External application interface in AlphaTcl

Frédéric Boulanger

November 9, 2001

Abstract

This file constitutes the literate programming source for the api code of AlphaTcl. This code allows to define APIs for external services, and to declare implementations of these APIs with applications.

Contents

1 Introduction **2**

2 Usage **2**

2.1 Basic functionalities 2

2.2 More about APIs 4

3 The implementation **5**

4 Examples **14**

Conventions used in this paper

In syntax descriptions, a `typewriter` font is used for explicit text. A named syntactic unit is written as ⟨*unit*⟩. In the special but very common case that the syntactic unit is precisely a word for Tcl, it is instead written as {*word*}, i.e., with braces instead of angle brackets. Optional and repeated elements in syntax descriptions are denoted as in regular expressions, using question marks, asterisks, and plus signs, e.g.

`set` {*var-name*} {*value*}$^{?}$
`list` {*item*}*
`append` {*var-name*} {*string*}$^{+}$

Parentheses can be used to group syntax elements, e.g.

`return` (`-code` {*code*})$^{?}$ {*string*}

The same conventions are used for specifying the structure of lists.

1 Introduction

The api code provides the minimal features that allow AlphaTcl to drive external applications through well defined interfaces (APIs).

An API describes a service which may depend on parameters. For instance, we can consider a `openURL` service which displays an URL. Its behavior depends obviously on the URL we want to display, so this service will have a `url` parameter.

There are generally several means to provide the service described by an API: we can use any browser to display an URL. However, each browser has its own syntax for performing the requested action.

For each possible implementation of an API, we need to describe how to build the request for the application from the parameters of the API.

Once an API has been declared and implementations of this API have been registered, we should be able to choose an implementation and to invoke the API. The choice of an implementation may be driven by personal taste or by the availability of some applications on a given platform.

2 Usage

2.1 Basic functionalities

`api::declare` (proc) An API is declared with :

`api::declare` {*api name*} {*description*} {*param*}*

where {*api name*} is the name of the new API, {*description*} is a textual description of the role of the API, and each {*param*} is either the name of a parameter or a list containing the name of a parameter and its default value.

For instance, the following code:

```
api::declare viewURL "Display an URL" url
```

declares an API named `viewURL` which takes a `url` parameter and provides a service described by the sentence "`Display an URL`"; while:

```
api::declare TeX "Typeset a file with TeX" file {format latex}
```

declares an API named `TeX` which takes a `file` and a `format` and provides a service described by the sentence "`Typeset a file with TeX`". If the `format` parameter is not given a value when the API is invoked, `latex` will be used as its value.

`api::registerApp` (proc) Once an API is declared, we can register applications that implement this API with :

`api::registerApp` {*api name*} {*app name*} {*app sig*} {*script*}

{*api name*} is the name of the API which is implemented by the application;

{*app name*} is the name of the application which implements the API. This is a symbolic name which is only meant to be used by human beings and may be different from the real program name;

{*app sig*} is the signature of the application. In *Alpha* on Mac OS, this is the 4-character creator code of the application and is used to launch it. On other operating systems, this may be the name of the interpreter that will understand the script;

{*script*} is the driver which translates the API request into something that the application understands. This script can use the names of the parameters of the API as variables and will be processed by `eval`. Its first parameter is always `apiTarget` and its value identifies the target application. `apiTarget` may be used in the driver to communicate with the application.

To register Internet Explorer as an implementation of the `viewURL` API on Mac OS, we could write:

```
api::registerApp viewURL Explorer MSIE {
  AEBuild $apiTarget WWW! OURL ---- [tclAE::build::TEXT $url]
}
```

The script uses the `OURL` Apple Event from the `WWW!` suite to ask Internet Explorer to display the URL. `$url` will expand to the value of the `url` parameter of the API when it is invoked. `apiTarget` will expand to `'MSIE'` here since the application was registered by its signature.

Of course, it is possible to register several applications for one API. If we want to use the Mac OS Help Viewer for `viewURL` (this will work only for `file::` URLs), we can write:

```
api::registerApp viewURL AppleHelp hbwr {
  AEBuild $apiTarget GURL GURL ---- [tclAE::build::TEXT $url]
}
```

`api::registerAppPath` (proc)

If several applications with the same creator code exist, it is possible to chose which one to register as an implementation of an API with :

`api::registerAppPath` {*api name*} {*app name*} {*app sig*} {*app path*} {*script*}

{*api name*} is the name of the API which is implemented by the application;

{*app name*} is the name of the application which implements the API. This is a symbolic name which is only meant to be used by human beings and may be different from the real program name;

{*app sig*} is the signature of the application. In *Alpha* on Mac OS, this is the 4-character creator code of the application and is used to launch it. On other operating systems, this may be the name of the interpreter that will understand the script.

