \xrightarrow{e} (a << b)$$

∼shr Takes two arguments and evaluates them. They should both be atoms. The second argument should be less than 31. Returns the first shifted right by the second number of bits.

$$(\langle{\sim}\texttt{shr}\rangle_e\ \ \langle a\rangle_e\ \ \langle b\rangle_e) \xrightarrow{e} (a >> b)$$

∼env Takes no arguments. Returns the environment.

$$(\langle{\sim}\texttt{env}\rangle_e) \xrightarrow{e} e$$

∼sys Takes one argument and evaluates it. Usually returns an implementation defined reciever based on its argument. See 5.3.3 The ∼sys Builtin for a complete definition.

$$((\langle{\sim}\texttt{sys}\rangle_e\ \ sym)\ \ \cdots) \xrightarrow{e} \text{See 5.3.3}$$

Each builtin takes a fixed number of arguments. Providing the wrong number of arguments to a builtin is undefined behavior.[14]

Attempting to use any atom that is not on this list of receivers is undefined behavior as is passing an argument of (or that evaluates to, where relevant) a type other than that expected.

[13] See 6 Required Optimizations.

[14] See 7 Undefined Behavior.

5.3.2 User Defined Receivers

If the receiver is a list then it represents a user-defined macro or function and should have exactly 2 or 3 elements, respectively.

The first of these elements is a pattern[15] and the second is the body. The third element has two purposes:

1. Its presence indicates that this receiver is a function rather than a macro.
2. It defines an environment to use when expanding the function body. Combined with the `~env` builtin, this allows lexical scoping of functions, scoped evaluation as a derived object, and more.[16]

To evaluate a call involving a user-defined receiver preform the following steps:

1. If the receiver is a function, evaluate all of its arguments, otherwise, leave them unevaluated.
2. Match the list of arguments against the included pattern to produce a new environment.[17]
    - If the receiver is a function, use the included environment as the base when constructing the new environment.
    - Otherwise, use the calling environment as the base when constructing the new environment.
3. Tail evaluate[18] the included body in the constructed environment.

Attempting to use any cons-cell which does not form a valid list of either exactly two or three elements where the first element is a valid pattern and the third (if present) is a valid environment as a receiver is undefined behavior.[19]

5.3.3 The `~sys` Builtin

`~sys` is the main point of extensibility built into μ_.

Calling the `~sys` with the 0 atom, e.g. as `((() ~sys) ())`, should return a list of pairs mapping *system operation names* to *system operation codes*.[20] This mapping should remain constant throughout a program's lifetime.

The heads of these pairs, the *system operation names*, are atoms which correspond to human readable names of a sort a programmer may write in their source code. The tails, meanwhile, the *system operation codes*, should also be atoms but which need not be human readable names.

Calling `~sys` with a system operation code should return a receiver which can be called to perform the desired system operation. The implementation may choose to represent this receiver using any μ_ object except one of the builtin atoms or a list of one or two elements. The receivers may be, in general, macro like, performing a more complex transform on their arguments than simple evaluation.

Using both a readable name and a code and delegating the mapping to the user's program this way allows for an extremely simple implementation of `~sys` using a jump table or similar structure while still allowing readable names to be used to refer to these operations.

6 Required Optimizations

Several points in 5 Evaluation specify to tail evaluate some expression. In such cases implementations MUST perform proper tail-call optimization. This means that evaluating an expression which causes a tail evaluation of another expression which itself causes the tail evaluation of another expression and so on should only take up $O(1)$ space for storage of the current evaluation state (e.g. call stack). This optimization is necessary to ensure programmers can write loops without fear of running out of space.

Implementations must not needlessly duplicate cons cells. Acquiring a cons cell from a binding multiple times should not require additional space for each instance.

Implementations must have a garbage collector. Concretely, repeatedly creating and then no longer using cons cells should not cause memory expenditure to grow without bound.

E Tests offers a list of test cases which, in addition to testing correctness of an implementation, test that these characteristics are met appropriately.

7 Undefined Behavior

Several points throughout this specification refer to certain occurrences as undefined behavior. Implementations are free to adopt arbitrary behavior in these instances including but not limited to:

- Extending the specification behavior
- Producing some kind of error
- Producing a nonsense result
- Halting and catching fire

[15] See 4 Pattern Matching.

[16] See TODO.

[17] See 4 Pattern Matching for the definition of this operation.

[18] See 6 Required Optimizations.

[19] See 7 Undefined Behavior.

[20] For a system which does not provide implementation specific functionality through this mechanism just return `()`.

Appendix A

Background

Appendix B

Implementation Strategies

Appendix C

Derivation of Higher-Level Functionality

Appendix D

Reference Implementation (WebAssembly)

This appendix contains a reference implementation of a µ_ interpreter written in WebAssembly and the Javascript bindings necessary to embed it on the Web or in Deno.

While the full source code of these files is included in this appendix for completeness, a reader interested in acquiring copies can much more easily obtain them from the `reference-implementation` directory of the µ_ project's git repository in clean, plain text.

