Tiny Lisp Interpreter

Brent Seidel
Phoenix, AZ

February 15, 2021

Contents

Chapter 1

Introduction

This document provides a definition of a Tiny-Lisp interpreter written in Ada. Without such a definition, it is difficult to determine if the language is actually doing what it should be doing. This makes debugging more complicated.

1.1 What is This?

This is a Tiny-Lisp interpreter written in Ada. It is designed to proved a language that can be embedded into other programs, including running on embedded systems without an operating system. As a result, effort has been made to remove dependencies on Ada packages that may not be available. A primary example is `Ada.Text_IO`. Another feature that may be missing is dynamic memory allocation.

1.2 Why is This?

As a young lad, I learned to program on 8-bit computers with minimal BASIC interpreters and 4-16K of RAM. With these simple systems, one had a hope of being able to understand the complete system at a fairly low level. Now, one can buy small computers like the Arduino Due with 32-bit processors, 96K of RAM, and 512K of flash memory (I'm ignoring systems like the Raspberry PI as they are full up Linux computer and thus are more complicated). This seemed like a reasonable platform for recreating the early experience.

1.2.1 Why Lisp?

Why not? My first thought was to use some flavor of Tiny BASIC which would have more in common with those early systems. I then realized that Lisp is much easier to parse. Being somewhat lazy and interested in various computer languages, I decided that some form of a "Tiny-Lisp" would be a good idea.

Tiny-Lisp can be thought of as a small subset of Common Lisp, with some extensions of use to embedded systems. Most of the more complex features of Common Lisp are not and probably

never will be available in this Tiny-Lisp. However, one should be able to write code in Tiny-Lisp and have it actually run on a Common Lisp system.

1.2.2 Why Ada?

Again, why not? I have developed an interest in Ada, especially for programming embedded systems. It has features, such as strong typing, which can help to catch errors early, thus saving time debugging. I would not claim to be the world's greatest programmer, so I need all the help that I can get.

Chapter 2

The Language

As a "Tiny-Lisp", some (many) of the features of Common Lisp are not available. Some of the lacks may be temporary while others will be permanent, and some may be added by the host program.

2.1 User Interface

The interpreter reads text from an input device, parses it, and and executes it. The function used to read the input must match the signature for `Ada.Text_IO.Get_Line()` and this will probably be used if that is available. On an embedded system without `Ada.Text_IO`, the user must provide a suitable function.

Comments

A comment starts with a semicolon character, ";", and extends to the end of the line. Any text in a comment is ignore by the interpreter.

Continuation

If a list isn't closed (number of open parentheses matches the number of close parentheses) by the end of the line, the interpreter will ask for more text. This will continue until the list is closed.

2.2 Optimization

None. Some could possibly be added, but right now the focus has been on getting things to work correctly.

2.3 Syntax

2.3.1 Special Characters

There are only a few characters with special significance. Parenthesis, "(" and ")", are used for delimiting lists. Quotation marks, "" are used for delimiting strings. The apostrophe " ' " is used

for quoting symbols or lists. The semicolon, ";", indicates a comment. The pound sign (octothorp) "#" is used to indicate certain special processing. Spaces are used to separate elements in a list. That's about it. However, it's probably best to avoid most symbol characters since some more special characters may be added. A good rule of thumb would be to avoid any special characters that are used by Common Lisp.

The language is case insensitive thus, `CAR`, `car`, `cAr`, etc all are considered identical by the language.

2.3.2 Reserved Words

There are almost none. *`T`* and *`NIL`* refer to the boolean true and false values, and you can't define a symbol that it already used for a builtin or special operation. However, even the builtin and special operations are not, strictly speaking, reserved words. Their names are strings that are added to the symbol table during program initialization. They can easily be changed (say to translate into a different language) and the interpreter recompiled.

2.3.3 Examples

The basic syntax for languages in the Lisp family is very simple. Everything is a list of elements, where each element may also be a list. Elements are separated by spaces and the list is contained in parentheses. Here is a simple list:

```
(+ 1 2 3)
```

The first element in the list is the symbol "+". The following elements are "1", "2", and "3". The "+" symbol is the addition operation and adds the following integers together. Thus, the example would return the integer "6".

A more complicated example:

```
(+ (* 2 3) (* 4 5))
```

This is equivalent to $2 * 3 + 4 * 5$. Breaking this down, the first element of the outside list is "+". The second element is the list `(* 2 3)` and the third element is the list `(* 4 5)`. Since "*" is the symbol for the multiplication operation, this returns a value of 26.

A final example:

```
(print "Hello World!")
```

This list consists of only two elements. The first is the symbol `print`. The second is the string "Hello World!". With strings, everything from the starting quotation mark to the next quotation mark is part of the string. This means that you can't have a string that contains a quotation mark (at some point, a work-around may be available).

2.4 Symbols and Variables

Elements that are not numbers, strings, or lists are symbols or variables. In determining what the element represents, the search order is:

1. Boolean literals are checked first.

2. Builtin or Special symbols are checked next.

3. Variables in the most recent stack frame.

4. Variables in older stack frames.

5. Variable symbols are checked last. These can be considered to be global variables.

All symbols share the same namespace. This makes this Tiny-Lisp a LISP-1 (for those who are interested in such things). It is possible that this will change at some point.

Another thing to be aware of is that if a function is defined within a function definition or local block, the inner definition may reference locals or parameters in the outer blocks. In Common Lisp, this creates a closure where the variables remain accessible. This does not work in Tiny-Lisp and may cause an error when the function is called. It is best to define functions are the top level for now. For example, consider the following:

```
(let ((a 10)) (defun test (b) (print "Sum is " (+ a b)) (terpri))
  (test 5))
(test 6)
(let ((a 20)) (test 7))
(let ((b 30)) (test 8))
```

The first call `(test 5)` produces "Sum is 15". The second `(test 6)` and fourth `(test 8)` calls produce an error. The third call `(test 7)` produces "Sum is 27".

2.5 Operations

A limited number of operations are defined. Note that this list will probably be expanded.

2.5.1 Normal Forms vs Special Forms

A number of normal forms are defined. The main difference between normal forms and special forms is that all active arguments for a normal form are evaluated. Thus:

```
(* (+ 1 2) (+ 3 4))
;
; Versus
;
(if (> 1 2) (+ 1 2) (+ 3 4))
```

"*" is a normal operation and both `(+ 1 2)` and `(+ 3 4)` are evaluated before "*" is evaluated. `If` is a special form so first `(> 1 2)` is evaluated, then depending on whether the result is `T` or *NIL*, either `(+ 1 2)` or `(+ 3 4)` is evaluated. For a simple example like this, it doesn't really matter, but if the operations have other effects, such as:

```
(if (> 1 2) (print "Greater") (print "Not greater"))
```

will only print "Not greater".

2.5.2 Arithmetic Operations

Four arithmetic operations are defined for operation on integers. The operations are addition, subtraction, multiplication, and division. For example:

```
(+ 1 2 3 4)
(- 1 2 3 4)
(* 1 2 3 4)
(/ 1 2 3 4)
```

These operations work on a list of one or more parameters, with the operation inserted between the parameters. Thus `(+ 1 2 3 4)` computes as $1+2+3+4$. The return value for each of these operations is an integer value.

Note that division by zero is not checked. If this occurs, an Ada exception will be thrown. In some cases, this might be useful.

2.5.3 Boolean Operations

Three basic boolean operations are provided. These work on either boolean or integer variables.

```
(not NIL)
(and 1 5 7)
(or 1 2 4)
```

The `not` operation operates on a single parameter. If the parameter is boolean, the return value is the inverse of the parameter (*NIL* $\rightarrow$ *T*, *T* $\rightarrow$ *NIL*). If the parameter is integer, the individual bits of the integer are inverted and the resulting value returned.

The `and` and `or` operations operate on either booleans or integers as long as they are not mixed. These perform the logical `and` and `or` operations. Both of these operations short circuit. As soon as the result is *T* or -1 or `or` or *NIL* or 0 for `and`, processing additional parameters will not change the result so evaluation of parameters stops and the result is returned.

2.5.4 Character Operations

The normal comparison operations work on characters. There are also some operations defined to operate on characters.

```
(char-downcase #\A)
(char-code #\B)
(char-upcase #\c)
(code-char 65)
```

The `char-code` and `code-char` operations convert between characters and their integer codes. Given an integer in the range 0-255, `code-char` returns the corresponding character value. Given a character value, the function `char-code` returns the corresponding integer (usually the ASCII code.

The `char-downcase` and `char-upcase` operations convert characters between upper and lower case. Non-alphabetic characters are not changed.

2.5.5 Comparison Operations

Four comparison operations are defined for integers, strings, and booleans Note that this is different from Common Lisp which has separate operations defined for different types. The operations are equals, not equals, greater than, and less than. Equality and not equality is also defined for quoted symbols. For example:

```
(= 1 2)
(/= 1 2)
(< 1 2)
(> 1 2)
```

These operations work on two parameters of the same type. The return value of each of these operations is a boolean.

2.5.6 Control Flow

A couple of control flow special forms are available. More will probably be added.

```
(if (> 1 2) (print "True") (print "False"))
(dowhile (> 1 2) (print "Forever") (terpri))
(dotimes (n 5 10) (print "This is printed 5 times") (terpri))
```

The `if` form has two or three parameters. The first parameter is a condition. If the condition evaluates to `T`, then the second parameter is evaluated. If the condition evaluates to `NIL`, then the third parameter, if present, is evaluated.

The `dowhile` form has two parameters. The first is a condition. The second is a list of operations to be evaluated. The second parameter is evaluated as long as the condition evaluates to *T*.

