

**ahist - a simple seach history for Acme**

(version 0.4.5)

Alexander Sychev (santucco@gmail.com)

**1. Introduction.** This is an implementation of `ahist` command for `Acme`. It tracks all search requests in `Acme`'s window to a separate window.

**2. Implementation.**

```

// This file is part of ahist
//
// Copyright (c) 2020 Alexander Sychev. All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// * Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above
// copyright notice, this list of conditions and the following disclaimer
// in the documentation and/or other materials provided with the
// distribution.
// * The name of author may not be used to endorse or promote products derived from
// this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
package main
import(
    ⟨Imports 4⟩
)
var(
    ⟨Global variables 5⟩
)
type(
    ⟨Types 44⟩
)

```

**3. Startup.**

```
func main(){
    ⟨Store a name of the program 13⟩
    ⟨Obtaining of id of a window 11⟩
    ⟨Open window w by id 19⟩
    ⟨Change the name of the program in the tag 14⟩
    ⟨Read name of the window 22⟩
    ⟨Start history processing 48⟩
    ⟨Processing window events 18⟩
}
```

**4.**

```
⟨Imports 4⟩ ≡
    "fmt"
    "os"
```

See also sections 9, 17, and 20.

This code is used in section 2.

**5.** Let's define *dbg* flag and will switch it by *ahist +* and *ahist -*.

```
⟨Global variables 5⟩ ≡
    dbg bool
```

See also sections 10, 12, 21, 35, and 45.

This code is used in section 2.

**6.**

```
⟨Switch debug output on 6⟩ ≡
    dbg = true
    debug("debug_has_been_switched_on\n")
```

This code is used in section 27.

**7.**

```
⟨Switch debug output off 7⟩ ≡
    debug("debug_has_been_switched_off\n")
    dbg = false
```

This code is used in section 27.

**8.**

```
func debug(f string, args ...interface{}){
    if dbg {
        fmt.Fprintf(os.Stderr, f, args...)
    }
}
```

**9.**

```
⟨Imports 4⟩ +≡
    "strconv"
```

**10.**

```
⟨Global variables 5⟩ +≡
    id int
```

**11.**

⟨Obtaining of *id* of a window 11⟩ ≡

```
{
  var err error
  id, err = strconv.Atoi(os.Getenv("winid"))
  if err != nil {
    return
  }
}
```

This code is used in section 3.

**12.**

⟨Global variables 5⟩ +≡

```
tagname string
```

**13.**

⟨Store a name of the program 13⟩ ≡

```
tagname = os.Args[0]
if n := strings.LastIndex(tagname, "/"); n != -1 {
  tagname = tagname[n:]
}
debug("tagname:%s\n", tagname)
```

This code is used in section 3.

**14.** We change **ahist** into **-ahist** to add a possibility to switch **ahist** off.

⟨Change the name of the program in the tag 14⟩ ≡

```
{
  del := []string{tagname, "-" + tagname, "-" + tagname + "+", "-" + tagname + "-"}
  add := []string{"-" + tagname}
  changeTag(w, del, add)
}
```

This code is used in section 3.

**15.** On exit we should make an opposite change.

⟨Cleanup 15⟩ ≡

```
{
  del := []string{tagname, "-" + tagname, "-" + tagname + "+", "-" + tagname + "-"}
  add := []string{tagname}
  changeTag(w, del, add)
}
```

See also sections 42 and 46.

This code is used in sections 18 and 27.

**16. Events handling.****17.**

```

⟨ Imports 4 ⟩ +=
    "github.com/santuccio/goacme"

```

**18.**

```

⟨ Processing window events 18 ⟩ ≡
    ⟨ Fix tag of the window 41 ⟩
    for{
        ev, err := w.ReadEvent()
        if err ≠ nil {
            ⟨ Cleanup 15 ⟩
            return
        }
        ⟨ Process main window 23 ⟩
    }

```

This code is used in section 3.