D.1 WebAssembly Text — `mu_.wat`

```
;; # `mu_.wat` — A reference implementation for the mu_ programming language
;;
;; `mu_.wat` is handcrafted, unfolded, and well commented WebAssembly text
;; program which can serve suitably both as an implementation in its own right
;; and as a template for ports to other assembly and programming languages.
;; See `mu_.mjs` for Javascript bindings
;;
;; This is version `r0.3i1` which implements `r0.3` of the mu_ specification.
;;
;; To compile, run: `wat2wasm --enable-multi-memory --enable-tail-call mu_.wat`
;;
;; ## Exposed Details
;;
;; All items are uniformly represented as 32 bit signed integers, with positive
;; numbers representing atoms in the obvious way and negative numbers
;; representing cons cells as their negative offsets into the cons cell stack.
;;
;; There is a string yard, a simple memory buffer and allocator `syalloc` which
;; the embedder can write strings into as a pre-condition to calling interpreter
;; methods which expect strings.
;;
;; `mu_.wat` exports the following bindings (in order of definition)
;;   - `cons`          : Construct a cons cell
;;   - `head`          : Take the head of a cons cell
;;   - `tail`          : Take the tail of a cons cell
;;   - `lookup`        : Look up a symbol in an environment
;;   - `match`         : Match a value against a pattern in a base environment
;;   - `stringyard`    : A buffer for the embedder to write strings (such as
;;                       program source code or atom names) into
;;   - `syalloc`       : Allocate space in the string yard for a new string
;;   - `inter_string`  : Take an offset and length (in bytes) into the string
;;                       yard and inter the string onto the internment stack,
;;                       returning its atom number
;;   - `lookup_interred_string` : Given an atom, return the offset and length
;;                                into the string yard of that atom's name,
;;                                yielding -1, -1 if it is unnamed
;;   - `parse`         : Parse a string in the string yard to a mu_ value
;;   - `eval`          : Evaluate an expression in an environment (then gc)
;;   - `register_system_operation` : Given an atom and a funcref register a
;;                                   system operation with that name handled by
;;                                   that handler
;;   - `gc_get_anchor` : Get an anchor for the garbage collector
;;                       Usually, you should call this and pass the result as
;;                       the third argument to `eval`. Read the comments in the
;;                       garbage collection section which offer more detail on
;;                       how garbage collection is implemented before using the
;;                       the garbage collection mechanism in any other way
;;   - `gc_collect`    : Run the garbage collector
;;
;; (C) 2025 Brielle Hoff --- Dual licensed under CC BY-NC 4.0 and MIT.

(module
    ;; ---------- Cons Cells ----------
    ;; Each cell is (head, tail)
    ;; $cons_cells_top points to the last allocated cons cell
    (memory $cons_cells 1)
    (data (memory $cons_cells) (i32.const 0) "\00\00\00\00\00\00\00\00")
    ;; Points to the last element of the stack
    (global $cons_cells_top (mut i32) (i32.const 0))

    ;; Construct a cell
    (func $cons (export "cons")
        (param $head i32) (param $tail i32)
        (result i32)
        (local $top i32)

        ;; Adjust the stack pointer
        global.get $cons_cells_top
        i32.const 8
        i32.add
        local.tee $top
        global.set $cons_cells_top

        ;; Write the head value
        local.get $top
        local.get $head
        i32.store (memory $cons_cells)

        ;; Write the tail value
        local.get $top
        i32.const 4
        i32.add
        local.get $tail
        i32.store (memory $cons_cells)

        ;; Negate the offset into the stack to produce the representation
        i32.const 0
        local.get $top
        i32.sub
    )

    ;; Take the head of a cell
    (func $head (export "head")
        (param $cell i32)
        (result i32)

        ;; Negate the representation to get the offset into the stack
        i32.const 0
        local.get $cell
        i32.sub

        ;; Read the head value
        i32.load (memory $cons_cells)
    )

    ;; Take the tail of a cell
    (func $tail (export "tail")
        (param $cell i32)
        (result i32)

        ;; Negate the representation to get the offset into the stack and
        ;; shift by four to get the tail instead of the head in one step
        i32.const 4
        local.get $cell
        i32.sub

        ;; Read the tail value
        i32.load (memory $cons_cells)
    )

    ;; ---------- Primitive Interpreter Operations ----------

    ;; Lookup a symbol in an environment
    (func $lookup (export "lookup")
        (param $symbol i32)
        (param $environment i32)
        (result i32)
        (local $binding i32)

        (loop $loop (result i32)
            local.get $environment
            i32.eqz
            (if (result i32)
                (then
                    ;; We've exhausted the environment, map a symbol to itself
                    local.get $symbol
                )
                (else
                    ;; Get the first binding in the environment
                    local.get $environment
                    call $head
                    local.tee $binding

                    call $head
                    local.get $symbol
                i32.eq
                (if (result i32)
                    (then
                        ;; If it matches, return the bound value
                        local.get $binding
                        call $tail
                    )
                    (else
                        ;; Otherwise, continue over the remaining bindings
                        local.get $environment
                        call $tail
                        local.set $environment
                        br $loop
                    )
                )
            ))
    )
)

;; Match a value against a pattern in a base environment
(func $match (export "match")
    (param $value i32)
    (param $pattern i32)
    (param $environment i32)
    (result i32)

    local.get $pattern
    i32.eqz
    (if (result i32)
        (then
            ;; The pattern is (), don't introduce any bindings
            local.get $environment
        )
        (else
            local.get $pattern
            i32.const 0
            i32.gt_s
            (if (result i32)
                (then
                    ;; The pattern is a positive atom, add a binding to the
                    ;; environment
                    local.get $pattern
                    local.get $value
                    call $cons
                    local.get $environment
                    call $cons
                )
                (else
                    ;; The pattern is a cons cell, recurse

                    ;; Match the value head against the pattern head
                    local.get $value
                    call $head
                    local.get $pattern
                    call $head

                    ;; Match the value tail against the pattern tail
                    local.get $value
                    call $tail
                    local.get $pattern
                    call $tail
