ahist - a simple seach history for Acme

(version 0.3)

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 43⟩
)
```

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 47 ⟩
    ⟨ 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, 34, and 44.

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 41 and 45.

This code is used in sections 18 and 27.

16. Events handling.

17.

⟨ Imports 4 ⟩ +≡
```
"github.com/santucco/goacme"
```

18.

⟨ Processing window events 18 ⟩ ≡
```
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␣%d:␣%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 29 ⟩
  ⟨ Read addr into b, e 37 ⟩
  ⟨ Show dot 39 ⟩
  ⟨ Write history 51 ⟩
```

This code is used in section 18.

24.

```
⟨ Process and continue if it is not Look in any form 24 ⟩ ≡
  debug("incoming␣event:␣%+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 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 40 ⟩
      continue
    default:
      ⟨ Unread event and continue 28 ⟩
  }
```

This code is used in section 23.

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

```
⟨ Process in case of a request by B3 mouse button in the tag 25 ⟩ ≡
  s = ev.Text
  if len(ev.Arg)⟩0 {
    s += "␣" + ev.Arg
  }
  ⟨ Set addr to dot 31 ⟩
```

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
  }
  b := ev.Begin
  e := ev.End
  ⟨ Set addr to b, e 33 ⟩
```

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":
      s = ev.Arg
      ⟨ Set addr to dot 31 ⟩
      break type_switch
    case tagname:
      continue
    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.

```
⟨ Unread event and continue 28 ⟩ ≡
  w.UnreadEvent(ev)
  continue
```

This code is used in sections 24, 31, 32, 33, 36, 37, 38, and 39.

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

```
⟨ Process Look 29 ⟩ ≡
  {
    ⟨ Read addr into b, e 37 ⟩
    if len(s)⟩0 {
      ⟨ Make a search of s 36 ⟩
    } else {
      ⟨ Look for selected string 30 ⟩
    }
  }
```

This code is used in section 23.

30.

⟨ Look for selected string 30 ⟩ ≡
```
{
  ⟨ Read selected string from "xdata" file to s 32 ⟩
  ⟨ Make a search of s 36 ⟩
}
```
This code is used in section 29.

31.

⟨ Set addr to dot 31 ⟩ ≡
```
if w.WriteCtl("addr=dot") ≠ nil {
  ⟨ Unread event and continue 28 ⟩
}
debug("set␣addr␣to␣dot\n")
```
This code is used in sections 25 and 27.

32.

⟨ Read selected string from "xdata" file to s 32 ⟩ ≡
```
{
  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 28 ⟩
  }
  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 30.

33.

⟨ Set addr to b, e 33 ⟩ ≡
```
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 28 ⟩
}
debug("set␣addr␣to␣%d,␣%d\n", b, e)
```
This code is used in sections 26 and 36.

34. We need to story 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
```

35. Let's add *empty* function for *entry*
```
func (this entry) empty() bool{
  return this.b ≡ this.e
}
```

36. 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 36 ⟩ ≡
  {
    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 33 ⟩
      }
    } else if b ≠ e {
      lentr = entry{b, e, s}
      ⟨ Write history 51 ⟩
    }
    es := ""
    for _, v := range s {
      if strings.ContainsRune("|\\/[].+?()*^$", v) {
        es += "\\"
      }
      es += string(v)
    }
    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 28 ⟩
    }
  }
```

This code is used in sections 29 and 30.

37.

```
⟨ Read addr into b, e 37 ⟩ ≡
  b, e, err := w.ReadAddr()
  if err ≠ nil {
    ⟨ Unread event and continue 28 ⟩
  }
  debug("read address b: %v, e: %v\n", b, e)
```

This code is used in sections 23 and 29.

38.

```
⟨ Set dot to addr 38 ⟩ ≡
  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 28 ⟩
  }
  debug("set dot to addr\n")
```

This code is used in section 39.

39.

```
⟨ Show dot 39 ⟩ ≡
  ⟨ Set dot to addr 38 ⟩
  if w.WriteCtl("show") ≠ nil {
    debug("cannot write to 'ctl' of the window with id %d: %s\n", id, err)
    ⟨ Unread event and continue 28 ⟩
  }
  debug("show dot\n")
```

This code is used in section 23.

40. `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 40 ⟩ ≡
  {
    _, _, _, _, 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)
    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 section 24.

41. Removing added commands on exit

```
⟨ Cleanup 15 ⟩ +≡
  {
    del := append([]string{}, "Put", "Undo", "Redo")
    changeTag(w, del, nil)
  }
```

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

43.
```
⟨ Types 43 ⟩ ≡
  entry struct{
    b, e int
    s string
  }
```
This code is used in section 2.

44. Special *histch* channel is received *entry* to print them in the window
```
⟨ Global variables 5 ⟩ +≡
  histch chan entry = make(chan entry)
```

45. On exit we should signal the goroutine to stop processing. It is made by closing *histch* channel
```
⟨ Cleanup 15 ⟩ +≡
  close(histch)
```

46.
```
⟨ Variables outside the loop 46 ⟩ ≡
  var hch ←chan ∗goacme.Event
```
See also sections 48 and 52.
This code is used in section 47.

47. The goroutine handles two variants of events.
```
⟨ Start history processing 47 ⟩ ≡
  go func(){
    ⟨ Variables outside the loop 46 ⟩
    for{
      select{
        case entr, ok :=← histch:
          ⟨ Process entr entry from histch 49 ⟩
        case ev, ok :=← hch:
          ⟨ Process ev event from hch event channel of the window 50 ⟩
      }
    }
  }()
```
This code is used in section 3.

48.
```
⟨ Variables outside the loop 46 ⟩ +≡
  var h ∗goacme.Window
```

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