**19.**

```

⟨ Open window w by id 19 ⟩ ≡
    w, err := goacme.Open(id)
    if err ≠ nil {
        debug("cannot open a window with id: %s\n", id, err)
        return
    }
    defer w.Close()

```

This code is used in section 3.

**20.**

```

⟨ Imports 4 ⟩ +=
    "strings"

```

**21.**

```

⟨ Global variables 5 ⟩ +=
    name string

```

**22.**

⟨Read *name* of the window 22⟩ ≡

```

{
  f, err := w.File("tag")
  if err ≠ nil {
    debug("cannot_read_from_'tag'_of_the_window_with_id_%d:_%s\n", id, err)
    return
  }
  if _, err := f.Seek(0, 0); err ≠ nil {
    debug("cannot_seek_to_the_start_'tag'_of_the_window_with_id_%d:_%s\n", id, err)
    return
  }
  var b [1000]byte
  n, err := f.Read(b[:])
  if err ≠ nil {
    debug("cannot_read_tag_of_the_window_with_id_%d:_%s\n", id, err)
    return
  }
  ss := strings.Split(string(b[:n]), " ")
  if len(ss) ≡ 0 {
    return
  }
  name = string(ss[0])
}

```

This code is used in section 3.

**23.**

⟨Process main window 23⟩ ≡

```

⟨Process and continue if it is not Look in any form 24⟩
⟨Process Look 30⟩
⟨Read addr into b, e 38⟩
⟨Show dot 40⟩
⟨Write history 53⟩

```

This code is used in section 18.

**24.**  $b, e$  address pair is taken from the  $ev$  event.

```

⟨Process and continue if it is not Look in any form 24⟩ ≡
  debug("incoming_event:␣%+v\n", ev)
  s := ""
  b := ev.Begin
  e := ev.End
  type_switch:
  switch{
    case ev.Type ≡ goacme.Look | goacme.Tag:
      ⟨Process in case of a request by B3 mouse button in the tag 25⟩
    case ev.Type ≡ goacme.Look:
      ⟨Process in case of a request by B3 command in the body 26⟩
    case ev.Type ≡ goacme.Execute ∨ ev.Type ≡ goacme.Execute | goacme.Tag:
      ⟨Process in case of executing a command in the body or tag 27⟩
    case ev.Type ≡ goacme.Insert ∨ ev.Type ≡ goacme.Delete:
      ⟨Fix tag of the window 41⟩
    continue
  default:
    ⟨Unread event and continue 29⟩
  }

```

This code is used in section 23.

**25.** We take a search string from  $ev$  event and set dot. Also we have to clean  $b, e$  because it is an address in the tag.

```

⟨Process in case of a request by B3 mouse button in the tag 25⟩ ≡
  b, e = 0, 0
  s = ev.Text
  if len(ev.Arg) > 0 {
    s += "␣" + ev.Arg
  }
  ⟨Set addr to dot 32⟩

```

This code is used in section 24.

**26.** We take a search string and address from  $ev$  event.

```

⟨Process in case of a request by B3 command in the body 26⟩ ≡
  s = ev.Text
  if len(ev.Arg) > 0 {
    s += "␣" + ev.Arg
  }
  ⟨Set addr to  $b, e$  34⟩

```

This code is used in section 24.



**27.** For *Look* command we set address and continue processing. *ahist* command we just ignore to avoid duplicates. *-ahist* command makes cleanups and processes to exit. *ahist +* and *ahist -* switch debug output on and off. All other commands are written back to "event" file and **fallthrough** to the next case, where a status of the window is checked.

⟨Process in case of executing a command in the body or tag 27⟩ ≡

```

switch strings.TrimSpace(ev.Text) {
  case "Look":
    ⟨Process in case of executing Look command 28⟩
    break type_switch
  case tagname:
    continue
  case "-" + tagname + "+":
    fallthrough
  case "-" + tagname + "-":
    fallthrough
  case "-" + tagname:
    debug("exiting\n")
    ⟨Cleanup 15⟩
    return
  case tagname + "+":
    ⟨Switch debug output on 6⟩
    continue
  case tagname + "-":
    ⟨Switch debug output off 7⟩
    continue
}
w.UnreadEvent(ev)
fallthrough

```

This code is used in section 24.

