Parsing Key=Value Pairs in Lisp

Gene Michael Stover

created Sunday, 24 October 2004
updated Sunday, 7 November 2004

Copyright © 2004 Gene Michael Stover. All rights reserved. Permission to copy, store, & view this document unmodified & in its entirety is granted.

Contents

1 Introduction

It seems like every week I need a function to parse strings that contain *key=value* pairs. Instead of writing the code from scratch each time because I can't remember where I stowed the last copy I wrote, I'm putting it in an article & documenting it once & for all.

Why don't I put it in a library? One reason is that I'm often programming for other people who place limitations on the libraries I'm allowed to use. The main reason, though, is that it seems like each of these *key=value* formats is slightly different[1], & I'm not certain I could make an API that's simultaneously simple but general enough to handle all of them. Rather than have a library containing a pair-parsing service that never works the way I need, I'd rather copy, paste, & modify the code here.[2]

2 Requirements

The code doesn't require anything except Common Lisp & some standard Common Lisp functions. If you have a Common Lisp system, you should be able to use these functions as they are.

3 License

The source code in this article & in the accompanying `pkvpl.lisp` file is licensed according to the terms of the Gnu Lesser General Public License ([2,)] (LGPL).

The documentation (the document you are reading now) is copyrighted by Gene Michael Stover. The copyright notice is at the beginning of the document. The documentation is not licensed according to the LGPL.

4 Obtaining

The source code in this article is available in a single file, http://lisp-p.org/pkvpl/pkvpl.lisp.

To use PARSE-KEY-VALUE-PAIRS in your programs, you might copy-&-paste the "`defun parse-key-value-pairs`" form from that file into your program.

5 What It Does (a few examples)

The PARSE-KEY-VALUE-PAIRS function eats *key=value* pairs from a stream & excretes an equivalent Lisp ASSOCIATION LIST . Examples of the *key=value*

[1] No no no, I'm not defining all these annoyingly similar-but-unique formats. Other people defined them. If it were up to me, they'd all be of the form "((name0 . value0) (name1 . value1))" so I could use Lisp's `read` function to parse them without having to write any code myself.

[2] Tim Kientzle once said to me something like "Copy-&-paste is the fundamental form of code re-use".

pairs I'm talking about might be:

- a configuration file in which each line is a *key=value* pair, or
- an HTTP Cookie header, which contains *key=value* pairs separated by semicolons or commas.

For example, if the input is "$Version=0; a= 1,b=2 ; c=3", which we might get from a sloppily formed HTTP Cookie header, PARSE-KEY-VALUE-PAIRS would return "`(("$Version" . "0") ("a" . "1") ("b" . "2") ("c" . "3"))`".

PARSE-KEY-VALUE-PAIRS has arguments that allow you to specify the characters that separate keys from values, the characters that separate *key=value* pairs from each other, & whether white-space characters are ignored.

For another example, if we have a two-line file named "equivs.txt", with contents like you see here:

```
$ cat equivs.txt
Romulus=Cain
Remus=Abel
$
```

then you might parse it with `parse-key-value-pairs` like this:

```
lisp> (with-open-file (strm "equivs.txt")
        (parse-key-value-pairs strm
          :pairsep '(#\Newline)
          :white-space '(#\Space)))
=> (("Romulus" . "Cain") ("Remus" . "Abel"))
```

Here's another example, in which white-space is significant, & we use "&" as the separator between pairs:

```
lisp> (defvar *x* " a = 1&b =2 &c=3&")
=> *X*
lisp> (with-input-from-string (strm *x*)
        (parse-key-value-pairs strm
          :pairsep '(#\&)))
;; The space characters are preserved!
=> ((" a " . " 1") ("b " . "2 ") ("c" . "3"))
```

Notice that the space characters are preserved in this last example, but they weren't in the previous examples because we didn't use the :WHITE-SPACE keyword argument in the previous examples.

In the next section, I'll discuss all the arguments for PARSE-KEY-VALUE-PAIRS.