The `dotimes` form also has two parameters. The first is a list with two or three elements. The first element is the name of the local variable used as a loop counter. The second element is a positive integer giving the number of times to loop. The third is a value to return at the end of the loop. If the return value is not provided, *NIL* is returned. The second parameter is a list of operations to be evaluated.

2.5.7 Debugging

Some additional operations are provided for debugging purposes. These control the display of some debugging information.

```
(dump)
(msg T)
(msg NIL)
```

The `dump` operation prints the contents of the cons, symbol, and string tables. The `msg` operation turn the display of debugging information on and off. These are helpful when trying to debug the interpreter and should not be necessary during normal operation.

2.5.8 Functions

```
(defun fib (n)
  (if (< n 2)
    1
    (+ (fib (- n 2)) (fib (- n 1)))))
(lambda (a b) (+ a b))
```

the `defun` form is used to create a user defined function. The first parameter is a symbol for the function name. The second parameter is a list of the parameters for the function. If the function has no parameters, the empty list "()" is used. Following this is a list of statements for the function. The function returns the value from the last statement to return a value.

The `lambda` form returns a user defined function. This function can be assigned to a variable or passed as a parameter to another function.

2.5.9 Input/Output

As this Lisp may run on systems without filesystems, only a few operations are provided for input and output. These are:

```
(print "Strings " 1 2 N)
(fresh-line)
(read-line)
(terpri)
```

The `print` form prints the list of objects to the standard output. No newline is added to the end. It returns *NIL_ELEM*.

The `fresh-line` prints a newline to the standard output if the output is not already at the start of a line. It returns *NIL_ELEM*.

The `read-line` reads a line of text from the standard input, terminated by a newline. It returns the text as a string without the newline.

The `terpri` prints a newline to the standard output. It returns *NIL_ELEM*.

2.5.10 List Operations

Basic list operations are provided.

```
(car (1 2 3 4))
(cdr (1 2 3 4))
(cons 1 (cons 2 ()))
(quote (+ 1 2) 3 4 (* 5 6 7 8))
(list (+ 1 2) 3 4 (* 5 6 7 8))
```

Each of `car` and `cdr` take one parameter that should be a list. `Car` returns the first item in the list. This item may be a single element or it may be a list. `Cdr` returns the remainder of the list.

The `cons` operation creates a *cons* cell and sets the *car* field to the first parameter and the *cdr* to the second parameter. This exposes a subtle difference between Tiny-Lisp and Common Lisp. In Tiny-Lisp, *NIL* is a constant of boolean type, while in Common Lisp, it also represents an empty list. Thus `(cons 1 NIL)` produce slightly different results, `(1 . NIL)` for Tiny-Lisp or `(1)` for Common Lisp. If you wish to produce the Common Lisp results, where the *car* points to a value and the *cdr* is an empty pointer, you can use `(cons 1 ())` or `(cons 1)`. The former is preferred as it is compatible with Common Lisp.

The `list` operation returns its parameters as a list after evaluating each of them. This is similar to `quote` except that quote does not evaluate the parameters. Thus `(quote (+ 1 2) 3 4 (* 5 6 7 8))` returns `((+ 1 2) 3 4 (* 5 6 7 8))`, while `(list (+ 1 2) 3 4 (* 5 6 7 8))` returns `(3 3 4 1680)`.

The `quote` operation returns its parameters as a list without evaluating any of them. In many cases this is not needed since if the first item in a list is not a symbol representing an operation or user defined function, the list simply evaluates to itself. At some point, this may change to be more compatible with Common Lisp.

2.5.11 Memory Access

Here be dragons! Use at your own risk. These operations are intended for use on embedded systems to access memory mapped peripheral devices. Access to a memory map is essential so that you know which locations to access.

```
(peek8 #x400E0940)
(peek16 #x400E0940)
(peek32 #x400E0940)
(poke8 #x100 5)
(poke16 #x110 10)
(poke32 #x1000 32)
```

The `peek` operations read 8, 16, or 32 bits from the specified memory location. Depending on the hardware, there may be memory alignment requirements, or certain operations will only work on some addresses. For example, the bytes of the Chip ID (CHIPID_CIDR) register on the SAM3X8E works using `peek8`, but hangs when using `peek16` or `peek32`. The returned value is the contents of memory at the specified location.

The `poke` operations write a 8, 16, or 32 bit value to the specified memory location. This is even more dangerous that the `peek` operations. ***You have been warned!*** The return value is the value written to the memory location.

2.5.12 Predicates

A wide variety of predicates are provided. These mostly match the ones in Common Lisp. Note that some of these will always return *`NIL`* due to missing features. There may also be some differences in corner cases due to implementation differences between Common Lisp and Tiny-Lisp.

```
;
;  The following will always return NIL as the data types or features
;   are not implemented.
;
(arrayp (1 3 5))
(bit-vector-p (1 2 3))
(complexp +)
(floatp 3)
(vectorp (1 2 3)
(rationalp "Hello")
(realp 4)
```

```
(simple-vector-p print)
(simple-bit-vector-p #x0F0F0F0F)
(packagep "package")
(vectorp (1 2 3)
;
;  The following will return NIL or T depending on the parameter.
;
(atomp 1)
(characterp #\A)
(compiled-function-p print)
(consp (1 2 3))
(errorp (+ 1 "A"))
(functionp functionp)
(integerp 3)
(listp (2 4 6))
(numberp 4)
(null ())
(simple-string-p "Hello")
(stringp "Hello")
(symbolp car)
```

Some corner cases to watch out for are:

1. Tiny-Lisp does not treat `()` and `NIL` exactly the same so `nullp` may not always do what it does in Common Lisp.

2. Tiny-Lisp does not have arrays or vectors. Strings are managed as linked lists in a separate allocation pool. Thus `stringp` and `simple-string-p` are treated the same and return `T` for any string and `NIL` for anything else.

3. Some of these operations evaluate the parameter to get a value to check and some do not. It's best not to get too creative with them.

2.5.13 String Related Operations

These operations are related to strings, but may have wider scope.

```
(length "Hello, this is a test")
(length (list 1 2 3 4 5))
(char "This is a test string" 5)
(parse-integer "42")
(string-downcase "HELLO")
(string-upcase "hello")
(subseq "This is a test of a subsequence" 5 10)
```

The `length` operation works on all types. For strings, it returns the number of characters in the string. For lists, it returns the number of elements in a list. For integers, characters, and booleans, it returns 1. For an empty list, it returns 0.

The `char` operation returns a specific character in a string, where the first character is character number 0.

The `parse-integer` operation parses the passed string as an integer. Positive and negative decimal integers are supported. Parsing ends when a non-decimal character is encountered.

The `string-downcase` and `string-upcase` operations make a copy of the passed string and convert it to all upper or all lower case. The original string is unchanged.

The `subseq` operation returns a substring of the original string. The first parameter is the string. The second parameter is the starting character (0 based). The third parameter is optional. If present, it is the index (not length of the substring) of the first character not part of the substring. If absent, the substring extends to the end of the original string.

2.5.14 Symbol Related Operations

Some operations use a quoted symbol to indicate what type of operation should be performed or what type of date should be returned. These thus require a bit more description than some of the other operations.

```
(coerce t 'integer)
(concatenate 'string "First string, " "second string, "
   "and finally the third string.")
```

The `coerce` operation is used to convert data of one type to another. The current supported conversions are:

- Boolean → Integer
- Boolean → String
- Character → String
- Integer → Boolean

Converting a type to itself is supported, but probably isn't very useful. Also of note is that coercing a string to a string returns a string object that points to the original string data structure, not a copy.

The `concatenate` operation works on both strings and lists. It constructs a new list or string that is the concatenation of the parameters. Note that in the case of a list, elements that are lists or strings are not copied. Only the references are copied.

2.5.15 Variables

Both global and local variables are supported.

```
(setq variable 1)
(let (var1 (var2 2) (var3))
  (print "var1 is " var1 " var2 is " var2 "var3 is " var3)
  (terpri))
```

The `setq` form sets a value for a symbol or stack variable. If a symbol and an active stack variable have the same name, the stack variable will be used. The first parameter is the symbol

and the second parameter is the value. If the symbol does not yet exist, it is created. Symbols that already exist as builtin or special can't be used for values. The second parameter is evaluated to return the value.

The `let` form creates local variables on the stack and an environment for other statements that use them. Variables can have an optional initial value. If no initial value is provided, the variable is set to *NIL*. The value returned from the `let` form is the value of the last statement executed.

2.5.16 Error Handling

In cases where the interpreter detects an error, the current operation returns an element of type *E_ERROR*. Currently, the only thing that can be done with this is to check if it is present using `errorp`. It is expected that this will eventually be expanded to include error codes that can help identify what sort of error occurred.

2.5.17 Other

There are a few operations that do things that can't be easily categorized.

```
(exit)
(sleep 1000)
```

The `exit` operation just exits the interpreter. It should mainly be used from the command line. It may cause problems in some cases if used in a function.

The `sleep` operation suspends program execution for the specified number of milliseconds. This is different from Common Lisp, where the parameter is a float in units of seconds. Since Tiny-Lisp is integer only, this doesn't work well, thus the difference.

2.6 Data Types

A limited selection of data types is provided. Think of the old Applesoft Integer BASIC.

2.6.1 Integer

This is a 32 bit signed integer. Integer literals can be given as either signed decimal integers, with a minus sign, "-", indicating negative numbers. This is just as one would expect, however don't use a plus sign, "+", to indicate positive numbers. Integers can also be expressed as unsigned hexadecimal numbers by preceding the number by "#x".