With this procedure, since the path to the application is given, the signature can be empty on Mac OS because it won't be used to launch the application.

{*script*} is the driver which translates the API request into something that the application understands. This script can use the names of the parameters of the API as variables and will be processed by `eval`. It can also use the `apiTarget` parameter, which will expand to the short name of the application in the driver script.

`api::chooseAppFor` (proc) To chose an implementation among the declared ones for an API, we use :

`api::chooseAppFor` {*api name*} {*app name*}

where {*api name*} is the name of the API, and {*app name*} is the name of the application, as declared with `api::registerApp`.

For instance, to choose Internet Explorer as the implementation of `viewURL`, we write:

```
api::chooseAppFor viewURL Explorer
```

`api::invoke` (proc) To request a service from an API, we use :

`api::invoke` ⟨*interaction*⟩ {*api name*} {*param*}*

⟨*interaction*⟩ tells whether we want the user to be able to interact with the application or not. `-foreground` indicates a possible interaction, and the application will be brought to front. `-background` deny the possibility to interact with the application, which will be launched in the background;

{*api name*} is the name of the API to invoke;

{*param*} is a pair (not a list) of items, the first being the name of a parameter of the API (it can be prefixed by a '-' sign to make it clear that it is a parameter specification), and the second being the value of this parameter for this invocation.

Thus, to display the `http://wwwsi.supelec.fr/` URL with the current implementation of `viewURL`, we write:

```
api::invoke -foreground viewURL -url http://wwwsi.supelec.fr/
```

2.2 More about APIs

The api code defines other commands to work with APIs. It is possible to suppress an
`api::forget` (proc) API with :

`api::forget` {*api name*}

suppresses the API named {*api name*} and all its implementations. Forgetting an API that does not exist causes no harm.

When you declare an API, it is first forgotten so that if this API already existed, its implementations are forgotten. This is necessary to avoid that former implementations be invoked and try to use parameters that do not exist any longer in the new API.

`api::getCurrentAppFor` (proc) To know which application currently implements an API, use :

```
api::getCurrentAppFor {api name}
```

which yields the name of the application that was chosen to implement the {*api name*} API. This name is one of the {*app name*} used in the `api::registerApp` calls for this API, or is empty if no application is chosen to implement the API.

`api::getSignatureFor` (proc)

If you want to know the signature of an application which is used as an implementation of an API, use `api::getSignatureFor` {*app name*}. The result will be the signature of {*app name*} or an empty string if {*app name*} is not registered to an API, or it it was registered by path with an empty signature.

`api::describe` (proc)

Last, you can get information about an API with :

```
api::describe {api name}
```

which returns an empty list if the API {*api name*} is unknown, or a list containing the following items:

0. the name of the API;
1. the list of the parameters of the API, with default value when applicable (the parameter is then a two element list, the first element being the name of the parameter and second its default value);
2. the textual description of the API;
* as many items as there are implementations of the API, each item being a list composed of:
    0. the name of the application;
    1. the signature of the application;
    2. the path of the application;
    3. the script that drives the application according to the parameters of the API.

3 The implementation

All the api code belongs to the `api` namespace;

```
1 ⟨*tcl⟩
2 namespace eval api {}
3
```

`api::declare` (proc)

The `api::declare` procedure declares a new API. Each API has a name, a textual description and a set of formal parameter names.

```
4 proc api::declare {apiname description args} {
```

{*apiname*} is the name of the new API, {*description*} is some text that describes what this API is for, and the remaining arguments ⟨*args*⟩ are the names of the parameters of the API. Each item in ⟨*args*⟩ may be either a single {*parameter name*} or a two item list {{*parameter name*} {*default value*}}.

`api::description` (`{`*api name*`}`)

The descriptions of the APIs are store in the `api::description` global array which is indexed by the name of the API. The description of the new API is stored in this array.

```
5  global api::description
6
7  set api::description($apiname) $description
8
```

An API is a Tcl procedure which passes its arguments to its current implementation. We build the effective argument list to pass to the implementation from the parameter list of the API by keeping only the name of the parameter (the first item) and prefixing it with a $ sign:

```
9  set arglist ""
10 foreach arg $args {
11   append arglist " \$[lindex $arg 0]"
12 }
13
```

Then, we build the body of the procedure which codes the API. This procedure must determine which implementation to use and call the corresponding procedure with the argument list computed above.

