A LISP interpreter in Awk

Roger Rohrbach

1592 Union St., #94
San Francisco, CA 94123
January 3, 1989

ABSTRACT

This note describes a simple interpreter for the LISP programming language, written in **awk**. It provides intrinsic versions of the basic functions on s-expressions, and many others written in LISP. It is compatible with the commonly available version of **awk** that is supplied with most UNIX systems. The interpreter serves to illustrate the use of **awk** for prototyping or implementing language translators, as well as providing a simple example of LISP implementation techniques.

Intrinsic functions.

The interpreter has thirteen built-in functions. These include the five elementary functions on s-expressions defined by McCarthy [1]; some conditional expression operators; an assignment operator, and some functions to control the evaluation process.

The intrinsic functions are summarized below. Familiarity with existing LISP systems is assumed in the descriptions of these functions.

`(car` *l*`)`
: Returns the first element of the list *l*. An error occurs if *l* is atomic.

`(cdr` *l*`)`
: Returns the remainder of the list *l*, *i.e.*, the sublist containing the second through the last elements. If *l* has only one element, **nil** is returned. **cdr** is undefined on atoms.

`(cons` *e l*`)`
: Constructs a new list whose **car** is *e* and whose **cdr** is *l*.

`(eq` *x y*`)`
: Returns **t** if *x* and *y* are the same LISP object, *i.e.*, are either atomic and have the same print name, or evaluate to the same list cell. Otherwise, **nil** is returned.

`(atom` *s*`)`
: Returns **t** if *s* is an atom, otherwise **nil**.

`(set` *x y*`)`
: Assigns the value *y* to *x* and returns *y*. *x* must be atomic, and may not be a constant or name an intrinsic function.

`(eval` *s*`)`
: Evaluates *s* and returns the result.

`(error` *s*`)`
: Halts evaluation and returns **nil**. The atom *s* is printed.

`(quote` *s*`)`
: Returns *s*, unevaluated. The form

´*expr*

is an abbreviation for

(quote *expr*)

(cond (*p1* [*e1*]) ... (*pN* [*eN*]))

Evaluates each *p* from left to right. If any evaluate to a value other than **nil**, the corresponding *e* is evaluated and the result is returned. If there is no corresponding *e*, the value of the *p* itself is returned instead. If no *p* has a non-null value, **nil** is returned.

(and *e1* ... *eN*)

Evaluates each *e* and returns **nil** if any evaluate to **nil**. Otherwise the value of the last *e* is returned.

(or *e1* ... *eN*)

Evaluates each *e* and returns the first whose value is non-null. If no such *e* is found, **nil** is returned.

(list *e1* ... *eN*)

Constructs a new list with elements *e1* ... *eN*. Equivalent to (cons *e1* (cons ... (cons *eN* nil)).

Lambda functions.

The following functions are written in LISP and are defined in the file **walk.w**. Most of these are commonly supplied with LISP systems.

(cadr *s*)

(cddr *s*)

(caar *s*)

(cdar *s*)

(cadar *s*)

(caddr *s*)

(cddar *s*)

(cdadr *s*)

These correspond to various compositions of **car** and **cdr**, *e.g.*, (cadr *s*) → (car (cdr *s*)).

(null *s*)

Equivalent to (eq *s* nil).

(not *s*)

Same as **null**.

(ff *s*)

Returns the first atomic symbol in *s*.

(subst *x y z*)

Substitutes *x* for all occurrences of the atom *y* in *z*. *x* and *z* are arbitrary s-expressions.

(equal *x y*)

Returns **t** if *x* and *y* are the same s-expression, otherwise **nil**.

(append *x y*)

Creates a new list containing the elements of x and y, which must both be lists.

(member *x y*)

Returns **t** if *x* is an element of the list *y*, otherwise **nil**.

(pair *x y*)

Pairs each element of the lists *x* and *y*, and returns a list of the resulting pairs. The number of pairs in the result will equal the length of the shorter of the two input lists.

`(assoc` *x y*`)`
: Association list selector function. *y* is a list of the form `((`*u1 v1*`)` ... `(`*uN vN*`))` where the *u*'s are atomic. If *x* is one of these, the corresponding pair `(`*u v*`)` is returned, otherwise **nil**.

`(sublis` *x y*`)`
: *x* is an association list. Substitutes the values in *x* for the keys in *y*.

`(last` *l*`)`
: Returns the last element of *l*.

`(reverse` *l*`)`
: Returns a list that contains the elements in *l*, in reverse order.

`(remove` *e l*`)`
: Returns a copy of *l* with all occurrences of the element *e* removed.

`(succ` *x y*`)`
: Returns the element that immediately follows the atom *x* in the list *y*. If *x* does not occur in *y* or is the last element, **nil** is returned.

`(pred` *x y*`)`
: Returns the element that immediately precedes the atom *x* in the list *y*. If *x* does not occur in *y* or is the first element, **nil** is returned.

`(before` *x y*`)`
: Returns the list of elements occurring before y in x. If *y* does not occur in *x* or is the first element, **nil** is returned.

`(after` *x y*`)`
: Returns the list of elements occurring after y in x. If *y* does not occur in *x* or is the last element, **nil** is returned.

`(plist` *x*`)`
: Returns the property list for the atom *x*.

`(get` *x i*`)`
: Returns the value stored on *x*'s property list under the indicator *i*.

`(putprop` *x v i*`)`
: Stores the value *v* on *x*'s property list under the indicator *i*.

`(remprop` *x i*`)`
: Remove the indicator *i* and any associated value from *x*'s property list.

`(mapcar` *f l*`)`
: Applies the function *f* to each element of *l* and returns the list of results.