2.6.2 Characters

Character literals are introduced by preceding the literal by "#\". The following character is the character used, with some exceptions. The end of a line is always the end of a line, so this cannot be used to create a character containing a newline. If the first character is alphabetic and is followed by further alphabetic characters, it is interpreted as a character name. The defined character names are:

- Space

- Newline
- Tab
- Page
- Rubout
- Linefeed
- Return
- Backspace

Thus, the correct way to create a character containing a newline is "#\newline". Note that the character names are case insensitive.

2.6.3 String

Strings are stored in linked lists of 8-bit characters/bytes. Each node in the list can hold 16 (adjustable by a parameter) bytes. Unicode is not currently supported.

2.6.4 Boolean

The Boolean values *NIL* and *T* correspond to *True* and *False*. An empty list "()" is also interpreted as *NIL*.

2.6.5 List

The list is the basic complex data type. A list element has two slots (historically called *car* and *cdr*). Typically the *car* slot contains a data value and the *cdr* slot contains a pointer to the next list element. The end of a list is indicated by a *NIL* value in the *cdr* slot.

2.6.6 Error

There is currently only one possible `error` value. This is used to signal that some sort of error has occurred. It is expected that this will eventually be expanded to include a code to help identify what sort of error occurred.

Chapter 3

Operation Reference

This is an alphabetical list of all the operations.

3.1 Template

This is the template for each operation.

3.1.1 Inputs

The inputs are listed here.

3.1.2 Output

Any output is listed here.

3.1.3 Example

An example of the operation is listed here.

3.1.4 Description

This describes the operation. In many cases, it will be fairly simple.

3.1.5 Common Lisp Compatibility

This subsection discusses compatibility with Common Lisp. Usually, this will be a subset of Common Lisp. In some cases, it may be a superset. For example the comparison operators work on more types than Common Lisp supports.

3.2 +

Addition

3.2.1 Inputs

Any number of `integers`.

3.2.2 Output

An `integer` representing the sum of the inputs.

3.2.3 Example

(+ 1 2 3)

Returns the value 6.

3.2.4 Description

This operation adds a series of `integers`. Note that there is a possibility for integer overflow.

3.2.5 Common Lisp Compatibility

This is a subset of Common Lisp in that it only works on `integers`.

3.3 -

Subtraction

3.3.1 Inputs

Any number of `integers`.

3.3.2 Output

An `integer` representing the difference of the inputs. Note that there is a possibility for integer overflow.

3.3.3 Example

(− 1 2 3)

Returns the value -4.

3.3.4 Description

This operation subtracts a series of `integers`. This is done by starting with the first value, then subtracting the second value (if any). The next value is subtracted from the result.

3.3.5 Common Lisp Compatibility

This is a subset of Common Lisp in that it only works on `integers`.

3.4 *

Multiplication

3.4.1 Inputs

Any number of `integers`.

3.4.2 Output

An `integer` representing the product of the inputs. Note that there is a possibility for integer overflow.

3.4.3 Example

```
(* 1 2 3)
```

Returns the value 6.

3.4.4 Description

This operation multiplies a series of `integers`.

3.4.5 Common Lisp Compatibility

This is a subset of Common Lisp in that it only works on `integers`.

3.5 /

Division

3.5.1 Inputs

Any number of `integers`.

3.5.2 Output

An `integer` representing the quotient of the inputs. Division by zero is not checked and will cause an exception.

3.5.3 Example

```
(/ 1 2 3)
```

Returns the value 0.

3.5.4 Description

This operation divides a series of `integers`. This is done by starting with the first value, then dividing by the second value (if any). The result is then divided by the next value, and so on.

3.5.5 Common Lisp Compatibility

This is a subset of Common Lisp in that it only works on `integers`.

3.6 =

Equals

3.6.1 Inputs

Compares two values of the same type.

3.6.2 Output

`T` if the values are equal, otherwise *`NIL`*.

3.6.3 Example

```
(= 1 2)
```

Returns the value *`NIL`*.

3.6.4 Description

This operation compares two values of the same type for equality.

3.6.5 Common Lisp Compatibility

This operation works on `integers`, `booleans`, `strings`, and quoted `symbols`.

3.7 /=

Not-equals

3.7.1 Inputs

Compares two values of the same type.

3.7.2 Output

`T` if the values are not equal, otherwise *`NIL`*.

3.7.3 Example

(= 1 2)

Returns the value *NIL*.

3.7.4 Description

This operation compares two values of the same type for not equality.

3.7.5 Common Lisp Compatibility

This operation works on `integers`, `booleans`, `strings`, and quoted `symbols`.

3.8 <

Less Than

3.8.1 Inputs

Compares two values of the same type.

3.8.2 Output

T if the first value is less than the second value, otherwise *NIL*.

3.8.3 Example

(< 1 2)

Returns the value *T*.

3.8.4 Description

This operation compares two values of the same type for less than.

3.8.5 Common Lisp Compatibility

This operation works on `integers`, `booleans`, and `strings`.

3.9 >

Greater Than

3.9.1 Inputs

Compares two values of the same type.

3.9.2 Output

T if the first value is greater than the second value, otherwise *NIL*.

3.9.3 Example

(> 1 2)

Returns the value *NIL*.

3.9.4 Description

This operation compares two values of the same type for greater than.

3.9.5 Common Lisp Compatibility

This operation works on `integers`, `booleans`, and `strings`.

3.10 and

Logical or bitwise `and`.

3.10.1 Inputs

Performs the logical or bitwise `and` on values of the same type.

3.10.2 Output

If the input parameters are `boolean` then the output is `boolean`. If the input parameters are *integer*, the output is `integer`.

3.10.3 Example

(**and** 1 3 4)

Returns the value 1.

3.10.4 Description

If the two parameters are `boolean`, the result is the logical and of the parameters. If the two parameters are `integer`, then the result is the bitwise and of the parameters. Processing of parameters stop when the result is either *NIL* of `boolean` values, or 0 (zero) for `integer` values.

3.10.5 Common Lisp Compatibility

This operation performs a bitwise `and` for integers. This is probably more useful for embedded systems.

3.11 arrayp

Is parameter an `array`?

3.11.1 Inputs

A single value. Any additional values are ignored..

3.11.2 Output

NIL.

3.11.3 Example

(**arrayp** 1 2 3)

Returns the value *NIL*.

3.11.4 Description

Since `arrays` are not a supported datatype, this always returns *NIL*.

3.11.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.12 atomp

Is parameter an atom?

3.12.1 Inputs

A single value. Any additional values are ignored..

3.12.2 Output

T or *NIL*

3.12.3 Example

(**atomp** 1 2 3)

Returns the value *T*.

3.12.4 Description

Returns *T* if the first value is an atom. Returns *NIL* otherwise. Since the only non-atom datatype supported is a list, this really just checks if the value is a list.

3.12.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.13 bit-vector-p

Is parameter a `bit vector`?

3.13.1 Inputs

A single value. Any additional values are ignored.

3.13.2 Output

NIL.

3.13.3 Example

```
(bit-vector-p 1 2 3)
```

Returns the value *NIL*.

3.13.4 Description

Since `bit vectors` are not a supported datatype, this always returns *NIL*.

3.13.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.14 car

Returns the first element of a list

3.14.1 Inputs

If the first value is a list, return the first value of that. Otherwise return the first value.

3.14.2 Output

The first value of a list.

3.14.3 Example

```
(car 1 2 3)
```

Returns the value 1.

3.14.4 Description

Returns the first value of a list. If the first value passed is a list, then return the first value of that. Otherwise the list of parameters is treated as a list and the first value is returned.

3.14.5 Common Lisp Compatibility

If multiple parameters are passed, the first one is returned. Compatible with Common Lisp if only one parameter is passed.

3.15 cdr

Returns all but the first element of a list

3.15.1 Inputs

If the first value is a list, return all but the first value of that. Otherwise return all but the first value.

3.15.2 Output

All but he first value of a list.

3.15.3 Example

```
(cdr 1 2 3)
```

Returns the value (2 3).

3.15.4 Description

Returns all but the first value of a list. If the first value passed is a list, then return all but the first value of that. Otherwise the list of parameters is treated as a list and all but the first value is returned.

3.15.5 Common Lisp Compatibility

If multiple parameters are passed, all but the first one is returned. Compatible with Common Lisp if only one parameter is passed.

3.16 char

Returns a specified character in a string.

3.16.1 Inputs

A string and an integer.

3.16.2 Output

A character.

3.16.3 Example

```
(char "This is a string" 5)
```

Returns the character "i".

3.16.4 Description

Returns the specified character in a string where the first character is number 0.

3.16.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.17 char-code

Returns the integer ASCII value of a character.

3.17.1 Inputs

A character.

3.17.2 Output

A character.

3.17.3 Example

```
(char-code #\A)
```

Returns the integer 65.

3.17.4 Description

This returns the integer ASCII (you might be able to find some odd systems where this is not true) code for the provided character. Unicode is not currently supported.

3.17.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.18 char-downcase

Converts a character to lower case.

3.18.1 Inputs

A character.

3.18.2 Output

A character.

3.18.3 Example

(**char−downcase** #\A)

Returns the character "a".

3.18.4 Description

If the character passed is uppercase, convert it to lowercase and return it. Otherwise return the character unchanged.

3.18.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.19 char-upcase

Converts a character to upper case.

3.19.1 Inputs

A character.

3.19.2 Output

A character.

3.19.3 Example

(**char−upcase** #\a)

Returns the character "A".