                    local.get $environment

                    call $match
                    call $match
                )
            )
        )
    )
)

;; ---------- String Yard ----------
;; The stringyard is a place for the embedding application to place strings
;; it wishes to pass to the interpreter.
;; Call syalloc(size) to acquire size bytes from the yard to place a string
(memory $stringyard (export "stringyard") 1)
(data (memory $stringyard) (i32.const 0)
    "~~true..~~false."
    "~~head..~~tail..~~cons.."
    "~~lte...~~eq...."
    "~~add...~~sub..."
    "~~and...~~or....~~not..."
    "~~shl...~~shr..."
    "~~env...~~sys...")
(global $stringyard_top (mut i32) (i32.const 128))

;; Allocate space on the string yard to place a string of size bytes
;; mu_ never uses this internally (nor modifies the string yard at all)
;; it only uses it for its initial strings and as a dumping ground for the
;; embedder to place strings into for it to use.
(func (export "syalloc")
    (param $size i32)
    (result i32)
    (local $top i32)

    ;; Adjust the available space top
    ;; This implementation is extremely simple because we don't do any
    ;; sophisticated memory management for strings.
    global.get $stringyard_top
    local.tee $top
    local.get $size
    i32.add
    global.set $stringyard_top

    local.get $top
)

;; Compare two strings in the string yard
(func $str_eq
    (param $a_off i32)
    (param $a_len i32)
    (param $b_off i32)
    (param $b_len i32)
    (result i32)
    (local $a_end i32)

    ;; Pre-check the string lengths
    local.get $a_len
    local.get $b_len
    i32.eq
    (if
        (then
            ;; Convert len to end for simpler iteration
            local.get $a_off
            local.get $a_len
            i32.add
            local.set $a_end

            ;; Loop over the bytes in the strings
            (block $break_loop
                (loop $loop
                    ;; If we've reached the end, the strings are equal
                    local.get $a_off
                    local.get $a_end
                    i32.eq
                    (if
                        (then
                            i32.const 1
                            return
                        )
                    )

                    ;; Compare bytes, break if unequal
                    local.get $a_off
                    i32.load8_u (memory $stringyard)
                    local.get $b_off
                    i32.load8_u (memory $stringyard)
                    i32.ne
                    br_if $break_loop

                    ;; Increment the offsets into the strings
                    local.get $a_off
                    i32.const 1
                    i32.add
                    local.set $a_off
                    local.get $b_off
                    i32.const 1
                    i32.add
                    local.set $b_off

                    br $loop
                )
            )
        )
    )

    ;; The strings are different
    i32.const 0
)

;; String internment
(memory $string_internment_stack 1)
;; [{ offset: i32, len: u16, system_opcode: u16 }]
(data (memory $string_internment_stack) (i32.const 0)
    "\00\00\00\00" "\00\00" "\00\00" ;; <ensure no string at 0>
    "\00\00\00\00" "\06\00" "\00\00" ;; ~~true  (6)
    "\08\00\00\00" "\07\00" "\00\00" ;; ~~false (7)
    "\10\00\00\00" "\06\00" "\00\00" ;; ~~head  (6)
    "\18\00\00\00" "\06\00" "\00\00" ;; ~~tail  (6)
    "\20\00\00\00" "\06\00" "\00\00" ;; ~~cons  (6)
    "\28\00\00\00" "\05\00" "\00\00" ;; ~~lte   (5)
    "\30\00\00\00" "\04\00" "\00\00" ;; ~~eq    (4)
    "\38\00\00\00" "\05\00" "\00\00" ;; ~~add   (5)
    "\40\00\00\00" "\05\00" "\00\00" ;; ~~sub   (5)
    "\48\00\00\00" "\05\00" "\00\00" ;; ~~and   (5)
    "\50\00\00\00" "\04\00" "\00\00" ;; ~~or    (4)
    "\58\00\00\00" "\05\00" "\00\00" ;; ~~not   (5)
    "\60\00\00\00" "\05\00" "\00\00" ;; ~~shl   (5)
    "\68\00\00\00" "\05\00" "\00\00" ;; ~~shr   (5)
    "\70\00\00\00" "\05\00" "\00\00" ;; ~~env   (5)
    "\78\00\00\00" "\05\00" "\00\00" ;; ~~sys   (5)
)
;; Points “One past the end” of the stack
(global $string_internment_stack_top (mut i32) (i32.const 136))

;; Inter a string from the stringyard onto the string internment stack
(func $inter_string (export "inter_string")
    (param $off i32)
    (param $len i32)
    (result i32)
    (local $idx i32)

    ;; Initialize idx to zero
    i32.const 0
    local.set $idx

    ;; Loop over the strings in the internment stack
    (block $scan
        (block $break_loop
            (loop $loop
                ;; If we reached the end of the internment stack, we need
                ;; to add a new string
                local.get $idx
                global.get $string_internment_stack_top
                i32.eq
                br_if $break_loop

                ;; Load the idx'th string from the stack and compare to the
                ;; string to be interred, if equal we found a match
                local.get $idx
                i32.load (memory $string_internment_stack)
                local.get $idx
                i32.const 4
                i32.add
                i32.load16_u (memory $string_internment_stack)
                local.get $off
                local.get $len
                call $str_eq
                br_if $scan

                ;; Increment idx
                local.get $idx
                i32.const 8
                i32.add
                local.set $idx

                br $loop
            )
        )

        ;; We need to add a new string

        ;; Store the string onto the internment stack
        local.get $idx
        local.get $off
        i32.store (memory $string_internment_stack)
        local.get $idx
        i32.const 4
        i32.add
        local.get $len
        i32.store16 (memory $string_internment_stack)
        local.get $idx
        i32.const 6
        i32.add
        i32.const 0
        i32.store16 (memory $string_internment_stack)

        ;; Adjust the top of the internment stack
        local.get $idx
        i32.const 8
        i32.add
        global.set $string_internment_stack_top
    )

    local.get $idx

    ;; Flip bit 29 to decrease the odds the numeric representation occurs
    ;; by chance when doing ordinary calculations
    i32.const 0x20_00_00_00
    i32.xor
)

;; Lookup an interred string or return -1 -1 if not an interred string atom
;; String atom detection is on a best effort basis as all string atoms have
;; a numeric value it is always possible that a string atom is created by
;; chance via a numeric calculation.
;; String atoms are given unusual numbers to minimize the likelihood of this
;; but it remains possible.
(func (export "lookup_interred_string")
    (param $idx i32)
    (result i32 i32)

    ;; Undo the flip of bit 29
    local.get $idx
    i32.const 0x20_00_00_00
    i32.xor
    local.set $idx

    (block $unnamed
        ;; Check that the index is in bounds
        local.get $idx
        global.get $string_internment_stack_top
        i32.ge_u
        br_if $unnamed

        ;; Check that the index is properly aligned
        local.get $idx
        i32.const 0x7
        i32.and
        i32.const 0
        i32.ne
        br_if $unnamed

        ;; This could be an interred string, load its properties

        ;; Load the offset
        local.get $idx
