

`ahist` - a simple seach history for Acme

(Version 0.1)

Alexander Sychev (santuccio@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 version 0.1
//
// 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 7⟩
)
type(
    ⟨Types 38⟩
)

```

3. Startup.

```

func main(){
    ⟨Store a name of the program 10⟩
    ⟨Obtaining of id of a window 8⟩
    ⟨Open window w by id 16⟩
    ⟨Change the name of the program in the tag 11⟩
    ⟨Read name of the window 19⟩
    ⟨Start history processing 42⟩
    ⟨Processing window events 15⟩
}

```

4.

```

⟨Imports 4⟩ ≡
    "fmt"
    "os"

```

See also sections 6, 14, and 17.

This code is used in section 2.

5.

```

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

```

6.

```

⟨Imports 4⟩ +≡
    "strconv"

```

7.

```

⟨Global variables 7⟩ ≡
    id int

```

See also sections 9, 18, and 39.

This code is used in section 2.

8.

```

⟨Obtaining of id of a window 8⟩ ≡
{
    var err error
    id, err = strconv.Atoi(os.Getenv("winid"))
    if err ≠ nil {
        return
    }
}

```

This code is used in section 3.

9.

```

⟨Global variables 7⟩ +≡
    tagname string

```

10.

⟨ Store a name of the program 10 ⟩ ≡

```

    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.

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

⟨ Change the name of the program in the tag 11 ⟩ ≡

```

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

```

This code is used in section 3.

12. On exit we should make an opposite change.

⟨ Cleanup 12 ⟩ ≡

```

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

```

See also sections 36 and 40.

This code is used in sections 15 and 24.

13. Events handling.**14.**

```

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

```

15.

```

⟨ Processing window events 15 ⟩ ≡
    for{
        ev, err := w.ReadEvent()
        if err ≠ nil {
            ⟨ Cleanup 12 ⟩
            return
        }
        ⟨ Process main window 20 ⟩
    }

```

This code is used in section 3.

16.

```

⟨ Open window w by id 16 ⟩ ≡
    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.

17.

```

⟨ Imports 4 ⟩ +=
    "strings"

```

18.

```

⟨ Global variables 7 ⟩ +=
    name string

```

19.

⟨Read *name* of the window 19⟩ ≡

```
{
  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.

20.

⟨Process main window 20⟩ ≡

```
⟨Process and continue if it is not Look in any form 21⟩
⟨Process Look 26⟩
⟨Read addr into b, e 32⟩
⟨Show dot 34⟩
⟨Write history 46⟩
```

This code is used in section 15.

21.

```

⟨Process and continue if it is not Look in any form 21⟩ ≡
  debug("ev:␣%#v\n", ev)
  s := ""
  type_switch:
  switch{
    case ev.Type ≡ goacme.Look | goacme.Tag:
      ⟨Process in case of a request by B3 mouse button in the tag 22⟩
    case ev.Type ≡ goacme.Look:
      ⟨Process in case of a request by B3 command in the body 23⟩
    case ev.Type ≡ goacme.Execute ∨ ev.Type ≡ goacme.Execute | goacme.Tag:
      ⟨Process in case of executing a command in the body or tag 24⟩
    case ev.Type ≡ goacme.Insert ∨ ev.Type ≡ goacme.Delete:
      ⟨Fix tag of the window 35⟩
    continue
  default:
    ⟨Unread event and continue 25⟩
  }

```

This code is used in section 20.

22. We take a search string from *ev* event and set dot

```

⟨Process in case of a request by B3 mouse button in the tag 22⟩ ≡
  s = ev.Text
  if len(ev.Arg))0 {
    s += "␣" + ev.Arg
  }
  ⟨Set addr to dot 28⟩

```

This code is used in section 21.