**28.** We take a search string from an argument of *Look* command. Current address is set to dot, then *b, e* pair is set to the current address.

⟨Process in case of executing *Look* command 28⟩ ≡

```

s = ev.Arg
⟨Set addr to dot 32⟩
⟨Read addr into b, e 38⟩

```

This code is used in section 27.

**29.**

⟨Unread event and continue 29⟩ ≡

```

w.UnreadEvent(ev)
continue

```

This code is used in sections 24, 32, 33, 34, 37, 38, 39, and 40.

**30.** If the *ev* event contains a search string, use it. Otherwise we should read selected the string from the window's body and read its address into *b, e*.

```

⟨Process Look 30⟩ ≡
{
  if len(s) > 0 {
    ⟨Make a search of s 37⟩
  } else {
    ⟨Look for selected string 31⟩
    ⟨Read addr into b, e 38⟩
  }
}

```

This code is used in section 23.

**31.**

```

⟨Look for selected string 31⟩ ≡
{
  ⟨Read selected string from "xdata" file to s 33⟩
  ⟨Make a search of s 37⟩
}

```

This code is used in section 30.

**32.**

```

⟨Set addr to dot 32⟩ ≡
if w.WriteCtl("addr=dot") ≠ nil {
  ⟨Unread event and continue 29⟩
}
debug("set_addr_to_dot\n")

```

This code is used in sections 25 and 28.

**33.**

```

⟨Read selected string from "xdata" file to s 33⟩ ≡
{
  d, err := w.File("xdata")
  if err ≠ nil {
    debug("cannot_read_from_xdata_of_the_window_with_id_%d:_%s\n", id, err)
    ⟨Unread event and continue 29⟩
  }
  buf := make([]byte, e - b + 1)
  for n, _ := d.Read(buf); n > 0; n, _ = d.Read(buf) {
    s += string(buf[:n])
  }
  debug("read_address_from_xdata_b:_%v, e:_%v\n", b, e)
}

```

This code is used in section 31.

**34.**

```

⟨Set addr to b, e 34⟩ ≡
  if err := w.WriteAddr("#%d, #%d", b, e); err ≠ nil {
    debug("cannot_write_to_'addr'_of_the_window_with_id_%d:_%s\n", id, err)
    ⟨Unread event and continue 29⟩
  }
  debug("set_addr_to_%d,_%d\n", b, e)

```

This code is used in sections 26 and 37.

**35.** We need to store previous history *entry* for the case, when *Look* in a tag is executed but without selected text. In the case a search string is taken from *Acme*. We take it from *lentr*

```

⟨Global variables 5⟩ +=
  lentr entry

```

**36.** Let's add *empty* function for *entry*

```

func (this entry) empty() bool{
  return this.b ≡ this.e
}

```

**37.** Search is processed by writing `"<regex>/"` to `"addr"` file, but before regex-specific symbols of *s* have to be escaped. In the case of empty search string we take it from *lentr*. Also we write the current position with the string to the history to track the search, because it already has a place.

```

⟨Make a search of s 37⟩ ≡
{
  debug("last_entry:_%v\n", lentr)
  if len(s) ≡ 0 {
    if ¬lentr.empty() {
      b = lentr.b
      e = lentr.e
      s = lentr.s
      ⟨Set addr to b, e 34⟩
    }
  } else if b ≠ e {
    lentr = entry{b, e, s}
    ⟨Write history 53⟩
  }
  es := escapeSymbols(s)
  debug("escaped_search_string:_%q\n", es)
  if err := w.WriteAddr("/%s/", es); err ≠ nil {
    debug("cannot_write_to_'addr'_of_the_window_with_id_%d:_%s\n", id, err)
    ⟨Unread event and continue 29⟩
  }
}

```

This code is used in sections 30 and 31.