        i32.load (memory $string_internment_stack)

        ;; Load the length
        local.get $idx
        i32.const 4
        i32.add
        i32.load16_u (memory $string_internment_stack)

        return
    )

    ;; This isn't an interred string, return the failure sentinel
    i32.const -1
    i32.const -1
    return
)

;; ---------- Parsing ----------

;; Parse mu_ source text to an expression
(func $parse (export "parse")
    (param $off i32)
    (param $len i32)
    (result i32)
    (local $end i32)

    ;; Convert len to end for easier iteration
    local.get $off
    local.get $off
    local.get $len
    i32.add
    local.tee $end

    ;; Skip leading whitespace
    call $parse_skip_ws

    ;; Parse an expression
    local.get $end
    call $parse_expr

    ;; Skip trailing whitespace
    local.get $end
    call $parse_skip_ws

    drop
)

;; Parse an expression
;; Returns expr, off
(func $parse_expr
    (param $off i32)
    (param $end i32)
    (result i32 i32)

    ;; Check for a left parenthesis
    local.get $off
    local.get $end
    call $parse_peek
    i32.const 0x28 ;; LEFT PARENTHESIS
    i32.eq
    (if (result i32)
        (then
            ;; There is a left parenthesis, this is a list/cell

            ;; Eat the left parenthesis
            local.get $off
            i32.const 1
            i32.add

            ;; Parse the rest of the list/cell
            local.get $end
            return_call $parse_list_inner
        )
        (else
            ;; There isn't a left parenthesis, this is an atom

            ;; Parse an atom
            local.get $off
            local.get $end
            return_call $parse_atom
        )
    )
)

;; Parse a list sans leading paren
;; Returns expr, off
(func $parse_list_inner
    (param $off i32)
    (param $end i32)
    (result i32 i32)
    (local $chr i32)

    ;; Skip leading whitespace
    local.get $off
    local.get $end
    call $parse_skip_ws

    ;; Check for a dot
    local.get $end
    call $parse_peek
    local.tee $chr
    i32.const 0x2e ;; FULL STOP
    i32.eq
    (if (param i32) (result i32)
        (then
            ;; There is a dot, parse one final tail expression

            ;; Eat the dot
            i32.const 1
            i32.add

            ;; Skip whitespace after the dot
            local.get $end
            call $parse_skip_ws

            ;; Parse an expression
            local.get $end
            call $parse_expr

            ;; Skip trailing whitespace before the closing parenthesis
            local.get $end
            call $parse_skip_ws

            ;; Eat the closing paren
            i32.const 1
            i32.add
            return
        )
    )

    ;; Check for a right parenthesis
    local.get $chr
    i32.const 0x29 ;; RIGHT PARENTHESIS
```

```
        i32.eq
        (if (param i32) (result i32)
            (then
                ;; There is a right parenthesis, return the 0 atom

                local.set $off

                ;; Load the value zero
                i32.const 0

                ;; Eat the right parenthesis
                local.get $off
                i32.const 1
                i32.add
                return
            )
        )

        ;; There are more items remaining

        ;; Parse an expression
        local.get $end
        call $parse_expr

        ;; Parse the rest of the list
        local.get $end
        call $parse_list_inner

        ;; Combine the results
        local.set $off
        call $cons
        local.get $off
    )

    ;; Parse an atom
    ;; Returns expr, off
    (func $parse_atom
        (param $off i32)
        (param $end i32)
        (result i32 i32)
        (local $chr i32)
        (local $nex i32)

        ;; Determine the extent of the atom
        local.get $off
        local.get $off
        (block $break_loop (param i32) (result i32)
            (loop $loop (param i32) (result i32)
                local.get $end
                call $parse_peek
                local.tee $chr

                ;; Break for whitespace or end
                i32.const 32
                i32.le_u
                br_if $break_loop

                ;; Break for lparen
                local.get $chr
                i32.const 0x28
                i32.eq
                br_if $break_loop

                ;; Break for rparen
                local.get $chr
                i32.const 0x29
                i32.eq
                br_if $break_loop

                ;; Break for dot
                local.get $chr
                i32.const 0x2e
                i32.eq
                br_if $break_loop

                i32.const 1
                i32.add
                br $loop
            )
        )

        ;; Inter the atom
        local.tee $nex
        local.get $off
        i32.sub
        call $inter_string

        local.get $nex
    )

    ;; Skip whitespace
    (func $parse_skip_ws
        (param $off i32)
        (param $end i32)
        (result i32)

        ;; Loop over bytes
        local.get $off
        (block $break_loop (param i32) (result i32)
            (loop $loop (param i32) (result i32)
                local.get $end
                call $parse_peek

                ;; If the value is in [1, 32] it is whitespace
                i32.const 1
                i32.sub
                i32.const 32
                i32.ge_u
                br_if $break_loop
                i32.const 1
                i32.add
                br $loop
            )
        )
    )

    ;; Peek at a byte -- off, byte
    (func $parse_peek
        (param $off i32)
        (param $end i32)
        (result i32 i32)

        local.get $off

        ;; If at the end of the input, return 0
        (block $not_end (param i32) (result i32)
            local.get $off
            local.get $end
            i32.ne
            br_if $not_end

            i32.const 0
            return
        )

        ;; Get a byte
        local.get $off
        i32.load8_u (memory $stringyard)
    )

    ;; ---------- Evaluation : The Heart of the Interpreter ----------

    ;; Table of builtins and system operations
    ;; The former take slots 0 - 16 while the latter start in slot 32
    (table 64 funcref)
    (elem (i32.const 0)
        $eval_builtin_quote
        $eval_builtin_true
        $eval_builtin_false
        $eval_builtin_head
        $eval_builtin_tail
        $eval_builtin_cons
        $eval_builtin_lte
        $eval_builtin_eq
        $eval_builtin_add
        $eval_builtin_sub
        $eval_builtin_and
        $eval_builtin_or
        $eval_builtin_not
        $eval_builtin_shl
        $eval_builtin_shr
        $eval_builtin_env
        $eval_builtin_sys
    )

    (type $invokable (func
        (param $arguments i32)
        (param $environment i32)
        (param $gc_anchor i32)
        (result i32)
    ))

    ;; Evaluate an expression in an environment, then perform garbage
    ;; collection from the given anchor.
    ;;
    ;; A garbage collection anchor is passed to this function so that the
    ;; needed garbage collection information can be maintained over tail calls.
    ;;
    ;; For the end user, you should probably just pass gc_get_anchor() unless
    ;; you would otherwise call gc_collect(<result of eval>, <some anchor>)
    ;; immediately after anyways.
    (func $eval (export "eval")
        (param $expression i32)
        (param $environment i32)