`api::currentApp` (`{`*api name*`}`)

The current implementation of an API is stored in the `api::currentApp` global array. The name of the procedure that converts the invocation of the API into something the application understands is:

`api::`⟨*app name*⟩`implementationOf`⟨*api name*⟩

```
14 set proctext "\
15   global api::currentApp;\
16   set implementation \
17    \"api::\[set api::currentApp($apiname)\]implementationOf$apiname\";\
18   \$implementation $arglist;\
19 "
20
```

After this, `proctext` contains the body of a procedure which computes the name of the procedure that implements the API in `implementation`, and then calls this procedure with the arguments of the API.

Since an API may be declared several times, and that each new declaration may have a different parameter list or semantics, we must invalidate previous implementations of the API when we declare it again. This is done with `api::forget`:

```
21 api::forget $apiname
22
```

The last thing to do is to declare the procedure that codes this API. Its name is `api::apiNamed`⟨*api name*⟩, it takes the parameters of the API as arguments, and its body has been computed in `proctext`:

```
23 proc api::apiNamed$apiname $args $proctext
```

```
24 }
25
```

`api::describe` (proc) The `api::describe` procedure returns the description of an API in the form of an empty list if the API does not exist or a list containing the following items:

0 the name of the API;

1 the list of the parameters of the API, with default value when applicable (the parameter is then a two element list, the first element being the name of the parameter and second its default value);

2 the textual description of the API;

* as many items as there are implementations of the API, each item being a list composed of:

  0 the name of the application;

  1 the signature of the application;

  2 the path to the application;

  3 the script that drives the application according to the parameters of the API.

```
26 proc api::describe {apiname} {
27   global api::applicationSigs
28   global api::applicationPath
29   global api::description
30
```

`api::applicationSigs` ({*app name*}) The `api::applicationSigs` global array contains the signature of the applications that implement at least an API. It is indexed by the name of the applications as declared with `api::registerApp`.

`api::applicationPath` ({*app name*}) The `api::applicationPath` global array contains the paths of the applications which are registered by path for at least an API. It is indexed by the name of the applications as declared with `api::registerAppPath`.

The `api::description` global array contains the description of the APIs, as seen in 3.

The description starts as an empty list, and is returned as is if the API does not exists:

```
31   set desc [list]
32   if {[info commands api::apiNamed$apiname] == ""} {
33     return $desc
34   }
35
```

To test the existence of an API, we just have to see if a procedure named `api::apiNamed`⟨*api name*⟩ exists.

Once we know the API exists, we can add its name as the first item of its description:

```
36   lappend desc $apiname
37
```

Then, we must build the list of parameters of the API, with their default value if any. Since `info args` omits the defaults values, we have to look for them with `info default` and put them back in the list:

```
38  set sublist [list]
39  foreach arg [info args api::apiNamed$apiname] {
40    if {[info default api::apiNamed$apiname $arg defval]} {
41      lappend sublist [list $arg $defval]
42    } else {
43      lappend sublist $arg
44    }
45  }
46  lappend desc $sublist
47
```

The next item is the textual description of the API:

```
48  lappend desc [set api::description($apiname)]
49
```

Then comes the items that describe the implementations of the API. We look for such implementation by asking for all procedures whose name is structured as `api::`⟨*app name*⟩`implementationOf$apiname` since this is the canonical form chosen for the implementation procedures:

```
50  set implementations [info commands api::*implementationOf$apiname]
51  foreach imp $implementations {
52    regexp -- "api::(.*)implementationOf$apiname" $imp dummy appName
53    set sublist $appName
54    lappend sublist [set api::applicationSigs($appName)]
55    if {[info exists api::applicationPath($appName)]} {
56      lappend sublist [set api::applicationPath($appName)]
57    } else {
58      if {![catch {set appPath\
                    [nameFromAppl [set api::applicationSigs($appName)]]}]} {
59        lappend sublist $appPath
60      } else {
61        lappend sublist {}
62      }
63    }
64    lappend sublist [info body $imp]
65    lappend desc $sublist
66  }
67  return $desc
68 }
69
```

`api::getCurrentAppFor` (proc)

The `api::currentAppFor` procedure returns the name of the application that is used as the current implementation of an API.

```
70 proc api::getCurrentAppFor {apiname} {
71   global api::currentApp
72
73   if {[info commands api::apiNamed$apiname] == ""} {
74     error "Undeclared API \"$apiname\""
```

```
75   }
```