6 How to Use It

PARSE-KEY-VALUE-PAIRS (*strm* &KEY (*pairsep* '(#\;)) (*nvsep* '(#\=)) (*white-space* ()))

strm is a character input stream. PARSE-KEY-VALUE-PAIRS parses characters from *strm*. *Strm* must support the PEEK-CHAR & READ-CHAR functions.

pairsep is a set (list) of characters which separate the pairs in the input. Whenever PARSE-KEY-VALUE-PAIRS encounters a character from *pairsep* in the input, it assumes that character separates two pairs. *Pairsep* defaults to `'(#\;)`.

nvsep is a set (list) of characters which separate names from values. Whenever PARSE-KEY-VALUE-PAIRS encounters a character from *nvsep*, it assumes that what precedes is a name (a.k.a. key) & what follows is the value associated with it. *Nvsep* defaults to `'(#\=)`.

white-space is a set (list) of characters to ignore. These characters are stripped from the input as if they did not exist. *White-space* defaults to NIL.

Given an input stream which contains name/value pairs encoded in a way that can be recognized with the PAIRSEP, NVSEP, & WHITE-SPACE values you supply, PARSE-KEY-VALUE-PAIRS returns an ASSOCIATION LIST of the name/value pairs.

There are no guarrantees about the order of the associations in the list. As long as names are not duplicated, this is unimportant, but if a name is duplicated, you won't know which of the values associated with that name will appear first in the ASSOCIATION LIST .

To use PARSE-KEY-VALUE-PAIRS in your programs, you might copy-&-paste the "`defun parse-key-value-pairs`" form from that file into your program.

7 Lots of Examples

Let's start by using the defaults for all the keyword arguments. That allows us to parse key/value pairs of the form "a=1;b=2;c=3". Here is an example:

```
lisp> (with-input-from-string (strm "a=1;b=2;c=3")
        (parse-key-value-pairs strm))
====> (("a" . "1") ("b" . "2") ("c" . "3"))
lisp>
```

Because the value of the WHITE-SPACE keyword argument defaults to the empty list, PARSE-KEY-VALUE-PAIRS will preserve white-space:

```
lisp> (with-input-from-string (strm "a =1;b= 2; c = 3 ")
        (parse-key-value-pairs strm))
====> (("a " . "1") ("b" . " 2") (" c " . " 3 "))
lisp>
```

In this most recent example, notice that there is a space character after the "a". Since PARSE-KEY-VALUE-PAIRS, as we've invoked it here, preserves spaces, that means that the name in the first name/value pair is "a ". There is a space before the 2, so the value of the second pair is " 2". There are spaces before & after the "c", so the name of the third name/value pair is " 2 ". Likewise, there are spaces around the 3, so the value of the third pair is " 3 ".

If we wanted to strip those spaces from the names & values, we would use the WHITE-SPACE keyword argument, like this:

```
lisp> (with-input-from-string (strm "a =1;b= 2; c = 3 ")
        (parse-key-value-pairs strm :white-space '(#\Space)))
====> (("a" . "1") ("b" . "2") ("c" . "3"))
lisp>
```

The strings we've been parsing, which have the form "a=1;b=2;c=3" have more or less the form of cookies sent from a web browser to the web server. Here's how we might parse name/value pairs embedded in an URL[3]. To do that, we simply need to use the PAIRSEP keyword argument to tell PARSE-KEY-VALUE-PAIRS that "&" is a separator character. Here's an example:

```
lisp> (with-input-from-string (strm "a=1&b=2&c=3")
        (parse-key-value-pairs strm :pairsep '(#\&)))
====> (("a" . "1") ("b" . "2") ("c" . "3"))
lisp>
```

Somewhere in an old edition of my favorite book about HTML[1], there is mention that using "&" to separate the pairs in an URL can cause confusion because "&" has special meaning within HTML. So, says the book, the W3C committee recommends that web browsers & servers allow semicolons (;) to separate pairs. I don't know how serious the recommendation was, & I'm pretty sure that no browsers & servers use semicolons to separate pairs. By using the PAIRSEP keyword argument, we could use PARSE-KEY-VALUE-PAIRS to allow both ampersands (&) and semicolons (;) to separate name/value pairs embedded in an URL:

```
lisp> (with-input-from-string
          (strm "action=view;user=joe&session=123")
        (parse-key-value-pairs strm :pairsep '(#\& #\;)))
====> (("action" . "view") ("user" . "joe")
       ("session" . "123"))
lisp>
```

Notice from this most recent example that a single input uses both of the name/value pair separators that we specified. In other words, it uses a semicolon

[3]In my strongly held but humorous opinion, URL should be pronounced like "earl". Otherwise FORTRAN must be pronounced "eff oh are tee are ay in" & BASIC must be pronounced "bee ay ess eye see".

(;) to separate the first two pairs, & an ampersand (&) to separate the last two. Because we specified both ampersand & semicolon with the PAIRSEP keyword argument, PARSE-KEY-VALUE-PAIRS treats ampersand & semicolon identically.