        (param $gc_anchor i32)
        (result i32)
        (local $receiver i32)

        ;; Determine whether the expression is an atom or cons cell
        local.get $expression
        i32.const 0
        i32.ge_s
        (if (result i32)
            (then
                ;; This is an atom, determine whether it is 0
                local.get $expression
                i32.eqz
                (if (result i32)
                    (then
                        ;; The 0 atom evaluates to itself
                        i32.const 0
                    )
                    (else
                        ;; Positive atoms are looked up in the environment
                        local.get $expression
                        local.get $environment
                        call $lookup
                    )
                )

                ;; Run garbage collection
                local.get $gc_anchor
                call $gc_collect
            )
            (else
                ;; This is a cons cell, it should be a list representing an
                ;; invocation of either a user defined receiver or a builtin

                ;; Evaluate the head to determine the receiver
            local.get $expression
            call $head
            local.get $environment
            global.get $cons_cells_top
            call $eval
            local.set $receiver

            ;; Check whether the receiver is an atom
            local.get $receiver
            i32.const 0
            i32.ge_s
            (if (result i32)
                (then
                    ;; The receiver is an atom, this is a builtin call
                    ;; Look up the builtin in the table of builtins

                    ;; Load the arguments
                    local.get $expression
                    call $tail

                    ;; Load the environment
                    local.get $environment

                    ;; Load the garbage collection anchor
                    local.get $gc_anchor

                    ;; Load the receiver
                    local.get $receiver

                    ;; Zero out bit 29
                    i32.const 0xDF_FF_FF_FF
                    i32.and

                    ;; Shift the value to get an index into the table
                    ;; (which increments by one each time) instead of the
                    ;; memory (which increments by eight each time)
                    i32.const 3
                    i32.shr_u

                    ;; Invoke the specified builtin
                    return_call_indirect (type $invokable)
                )
                (else
                    ;; The receiver is a cons cell, it is either a system
                    ;; defined receiver (if its tail is ~~sys) or a user
                    ;; defined receiver otherwise
                    local.get $receiver
                    call $tail
                    i32.const 0x20_00_00_80 ;; ~~sys
                    i32.eq
                    (if (result i32)
                        (then
                            ;; Run the system defined receiver handler

                            ;; Load the arguments
                            local.get $expression
                            call $tail

                            ;; Load the environment
                            local.get $environment

                            ;; Dynamically dispatch
                            local.get $receiver
                            call $head
                            i32.const 32
                            i32.add
                            call_indirect (type $system_operation_handler)

                            ;; Run garbage collection
                            local.get $gc_anchor
                            call $gc_collect
                        )
                        (else
                            ;; Load the arguments
                            local.get $expression
                            call $tail

                            ;; Load the environment
                            local.get $environment

                            ;; Load the garbage collection anchor
                            local.get $gc_anchor

                            ;; Load the receiver
                            local.get $receiver

                            ;; Run the user defined receiver handling
                            return_call $eval_invoke_user_defined
                        )
                    )
                )
            )
        )
    )
)

;; Evaluates each item in a list, forming a new list
(func $eval_list
    (param $expression_list i32)
    (param $environment i32)
    (result i32)

    ;; Check whether the list is empty
    local.get $expression_list
    i32.eqz
    (if (result i32)
        (then
            ;; This is an empty list, we've evaluated the entire list
            i32.const 0
        )
        (else
            ;; This is a non-empty list, evaluate the first element,
            ;; recurse over the remaining elements, and recombine

            ;; Evaluate the head
            local.get $expression_list
            call $head
            local.get $environment
            global.get $cons_cells_top
            call $eval

            ;; Evaluate each item in the tail
            local.get $expression_list
            call $tail
            local.get $environment
            call $eval_list

            ;; Recombine the evaluated head and tail
            call $cons
        )
    )
)

;; Evaluate an invocation of a user defined receiver (function or macro)
(func $eval_invoke_user_defined
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (param $receiver i32)
    (result i32)
    (local $function_environment i32)
    (local $preserved i32)

    (block $skip_function_ops
        ;; Check if the receiver has a third element (is a function)
        local.get $receiver
        call $tail
        call $tail
        local.tee $function_environment
        i32.eqz
        br_if $skip_function_ops

        ;; If it is, do the following, otherwise skip

        ;; Evaluate each argument in the list of arguments
        local.get $arguments
        local.get $environment
        call $eval_list
        local.set $arguments

        ;; Update the environment to use to the specified one instead of
        ;; the one belonging to the caller
        local.get $function_environment
        call $head
        local.set $environment
    )

    ;; Get the body
    local.get $receiver
    call $tail
    call $head

    ;; Get the arguments
    local.get $arguments

    ;; Get the pattern
    local.get $receiver
    call $head

    ;; Match the arguments against the pattern
    local.get $environment
    call $match

    ;; Run garbage collection
    ;; This garbage collection is unnecessary for ensuring the final size
    ;; of the cons cell stack as a garbage collection from the same point
    ;; will be run downstream of the tail call which will be made.
    ;; However, this garbage collection does help to keep down the level of
    ;; garbage when repeatedly tail evaluating from one user defined
    ;; receiver into another as is extremely common in real mu_ programs.
    call $cons
    local.get $gc_anchor
    call $gc_collect
    local.tee $preserved
    call $head
    local.get $preserved
    call $tail

    ;; Tail evaluate the body