**38.**

```

⟨ Read addr into b, e 38 ⟩ ≡
  b, e, err = w.ReadAddr()
  if err ≠ nil {
    ⟨ Unread event and continue 29 ⟩
  }
  debug("read_address_b:_%v, _e:_%v\n", b, e)

```

This code is used in sections 23, 28, and 30.

**39.**

```

⟨ Set dot to addr 39 ⟩ ≡
  if w.WriteCtl("dot=addr") ≠ nil {
    debug("cannot_write_to_ 'ctl' _of _the _window _with _id _%d:_%s\n", id, err)
    ⟨ Unread event and continue 29 ⟩
  }
  debug("set_dot_to_addr\n")

```

This code is used in section 40.

**40.**

```

⟨ Show dot 40 ⟩ ≡
  ⟨ Set dot to addr 39 ⟩
  if w.WriteCtl("show") ≠ nil {
    debug("cannot_write_to_ 'ctl' _of _the _window _with _id _%d:_%s\n", id, err)
    ⟨ Unread event and continue 29 ⟩
  }
  debug("show_dot\n")

```

This code is used in section 23.

**41.** Acme does not produce standard commands in case of opened "event" file. So we have to add command "Put" in case of the window is modified and "Undo" and "Redo" commands too.

```

⟨ Fix tag of the window 41 ⟩ ≡
  {
    _, _, _, d, _, _, err := w.ReadCtl()
    if err ≠ nil {
      debug("cannot_read_from_ 'ctl' _of _the _window _with _id _%d:_%s\n", id, err)
    } else {
      debug("dirty:_%v\n", d)
      del := []string{"Put", "Undo", "Redo"}
      var add []string
      if d {
        add = append(add, "Put")
      }
      add = append(add, "Undo", "Redo")
      changeTag(w, del, add)
    }
  }

```

This code is used in sections 18 and 24.

**42.**    Removing added commands on exit

⟨ Cleanup 15 ⟩ +≡

```
{  
  del := append([]string{}, "Put", "Undo", "Redo")  
  changeTag(w, del, nil)  
}
```

**43. Tracking search requests .**

We create a window with history of search requests and make separated goroutine to process events from the window.

**44.**

```

⟨Types 44⟩ ≡
    entry struct{
        b, e int
        s string
    }

```

This code is used in section 2.

**45.** Special *histch* channel is received *entry* to print them in the window

```

⟨Global variables 5⟩ +=
    histch chan entry = make(chan entry)

```

**46.** On exit we should signal the goroutine to stop processing. It is made by closing *histch* channel

```

⟨Cleanup 15⟩ +=
    close(histch)

```

**47.**

```

⟨Variables outside the loop 47⟩ ≡
    var hch ← chan *goacme.Event

```

See also section 49.

This code is used in section 48.

**48.** The goroutine handles two variants of events.

```

⟨Start history processing 48⟩ ≡
    go func(){
        ⟨Variables outside the loop 47⟩
        for{
            select{
                case entr, ok :=← histch:
                    ⟨Process entr entry from histch 50⟩
                case ev, ok :=← hch:
                    ⟨Process ev event from hch event channel of the window 51⟩
            }
        }
    }()

```

This code is used in section 3.

**49.**

```

⟨Variables outside the loop 47⟩ +=
    var h *goacme.Window

```

**50.** Events from *histch* channel is written to the history. Before writing a history entry we look for the address in the history window and write the entry only if it has not been found.

⟨Process *entr* entry from *histch* 50⟩ ≡

```

if ¬ok {
  if h ≠ nil {
    h.Del(true)
    h.Close()
    h = nil
  }
  return
}
⟨Open history window, if it does not exist 54⟩
if h.WriteAddr("/#%d, #d/", entr.b, entr.e) ≠ nil {
  debug("writing_to_the_history_%d,%d\n", entr.b, entr.e)
  h.Write([]byte(fmt.Sprintf("%s: #d, #d%q\n", name, entr.b, entr.e, entr.s)))
  h.WriteCtl("clean")
}
debug("selecting_the_current_position_%d, #d_in_the_history\n", entr.b, entr.e)
es := fmt.Sprintf("#%d, #d", entr.b, entr.e)
⟨Make a selection of the current search request 52⟩

```

This code is used in section 48.