We can use PARSE-KEY-VALUE-PAIRS to extract the associations from a file that contains name/value pairs each on a line by itself. In other words, if we have a file called `input.txt` whose contents are like this:

```
apple:red
orange:orange
asparagus:gus
you-should:read-more-poetry
```

... We could apply PARSE-KEY-VALUE-PAIRS to that file like this:

```
lisp> (with-open-file (strm "input.txt")
        (parse-key-value-pairs strm
          :pairsep '(#\Newline)
          :nvsep '(#\:)
          :white-space '(#\Space)))
====> (("apple" . "red") ("orange" . "orange")
       ("asparagus" . "gus")
       ("you-should" . "read-more-poetry"))
```

Let's look at some unusual situations.

What if the input has duplicate names? In other words, what if the input associates a name with a particular value, then associates the same name with another value? In that case, PARSE-KEY-VALUE-PAIRS should return a valid ASSOCIATION LIST , but you can't predict which of the associations is first. So if you try to fetch the value associated with that name with ASSOC , you don't know which value you will find.

```
lisp> (with-input-from-string (strm "a=1;b=2;c=3;a=4")
        (parse-key-value-pairs strm))
====> (("a" . "1") ("b" . "2") ("c" . "3") ("a" . "4"))
```

In this latest example, the associations in the result have the same order as the associations in the input, but that could change. Like I said, PARSE-KEY-VALUE-PAIRS make not guarrantees about the order of the associations in the result.

Another strange case might be an input that contains pair separators without pairs between them. Let's see what happens.

Here's an input that ends with a separator:

```
lisp> (with-input-from-string (strm "a=1;b=2;c=3;")
        (parse-key-value-pairs strm))
====> (("a" . "1") ("b" . "2") ("c" . "3"))
```

So when the input ends with a separator, that last separator does not create an association.

Here's an input that contains consecutive separators:

```
lisp> (with-input-from-string (strm "a=1;b=2;;c=3;")
        (parse-key-value-pairs strm))
====> (("a" . "1") ("b" . "2") ("c" . "3"))
```

Again, the excess separators do not create associations.

Here's an input that contains an invalid pair. This pair has one token which could be a key or a value, but no name/value separator.

```
lisp> (with-input-from-string (strm "a=1;b;c=3")
        (parse-key-value-pairs strm))
====> (("a" . "1"))
```

We get just the first pair. What happened here is that the invalid pair caused NEXT-PAIR (8.7) to return NIL, which the loop in PARSE-KEY-VALUE-PAIRS interprets as end-of-input. So the loop stops & returns what it has parsed at that point.

8 How It Works

I'll discuss different features of the source code. Why? Partly because how I programmed it might be of use to other programmers. Mostly because I have no doubt that in a few months or a few years or whenever, I myself will look at the source code & have no idea how it works unless I tell myself here & now.

The source code is in the file http://lisp-p.org/pkvpl/pkvpl.lisp.

8.1 Single DEFUN

If you look at the source code, the first think you might notice is that it is a single DEFUN PARSE-KEY-VALUE-PAIRS form containing a bunch of local functions within it. I did as a single DEFUN with a bunch of local functions to avoid polluting the name space. My hope is that wherever PARSE-KEY-VALUE-PAIRS is needed, the programmer can drop the DEFUN PARSE-KEY-VALUE-PAIRS form into her existing source code with minimal risk of interference with other functions & symbols.

8.2 Test Programs

You might also notice that a bunch of test-related definitions follow the DEFUN PARSE-KEY-VALUE-PAIRS. Those are unit tests. To use them, LOAD `pkvpl.lisp` into your Lisp, then evaluate (CHECK). That will run all of the test programs. It prints each test's name before running it. If all the tests works, CHECK returns true. Otherwise, it returns NIL.

My convention is that a test program prints nothing if it works. If it fails, it might print an error message that will help track down the problem later. It is important that successful tests print nothing. If every test prints something & there are a lot of tests, it is difficult to distinguish the true error messages from the unnecessary messages. If tests print messages on failure only, a glance at the screen will reveal whether there were any errors. That's why my test programs print nothing when they succeed.