3.19.4 Description

If the character passed is lowercase, convert it to uppercase and return it. Otherwise return the character unchanged.

3.19.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.20 characterp

Is parameter a `character`?

3.20.1 Inputs

A single value. Any additional values are ignored.

3.20.2 Output

A boolean value

3.20.3 Example

```
(characterp 1)
```

Returns the value *NIL*.

3.20.4 Description

Returns *T* if the first value is a character. Otherwise it returns *NIL*. Note that a string containing a single character is not the same as a character.

3.20.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.21 code-char

Converts an integer to a character where the integer is the ASCII representation of the character. The integer is limited to the range 0-255.

3.21.1 Inputs

A single integer. Any other parameters are ignored.

3.21.2 Output

The character represented by the ASCII code input.

3.21.3 Example

```
(code-char 65)
```

Returns the character 'A'.

3.21.4 Description

This can be used to generate any 8 bit ASCII character.

3.21.5 Common Lisp Compatibility

Common Lisp allows a larger range than 0-255 since Unicode is supported.

3.22 coerce

Converts a value of one type to another type

3.22.1 Inputs

Two values. The first value is the item to be converted. The second value is a quoted symbol representing the result type.

3.22.2 Output

A value of the desired type.

3.22.3 Example

```
(coerce NIL 'integer)
```

Returns the integer value 0.

3.22.4 Description

The result is a representation of the first value in the desired type.

3.22.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

Only the following coercions are supported:

- character → string
- boolean → string
- boolean → integer (*NIL* → 0, *T* → 1).
- integer → boolean (0 → *NIL*, ≠ 0 → *T*)

3.23 compiled-function-p

Is parameter a compiled function?

3.23.1 Inputs

A single value. Any additional values are ignored.

3.23.2 Output

A boolean value

3.23.3 Example

(**compiled−function−p print**)

Returns the value *T*.

3.23.4 Description

Returns *T* if the first value is a compiled function. Otherwise it returns *NIL*. Tiny-Lisp() considers the builtin intrinsic functions to be compiled. User defined functions are not compiled..

3.23.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.24 complexp

Is parameter a `complex number`?

3.24.1 Inputs

A single value. Any additional values are ignored.

3.24.2 Output

NIL.

3.24.3 Example

(**complexp** 1 2 3)

Returns the value *NIL*.

3.24.4 Description

Since `complex numbers` are not a supported datatype, this always returns *NIL*.

3.24.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.25 concatenate

Concatenates strings or lists.

3.25.1 Inputs

A quoted symbol (either *LIST* or *STRING*) followed by either lists or strings.

3.25.2 Output

A list or string consisting of the concatenation of the lists or strings

3.25.3 Example

```
(concatenate 'string "One " "Two")
```

Returns the string "One Two".

3.25.4 Description

Concatenates either string or lists. The first parameter is a symbol that specifies what to concatenate. The following parameters must be of the appropriate type.

3.25.5 Common Lisp Compatibility

This is probably a subset of the Common Lisp function. Normal cases will operate the same, but error handling is different.

3.26 cons

Combines elements into a list.

3.26.1 Inputs

One or two values

3.26.2 Output

A list consisting of the provided inputs.

3.26.3 Example

(**cons** 1 2)

Returns the list (1 . 2).

3.26.4 Description

The `cons` operation creates a *`cons`* cell and sets the *`car`* field to the first parameter and the *`cdr`* to the second parameter..

3.26.5 Common Lisp Compatibility

There is a subtle difference between Tiny-Lisp and Common Lisp. In Tiny-Lisp, *`NIL`* is a constant of boolean type, while in Common Lisp, it also represents an empty list. Thus `(cons 1 NIL)` produce slightly different results, `(1 . NIL)` for Tiny-Lisp or `(1)` for Common Lisp. If you wish to produce the Common Lisp results, where the *`car`* points to a value and the *`cdr`* is an empty pointer, you can use `(cons 1 ())` or `(cons 1)`. The former is preferred as it is compatible with Common Lisp.

3.27 consp

Is parameter a `cons`?

3.27.1 Inputs

A single value. Any additional values are ignored.

3.27.2 Output

A boolean.

3.27.3 Example

(**consp** (1 2 3))

Returns the value *`T`*.

3.27.4 Description

If the supplied parameter is a `cons` (a list), return *`T`*, otherwise return *`NIL`*.

3.27.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.28 defun

Defines a function.

3.28.1 Inputs

Three or more values. The first is a symbol that becomes the function's name. The second is a list of parameters for the function. The remaining values are the code for the function

3.28.2 Output

NIL.

3.28.3 Example

```
(defun hello (name) (print "Hello " name))
```

Returns the value *NIL*.

3.28.4 Description

This creates a user defined function.

3.28.5 Common Lisp Compatibility

This is more or less a subset of Common Lisp, except that there are probably many corner cases where things don't quite match..

3.29 dowhile

Repeats a series of statements while a condition is *T*.

3.29.1 Inputs

Two or more values. The first value is evaluated as the condition, the remaining values are the statements to be executed while the condition is true.

3.29.2 Output

A value.

3.29.3 Example

```
(dowhile (> (- max min) 1)
  (setq mid (/ (+ min max) 2))
  (if (> mid (/ n mid))
    (setq max mid)
    (setq min mid))
```

```
(+ 0 min))
```

Returns the value of the last statement in the loop, `min`

3.29.4 Description

The condition is evaluated on each pass through the loop. If the condition evaluates to *T*, the rest of the statements are executed. If the condition evaluates to *NIL*, the loop is exited. Thus, if the first time the condition is evaluated it returns *NIL*, the statements in the loop are never executed.

3.29.5 Common Lisp Compatibility

This doesn't appear to exist in Common Lisp. It is similar to the Common Lisp `do` loop, except that the condition comes first.

3.30 dotimes

Repeats a series of statements a specific number of times.

3.30.1 Inputs

Two or more values. The first value is a list containing a local variable and the numb, the remaining values are the statements to be executed the specified number of times.

3.30.2 Output

A constant *NIL*.

3.30.3 Example

```
(setq sum 0)
(dotimes (x 10)
  (print "The sum is " sum)
  (terpri)
  (setq sum (+ sum x)))
```

Returns the constant *NIL*.

3.30.4 Description

On each pass through the loop, the local variable is set to the next value in the range 0 to the loop limit. The supplied statements are evaluated. It is not recommended to change the value of the local variable in the body of the loop.

3.30.5 Common Lisp Compatibility

The result-form, declarations, and tags are not supported.

3.31 dump

Prints out some internal tables.

3.31.1 Inputs

None.

3.31.2 Output

A value.

3.31.3 Example

(**dump**)

Returns the constant *NIL*.

3.31.4 Description

This is intended for debugging purposes. It prints the contents of the cons, symbol, and string tables.

3.31.5 Common Lisp Compatibility

This operation does not exist in Common Lisp.

3.32 errorp

Is parameter a `error`?

3.32.1 Inputs

A single value. Any additional values are ignored.

3.32.2 Output

A boolean.

3.32.3 Example

```
(error (1 2 3))
```

Returns the value *NIL*.

3.32.4 Description

If the parameter represents an error condition, return *T*, otherwise return *NIL*. This offers Tiny-Lisp programs a rudimentary way to check for errors.

3.32.5 Common Lisp Compatibility

This operation does not exist in Common Lisp. Common Lisp provides more comprehensive error handling, signaling, and trapping.

3.33 exit

Exits the Tiny-Lisp interpreter.

3.33.1 Inputs

None.

3.33.2 Output

A value.

3.33.3 Example

(**exit**)

No value can be returned as the interpreter exits..

3.33.4 Description

This is a way to exit the Tiny-Lisp iterpreter.

3.33.5 Common Lisp Compatibility

This operation does not exist in Common Lisp, but some implementations (i.e. SBCL) do have it.

3.34 floatp

Is parameter a `floating point number`?

3.34.1 Inputs

A single value. Any additional values are ignored.

3.34.2 Output

`NIL`.

3.34.3 Example

```
(floatp 1 2 3)
```

Returns the value *NIL*.

3.34.4 Description

Since `floating point numbers` are not a supported datatype, this always returns *NIL*.

3.34.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.35 fresh-line

Prints a newline if not already at the start of a line.

3.35.1 Inputs

None, any values are ignored.

3.35.2 Output

NIL.

3.35.3 Example

```
(fresh-line)
```

Returns the value *NIL*.

3.35.4 Description

Checks if the internal first character flag is set. If not, prints a newline, otherwise does nothing.

3.35.5 Common Lisp Compatibility

There is no optional output-stream parameter as Tiny-Lisp only has one output stream. It also always returns *NIL*.

3.36 functionp

Is parameter a function?

3.36.1 Inputs

A single value. Any additional values are ignored.

3.36.2 Output

NIL.

3.36.3 Example

```
(functionp 1 2 3)
```

Returns the value *NIL*.

3.36.4 Description

Returns *T* if the value is a builtin function, a user defined function, or a lambda function.

3.36.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.37 if

Conditionally executes a statement.

3.37.1 Inputs

Two or three parameters. The first is a condition. If the condition evaluates to *T*, the second parameter is evaluated. If the condition evaluates to *NIL*, the third parameter, if present, is evaluated.

3.37.2 Output

Returns the value of the parameter evaluated.

3.37.3 Example

```
(if (> 1 2)
  (print "Greater")
  (print "Not greater"))
```

Prints the string "Not greater".