It is an error to call this procedure on an API that does not exist. If the API exists and no application has been chosen for it yet, we return an empty string:

```
76   if {[info exists api::currentApp($apiname)]} {
77     return [set api::currentApp($apiname)]
78   } else {
79     return ""
80   }
81 }
82 

83 # Return the signature used by the APIs for an application
84 proc api::getSignatureFor {appname} {
85   global api::applicationSigs
86 
87   if {[info exists api::applicationSigs($appname)]} {
88     return [set api::applicationSigs($appname)]
89   } else {
90     return ""
91   }
92 }
93 
94 # Return the application path used by the APIs for an application
95 proc api::getPathFor {appname} {
96   global api::applicationPath
97   global api::applicationSigs
98 
99   if {[info exists api::applicationPath($appname)]} {
100    return [set api::applicationSigs($appname)]
101  } else {
102    if {![catch {
103      set appPath [nameFromAppl [set api::applicationSigs($appname)]]
104    }} {
105      return $appPath
106    } else {
107      return ""
108    }
109  }
110 }
111 
112 # Forget an API.
113 proc api::forget {apiname} {
114   global api::currentApp
115 
116   # Forget any implementation of a possible previous version of the API
117   set implementations [info commands api::*implementationOf$apiname]
118   foreach imp $implementations {
119     rename $imp api::unimplemented
120   }
121 
122   # Forget the API proc
123   if {[info commands api::apiNamed$apiname] != ""} {
```

```
    rename api::apiNamed$apiname api::undefined
  }

  # Forget the current app for this API
  if {[info exists api::currentApp($apiname)]} {
    unset api::currentApp($apiname)
  }
}

# Register an application for an API. This means that the
# application supports the API, and a script is given to
# turn the invocation of the API into a run of the application.
#
# For instance, if application CMacTeX of signature "*XeT" supports the
# "tex" API, it can be registered with the following call:
#
#  api::registerApp tex CMacTeX *XeT {
#    buildNewCMacTeXAE "tex $options &$format" $filename
#  }
#
# The last argument says that to implement the "tex" API with CMacTeX, one
# should execute "buildNewCMacTeXAE ...". The variable used in the script
# have the same name as the formal arguments of the API and their value
# will be the actual arguments when the script is executed.
proc api::registerApp {apiname appName appSig script} {
  global api::applicationSigs

  if {[info commands api::apiNamed$apiname] == ""} {
    error "Undeclared API \"$apiname\""
  }

  # Remember the signature of this application to be able to
  # launch it later.
  set api::applicationSigs($appName) $appSig

  # Declare the proc that uses "script" to implement the API
  proc api::${appName}implementationOf$apiname \
   [linsert [info args api::apiNamed$apiname] 0 apiTarget] \
   $script

}

# Register an application for an API. This means that the
# application supports the API, and a script is given to
# turn the invocation of the API into a run of the application.
#
# For instance, if application CMacTeX of signature "*XeT" supports the
# "tex" API, it can be registered with the following call:
#
#  api::registerAppPath tex CMacTeX *XeT "MyDisk:CMacTeX 4.0:CMacTeX" {
#    buildNewCMacTeXAE "tex $options &$format" $filename
#  }
#
# The last argument says that to implement the "tex" API with CMacTeX, one
```

```
178 # should execute "buildNewCMacTeXAE ...". The variable used in the script
179 # have the same name as the formal arguments of the API and their value
180 # will be the actual arguments when the script is executed.
181 proc api::registerAppPath {apiname appName appSig appPath script} {
182   global api::applicationSigs
183   global api::applicationPath
184 
185   if {[info commands api::apiNamed$apiname] == ""} {
186     error "Undeclared API \"$apiname\""
187   }
188 
189   # Remember the path to this application to be able to
190   # launch it later.
191   set api::applicationPath($appName) $appPath
192 
193   # Remember the signature of this application.
194   set api::applicationSigs($appName) $appSig
195 
196   # Declare the proc that uses "script" to implement the API
197   proc api::${appName}implementationOf$apiname \
198    [linsert [info args api::apiNamed$apiname] 0 apiTarget \
199    $script
200 
201 }
202 
203 # Choose an application to implement an API.
204 # Since several applications may support one API, one must choose an
205 # application before being able to invoke an API.
206 # Exemple:    api::chooseAppFor tex CMacTeX
207 proc api::chooseAppFor {apiname appName} {
208   global api::currentApp
209 
210   if {[info commands api::apiNamed$apiname] == ""} {
211     error "Undeclared API \"$apiname\""
212   }
213   if {[info commands api::${appName}implementationOf$apiname] == ""} {
214     error "Application $appName does not support API \"$apiname\""
215   }
216 
217   set api::currentApp($apiname) $appName
218 }
219 
220 # Invoke an API, e.g. ask for something to be done using the current
221 # application for the API. The "interact" parameter should be set to
222 #   '-foreground'
223 # if the user is expected to interact with the application. It should be
224 #   '-background'
225 # if the user shouldn't interact with the application.
226 # The arguments must be in the form "-key value", where "key" is the
227 # name of a formal parameter of the API, and value is the value of this
228 # parameter:
229 #
230 #   api::invoke -foreground tex -filename hello.tex -format latex\
                                                              -options --src
```

```
#
# The '-' in front of the keys can be omitted, so:
#
#   api::invoke -foreground tex filename hello.tex format latex options\
                                                                    --src
#
# is valid too.
#
# If a parameter is not set, its default value (as declared in the API) is
# used. If no default value is declared, this is an error (just like for a
# proc).
proc api::invoke {interact apiname args} {
  global api::currentApp
  global api::applicationSigs
  global api::applicationPath

  if {[info commands api::apiNamed$apiname] == ""} {
    error "Undeclared API \"$apiname\""
  }
  if {[info commands\
     api::[set api::currentApp($apiname)]implementationOf$apiname] == ""} {
    error "Application \"$appName\" does not support API \"$apiname\""
  }

  switch -- "$interact" {
    "-foreground"  {set interaction 1}
    "-background"  {set interaction 0}
    default {
      error "Unknown value \"$interact\" for 'interact' parameter"
    }
  }

  # Get the list of formal parameters from the proc of the API
  set formalargs [info args api::apiNamed$apiname]

  # Find the value of the effective parameter by scanning the arguments
  set nargs [llength $args]
  for {set i 0} {$i < $nargs} {incr i} {
    # Ignore a leading '-' for convenience
    regexp -- {-?(.*)} [lindex $args $i] dummy argname
    # Search the name of the formal parameter in the list
    set argidx [lsearch $formalargs $argname]
    if {$argidx == -1} {
      error "Invalid argument name \"$argname\" for API \"$apiname\""
    }
    # Next argument is the value for $argname
    incr i
    if {$i < $nargs} {
      set effectiveargs($argname) [lindex $args $i]
    } else {
      error "Missing value after \"$argname\" key"
    }
  }