23. We take a search string and address from *ev* event. Then we write the current position with the string to the history to track the search, because it already has a place.

```

⟨Process in case of a request by B3 command in the body 23⟩ ≡
  s = ev.Text
  if len(ev.Arg))0 {
    s += "␣" + ev.Arg
  }
  b := ev.Begin
  e := ev.End
  ⟨Write history 46⟩
  ⟨Set addr to b, e 30⟩

```

This code is used in section 21.

24. 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. 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 24⟩ ≡

```
switch ev.Text {
  case "Look":
    s = ev.Arg
    ⟨Set addr to dot 28⟩
    break type_switch
  case tagname:
    continue
  case "-" + tagname:
    ⟨Cleanup 12⟩
    return
}
```

w.UnreadEvent(ev)
fallthrough

This code is used in section 21.

25.

⟨Unread event and continue 25⟩ ≡

```
w.UnreadEvent(ev)
continue
```

This code is used in sections 21, 28, 29, 30, 31, 32, 33, and 34.

26. If the *ev* event contains a search string, use it. Otherwise we should read selected the string from the window's body.

⟨Process *Look* 26⟩ ≡

```
{
  ⟨Read addr into b, e 32⟩
  if len(s)>0 {
    ⟨Make a search of s 31⟩
  } else {
    ⟨Look for selected string 27⟩
  }
}
```

This code is used in section 20.

27.

⟨Look for selected string 27⟩ ≡

```
{
  ⟨Read selected string from "xdata" file to s 29⟩
  ⟨Make a search of s 31⟩
}
```

This code is used in section 26.

28.

```

⟨ Set addr to dot 28 ⟩ ≡
  if w.WriteCtl("addr=dot") ≠ nil {
    ⟨ Unread event and continue 25 ⟩
  }
  debug("set_addr_to_dot\n")

```

This code is used in sections 22 and 24.

29.

```

⟨ Read selected string from "xdata" file to s 29 ⟩ ≡
{
  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 25 ⟩
  }
  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 27.

30.

```

⟨ Set addr to b, e 30 ⟩ ≡
  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 25 ⟩
  }
  debug("set_addr_to_%d, %d\n", b, e)

```

This code is used in section 23.

31. Search is processed by writing `"/<regex>/"` to "addr" file, but before regex-specific symbols of *s* have to be escaped

```

⟨ Make a search of s 31 ⟩ ≡
{
  es := ""
  for _, v := range s {
    if strings.ContainsRune("|\\\/[].+?()*~$", v) {
      es += "\\\"
    }
    es += string(v)
  }
  debug("es:_%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 25 ⟩
  }
}

```

This code is used in sections 26 and 27.

32.

```

⟨ Read addr into b, e 32 ⟩ ≡
  b, e, err := w.ReadAddr()
  if err ≠ nil {
    ⟨ Unread event and continue 25 ⟩
  }
  debug("read_address_b:_%v,e:_%v\n", b, e)

```

This code is used in sections 20 and 26.

33.

```

⟨ Set dot to addr 33 ⟩ ≡
  if w.WriteCtl("dot=addr\nshow") ≠ nil {
    debug("cannot_write_to_ctl'of_the_window_with_id_%d:_%s\n", id, err)
    ⟨ Unread event and continue 25 ⟩
  }
  debug("set_dot_to_addr\n")

```

This code is used in section 34.

34.

```

⟨ Show dot 34 ⟩ ≡
  ⟨ Set dot to addr 33 ⟩
  if w.WriteCtl("show") ≠ nil {
    debug("cannot_write_to_ctl'of_the_window_with_id_%d:_%s\n", id, err)
    ⟨ Unread event and continue 25 ⟩
  }
  debug("show_dot\n")

```

This code is used in section 20.

35. 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 35 ⟩ ≡
  {
    →, →, →, →, d, →, →, →, err := w.ReadCtl()
    if err ≠ nil {
      debug("cannot_read_from_ctl'of_the_window_with_id_%d:_%s\n", id, err)
      continue
    }
    debug("dirty:_%v\n", d)
    var add, del []string
    if d {
      add = append(add, "Put")
    } else {
      del = append(del, "Put")
    }
    add = append(add, "Undo", "Redo")
    changeTag(w, del, add)
  }

```

This code is used in section 21.