3.37.4 Description

Evaluates the condition and then depending on the condition, evaluates one of the other parameters. If the third parameter is omitted, this is approximately equivalent to an `IF-THEN` statement in other languages. If the third parameter is present, this is similar to an `IF-THEN-ELSE` statement. The value of the evaluated parameter is returned. If no parameter is evaluated (only two parameters passed and the condition evaluates to *NIL*), then *NIL* is returned.

3.37.5 Common Lisp Compatibility

This seems to be mostly compatible with Common Lisp.

3.38 integerp

Is the parameter an integer?

3.38.1 Inputs

A single value. Any additional values are ignored.

3.38.2 Output

The value *T* if the parameter is an integer, otherwise *NIL*.

3.38.3 Example

```
(integerp 1 2 3)
```

Returns the value *T*.

3.38.4 Description

This is used to check if a parameter is of integer type or not. Currently the only number type supported is integer so this is equivalent to `numberp` in Tiny-Lisp.

3.38.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.39 lambda

Creates a variable record that can be assigned to a variable or passed as a parameter to a function.

3.39.1 Inputs

The first parameter is a list of the parameters for the function. The remaining parameters are the operation for the function.

3.39.2 Output

A variable record pointing to the created function.

3.39.3 Example

```
(setq hello (lambda (name) (print "Hello " name)))
```

Sets the symbol (or variable depending on context) `hello` to point to the created function.

3.39.4 Description

Creates a function that can be assigned to a variable or passed as a parameter. Note that `(setq var (lambda ...))` is slightly different from `(defun var ...)`. The first creates a variable record and assigns it to `var` while the second directly sets `var` to point to the function. This difference may be removed in future versions.

3.39.5 Common Lisp Compatibility

This is mostly a subset of Common Lisp.

3.40 length

Returns the length of an object.

3.40.1 Inputs

The first parameter is a list of the object to measure.

3.40.2 Output

An integer.

3.40.3 Example

```
(length "Hello")
```

Returns the value 5.

3.40.4 Description

Returns the length of an object. For `strings` this is the number of characters in a string. For `lists` this is the number of items in a list not descending into sublists. All other datatypes return a value of 1.

3.40.5 Common Lisp Compatibility

This is mostly compatible with Common Lisp, except that errors are not thrown if the parameter is not a sequence.

3.41 let

Creates local variables.

3.41.1 Inputs

A list of variable names and optional initial values followed by statements to be executed with the local variables.

3.41.2 Output

Returns the value of the last statement evaluated.

3.41.3 Example

```
(defun fibi (n)
  (let (temp (n1 0) (n2 1))
    (dotimes (iter n)
      (setq temp (+ n1 n2))
      (setq n1 n2)
      (setq n2 temp))
    n2))
```

Defines a function to evaluate Fibonacci numbers using iteration. The variables `temp`, `n1`, and `n2` are local variables with `n1` being initialized to the value 0 and `n2` initialized to the value 1.

3.41.4 Description

Creates a stack frame containing variables that are local to the statements in the block. Outside of the block the variables do not exist.

3.41.5 Common Lisp Compatibility

Closures are not supported.

3.42 list

Creates a list

3.42.1 Inputs

Any number of parameters.

3.42.2 Output

A list.

3.42.3 Example

```
(list 1 2 3)
```

Returns the list `(1 2 3)`

3.42.4 Description

Creates a list of the passed parameters.

3.42.5 Common Lisp Compatibility

Unlike Common Lisp, the `list` operation is optional in Tiny-Lisp.

3.43 listp

Is the parameter a list?

3.43.1 Inputs

A single value. Any additional values are ignored.

3.43.2 Output

The value *`T`* if the parameter is a list, otherwise *`NIL`*.

3.43.3 Example

```
(listp 1 2 3)
```

Returns the value *`NIL`*.

3.43.4 Description

This is used to check if a parameter is a list or not.

3.43.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.44 msg

Turns display of debugging messages on or off.

3.44.1 Inputs

A single boolean value. Any additional values are ignored.

3.44.2 Output

The value *`NIL`*.

3.44.3 Example

(**msg** T)

Returns the value *NIL*.

3.44.4 Description

This is intended for use in debugging the interpreter to turn the display of some debugging messages on or off.

3.44.5 Common Lisp Compatibility

This operation does not exist in Common Lisp.

3.45 not

Logical or bitwise `not`.

3.45.1 Inputs

A single value of `boolean` or `integer`.

3.45.2 Output

A value of `boolean` or `integer`.

3.45.3 Example

(**not** T)

Returns the value *NIL*.

3.45.4 Description

If the parameter is `boolean`, perform a logical `not` operation. If the parameter is `integer`, perform a bitwise `not` operation.

3.45.5 Common Lisp Compatibility

This operation performs a bitwise `not` for integers. This is probably more useful for embedded systems.

3.46 null

Is the parameter null?

3.46.1 Inputs

A single value. Any additional values are ignored.

3.46.2 Output

The value *T* if the parameter is null, otherwise *NIL*.

3.46.3 Example

```
(null 1 2 3)
```

Returns the value *NIL*.

3.46.4 Description

This is used to check if a parameter is null or not. The empty list is considered to be null while an explicit *NIL* is not.

3.46.5 Common Lisp Compatibility

In Tiny-Lisp only the empty list () is treated as null, while Common Lisp also treats *NIL* as null. This may be changed in Tiny-Lisp to make it more compatible.

3.47 numberp

Is the parameter a number?

3.47.1 Inputs

A single value. Any additional values are ignored.

3.47.2 Output

The value *T* if the parameter is a number, otherwise *NIL*.

3.47.3 Example

```
(numberp 1 2 3)
```

Returns the value *T*.

3.47.4 Description

This is used to check if a parameter is of number type or not. Currently the only number type supported is integer so this is equivalent to `integerp` in Tiny-Lisp.

3.47.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.48 or

Logical or bitwise `or`.

3.48.1 Inputs

Performs the logical or bitwise `and` on values of the same type.

3.48.2 Output

If the input parameters are `boolean` then the output is `boolean`. If the input parameters are *`integer`*, the output is `integer`.

3.48.3 Example

(**or** 1 3 4)

Returns the value 7.

3.48.4 Description

If the two parameters are `boolean`, the result is the logical and of the parameters. If the two parameters are `integer`, then the result is the bitwise and of the parameters. Processing of parameters stop when the result is either *`NIL`* of `boolean` values, or 0 (zero) for `integer` values.

3.48.5 Common Lisp Compatibility

This operation performs a bitwise `or` for integers. This is probably more useful for embedded systems.

3.49 packagep

Is parameter a `package`?

3.49.1 Inputs

A single value. Any additional values are ignored.

3.49.2 Output

`NIL`.

3.49.3 Example

(**packagep** 1 2 3)

Returns the value *NIL*.

3.49.4 Description

Since `packages` are not a supported datatype, this always returns *NIL*.

3.49.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.50 parse-integer

Parses a `string` containing an integer as text to an `integer` value.

3.50.1 Inputs

A `string` containing an integer.

3.50.2 Output

An `integer` representing the value in the `string`.

3.50.3 Example

(**parse-integer** "1024")

Returns the `integer` 1024.

3.50.4 Description

This is used to get an integer value from a string containing the digits of an integer. To simplify the coding, only the first fragment of the string is examined for digits. Parameters not of `string` cause an `error` to be returned. Strings starting with non-integer values return 0. Parsing is terminated when a non digit character is encountered (thus the `string` "123abc" is parsed to the `integer` 123).

3.50.5 Common Lisp Compatibility

The Tiny-Lisp version is a subset of the Common Lisp version. None of the Common Lisp optional parameters are allowed. It operates similarly to having `:junk-allowed` set to *T*. Leading spaces are not allowed. A leading plus sign ('+') is not allowed. Only a single value is returned.

3.51 peek8

Reads an 8 bit byte from the specified address in memory.

3.51.1 Inputs

An `integer` representing the address to read from.

3.51.2 Output

An 8 bit `integer` representing the value at that address.

3.51.3 Example

```
(peek 1)
```

Returns the value at address 1. The actual value is system dependent.

3.51.4 Description

This is used to read memory locations. It is intended to be used with memory mapped devices to allow drivers to be developed using Tiny-Lisp. This should be used with caution as the results are strongly system dependent. No protection is provided by Tiny-Lisp to prevent attempting to read from protected or non-existent addresses.

3.51.5 Common Lisp Compatibility

This operation does not exist in Common Lisp.

3.52 peek16

Reads a 16 bit word from the specified address in memory.

3.52.1 Inputs

An `integer` representing the address to read from.

3.52.2 Output

A 16 bit `integer` representing the value at that address.

3.52.3 Example

```
(peek 1)
```

Returns the value at address 1. The actual value is system dependent.

3.52.4 Description

This is used to read memory locations. It is intended to be used with memory mapped devices to allow drivers to be developed using Tiny-Lisp. This should be used with caution as the results are strongly system dependent. No protection is provided by Tiny-Lisp to prevent attempting to read from protected or non-existent addresses. Some systems may also throw exceptions for misaligned access to some or all of the addresses.

3.52.5 Common Lisp Compatibility

This operation does not exist in Common Lisp.

3.53 peek32

Reads a 32 bit word from the specified address in memory.

3.53.1 Inputs

An `integer` representing the address to read from.

3.53.2 Output

A 32 bit `integer` representing the value at that address.

3.53.3 Example

```
(peek 1)
```

Returns the value at address 1. The actual value is system dependent.

3.53.4 Description

This is used to read memory locations. It is intended to be used with memory mapped devices to allow drivers to be developed using Tiny-Lisp. This should be used with caution as the results are strongly system dependent. No protection is provided by Tiny-Lisp to prevent attempting to read from protected or non-existent addresses. Some systems may also throw exceptions for misaligned access to some or all of the addresses.