    local.get $gc_anchor
    return_call $eval
)

;; Evaluate an invocation of the () builtin
(func $eval_builtin_quote
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~~true builtin
(func $eval_builtin_true
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Tail evaluate it
    local.get $environment
    local.get $gc_anchor
    return_call $eval
)

;; Evaluate an invocation of the ~false builtin
(func $eval_builtin_false
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the second argument
    local.get $arguments
    call $tail
    call $head

    ;; Tail evaluate it
    local.get $environment
    local.get $gc_anchor
    return_call $eval
)

;; Evaluate an invocation of the ~head builtin
(func $eval_builtin_head
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Get its head
    call $head

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~tail builtin
(func $eval_builtin_tail
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Get its tail
    call $tail

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~cons builtin
(func $eval_builtin_cons
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Get the second argument
    local.get $arguments
    call $tail
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Cons them together
    call $cons

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)
```

```
;; Evaluate an invocation of the ~lte builtin
(func $eval_builtin_lte
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Prepare the ~true and ~false atoms
    i32.const 0x20_00_00_08 ;; ~true
    i32.const 0x20_00_00_10 ;; ~false

    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Get the second argument
    local.get $arguments
    call $tail
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Select the appropriate atom (~true or ~false)
    ;; based on whether the first is less than or equal to the second
    i32.le_s
    select

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~eq builtin
(func $eval_builtin_eq
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Prepare the ~true and ~false atoms
    i32.const 0x20_00_00_08 ;; ~true
    i32.const 0x20_00_00_10 ;; ~false

    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Get the second argument
    local.get $arguments
    call $tail
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Select the appropriate atom (~true or ~false)
    ;; based on whether the first is equal to the second
    i32.eq
    select

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~add builtin
(func $eval_builtin_add
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Get the second argument
    local.get $arguments
    call $tail
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Add the arguments and take the result modulo 2 ^ 31
    i32.add
    i32.const 0x7F_FF_FF_FF
    i32.and

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~~sub builtin
(func $eval_builtin_sub
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Get the second argument
    local.get $arguments
    call $tail
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Subtract the arguments and take the result modulo 2 ^ 31
    i32.sub
    i32.const 0x7F_FF_FF_FF
    i32.and

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~~and builtin
(func $eval_builtin_and
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Get the second argument
    local.get $arguments
    call $tail
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Take the bitwise and of the arguments
    i32.and

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~~or builtin
(func $eval_builtin_or
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Get the second argument
    local.get $arguments
    call $tail
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Take the bitwise or of the arguments
    i32.or

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~~not builtin
(func $eval_builtin_not
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Take the bitwise negation of the least significant 31 bits
    i32.const 0x7F_FF_FF_FF
    i32.xor

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~~shl builtin
(func $eval_builtin_shl
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Get the second argument
    local.get $arguments
    call $tail
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Bit shift left the first argument by the second, keeping only the
    ;; least significant 31 bits
    i32.shl
    i32.const 0x7F_FF_FF_FF
    i32.and

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~~shr builtin
(func $eval_builtin_shr
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the first argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Get the second argument
    local.get $arguments
    call $tail
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval

    ;; Bit shift right the first argument by the second
    i32.shr_u

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~~env builtin
(func $eval_builtin_env
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    ;; Get the environment
    local.get $environment

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; Evaluate an invocation of the ~~sys builtin
(func $eval_builtin_sys
    (param $arguments i32)
    (param $environment i32)
    (param $gc_anchor i32)
    (result i32)
    (local $code i32)

    ;; Get the argument
    local.get $arguments
    call $head

    ;; Evaluate it
    local.get $environment
    global.get $cons_cells_top
    call $eval
    local.tee $code

    i32.eqz
    (if (result i32)
        (then
            call $construct_system_operation_table
        )
        (else
            ;; Construct a system receiver
            ;; (this implementation represents these as (n . ~~sys))
            local.get $code
            i32.const 0x20_00_00_80 ;; ~~sys
            call $cons
        )
    )

    ;; Run garbage collection
    local.get $gc_anchor
    call $gc_collect
)

;; ---------- System Operation Registration ----------

(type $system_operation_handler (func
    (param $arguments i32)
    (param $environment i32)
    (result i32)
))

(global $highest_system_opcode (mut i32) (i32.const 0))

;; Register a system operation under the designated name using the provided
;; handler. To be fully spec compliant this function should only be called
;; before any mu_ code has been evaluated.
(func (export "register_system_operation")
    (param $operation_name i32)
    (param $handler funcref)
    (local $opcode i32)
    (local $size_delta i32)

    ;; Get the next available opcode
    global.get $highest_system_opcode
    i32.const 1
    i32.add
    local.tee $opcode
    global.set $highest_system_opcode

    ;; Resize the function table as necessary

    i32.const 1
    i32.const 32

    ;; Compute the table index
    local.get $opcode
    i32.const 32
    i32.add

    ;; Round up to the next power of two
    i32.clz
    i32.sub
    i32.shl

    ;; Compute the size delta
    table.size
    i32.sub
    local.tee $size_delta

    ;; Resize if the delta is positive
    i32.const 0
    i32.gt_s
    (if
        (then
            ref.null func
            local.get $size_delta
            table.grow
            drop
        )
    )

    ;; Set the designated slot
    local.get $opcode
    i32.const 32
    i32.add
    local.get $handler
    table.set

    ;; Update the string internment entry to indicate the opcode