36. Removing added commands on exit

⟨ Cleanup 12 ⟩ +≡

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

37. Tracking search requests .

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

38.

```

⟨Types 38⟩ ≡
    entry struct{
        b, e int
        s string
    }

```

This code is used in section 2.

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

```

⟨Global variables 7⟩ +=
    histch chan entry = make(chan entry)

```

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

```

⟨Cleanup 12⟩ +=
    close(histch)

```

41.

```

⟨Variables outside the loop 41⟩ ≡
    var hch ← chan *goacme.Event

```

See also sections 43 and 47.

This code is used in section 42.

42. The goroutine handles two variants of events.

```

⟨Start history processing 42⟩ ≡
    go func(){
        ⟨Variables outside the loop 41⟩
        for{
            select{
                case entr, ok :=← histch:
                    ⟨Process entr entry from histch 44⟩
                case ev, ok :=← hch:
                    ⟨Process ev event from hch event channel of the window 45⟩
            }
        }
    }()

```

This code is used in section 3.

43.

```

⟨Variables outside the loop 41⟩ +=
    var h *goacme.Window

```

44. Events from *histch* channel is written to the history.

⟨Process *entr* entry from *histch* 44⟩ ≡

```

if  $\neg ok$  {
  if  $h \neq \text{nil}$  {
    h.Del(true)
    h.Close()
     $h = \text{nil}$ 
  }
  return
}
⟨Open history window, if it does not exist 48⟩
if  $ee, ok := \text{history}[entr.b]; ok \wedge ee \equiv entr.e$  {
  continue
}
 $\text{history}[entr.b] = entr.e$ 
debug("writing to the history %d,%d\n", entr.b, entr.e)
h.Write([]byte(fmt.Sprintf("s: %d, %d\n", name, entr.b, entr.e, entr.s)))
h.WriteCtl("clean")

```

This code is used in section 42.

45. 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*, *hch* and *history*. Otherwise we just write the event back.

⟨Process *ev* event from *hch* event channel of the window 45⟩ ≡

```

if  $\neg ok$  {
  debug("history is closed\n")
  h.Del(true)
  h.Close()
   $h = \text{nil}$ 
   $hch = \text{nil}$ 
   $history = \text{nil}$ 
  continue
}
h.UnreadEvent(ev)

```

This code is used in section 42.

46.

⟨Write history 46⟩ ≡

```

 $histch \leftarrow \text{entry}\{b: b, e: e, s: s\}$ 

```

This code is used in sections 20 and 23.

47.

⟨Variables outside the loop 41⟩ +≡

```

var history map[int]int

```

48. If the history window h does not exist, we create it and (re)create *history* map too.

```

⟨ Open history window, if it does not exist 48 ⟩ ≡
  if  $h \equiv \text{nil}$  {
    var  $err$  error
    if  $h, err = goacme.New(); err \neq \text{nil}$  {
      return
    }
     $h.WriteCtl("name\_%s", name + "+History")$ 
    if  $hch, err = h.EventChannel(1, goacme.AllTypes); err \neq \text{nil}$  {
      return
    }
     $history = \text{make}(\text{map}[\text{int}]\text{int})$ 
  }

```

This code is used in section 44.

49. *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  $add \equiv \text{nil} \wedge del \equiv \text{nil}$  {
    return
  }
  ⟨ Read a tag of  $w$  into  $s$  50 ⟩
  ⟨ Split tag into  $tag$  fields after the pipe symbol 51 ⟩
  ⟨ Compose  $newtag$  52 ⟩
  ⟨ Clear the tag and write  $newtag$  to the tag 55 ⟩
}

```

50.