**51.** Event from *hch* channel is checked for a case the channel is close. In the case that means the history window is closed and we clear *h* and *hch*. Otherwise we just write the event back.

⟨Process *ev* event from *hch* event channel of the window 51⟩ ≡

```

if ¬ok {
  debug("history_is_closed\n")
  h.Del(true)
  h.Close()
  h = nil
  hch = nil
  continue
}
h.UnreadEvent(ev)
if ev.Type ≡ goacme.Look {
  debug("incoming_event:_%+v\n", ev)
  debug("selecting_the_current_position_%q_in_the_history\n", ev.Text)
  es := escapeSymbols(ev.Text)
  ⟨Make a selection of the current search request 52⟩
}

```

This code is used in section 48.

**52.**

⟨Make a selection of the current search request 52⟩ ≡

```

if err := h.WriteAddr("/%s/-+", es); err ≠ nil {
  debug("writing_of_addr_failed:_%s\n", err)
} else if err := h.WriteCtl("dot=addr\nshow"); err ≠ nil {
  debug("writing_of_ctl_failed:_%s\n", err)
}

```

This code is used in sections 50 and 51.

53.

⟨ Write history 53 ⟩ ≡  
`debug("request_to_store_a_history: %v, %v %q\n", b, e, s)`  
`histch ← entry{b: b, e: e, s: s}`

This code is used in sections 23 and 37.

54. If the history window  $h$  does not exist, we create it.

⟨ Open history window, if it does not exist 54 ⟩ ≡  
`if h ≡ nil {`  
`var err error`  
`if h, err = goacme.New(); err ≠ nil {`  
`return`  
`}`  
`h.WriteCtl("name_%s", name + "+History")`  
`if hch, err = h.EventChannel(1, goacme.AllTypes); err ≠ nil {`  
`return`  
`}`  
`}`

This code is used in section 50.

55. *changeTag* function.

We read the tag of  $w$  window, remove all commands from *del* list and add all commands from *add* list.

```
func changeTag(w *goacme.Window, del []string, add []string){
  if add ≡ nil ∧ del ≡ nil {
    return
  }
  ⟨ Read a tag of  $w$  into  $s$  56 ⟩
  ⟨ Split tag into  $tag$  fields after the pipe symbol 57 ⟩
  ⟨ Compose  $newtag$  58 ⟩
  ⟨ Clear the tag and write  $newtag$  to the tag 60 ⟩
}
```

56.

⟨ Read a tag of  $w$  into  $s$  56 ⟩ ≡  
`f, err := w.File("tag")`  
`if err ≠ nil {`  
`debug("cannot_read_from 'tag' of the window with id %d: %s\n", id, err)`  
`return`  
`}`  
`if _, err := f.Seek(0,0); err ≠ nil {`  
`debug("cannot_seek_to the start 'tag' of the window with id %d: %s\n", id, err)`  
`return`  
`}`  
`var b [1000]byte`  
`n, err := f.Read(b[:])`  
`if err ≠ nil {`  
`debug("cannot_read_tag of the window with id %d: %s\n", id, err)`  
`return`  
`}`  
`s := string(b[:n])`

This code is used in section 55.



**57.**

⟨ Split tag into *tag* fields after the pipe symbol 57 ⟩ ≡

```

if n = strings.LastIndex(s, "|"); n ≡ -1 {
    n = 0
} else {
    n++
}
tag := strings.Fields(s[n:])

```

This code is used in section 55.

**58.**

⟨ Compose *newtag* 58 ⟩ ≡

⟨ Every part is contained in *del* we remove from *tag* 59 ⟩

```

newtag := append([]string{" "}, add ...)
newtag = append(newtag, tag ...)

```

This code is used in section 55.