```
⟨ Process entr entry from histch 49 ⟩ ≡
  if ¬ok {
    if h ≠ nil {
      h.Del(true)
      h.Close()
      h = nil
    }
    return
  }
  ⟨ Open history window, if it does not exist 53 ⟩
  if ee, ok := history[entr.b]; ok ∧ ee ≡ entr.e {
    continue
  }
  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 %q\n", name, entr.b, entr.e, entr.s)))
  h.WriteCtl("clean")
```

This code is used in section 47.

50. 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 50 ⟩ ≡
  if ¬ok {
    debug("history is closed\n")
    h.Del(true)
    h.Close()
    h = nil
    hch = nil
    history = nil
    continue
  }
  h.UnreadEvent(ev)
```

This code is used in section 47.

51.

```
⟨ Write history 51 ⟩ ≡
  debug("request to store a history: %v,%v %q\n", b, e, s)
  histch ← entr{b: b, e: e, s: s}
```

This code is used in sections 23 and 36.

52.

```
⟨ Variables outside the loop 46 ⟩ +≡
  var history map[int]int
```

53. 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 53 ⟩ ≡
  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
    }
    history = make(map[int]int)
  }
```

This code is used in section 49.

54. *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 55 ⟩
  ⟨ Split tag into tag fields after the pipe symbol 56 ⟩
  ⟨ Compose newtag 57 ⟩
  ⟨ Clear the tag and write newtag to the tag 59 ⟩
}
```

55.

```
⟨ Read a tag of w into s 55 ⟩ ≡
  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 54.

56.

⟨ Split tag into *tag* fields after the pipe symbol 56 ⟩ ≡

```
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 54.

57.

⟨ Compose *newtag* 57 ⟩ ≡

```
newtag := append([]string{}, "")
⟨ Every part is contained in del we remove from tag 58 ⟩
newtag = append(newtag, add ...)
newtag = append(newtag, tag ...)
```

This code is used in section 54.

58.

⟨ Every part is contained in *del* we remove from *tag* 58 ⟩ ≡

```
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 57.

59.

⟨ Clear the tag and write *newtag* to the tag 59 ⟩ ≡

```
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 54.

⟨ Change the name of the program in the tag 14 ⟩ Used in section 3.
⟨ Cleanup 15, 41, 45 ⟩ Used in sections 18 and 27.
⟨ Clear the tag and write *newtag* to the tag 59 ⟩ Used in section 54.
⟨ Compose *newtag* 57 ⟩ Used in section 54.
⟨ Every part is contained in *del* we remove from *tag* 58 ⟩ Used in section 57.
⟨ Fix tag of the window 40 ⟩ Used in section 24.
⟨ Global variables 5, 10, 12, 21, 34, 44 ⟩ Used in section 2.
⟨ Imports 4, 9, 17, 20 ⟩ Used in section 2.
⟨ Look for selected string 30 ⟩ Used in section 29.
⟨ Make a search of *s* 36 ⟩ Used in sections 29 and 30.
⟨ Obtaining of *id* of a window 11 ⟩ Used in section 3.
⟨ Open history window, if it does not exist 53 ⟩ Used in section 49.
⟨ 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 main window 23 ⟩ Used in section 18.
⟨ Process *Look* 29 ⟩ Used in section 23.
⟨ Process *entr* entry from *histch* 49 ⟩ Used in section 47.
⟨ Process *ev* event from *hch* event channel of the window 50 ⟩ Used in section 47.
⟨ Processing window events 18 ⟩ Used in section 3.
⟨ Read a tag of *w* into *s* 55 ⟩ Used in section 54.
⟨ Read addr into b, e 37 ⟩ Used in sections 23 and 29.
⟨ Read selected string from `"xdata"` file to *s* 32 ⟩ Used in section 30.
⟨ Read *name* of the window 22 ⟩ Used in section 3.
⟨ Set addr to dot 31 ⟩ Used in sections 25 and 27.
⟨ Set addr to b, e 33 ⟩ Used in sections 26 and 36.
⟨ Set dot to addr 38 ⟩ Used in section 39.
⟨ Show dot 39 ⟩ Used in section 23.
⟨ Split tag into *tag* fields after the pipe symbol 56 ⟩ Used in section 54.
⟨ Start history processing 47 ⟩ 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 43 ⟩ Used in section 2.
⟨ Unread event and continue 28 ⟩ Used in sections 24, 31, 32, 33, 36, 37, 38, and 39.
⟨ Variables outside the loop 46, 48, 52 ⟩ Used in section 47.
⟨ Write history 51 ⟩ Used in sections 23 and 36.

ahist - a simple search history for Acme

(version 0.2)