```

⟨ Read a tag of  $w$  into  $s$  50 ⟩ ≡
   $f, err := w.File("tag")$ 
  if  $err \neq \text{nil}$  {
     $debug("cannot\_read\_from\_tag\_of\_the\_window\_with\_id\_%d:\_%s\n", id, err)$ 
    return
  }
  if  $\_, err := f.Seek(0, 0); err \neq \text{nil}$  {
     $debug("cannot\_seek\_to\_the\_start\_tag\_of\_the\_window\_with\_id\_%d:\_%s\n", id, err)$ 
    return
  }
  var  $b$  [1000]byte
   $n, err := f.Read(b[:])$ 
  if  $err \neq \text{nil}$  {
     $debug("cannot\_read\_tag\_of\_the\_window\_with\_id\_%d:\_%s\n", id, err)$ 
    return
  }
   $s := \text{string}(b[:n])$ 

```

This code is used in section 49.

51.

⟨ Split tag into *tag* fields after the pipe symbol 51 ⟩ ≡

```

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

```

This code is used in section 49.

52.

⟨ Compose *newtag* 52 ⟩ ≡

```

newtag := append([]string{}, "")
⟨ Every part is contained in del we remove from tag 53 ⟩
⟨ Every part is contained in add we remove from tag 54 ⟩
newtag = append(newtag, add ...)
newtag = append(newtag, tag ...)

```

This code is used in section 49.

53.

⟨ Every part is contained in *del* we remove from *tag* 53 ⟩ ≡

```

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 52.

54.

⟨ Every part is contained in *add* we remove from *tag* 54 ⟩ ≡

```

for _, v := range add {
    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 52.

55.

```

⟨ Clear the tag and write newtag to the tag 55 ⟩ ≡
  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 49.

add: 11, 12, 35, 49, 52, 54.

addr: 31.

ahist: 24.

AllTypes: 48.

Arg: 22, 23, 24.

Args: 10.

args: 5.

Atoi: 8.

Begin: 23.

buf: 29.

changeTag: 11, 12, 35, 36, 49.

Close: 16, 44, 45.

ContainsRune: 31.

debug: 5, 10, 16, 19, 21, 28, 29, 30, 31, 32, 33, 34, 35, 44, 45, 50, 55.

del: 11, 12, 35, 36, 49, 53.

Del: 44, 45.

Delete: 21.

ee: 44.

End: 23.

entr: 42, 44.

entry: 38, 39, 46.

err: 8, 15, 16, 19, 29, 30, 31, 32, 33, 34, 35, 48, 50, 55.

es: 31.

ev: 15, 21, 22, 23, 24, 25, 26, 42, 45.

event: 24, 35.

Event: 41.

EventChannel: 48.

Execute: 21.

File: 19, 29, 50.

fmt: 4, 44.

Getenv: 8.

goacme: 14, 16, 21, 41, 43, 48, 49.

hch: 41, 42, 45, 48.

histch: 39, 40, 42, 44, 46.

history: 44, 45, 47, 48.

id: 7, 8, 16, 19, 29, 30, 31, 33, 34, 35, 50, 55.

Insert: 21.

Join: 55.

LastIndex: 10, 51.

Look: 21, 24.

main: 2, 3.

name: 18, 19, 44, 48.

New: 48.

newtag: 52, 55.

ok: 42, 44, 45.

Open: 16.

os: 4, 8, 10.

Put: 35.

Read: 19, 29, 50.

ReadAddr: 32.

ReadCtl: 35.

ReadEvent: 15.

Redo: 35.

Seek: 19, 50.

Split: 19, 51.

Sprintf: 44.

ss: 19.

strconv: 6, 8.

strings: 17, 10, 19, 31, 51, 55.

tag: 51, 52, 53, 54.

Tag: 21.

tagname: 9, 10, 11, 12, 24.

Text: 22, 23, 24.