3.53.5 Common Lisp Compatibility

This operation does not exist in Common Lisp.

3.54 poke8

Writes an 8 bit byte to the specified address in memory.

3.54.1 Inputs

Two `integer`s representing the address to write to and the value to write, respectively.

3.54.2 Output

The `integer` value written.

3.54.3 Example

```
(poke 1 2)
```

Returns the value 2. There may be other effects due to the memory being changed.

3.54.4 Description

This is used to write to memory locations. It is intended to be used with memory mapped devices to allow drivers to be developed using Tiny-Lisp. This should be used with caution as the results are strongly system dependent. No protection is provided by Tiny-Lisp to prevent attempting to write to protected or non-existent addresses. Some systems may also throw exceptions for misaligned access to some or all of the addresses.

3.54.5 Common Lisp Compatibility

This operation does not exist in Common Lisp.

3.55 poke16

Writes a 16 bit word to the specified address in memory.

3.55.1 Inputs

Two `integer`s representing the address to write to and the value to write, respectively.

3.55.2 Output

The `integer` value written.

3.55.3 Example

```
(poke 4 2)
```

Returns the value 2. There may be other effects due to the memory being changed.

3.55.4 Description

This is used to write to memory locations. It is intended to be used with memory mapped devices to allow drivers to be developed using Tiny-Lisp. This should be used with caution as the results are strongly system dependent. No protection is provided by Tiny-Lisp to prevent attempting to write

to protected or non-existent addresses. Some systems may also throw exceptions for misaligned access to some or all of the addresses.

3.55.5 Common Lisp Compatibility

This operation does not exist in Common Lisp.

3.56 poke32

Writes a 32 bit word to the specified address in memory.

3.56.1 Inputs

Two `integer`s representing the address to write to and the value to write, respectively.

3.56.2 Output

The `integer` value written.

3.56.3 Example

```
(poke 4 2)
```

Returns the value 2. There may be other effects due to the memory being changed.

3.56.4 Description

This is used to write to memory locations. It is intended to be used with memory mapped devices to allow drivers to be developed using Tiny-Lisp. This should be used with caution as the results are strongly system dependent. No protection is provided by Tiny-Lisp to prevent attempting to write to protected or non-existent addresses. Some systems may also throw exceptions for misaligned access to some or all of the addresses.

3.56.5 Common Lisp Compatibility

This operation does not exist in Common Lisp.

3.57 print

Prints objects.

3.57.1 Inputs

Any number of parameters.

3.57.2 Output

`NIL`.

3.57.3 Example

```
(print "Hello world!")
```

Returns the value *NIL*. "Hello world!" is sent to the output stream.

3.57.4 Description

This loops through the provided parameters and prints each one with no newline or space between and no trailing newline. Note that if a newline is contained in one of the items printed, the internal flag `first_char_flag` is not set. This may cause `fresh-line` to output an unneeded newline.

3.57.5 Common Lisp Compatibility

The output is not preceded by a newline and followed by a space. The optional `output-stream` parameter is not available as there is only one output stream. Multiple parameters are permitted. And, there is no implicit binding of parameters to values.

3.58 quote

Returns a list created from the supplied parameters.

3.58.1 Inputs

Any number of parameters.

3.58.2 Output

A list generated from the input parameters.

3.58.3 Example

```
(quote 1 2 3 4)
```

Returns the list (1 2 3 4).

3.58.4 Description

Returns a list generated from the passed parameters. Internally, this returns the index of the cons cell for the first parameter and the rest of the parameter list follows along in the linked list. The parameters are not evaluated. In many cases, this may not be needed in Tiny-Lisp as lists that do not start with a function parameter are simply returned as-is.

3.58.5 Common Lisp Compatibility

Mostly compatible with Common Lisp, except that multiple parameters are permitted.

3.59 rationalp

Is parameter a `rational number`?

3.59.1 Inputs

A single value. Any additional values are ignored.

3.59.2 Output

NIL.

3.59.3 Example

```
(rationalp 1 2 3)
```

Returns the value *NIL*.

3.59.4 Description

Since `rational numbers` are not a supported datatype, this always returns *NIL*.

3.59.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.60 read-line

Reads a line of text from the input stream.

3.60.1 Inputs

None.

3.60.2 Output

A `string` read from the input stream.

3.60.3 Example

```
(read-line)
```

Returns the text read from the input.

3.60.4 Description

Reads input into a `string` and returns the `string`. The newline that ends the string is not included in the string.

3.60.5 Common Lisp Compatibility

None of the Common Lisp optional parameters are supported.

3.61 realp

Is parameter a `real number`?

3.61.1 Inputs

A single value. Any additional values are ignored.

3.61.2 Output

NIL.

3.61.3 Example

```
(realp 1 2 3)
```

Returns the value *NIL*.

3.61.4 Description

Since `real numbers` are not a supported datatype, this always returns *NIL*.

3.61.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.62 setq

Assigns a value to a variable.

3.62.1 Inputs

Two parameters. The first is the variable to be set. If this is not a stack variable, it will be interpreted as a symbol. The second is the value to assign to the variable. It is evaluated.

3.62.2 Output

NIL

3.62.3 Example

```
(setq counter (+ 1 counter))
```

Returns the value *NIL* and increments the variable `counter`.

3.62.4 Description

This provides a way to assign values to symbols or stack variables. The previous value of the variable is lost. Note that symbols representing builtin or special functions cannot be assigned.

3.62.5 Common Lisp Compatibility

Only one variable can be set at a time. It returns *NIL*, not the value set.

3.63 simple-bit-vector-p

Is parameter a `simple bit vector`?

3.63.1 Inputs

A single value. Any additional values are ignored.

3.63.2 Output

NIL.

3.63.3 Example

```
(simple-bit-vector-p 1 2 3)
```

Returns the value *NIL*.

3.63.4 Description

Since `bit vectors` (simple or otherwise) are not a supported datatype, this always returns *NIL*.

3.63.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.64 simple-string-p

Is the parameter a simple string?

3.64.1 Inputs

A single value. Any additional values are ignored.

3.64.2 Output

The value *T* if the parameter is a string, otherwise *NIL*.

3.64.3 Example

(**simple-string-p** 1 2 3)

Returns the value *NIL*.

3.64.4 Description

This is used to check if a parameter is a simple string or not. All strings in Tiny-Lisp are considered to be simple strings.

3.64.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.65 simple-vector-p

Is parameter a `simple vector`?

3.65.1 Inputs

A single value. Any additional values are ignored.

3.65.2 Output

NIL.

3.65.3 Example

(**simple-vector-p** 1 2 3)

Returns the value *NIL*.

3.65.4 Description

Since `vectors` (simple or otherwise) are not a supported datatype, this always returns *NIL*.

3.65.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.66 sleep

Suspend execution for the specified number of milliseconds.

3.66.1 Inputs

A single `integer` value. Any additional values are ignored.

3.66.2 Output

NIL.

3.66.3 Example

```
(sleep 100)
```

Returns the value *NIL* after waiting 100 mS.

3.66.4 Description

Suspends execution for the specified number of milliseconds. Since Tiny-Lisp only supports integers, this difference from Common Lisp was done in order to allow finer resolutions in delay.

3.66.5 Common Lisp Compatibility

The delay value is in milliseconds, not seconds.

3.67 string-downcase

Converts a string to lowercase ASCII.

3.67.1 Inputs

A single `string` value. Any additional values are ignored.

3.67.2 Output

A `string`.

3.67.3 Example

```
(string-downcase "Hello World!")
```

Returns the `string` "hello world!".

3.67.4 Description

Creates a copy of the input `string` converting any uppercase characters to lowercase.

3.67.5 Common Lisp Compatibility

The optional `start` and `end` parameters are not supported. Only ASCII characters are supported and only the characters 'A' through 'Z' are converted.

3.68 string-upcase

Converts a string to uppercase ASCII.

3.68.1 Inputs

A single `string` value. Any additional values are ignored.

3.68.2 Output

A `string`.

3.68.3 Example

```
(string-upcase "Hello World!")
```

Returns the `string` "HELLO WORLD!".

3.68.4 Description

Creates a copy of the input `string` converting any lowercase characters to uppercase.

3.68.5 Common Lisp Compatibility

The optional `start` and `end` parameters are not supported. Only ASCII characters are supported and only the characters 'a' through 'z' are converted.

3.69 stringp

Is the parameter a string?

3.69.1 Inputs

A single value. Any additional values are ignored.

3.69.2 Output

The value *T* if the parameter is a string, otherwise *NIL*.

3.69.3 Example

```
(stringp 1 2 3)
```

Returns the value *NIL*.

3.69.4 Description

This is used to check if a parameter is a string or not..

3.69.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.70 subseq

Return a subsequence of the input.

3.70.1 Inputs

A single `string` followed by one or two `integers` representing the starting and (optional) ending positions.

3.70.2 Output

A `string` containing a sequence copied from the input.

3.70.3 Example

```
(subseq "Hello world!" 3 7)
```

Returns the `string` "lo w"..

3.70.4 Description

Copies the selected text from the input string and returns it.

3.70.5 Common Lisp Compatibility

In Tiny-Lisp, `subseq` only works on `strings`. At some point, it may be extended to also work on `lists`.

3.71 symbolp

Is the parameter a symbol?

3.71.1 Inputs

A single value. Any additional values are ignored.

3.71.2 Output

The value *`T`* if the parameter is a symbol, otherwise *`NIL`*.