**59.**

⟨ Every part is contained in *del* we remove from *tag* 59 ⟩ ≡

```

for _, v := range del {
    for i := 0; i < len(tag); {
        if tag[i] ≠ v {
            i++
            continue
        }
        copy(tag[i:], tag[i + 1:])
        tag = tag[:len(tag) - 1]
    }
}

```

This code is used in section 58.

**60.**

⟨ Clear the tag and write *newtag* to the tag 60 ⟩ ≡

```

s = strings.Join(newtag, " ")
if err := w.WriteCtl("cleartag"); err ≠ nil {
    debug("cannot_clear_tag_of_the_window_with_id_%d:_%s\n", id, err)
    return
}
if _, err := f.Write([]byte(s)); err ≠ nil {
    debug("cannot_write_tag_of_the_window_with_id_%d:_%s\n", id, err)
    return
}

```

This code is used in section 55.

61.

```

func escapeSymbols(s string) (es string){
    for _,v := range s {
        if strings.ContainsRune("|\\[/[].*?$",v) {
            es += "\\\"
        }
        es += string(v)
    }
    return
}

```

*add*: 14, 15, 41, 55, 58.*addr*: 37.*ahist*: 5, 27.*AllTypes*: 54.*Arg*: 25, 26, 28.*Args*: 13.*args*: 8.*Atoi*: 11.*Begin*: 24.*buf*: 33.*changeTag*: 14, 15, 41, 42, 55.*Close*: 19, 50, 51.*ContainsRune*: 61.*dbg*: 5, 6, 7, 8.*debug*: 6, 7, 8, 13, 19, 22, 24, 27, 32, 33, 34, 37, 38, 39, 40, 41, 50, 51, 52, 53, 56, 60.*del*: 14, 15, 41, 42, 55, 59.*Del*: 50, 51.*Delete*: 24.*empty*: 36, 37.*End*: 24.*entr*: 48, 50.*entry*: 35, 36, 37, 44, 45, 53.*err*: 11, 18, 19, 22, 33, 34, 37, 38, 39, 40, 41, 52, 54, 56, 60.*es*: 37, 50, 51, 52, 61.*escapeSymbols*: 37, 51, 61.*ev*: 18, 24, 25, 26, 27, 28, 29, 30, 48, 51.*event*: 27, 41.*Event*: 47.*EventChannel*: 54.*Execute*: 24.*Fields*: 57.*File*: 22, 33, 56.*fmt*: 4, 8, 50.*Fprintf*: 8.*Getenv*: 11.*goacme*: 17, 19, 24, 47, 49, 51, 54, 55.*hch*: 47, 48, 51, 54.*histch*: 45, 46, 48, 50, 53.*id*: 10, 11, 19, 22, 33, 34, 37, 39, 40, 41, 56, 60.*Insert*: 24.*Join*: 60.*LastIndex*: 13, 57.*lentr*: 35, 37.*Look*: 24, 27, 28, 35, 51.*main*: 2, 3.*name*: 21, 22, 50, 54.*New*: 54.*newtag*: 58, 60.*ok*: 48, 50, 51.*Open*: 19.*os*: 4, 8, 11, 13.*Put*: 41.*Read*: 22, 33, 56.*ReadAddr*: 38.*ReadCtl*: 41.*ReadEvent*: 18.*Redo*: 41.*Seek*: 22, 56.*Split*: 22.*Sprintf*: 50.*ss*: 22.*Stderr*: 8.*strconv*: 9, 11.*strings*: 20, 13, 22, 27, 57, 60, 61.*tag*: 57, 58, 59.*Tag*: 24.*tagname*: 12, 13, 14, 15, 27.*Text*: 25, 26, 27, 51.*this*: 36.*TrimSpace*: 27.*Type*: 24, 51.*type\_switch*: 24, 27.*Undo*: 41.*UnreadEvent*: 27, 29, 51.*Window*: 49, 55.*Write*: 50, 60.*WriteAddr*: 34, 37, 50, 52.*WriteCtl*: 32, 39, 40, 50, 52, 54, 60.*xdata*: 31, 33.