TrimLeft: 51.

Type: 21.

type_switch: 21, 24.

Undo: 35.

UnreadEvent: 24, 25, 45.

Window: 43, 49.

Write: 44, 55.

WriteAddr: 30, 31.

WriteCtl: 28, 33, 34, 44, 48, 55.

xdata: 27, 29.

- ⟨ Change the name of the program in the tag 11 ⟩ Used in section 3.
- ⟨ Cleanup 12, 36, 40 ⟩ Used in sections 15 and 24.
- ⟨ Clear the tag and write *newtag* to the tag 55 ⟩ Used in section 49.
- ⟨ Compose *newtag* 52 ⟩ Used in section 49.
- ⟨ Every part is contained in *add* we remove from *tag* 54 ⟩ Used in section 52.
- ⟨ Every part is contained in *del* we remove from *tag* 53 ⟩ Used in section 52.
- ⟨ Fix tag of the window 35 ⟩ Used in section 21.
- ⟨ Global variables 7, 9, 18, 39 ⟩ Used in section 2.
- ⟨ Imports 4, 6, 14, 17 ⟩ Used in section 2.
- ⟨ Look for selected string 27 ⟩ Used in section 26.
- ⟨ Make a search of *s* 31 ⟩ Used in sections 26 and 27.
- ⟨ Obtaining of *id* of a window 8 ⟩ Used in section 3.
- ⟨ Open history window, if it does not exist 48 ⟩ Used in section 44.
- ⟨ Open window *w* by *id* 16 ⟩ Used in section 3.
- ⟨ Process and continue if it is not *Look* in any form 21 ⟩ Used in section 20.
- ⟨ Process in case of a request by B3 command in the body 23 ⟩ Used in section 21.
- ⟨ Process in case of a request by B3 mouse button in the tag 22 ⟩ Used in section 21.
- ⟨ Process in case of executing a command in the body or tag 24 ⟩ Used in section 21.
- ⟨ Process main window 20 ⟩ Used in section 15.
- ⟨ Process *Look* 26 ⟩ Used in section 20.
- ⟨ Process *entr* entry from *histch* 44 ⟩ Used in section 42.
- ⟨ Process *ev* event from *hch* event channel of the window 45 ⟩ Used in section 42.
- ⟨ Processing window events 15 ⟩ Used in section 3.
- ⟨ Read a tag of *w* into *s* 50 ⟩ Used in section 49.
- ⟨ Read addr into *b, e* 32 ⟩ Used in sections 20 and 26.
- ⟨ Read selected string from "**xdata**" file to *s* 29 ⟩ Used in section 27.
- ⟨ Read *name* of the window 19 ⟩ Used in section 3.
- ⟨ Set addr to dot 28 ⟩ Used in sections 22 and 24.
- ⟨ Set addr to *b, e* 30 ⟩ Used in section 23.
- ⟨ Set dot to addr 33 ⟩ Used in section 34.
- ⟨ Show dot 34 ⟩ Used in section 20.
- ⟨ Split tag into *tag* fields after the pipe symbol 51 ⟩ Used in section 49.
- ⟨ Start history processing 42 ⟩ Used in section 3.
- ⟨ Store a name of the program 10 ⟩ Used in section 3.
- ⟨ Types 38 ⟩ Used in section 2.
- ⟨ Unread event and continue 25 ⟩ Used in sections 21, 28, 29, 30, 31, 32, 33, and 34.
- ⟨ Variables outside the loop 41, 43, 47 ⟩ Used in section 42.
- ⟨ Write history 46 ⟩ Used in sections 20 and 23.

ahist - a simple search history for Acme

(version 0.1)

| | Section | Page |
|--------------------------------|--------------------|------|
| Introduction | 1 | 2 |
| Implementation | 2 | 3 |
| Startup | 3 | 4 |
| Events handling | 13 | 6 |
| Tracking search requests | 37 | 13 |

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.