Tiny Lisp Interpreter

Brent Seidel
Phoenix, AZ

June 28, 2020

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.

1.2.2 Why Ada?

Again, why non? 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 Syntax

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).

2.2 Operations

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

2.2.1 Normal Operation

A number of normal operation are defined.

Arithmatic Operations

Four arithmatic 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.

Comparison Operations

Four comparison operations are defined for both integers and strings. The operations are equals, not equals, greater than, and less than. 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.

Input/Output

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

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

The print operation evaluates each item in its parameter list and prints the value. The new-line operation simply prints a new line. The read-line operation reads a string from the input device and returns it. Both print and new-line return NIL. Read-line returns the entered string with no line terminator.

List Operations

Basic list operations are provided.

```
(car 1 2 3 4)
(cdr 1 2 3 4)
```

Each of car and cdr take a list of parameters. Car returns the first item in the list. Cdr returns the remainder of the list.

Other

```
add_builtin("dump", BBS.lisp.evaluate.dump'Access);
add_builtin("t", BBS.lisp.evaluate.true'Access);
add_builtin("msg-on", BBS.lisp.evaluate.msg_on'Access);
add_builtin("msg-off", BBS.lisp.evaluate.msg_off'Access);
```

2.2.2 Special Forms

```
add_builtin("setq", BBS.lisp.evaluate.setq'Access);
add_builtin("if", BBS.lisp.evaluate.eval_if'Access);
add_builtin("dowhile", BBS.lisp.evaluate.dowhile'Access);
add_builtin("dotimes", BBS.lisp.evaluate.dotimes'Access);
add_builtin("defun", BBS.lisp.evaluate.defun'Access);
add_builtin("exit", BBS.lisp.evaluate.quit'Access);
add_builtin("reset", BBS.lisp.evaluate.reset'Access);
add_builtin("quote", BBS.lisp.evaluate.quote'Access);
```

2.3 Data Types

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

2.3.1 Integer

This is the same as the Ada Integer type. Typically it is a 32 bit signed integer.

2.3.2 String

Strings are stored in linked lists of 8-bit characters/bytes. Unicode is not currently supported.

2.3.3 Boolean

2.3.4 List

2.3.5 Possible Additions

2.4 Internals

2.4.1 Data Structures

Elements

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

```
type element_type(kind : ptr_type := E_NIL) is
   record
      case kind is
         when E_CONS =>
            ps : cons_index;
         when E_NIL =>
            null;
         when E_VALUE =>
            v : value;
         when E_SYMBOL =>
            sym : symb_index;
         when E_TEMPSYM =>
            tempsym : tempsym_index;
         when E_PARAM =>
            p_name : string_index;
            p_value : value;
         when E_LOCAL =>
            l_name : string_index;
            l_value : value;
      end case;
   end record;
```

The referenced types are defined as follows:

```
max_cons : constant Integer := 300;
max_symb : constant Integer := 200;
max_tempym : constant Integer := 50;
max_string : constant Integer := 500;
type cons_index is range 0 .. max_cons;
type symb_index is range 0 .. max_symb;
type tempsym_index is range 0 .. max_tempym;
type string_index is range 0 .. max_string;
type ptr_type is (E_CONS, E_NIL, E_VALUE, E_SYMBOL,
                  E_TEMPSYM, E_PARAM, E_LOCAL);
type value_type is (V_INTEGER, V_STRING, V_CHARACTER, V_BOOLEAN, V_LIST);
type value(kind : value_type := V_INTEGER) is
   record
      case kind is
      when V_INTEGER =>
         i : Integer;
      when V_CHARACTER =>
         c : Character;
      when V_STRING =>
         s : string_index;
      when V_BOOLEAN =>
         b : Boolean;
      when V_LIST =>
```

```
            l : cons_index;
        end case;
    end record;
```

Cons

Symbols

Strings

2.4.2 Utility Functions

2.4.3 Embedding

Chapter 3

Code Examples

3.1 Lisp Code

Here is some sample Lisp code

```
;
;  Read an analog pin and print the value repeatedly.
;  Digital pin 10 is tied high to keep looping and tied low to exit the loop.
;
(defun monitor-analog (n)
  (pin-mode 10 0)
  (print "Connect digital pin 10 to high to continue looping or to gnd to exit")
  (new-line)
  (print "Connect analog pin " n " to the analog value to monitor")
  (new-line)
  (print "Press <return> to continue")
  (read-line)
  (dowhile (= (read-pin 10) (+ 0 1))
    (print "Analog value is " (read-analog n))
    (new-line))
  (print "Exiting")
  (new-line))
```

3.2 Ada Code

Here is some sample Ada code

```
   procedure first_value(e : element_type;
                         car : out element_type;
                         cdr : out element_type) is
      first : element_type;
      temp : element_type;
```

```
      s : cons_index;
   begin
      if e.kind = E_NIL then
         car := NIL_ELEM;
         cdr := NIL_ELEM;
      elsif e.kind /= E_CONS then
         car := indirect_elem(e);
         cdr := NIL_ELEM;
      else -- The only other option is E_CONS
         s := e.ps;
         first := cons_table(s).car;
         cdr :=  cons_table(s).cdr;
         if first.kind = E_NIL then
            car := NIL_ELEM;
         elsif first.kind /= E_CONS then
            car := indirect_elem(first);
         else -- The first item is a E_CONS
            temp := eval_dispatch(first.ps);
            if temp.kind = E_NIL then
               car := NIL_ELEM;
            elsif temp.kind /= E_CONS then
               car := temp;
            else
               car := cons_table(temp.ps).car;
               cdr := cons_table(temp.ps).cdr;
            end if;
         end if;
      end if;
   end;
```