- ⟨ Change the name of the program in the tag 14 ⟩ Used in section 3.
- ⟨ Cleanup 15, 42, 46 ⟩ Used in sections 18 and 27.
- ⟨ Clear the tag and write *newtag* to the tag 60 ⟩ Used in section 55.
- ⟨ Compose *newtag* 58 ⟩ Used in section 55.
- ⟨ Every part is contained in *del* we remove from *tag* 59 ⟩ Used in section 58.
- ⟨ Fix tag of the window 41 ⟩ Used in sections 18 and 24.
- ⟨ Global variables 5, 10, 12, 21, 35, 45 ⟩ Used in section 2.
- ⟨ Imports 4, 9, 17, 20 ⟩ Used in section 2.
- ⟨ Look for selected string 31 ⟩ Used in section 30.
- ⟨ Make a search of *s* 37 ⟩ Used in sections 30 and 31.
- ⟨ Make a selection of the current search request 52 ⟩ Used in sections 50 and 51.
- ⟨ Obtaining of *id* of a window 11 ⟩ Used in section 3.
- ⟨ Open history window, if it does not exist 54 ⟩ Used in section 50.
- ⟨ Open window *w* by *id* 19 ⟩ Used in section 3.
- ⟨ Process and continue if it is not *Look* in any form 24 ⟩ Used in section 23.
- ⟨ Process in case of a request by B3 command in the body 26 ⟩ Used in section 24.
- ⟨ Process in case of a request by B3 mouse button in the tag 25 ⟩ Used in section 24.
- ⟨ Process in case of executing a command in the body or tag 27 ⟩ Used in section 24.
- ⟨ Process in case of executing *Look* command 28 ⟩ Used in section 27.
- ⟨ Process main window 23 ⟩ Used in section 18.
- ⟨ Process *Look* 30 ⟩ Used in section 23.
- ⟨ Process *entr* entry from *histch* 50 ⟩ Used in section 48.
- ⟨ Process *ev* event from *hch* event channel of the window 51 ⟩ Used in section 48.
- ⟨ Processing window events 18 ⟩ Used in section 3.
- ⟨ Read a tag of *w* into *s* 56 ⟩ Used in section 55.
- ⟨ Read *addr* into *b, e* 38 ⟩ Used in sections 23, 28, and 30.
- ⟨ Read selected string from "*xdata*" file to *s* 33 ⟩ Used in section 31.
- ⟨ Read *name* of the window 22 ⟩ Used in section 3.
- ⟨ Set *addr* to *dot* 32 ⟩ Used in sections 25 and 28.
- ⟨ Set *addr* to *b, e* 34 ⟩ Used in sections 26 and 37.
- ⟨ Set *dot* to *addr* 39 ⟩ Used in section 40.
- ⟨ Show *dot* 40 ⟩ Used in section 23.
- ⟨ Split tag into *tag* fields after the pipe symbol 57 ⟩ Used in section 55.
- ⟨ Start history processing 48 ⟩ Used in section 3.
- ⟨ Store a name of the program 13 ⟩ Used in section 3.
- ⟨ Switch debug output off 7 ⟩ Used in section 27.
- ⟨ Switch debug output on 6 ⟩ Used in section 27.
- ⟨ Types 44 ⟩ Used in section 2.
- ⟨ Unread event and continue 29 ⟩ Used in sections 24, 32, 33, 34, 37, 38, 39, and 40.
- ⟨ Variables outside the loop 47, 49 ⟩ Used in section 48.
- ⟨ Write history 53 ⟩ Used in sections 23 and 37.

# ahist - a simple seach history for Acme

(version 0.4.5)

	Section	Page
<b>Introduction</b> .....	<a href="#">1</a>	2
<b>Implementation</b> .....	<a href="#">2</a>	3
Startup .....	<a href="#">3</a>	4
Events handling .....	<a href="#">16</a>	6
Tracking search requests .....	<a href="#">43</a>	14

Copyright © 2020 Alexander Sychev. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The name of author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.