3.71.3 Example

(**symbolp** 1 2 3)

Returns the value *NIL*.

3.71.4 Description

This is used to check if a parameter is a symbol or not.

3.71.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

3.72 terpri

Prints a newline.

3.72.1 Inputs

None

3.72.2 Output

A newline.

3.72.3 Example

(**terpri**)

Returns the value *NIL*.

3.72.4 Description

Prints a newline to the output stream.

3.72.5 Common Lisp Compatibility

Tiny-Lisp() has only one output stream so the optional output stream designator is ignored.

3.73 vectorp

Is parameter a `vector`?

3.73.1 Inputs

A single value. Any additional values are ignored.

3.73.2 Output

NIL.

3.73.3 Example

(**vectorp** 1 2 3)

Returns the value *NIL*.

3.73.4 Description

Since `vectors` are not a supported datatype, this always returns *NIL*.

3.73.5 Common Lisp Compatibility

This is compatible with Common Lisp, except that no error is thrown with extra parameters. They are just silently ignored.

Chapter 4

Internals

As the interpreter is under active development, this section is subject to change without notice.

4.1 Operation

Processing consists of four phases.

4.1.1 Read

Text is read from the input stream and passed to the parser.

4.1.2 Parse

The parse phase examines the text and converts it into the internal representation of a list. If the list is not complete (parentheses are unbalanced), more text is requested until the list can be completed. Parsing of sub-lists is done by recursively calling the parser.

4.1.3 Evaluate

The list from the parse phase is evaluated. If the first in the list is not a symbol that represents a function, the list is simply returned as is. Otherwise the function is evaluated and the returned value is passed to the print phase. Evaluation of sub-lists is done recursively. Note that, depending on the function, not all sub-lists are evaluated.

4.1.4 Print

The value returned from the evaluation is printed. Once this is done, the read phase is reentered and more text requested.

4.2 Package Organization

In order to help modularize and organize the code, it has been divided into several packages. The root package for Tiny-Lisp is `BBS.lisp`. The `BBS` package is basically a bucket for my projects to help prevent name collisions with any other packages.

4.2.1 `BBS.lisp`

This package is the root package for Tiny-Lisp. It contains most of the data structures and the public interface for the interpreter. In addition, a number of common utility function are defined here so that they can be used in all the child packages.

`bbs.lisp.evaluate`

This package contains child packages for evaluating the Tiny-Lisp operations as well as common functions used by its children. To keep it a reasonable size, the following child packages have been broken out:

- `BBS.lisp.evaluate.bool` - Contains operations relating to `boolean` values.
- `BBS.lisp.evaluate.char` - Contains operations relating to `character` values.
- `BBS.lisp.evaluate.cond` - Contains Tiny-Lisp conditional operations.
- `BBS.lisp.evaluate.func` - Contains operations related to defining functions.
- `BBS.lisp.evaluate.io` - Contains Tiny-Lisp input/output operations.
- `BBS.lisp.evaluate.list` - Contains operations relating to `list` values.
- `BBS.lisp.evaluate.loops` - Contains Tiny-Lisp loop operations.
- `BBS.lisp.evaluate.math` - Contains Tiny-Lisp math operations.
- `BBS.lisp.evaluate.mem` - Contains Tiny-Lisp memory access operations.
- `BBS.lisp.evaluate.misc` - Contains operations that don't fit into any other catagory.
- `BBS.lisp.evaluate.pred` - Contains predicate (test) operations.
- `BBS.lisp.evaluate.str` - Contains operations relating to `string` values.
- `BBS.lisp.evaluate.symb` - Contains operations relating to symbols.
- `BBS.lisp.evaluate.vars` - Contains operations relating to variables.

`BBS.lisp.memory`

This package contains the memory manager. This is mainly allocating items and incrementing and decrementing the reference count.

`BBS.lisp.parser`

This package contains the parser.

`BBS.lisp.stack`

This package contains the stack and functions for accessing the stack.

`BBS.lisp.strings`

This package contains some utility functions for strings.

`BBS.lisp.utilities`

This package contains some general utility functions.

4.3 Data Structures

Most of the data structures are defined in the `BBS.lisp` package, except for the stack, which is defined in `BBS.lisp.stack`.

The main arrays have size limits and data types defined for accessing them. These may change if the `bbs.lisp` package gets turned into a generic package. Should that happen The four constants will be the generic parameters. This will make adjusting the size of the structures a little easier when embedding.

```
max_cons : constant Integer := 300;
max_symb : constant Integer := 200;
max_string : constant Integer := 500;
max_stack : constant Integer := 100;
--
type cons_index is range -1 .. max_cons;
type symb_index is range -1 .. max_symb;
type string_index is range -1 .. max_string;
type stack_index is range -1 .. max_stack;
```

The arrays are defined from *..._index'First + 1* to *..._index'Last*. The value *..._index'First* is used to represent an invalid or null index. The following constants are defined for this:

```
NIL_CONS : constant cons_index := cons_index'First;
NIL_STR : constant string_index := string_index'First;
NIL_SYM : constant symb_index := symb_index'First;
```

4.3.1 Elements

The basic data type is the element. It is defined as follows:

```
type ptr_type is (E_CONS, E_ERROR, E_NIL, E_STACK, E_SYMBOL,
                  E_TEMPSYM, E_VALUE);
type element_type(kind : ptr_type := E_NIL) is
```

```
record
   case kind is
      when E_CONS =>
         ps : cons_index;
      when E_ERROR =>
         null;
      when E_NIL =>
         null;
      when E_TEMPSYM =>
         tempsym : string_index;
      when E_SYMBOL =>
         sym : symb_index;
      when E_STACK =>
         st_name : string_index;
         st_offset : stack_index;
      when E_VALUE =>
         v : value;
   end case;
end record;
```

The different types of elements are:

E_CONS Contains an index into the array of cons cells as described in section 4.3.2. This may eventually go away.

E_ERROR This indicates that some operation has encountered an error of some sort.

E_NIL This represents an empty element.

E_TEMPSYM This contains an index into the *string* table for representing a temporary symbol name. This is used during parsing to represent an item where the type has not yet been determined. It should never appear once parsing is complete.

E_STACK This represents a stack variable. It contains an index into the *string* table for the variable's name and a stack frame offset.

E_SYMBOL This contains an index into the *symbol* table thus representing a symbol as described in section 4.3.3.

E_VALUE This represents a value as described in section 4.3.4. It can contain any of the defined data types. Note that for *V_STRING* or *V_LIST* data types, the value actually contains an index into the *string* or *cons* array.

There is a bit of ambiguity right now about lists. Since recursively defined records aren't possible, elements of type *E_STACK* can't contain an `element_type`. So in order for them to be able to have lists, the `value` type also contains a list pointer. This means that right now, an `element_type` can point to a list either directly by having a kind of *E_CONS*, or by having a kind that contains a `value` with a kind of *V_LIST*. This really should be fixed at some point. On the other hand, one could make the distinction that the kind *E_CONS* represents a list that can be evaluated, while a `value` of kind *V_LIST* is just data.

4.3.2 Cons

Cons elements are used to make lists. A cons cell is defined as

```
type cons is
   record
      ref : Natural;
      car : element_type;
      cdr : element_type;
   end record;
```

4.3.3 Symbols

Symbols are defined as:

```
type symbol_type is (SY_SPECIAL,  -- A special form that needs
                                  --  support during parsing
                     SY_BUILTIN,  -- A normal builtin function
                     SY_LAMBDA,   -- A user defined function
                     SY_VARIABLE, -- A value, not a function
                     SY_EMPTY);   -- No contents
type execute_function is access function(e : element_type)
  return element_type;
type special_function is access function(e : element_type;
                                         p : phase)
 return element_type;
type symbol(kind : symbol_type := SY_EMPTY) is
   record
      ref : Natural;
      str : string_index;
      case kind is
         when SY_SPECIAL =>
            s : special_function;
         when SY_BUILTIN =>
            f : execute_function;
         when SY_LAMBDA =>
            ps : cons_index;
         when SY_VARIABLE =>
            pv : element_type;
         when SY_EMPTY =>
            null;
      end case;
   end record;
```

SY_BUILTIN vs SY_SPECIAL

Some functions need to be able to access some of their parameters during parsing so that the rest of the parameters can be properly parsed. Usually, but not always, this involves building a stack

frame with the parameters so that they will be properly identified during further processing. These functions are passed an extra parameter p for phase. The possible values are:

```
type phase is (PH_QUERY, PH_PARSE_BEGIN, PH_PARSE_END, PH_EXECUTE);
```

The phases are:

PH_QUERY Initial call to the function to query the function when it wants to be called again. The function returns an integer value indicating the parameter after which it should be called.

PH_PARSE_BEGIN This is the call after the desired parameter has been parsed. The function can then examine this parameter and make any needed changes.

PH_PARSE_END This is the call at the end of parsing for the function. Usually this just clears the stack frame. It could also be used for things like preprocessing the parameter list.

PH_EXECUTE This is the call for execution where the function performs its normal operation.

4.3.4 Values

The value type represent a (surprise) value. It can be either an atomic type such as integer or boolean, or a more complex type such as a list or a string.

```
type value_type is (V_INTEGER, V_STRING, V_CHARACTER, V_BOOLEAN,
                    V_LIST, V_LAMBDA, V_SYMBOL, V_QSYMBOL, V_NONE);
type int32 is range -(2**31) .. 2**31 - 1
  with Size => 32;
type value(kind : value_type := V_INTEGER) is
   record
      case kind is
      when V_INTEGER =>
         i : int32;
      when V_CHARACTER =>
         c : Character;
      when V_STRING =>
         s : string_index;
      when V_BOOLEAN =>
         b : Boolean;
      when V_LIST =>
         l : cons_index;
      when V_LAMBDA =>
         lam : cons_index;
      when V_SYMBOL =>
         sym : symb_index;
      when V_QSYMBOL =>
         qsym : symb_index;
      when V_NONE =>
         null;
      end case;
   end record;
```

The data types available are:

V_INTEGER is the basic integer numeric type. It is defined as a 32 bit signed integer. Basic math operations can be performed on it and integers can be compared.