`(apply` *f args*`)`
: Calls *f* with the arguments *args*, *e.g.*,

```
(apply 'cons '(a (b)))
```

is equivalent to

```
(cons 'a '(b))
```

Syntactic conventions.

Atoms take the following forms:

Regular identifiers
: Atoms matching the regular expression

```
[_A-Za-z][-A-Za-z_0-9]*
```

The initial value of an identifier is **nil**.

Integers

Atoms matching the regular expression `[0-9][0-9]*`. Integers are constants, *i.e.*, evaluate to themselves.

Weird atoms

Identifiers matching the regular expression `".*"`. Weird atoms are not constants.

A semicolon introduces a comment, which continues for the rest of the line.

Usage.

The command for running the interpreter is

```
walk [ files ]
```

on BSD UNIX and derivative systems, or

```
awk -f walk [ files ]
```

on UNIX System V. The file name − represents the standard input. This can be omitted if no other files are being read in, or if the interpreter is being run non-interactively.

Normally, the interpreter is used interactively, augmented with the functions defined in **walk.w**, and, perhaps, other files. The command line to use for this purpose is

```
walk walk.w [ other files ] p -
```

The interpreter will first read **walk.w**, printing the results of evaluating the function definitions therein. Then it will read **p**. This file contains no LISP definitions; the interpreter recognizes it by name and prints a prompt to signal the user that all the prerequisite files have been read and that the interpreter is waiting for input. (This is the only way to get **awk** to do this; this can be hidden from the user with a shell program that invokes the interpreter if desired.) Thereafter, it will evaluate expressions typed in by the user, printing a prompt after each one. Normally the prompt is `->`; the first character of the prompt changes when appropriate to an integer that represents the number of unmatched left parentheses read in so far.

The interpreter exits when it encounters the end of its last input file. If this file is the standard input, the number of LISP objects created is reported.

Several files defining auxiliary functions are provided.

Implementation.

So that it can run on any UNIX system, The LISP interpreter has been written using the UNIX V7 version of **awk**, which predates the version described in *The Awk Programming Language* [2]. The only complex data type provided by this language is the array. Data that in C might be stored in structures is represented, therefore, using multiple arrays, one for each field. For example, the C code

```
p = allocate_cell();
p->car = s;
p->cdr = NIL;
```

can be approximated with:

```
p = ++cell;
car[p] = s;
cdr[p] = nil;
```

Lists (using nested array references) and stacks are also simulated with arrays. The most important data structures are explained in the program and in the following description.

As is usual for LISP implementations, the interpreter is constructed as a loop that reads an s-expression, evaluates it, and prints the result. The reader collects an s-expression, reading multiple input lines if necessary. Like the other two phases of the interpreter, this is a recursive procedure and in **awk** this must be managed explicitly. When an s-expression is read, its internal representation in list structure is formed using the stack **read[]**. Atoms and **cons** operators are pushed onto the read stack and periodically ‘reduced’

or replaced with list cells when a complete list has been read; the reader returns an atom or list on the top of the stack. The reader must be able to return an s-expression in the middle of an input line, so the entire interpreter is enclosed in a loop that allows the current input line to be completely scanned before the next input record is read. The general outline is:

```
BEGIN {
     initialize interpreter
     say hello if interactive
}

{
     initialize reader variables

     while (chars left on this line)
     {
          read

          if (have read an s-expression)
          {
               eval

               print
          }
     }

     prompt if interactive
}

END {
     say goodbye if interactive
     exit
}
```

The evaluator maintains two stacks, one for input and one for output. The result returned by the reader is copied onto the input stack (**eval[]**), and evaluated according to the usual LISP rules. Evaluated s-expressions are placed on the output stack, **arg[]**. When an intrinsic function that takes evaluated arguments appears on the top of the evaluation stack, its arguments are popped from the argument stack. Functions (like **cond**) that take unevaluated arguments are handled as special forms before their arguments have been pushed onto **eval[]**. The arguments are handled differently depending on the semantics of the function. Lambda (user-defined) functions are evaluated by temporarily binding the formal parameters in the function definition to the results of evaluating the actual arguments with which the function was called, and then evaluating the body of the function. Temporary bindings only are kept on a special pushdown list (the *alist*). Atoms have a global value that is stored separately; this keeps the alist small.

The evaluation procedure is sketched below:

atom?
    `lambda`
        restore previous environment (lambda function
        body has been evaluated already)
    constant?
        return
    bound?
        look up local value
    otherwise
        return global value

intrinsic function?
    apply to already evaluated arguments

lambda function?
    bind formal parameters to already evaluated arguments
    evaluate function body

form?
    intrinsic function application?
        `quote`
            return unevaluated argument
        `cond`
        `and`
        `or`
            begin evaluating arguments according to operator semantics
        `list`
            expand to repeated applications of cons
    other?
        push function variable, arguments

lambda function application?
    push lambda function, body
other?
    eval **car**, **cdr**

When the evaluation stack is emptied, the result is popped from the argument stack and printed. A stack is again used to manage recursion.

Conclusion.

The goal of writing a small LISP interpreter and extending it in LISP has been realized. Though it was not my original intention, it would be easy to incorporate the LISP functions as intrinsics, and many other extensions (such as numeric functions) could be made, in which case the interpreter might fulfill more than a pedagogic function. Even so, it can be used as is for an introduction to LISP programming and implementation concepts. I hope it also inspires more of us to learn how to program in **awk**!

References.

[1]
McCarthy, J. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. *Comm. ACM*, 3, 4, pp. 185-195 April 1960

[2]
Aho, A., Weinberger, P., & Kernighan, B.W. *The Awk Programming Language*. Addison-Wesley, Reading, MA 1988