I usually number the test programs. I could try to give them descriptive names, but that would make for some very long names. Whats more, on many projects, I have literally hundreds of test programs. Giving a descriptive name to each of them would be nealry impossible. It's easier to number them. It doesn't matter what a test program does as long as it works. The only time you need to know what it does is when it fails & you are debugging the problem, & when you do that, you will surely be looking in the source code for the test program.[4]

When you copy PARSE-KEY-VALUE-PAIRS into your program, you do not need the test programs, the CHECK function, the DEFTEST macro, or the *TESTS* variable. To use PARSE-KEY-VALUE-PAIRS in your program, you just need the DEFUN PARSE-KEY-VALUE-PAIRS form.

8.3 Helper Functions

Let's look at those functions which are local to PARSE-KEY-VALUE-PAIRS.

8.4 NEXT-CHAR

NEXT-CHAR returns the next character from the input stream. If there is an error or there is no more input, NEXT-CHAR returns NIL.

It's a trivially simple function. The only form in its body is "`(peek-char nil strm nil nil)`". That's okay because NEXT-CHAR exists for brevity & to help the rest of the code be a little more self-documenting.

8.5 NEXT-TOKEN

If PARSE-KEY-VALUE-PAIRS is a parser, then NEXT-TOKEN is its lexer.

The first thing NEXT-TOKEN does is consume & discard white-space characters. For NEXT-TOKEN, a white-space character is any character in the WHITE-SPACE argument of the enclosing PARSE-KEY-VALUE-PAIRS function.

After discarding white-space characters, NEXT-TOKEN checks the next character.

If the next character is NIL (which isn't a character), we're at end-of-input, SO NEXT-TOKEN returns NIL.

[4] I have encountered two kinds of programmers in my career: Those who implicitly agree with this idea or who come to agree with it after writing a few test programs of their own, & whose who will never, ever understand it, much less agree with it.

If the next character is any of those which separate a name from its value, NEXT-TOKEN returns the symbol PAIRSEP. How does NEXT-TOKEN know what characters may separate names from values? Any character in the list PAIRSEP, which is an argument to the PARSE-KEY-VALUE-PAIRS function, may separate names from values.

If the next character is in the list NVSEP, which is also an argument of PARSE-KEY-VALUE-PAIRS, we have a "name/value separator", so NEXT-TOKEN returns the symbol NVSEP.

Otherwise, NEXT-TOKEN collects characters as long as they are not pair separators, name/value separators, white-space, or end-of-stream. Then it returns the collected characters as a string.

8.6 NEXT-NAME

NEXT-NAME skips over pair separators & returns the next thing which is not a pair separator. Presumably, that next thing is the name in a name/value pair, but I suppose NEXT-NAME could return the symbol NVSEP if the thing we're parsing is malformed.

8.7 NEXT-PAIR

NEXT-PAIR uses NEXT-NAME to skip the crap & return the name of the next name/value pair, NEXT-TOKEN to get the separator, & NEXT-TOKEN again to get the value in the name/value pair. If all three items are as expected, with the name & value being strings & the separator being the symbol NVSEP, it CONSes the name & the value together & returns that.

NEXT-PAIR uses LET* to read the three items before checking that they are what they should be. So if any of the items are not what they should be, the input stream has been modified & further parsing is likely to fail. Then again, if the tokens are not what they should be, the input stream was probably incorrect as far as PARSE-KEY-VALUE-PAIRS is concerned the input was wrong, anyway.

8.8 PARSE-KEY-VALUE-PAIRS itself

Armed with those helper functions, PARSE-KEY-VALUE-PAIRS is about as simple as they come. Check it out:

```
  (loop for pair = (next-pair)
while pair
collect pair)
```

This loop uses NEXT-PAIR to fetch CONSes from the input stream, collects them into a list, & stops when NEXT-PAIR returns NIL at the end of the input. Nothing to it!

A Change Log

2004-Nov-03 Did a little more work.

2004-Nov-07 Finished, I hope.

B Other File Formats

- This document is available in multi-file HTML format at http://lisp-p.org/pkvpl/.
- This document is available in Pointless Document Format (PDF) at http://lisp-p.org/pkvpl/pkvpl.pdf.

References

[1] Chuck Musicano Bill Kennedy. *HTML: The Definitive Guide*. O'Reilly & Associates, Inc., second edition, 1997. ISBN 1-56592-235-2.

[2] Free Software Foundation. Lesser general public license. world wide web. http://www.gnu.org/licenses/licenses.html#LGPL.

[3] ANSI Committee X3J13. Common lisp hyperspec. Xanalys Web site. http://www.lispworks.com/reference/HyperSpec/.