    ;; Get the index into the internment stack from the atom number
    local.get $operation_name
    i32.const 0xDF_FF_FF_FF
    i32.and

    ;; Offset to the opcode entry
    i32.const 6
    i32.add

    ;; Write the opcode
    local.get $opcode
    i32.store16 (memory $string_internment_stack)
)

;; (~~sys ()) -- get mappings from system operation names to codes
```

```
(func $construct_system_operation_table
    (result i32)
    (local $idx i32)
    (local $acc i32)
    (local $opcode i32)

    i32.const 0
    local.set $acc

    i32.const 0
    local.set $idx
    (loop $loop
        ;; Load the opcode
        local.get $idx
        i32.const 6
        i32.add
        i32.load16_u (memory $string_internment_stack)
        local.tee $opcode

        (block $skip_add (param i32)
            ;; If opcode is zero, this isn't a system operation, continue
            i32.eqz
            br_if $skip_add

            ;; Flip bit 29 of the index to get the atom number
            local.get $idx
            i32.const 0x20_00_00_00
            i32.xor

            ;; Add mapping from name to opcode to accumulator
            local.get $opcode
            call $cons
            local.get $acc
            call $cons
            local.set $acc
        )

        ;; Increment idx
        local.get $idx
        i32.const 8
        i32.add
        local.set $idx

        local.get $idx
        global.get $string_internment_stack_top
        i32.lt_u
        br_if $loop
    )

    local.get $acc
)

;; ---------- Garbage Collector ----------
;; The garbage collection mechanism operates as follows:
;;   1. Acquire a garbage collection anchor (gc_get_anchor)
;;   2. Do something that might cause the allocation of garbage
;;   3. Call gc_collect(<preserve>, <anchor>) passing the value you want to
;;      keep as preserve. The response is a new value (earlier on the stack
;;      where possible) which is equivalent to the passed preserve value.
;;      All cons cells which are not directly or indirectly a dependency of
;;      the preserved value and which were allocated after the anchor was
;;      taken are removed.
;;
;; Note that evaluation performs a garbage collection step itself so the
;; manual use of this mechanism by the embedder is only necessary to clear
;; cells created through a means other than evaluation such as parsing.

;; Get a garbage collection anchor
(func $gc_get_anchor (export "gc_get_anchor")
    (result i32)

    global.get $cons_cells_top
)

;; Run the garbage collector
(func $gc_collect (export "gc_collect")
    (param $preserve i32)
    (param $anchor i32)
    (result i32)
    (local $anchor_2 i32)

    ;; Take a second anchor, this represents where the cons stack grew to
    global.get $cons_cells_top
    local.set $anchor_2

    ;; Recursively copy the preserved element
    local.get $preserve
    local.get $anchor
    local.get $anchor_2
    call $gc_copy

    ;; Move the copied cells (the ones to be kept) down, overwriting the
    ;; range between the first and second anchors
    local.get $anchor
    i32.const 8
    i32.add
    local.get $anchor_2
    i32.const 8
    i32.add
    global.get $cons_cells_top
    local.get $anchor_2
    i32.sub
    memory.copy (memory $cons_cells)

    ;; Adjust the top of the stack down
    global.get $cons_cells_top
    local.get $anchor_2
    i32.sub
    local.get $anchor
    i32.add
```

```
        global.set $cons_cells_top
    )

    ;; Create copies of everything subsidiary to item above anchor
    ;; Return their new position minus the difference between anchor_2
    ;; and anchor.
    ;; Items below anchor are returned unchanged.
    ;;
    ;; This mechanism uses a sentinel (0x80_00_00_00) which cannot occur as a
    ;; normal item (since it would represent the last possible cons cell
    ;; assuming the full space was taken up) unless the program is almost
    ;; certainly about to crash from resource exhaustion anyway.
    ;; It uses this sentinel to mark a pointer at the original location of a
    ;; cons cell pointing to the new location of the cell.
    ;; This allows it to avoid making multiple distinct copies of shared cells
    ;; which is critical for meeting the required memory characteristics.
    (func $gc_copy
        (param $item i32)
        (param $anchor i32)
        (param $anchor_2 i32)
        (result i32)
        (local $copied_item i32)

        ;; Check if the item is above the anchor
        i32.const 0
        local.get $item
        i32.sub
        local.get $anchor
        i32.gt_s
        (if (result i32)
            (then
                ;; The item is above the anchor

                ;; Check if the head is the magic value 0x80_00_00_00 which
                ;; cannot occur normally
                local.get $item
                call $head
                i32.const 0x80_00_00_00
                i32.eq
                (if (result i32)
                    (then
                        ;; The head is the sentinel, this cell was already
                        ;; copied, load the cached copy
                        local.get $item
                        call $tail
                    )
                    (else
                        ;; Recursively copy the head
                        local.get $item
                        call $head
                        local.get $anchor
                        local.get $anchor_2
                        call $gc_copy

                        ;; Recursively copy the tail
                        local.get $item
                        call $tail
                        local.get $anchor
                        local.get $anchor_2
                        call $gc_copy

                        ;; Reconstruct the cons cell
                        call $cons

                        ;; Offset the cons cell to where it will go after being
                        ;; shifted down at the end of gc_collect
                        local.get $anchor_2
                        i32.add
                        local.get $anchor
                        i32.sub
                        local.set $copied_item

                        ;; Overwrite the original item with a magic sentinel
                        ;; indicating the item has already been copied and
                        ;; caching the copy.
                        ;; This step (and checking for these sentinels) is
                        ;; critical to ensure that multiple references to the
                        ;; same object don't become multiple separate copies
                        ;; and therefore that the garbage collector cannot
                        ;; inadvertently create more cons cells than there were
                        ;; to begin with.
                        i32.const 0
                        local.get $item
                        i32.sub
                        i32.const 0x80_00_00_00
                        i32.store (memory $cons_cells)

                        i32.const 4
                        local.get $item
                        i32.sub
                        local.get $copied_item
                        i32.store (memory $cons_cells)

                        local.get $copied_item
                    )
                )
            )
            (else
                ;; The item is below the anchor, do not adjust it
                local.get $item
            )
        )
    )
)
```

