# Tiny Lisp Interpreter

Brent Seidel
Phoenix, AZ

September 28, 2020

Note that this is a draft version and not the final version for publication.

# 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-int #\B)
(char-upcase #\c)
(int-char 65)
```

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

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.

The `reset` operation is intended to reset the interpreter to an initial state. Currently, it just calls the `init()` function. It needs to go through the symbol table and remove everything that's not a `builtin` and clear out the cons table. Do not use until this is fixed.

### 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))
(functionp functionp)
(integerp 3)
(listp (2 4 6))
(numberp 4)
(nullp ())
(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)
(concatinate '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 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.

# Chapter 3

# Internals

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

## 3.1 Data Structures

### 3.1.1 Elements

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

```
max_cons : constant Integer := 300;
max_symb : constant Integer := 200;
max_string : constant Integer := 500;
type cons_index is range 0 .. max_cons;
type symb_index is range 0 .. max_symb;
type string_index is range 0 .. max_string;
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 3.1.2. 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 3.1.3.

**E_VALUE** This represents a value as described in section 3.1.4. 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.

### 3.1.2 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;
```

### 3.1.3 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.

### 3.1.4 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 3.1.5.

**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 3.1.2). 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.

### 3.1.5 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;
```

### 3.1.6 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.

### 3.1.7 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.

### 3.1.8 Global Data

### 3.1.9 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.

## 3.2 Utility Functions

## 3.3 Embedding

This section covers how to 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.

## 3.4 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.

### 3.4.1 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.

### 3.4.2 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