V_STRING is the string type. These are unbounded strings. The value structure contains an index into the string fragment array. Details of strings are described in section 4.3.5.

V_CHARACTER will represent a character data type when implemented. It is currently not implemented.

V_BOOLEAN is a boolean data type that can represent false or true. Comparison operations return boolean values and certain functions expect boolean values.

V_LIST is a list data type. It is approximately equal to an element type of *E_CONS* (see section 4.3.2). The value structure contains an index into the cons cell array.

V_LAMBDA is a list data type that is used to represent a user defined function. It is approximately equivalent to the symbol type of *SY_LAMBDA*, except that it can be assigned to stack variables.

V_QSYMBOL is a quoted symbol.

V_SYMBOL is a symbol. This is currently not used and may be deleted.

4.3.5 Strings

Strings are stored as a set of string fragments in a linked list. Thus, the length of a string is limited only by the number of fragments available. Strings are defined as:

```
fragment_len : constant Integer := 16;
type fragment is
   record
      ref : Natural;
      next : Integer range -1 .. Integer(string_index'Last);
      len : Integer range 0..fragment_len;
      str : String (1..fragment_len);
   end record;
```

4.3.6 Functions

A function is a list that contains two elements. The first element is a list of the function parameters. The second element is a list of the function's statements.

4.3.7 The Stack

A stack is defined for storing function parameters and local variables. The function parameters are used only for user defined functions. Builtin and Special functions are handled within the Ada code directly from the *cons* cells of the function parameter list. Stack entries are defined as follow:

```
type stack_entry_type is (ST_EMPTY, ST_FRAME, ST_VALUE);
type stack_entry(kind : stack_entry_type := ST_EMPTY) is
   record
      case kind is
         when ST_EMPTY =>
            null;
         when ST_FRAME =>
            number: Natural;
            next : stack_index;
         when ST_VALUE =>
            st_name : string_index;
            st_value : value;
      end case;
   end record;
```

Each stack entry can be empty, a stack frame boundary, or a variable. Stack variables have a name and a value.

4.3.8 Global Data

The various data arrays are defined as follows.

The actual arrays are (in `bbs.lisp`):

```
--
--  Since this interpreter is designed to be used on embedded computers
--  with no operating system and possibly no dynamic memory allocation,
--  The statically allocated data structures are defined here.
--
cons_table : array (cons_index'First + 1 .. cons_index'Last) of cons;
symb_table : array (symb_index'First + 1 .. symb_index'Last) of symbol;
string_table : array (string_index'First + 1 .. string_index'Last)
                 of fragment;
```

And in the stack package (`bbs.lisp.stack`):

```
--
--  The stack array
--
stack : array (stack_index'First + 1 .. stack_index'Last) of
        stack_entry := (others => (kind => ST_EMPTY));
```

Note that all the arrays have a lower bound of *...'first+1*. This allows an index value equal to *...'first* to be used to indicate a null entry.

4.3.9 Memory Management

Memory management is done by reference counting. When the number of references goes to zero, the item is deallocated. Items in the cons table and the strings table are reference counted.

4.4 Utility Functions

There are a number of functions that are available for use when embedding and extending Tiny-Lisp. These are primarily only in a few packages and they may be moved to improve organization.

4.4.1 BBS.lisp

The functions available here are primarily concerned with the overall operation of the interpreter. The first procedure to call is:

```
procedure init(p_put_line : t_put_line; p_put : t_put_line;
               p_new_line : t_newline; p_get_line : t_get_line);
```

This routine is used to establish pointers to the I/O functions used and to define the symbols for builtin and special functions. After this symbols for custom functions can be added. The following procedure is used for that:

```
procedure add_builtin(n : String; f : execute_function);
```

To pass control to the Tiny-Lisp read-execute-print-loop, the following procedure is used:

```
procedure repl;
```

If more control is needed, the read-execute-print-loop can be broken out using the following functions and procedure:

```
function read return Element_Type;
function eval(e : element_type) return element_type;
procedure print(e : element_type; d : Boolean; nl : Boolean);
function exit_lisp return Boolean;
```

These would be used in a loop as follows:

```
procedure repl is
   e : element_type;
   r : element_type;
begin
   exit_flag := False;
   break_flag := false;
   while True loop
      BBS.lisp.stack.reset;
      e := read;
      if e.kind /= E_ERROR then
         r := eval(e);
         if not first_char_flag then
            new_line;
         end if;
         print(r, True, True);
      end if;
      exit when exit_lisp;
   end loop;
end;
```

For writing custom functions, the following functions may be useful:

```
procedure error(f : String; m : String);
procedure msg(f : String; m : String);
procedure print(e : element_type; d : Boolean; nl : Boolean);
```

These support printing error and informational messages as well as printing Tiny-Lisp elements. There are other useful functions in some other packages as well.

4.4.2 BBS.lisp.evaluate

This package contains functions useful in the evaluation of Tiny-Lisp operations. The most useful, when adding custom operations, is:

```
function first_value(s : in out cons_index) return element_type;
```

It extracts the first element from the list pointed to by `s` and updates `s` to point to the next element in the list. If the first element is a variable, the value of the variable is returned. If the first element is a Tiny-Lisp operation, it is evaluated and the result of the evaluation is returned.

4.4.3 BBS.lisp.utilities

4.5 Embedding

This section covers how too embed the list interpreter in another program. Here is a minimal host program:

```
with Ada.Text_IO;
with bbs.lisp;
with new_line;
--
--  This is a simple shell routine to call the embedded lisp
--  interpreter.
--
procedure Lisp is
begin
   Ada.Text_IO.Put_Line("Tiny lisp interpreter written in Ada.");
   bbs.lisp.init(Ada.Text_IO.Put_Line'Access, Ada.Text_IO.Put'Access,
                 new_line.New_Line'Access, Ada.Text_IO.Get_Line'Access);
   bbs.lisp.repl;
end Lisp;
```

With `new_line` defined as:

```
--
--  The text_io version of newline contains an optional parameter
--  indicating the number of lines to skip.  The type of this parameter
--  is defined in Ada.Text_IO.  This makes it awkward to define a
--  function prototype that can be used both when Ada.Text_IO is
--  available and when it isn't.  This is a crude hack to define
```

```
--  locally a new_line that has no parameters and uses the
--  Ada.Text_IO new_line with the default value.
--
package new_line is
   procedure new_line;
end new_line;

with Ada.Text_IO;
package body new_line is

   procedure new_line is
   begin
      Ada.Text_IO.New_Line;
   end;

end new_line;
```

It's fairly simple. Initialize the interpreter and call it. The only wrinkle is the need to define `new_line`. The Ada version has an optional parameter of a type defined in `Ada.Text_IO`. This is a problem when trying to eliminate dependencies on `Ada.Text_IO`. A more complex example of embedding is found in the `https://github.com/BrentSeidel/Ada-Arduino-Due` repository. This repository contains code that runs on an Arduino Due and includes the definition of several Tiny-Lisp operations to access attached hardware.

4.5.1 Adding Custom Operations

The Ada functions that implement the Tiny-Lisp operations are defined using one of the two following prototypes:

```
   --
   --  Type for access to function that implement lisp operations.
   --
   type execute_function is access procedure(e : out element_type;
            s : cons_index);
   --
   --  Type for access to functions that implement lisp special
   --  operations
   --
   type special_function is access procedure(e : out element_type;
            s : cons_index; p : phase);
```

In most cases, an `execute_function` is the type to use. To install the operation, add something like the following line after the main Tiny-Lisp initialization function is called.

```
BBS.lisp.add_builtin("due-flash", due_flash'Access);
```

The first parameter to `add_builtin` is a string giving the Tiny-Lisp operation name. The second parameter is an access to the Ada function to call.

In the function that you write, the parameter `s` is an index pointing to the start of the parameter list. Thus a Tiny-Lisp expression like:

```
(some-function 1 2 3)
```

is translated into a linked list approximately like:

```
symbol.builtin("some-function")->
  value.integer(1)->
    value.integer(2)->
      value.integer(3)->
        NIL_CONS
```

The first element is turned into the Ada function call with (s) pointing to the second element (value.integer(1)). The Ada function can then traverse the list and extract the Tiny-Lisp parameters.

4.6 Opportunities for Optimizing

No big effort has gone into optimizing the interpreter. Should the need arise, there are a few places where things could be optimized.

4.6.1 Memory Management

If allocation becomes a bottleneck, the free items could be linked together in a list. That way a new item could be picked off the head of the list instead of searching through all the items. This would also require the list to be created at initialization.

4.6.2 Constant expressions

During parsing, it may be possible to recognize some constant expressions are replace them by their result. For example:

```
(+ 1 2 3) -> 6
```

4.6.3 The Symbol Table

An obvious target for optimization would be to sort the symbol table. Then a binary search could be done to locate symbols. The reason that this is not done is that searching for symbols is only done during parsing. The parser locates the symbol in the table and replaces it by its index. During execution, the symbol index is used to directly access the symbol without doing a search. This means that once a symbol is defined, it must never change its location in the table.