D.2 Javascript Bindings — mu_.mjs

```
// # `mu_.mjs` -- Javascript bindings for the `mu_.wasm` interpreter
//
// All uncommented public methods are thin wrappers around their WebAssembly
// counterparts, see `mu_.wat` for descriptions
//
// (C) 2025 Brielle Hoff --- Dual licensed under CC BY-NC 4.0 and MIT.

// Magic to import the WebAssembly module in either Deno or on the web
const module_url = URL.parse("./mu_.wasm", import.meta.url);
const module = typeof Deno === "object"
             ? await WebAssembly.compile(await Deno.readFile(module_url))
             : await WebAssembly.compileStreaming(fetch(module_url));

// You can't just pass Javascript functions to WebAssembly as funcrefs (no that
// would be too easy) so you have to use this incredibly janky hack where you
// import the function into a tiny module which does nothing but re-export the
// function in order to wrap it in the magic metadata that tells WebAssembly
// that it is permissible to use. Hopefully there will be a better way to do
// this at some point in the future.
const launder_js_function = (tiny_module => f => {
    const i = new WebAssembly.Instance(tiny_module, { ns: { f } });
    return i.exports.f;
})(await WebAssembly.compile(new Uint8Array([
    0x00, 0x61, 0x73, 0x6d, // wasm magic
    0x01, 0x00, 0x00, 0x00, // version
    0x01, 0x07,             // types section (id 1) --- 7 bytes
    0x01,                   //     1 type
    0x60,                   //     function (0x60)
    0x02, 0x7f, 0x7f,       //         2 inputs: i32 (0x7f) i32 (0x7f)
    0x01, 0x7f,             //         1 output: i32 (0x7f)
    0x02, 0x08,             // import section (id 2) --- 8 bytes
    0x01,                   //     1 import
    0x02, 0x6e, 0x73,       //     namespace: "ns"
    0x01, 0x66,             //     name: "f"
    0x00, 0x00,             //     function (magic 0) #0
    0x07, 0x05,             // export section (id 7) --- 5 bytes
    0x01,                   //     1 export
    0x01, 0x66,             //     name: "f"
    0x00, 0x00              //     function (0) #0
])));


export class Interpreter {
    static #builder_token = Symbol();

    // The builder method for Interpreter
    // This needs to be asynchronous for maximal efficiency and thus cannot be
    // a constructor, necessitating the use of a builder.
    //
    // To add system operations add them as entries of the sys property when
    // calling. Handlers should be functions taking in a (interpreter, arg, env)
    // triple and returning a result.
    //
    // For example you could instantiate using:
    // ```
    // Interpreter.instantiate({ sys: {
    //     "console:log": (mu_, arg, env) => {
    //         console.log(mu_.show(mu_.eval(arg, env)));
    //     },
    //     "math:random": (mu_, arg, env) => {
    //         return Math.floor(Math.random() * (2 ** 30));
    //     },
    // } })
    // ```
    // to provide a console log system function and random system function
    static async instantiate({ sys = {} } = {}) {
        const module_instance = await WebAssembly.instantiate(module);
        const instance = new this(Interpreter.#builder_token, module_instance);
        for (const [name, op] of Object.entries(sys)) {
            module_instance.exports.register_system_operation(
                instance.#inter_string(name),
                launder_js_function((arg, env) =>
                    instance.#conv(op(instance, arg, env)))
            );
        }
        return instance;
    }

    #bindings;

    constructor(builder_token, instance) {
        if (Interpreter.#builder_token !== builder_token) {
            throw new TypeError(
                "use await Interpreter.instantiate() to instantiate an interpreter"
            );
        }
        this.#bindings = instance.exports;
    }

    cons(head, tail) {
        return this.#bindings.cons(this.#conv(head), this.#conv(tail));
    }

    head(cell) {
        return this.#bindings.head(this.#conv(cell));
    }

    tail(cell) {
        return this.#bindings.tail(this.#conv(cell));
    }

    lookup(symbol, environment) {
        return this.#bindings.lookup(
            this.#conv(symbol),
            this.#conv(environment)
        );
```

```
    }

    match(value, pattern, environment = 0) {
        return this.#bindings.match(
            this.#conv(value),
            this.#conv(pattern),
            this.#conv(environment)
        );
    }

    eval(expression, environment = 0, anchor = null) {
        anchor ??= this.#bindings.gc_get_anchor();
        return this.#bindings.eval(
            this.#conv(expression),
            this.#conv(environment),
            anchor
        );
    }

    gc_get_anchor() {
        return this.#bindings.gc_get_anchor();
    }

    gc_collect(preserve, anchor) {
        return this.#bindings.gc_collect(this.#conv(preserve), anchor);
    }

    parse(str) {
        const { off, len } = this.#syalloc_string(str);
        return this.#bindings.parse(off, len);
    }

    // Method for rendering a mu_ object (which will just be an i32) as a
    // readable string. Implemented directly in Javascript as it is not part of
    // the core interpreter.
    //
    // The method chooses to render atoms without corresponding names as their
    // numeric representation as a u31 prefixed by the unicode symbol №.
    // Since the specification does not say anything about the interpretation
    // of characters outside the ascii range, this has no chance of colliding
    // with any code which is fully spec-compliant.
    //
    // This method only uses the dot symbol when it has to, always preferring
    // to render lists as much as it can.
    show(obj) {
        obj = this.#conv(obj);
        if (obj > 0) {
            const [ off, len ] = this.#bindings.lookup_interred_string(obj);
            if (len === -1) {
                return '№' + obj.toString();
            } else {
                const buf = new Uint8Array(
                    this.#bindings.stringyard.buffer,
                    off,
                    len
                );
                return this.#text_decoder.decode(buf);
            }
        } else {
            let str = '(';
            let first = true;
            while (obj < 0) {
                if (first) first = false;
                else       str += ' ';
                str += this.show(this.head(obj));
                obj  = this.tail(obj);
            }

            if (obj > 0) {
                str += " . ";
                str += this.show(obj);
            }

            str += ')';
            return str;
        }
    }

    // Convert a Javascript object into a mu_ one as best as possible
    // Converts arrays to lists and atoms to their atom numbers.
    #conv(obj) {
        if (typeof obj == "number") {
            return obj;
        } else if (typeof obj == "string") {
            return this.parse(obj);
        } else if (Array.isArray(obj)) {
            return obj.reduceRight(
                (acc, val) => this.cons(val, acc),
                this.#conv(obj["tail"] ?? 0)
            );
        } else if (obj == null) {
            return 0;
        }
    }

    // Inter a Javascript string into the mu_ interpreter
    #inter_string(str) {
        const { off, len } = this.#syalloc_string(str);
        return this.#bindings.inter_string(off, len);
    }

    #text_encoder = new TextEncoder();
    #text_decoder = new TextDecoder();

    // Allocate a javascript string into the mu_ interpreter's space
    #syalloc_string(str) {
        // We ask for 3 bytes per UTF-16 unit since that is an upper bound
        const size = str.length * 3;
        const off  = this.#bindings.syalloc(size);
        const buf  = new Uint8Array(
            this.#bindings.stringyard.buffer,
            off,
            size
        );

        const { written: len } = this.#text_encoder.encodeInto(str, buf);

        return { off, len };
    }
}
```

Appendix E

Tests