```

```
283  # Use default value for unspecified parameters
284  foreach arg $formalargs {
285    if {![info exists effectiveargs($arg)]} {
286      if {[info default api::apiNamed$apiname $arg defval]} {
287        set effectiveargs($arg) $defval
288      } else {
289        error "No default value for unspecified parameter \"$arg\""
290      }
291    }
292  }
293
294  set arglist [list]
295  foreach arg $formalargs {
296    lappend arglist $effectiveargs($arg)
297  }
298
299  if\
       {[info exists api::applicationPath([set api::currentApp($apiname)])]}\
                                                                              {
300    set appPath\
                  [set api::applicationPath([set api::currentApp($apiname)])]
301    set apiTarget "[file tail $appPath]"
302  } else {
303    set appPath {}
304    set apiTarget  "'[string trim\
           [set api::applicationSigs([set api::currentApp($apiname)])] \']'"
306  }
307  api::executeApp \
308    "[set api::applicationSigs([set api::currentApp($apiname)])]" \
309    $appPath \
310    $interaction \
311    [eval list \
312      api::[set api::currentApp($apiname)]implementationOf$apiname \
313      $apiTarget $arglist\
314    ]
315 }
316
317 # Execute an application.
318 # On MacOS, appsig is the creator of the application to launch.
319 # On other systems, I use it to give the name of the shell that should be
320 # used to interpret the commands. I hope Unix and Windows users will be
321 # able to adapt this to their needs.
322 #
323 # interact says if the user should interact with the application.
324 # If so, the application is brought to the foreground.
325 #
326 # command is the command to execute. It may refer to the formal
327 # parameters of the API which have been evaluated by api::invoke.
328 proc api::executeApp {appsig appPath interact command} {
329   global alpha::platform
330
331   if {${alpha::platform} == "alpha"} {
332     if {$appPath == ""} {
333       # We only have a signature for this application.
```

```
      # Check if it is running.
      app::ensureRunning $appsig $interact
    } else {
      # We have a path, so we must launch *this* application
      # even if another with the same signature is running.
      set cmd "launch"
      if {$interact} {
        lappend cmd "-f"
      }
      lappend cmd $appPath
      if {[catch {eval $cmd}]} {
        error "Could not launch $appPath"
      }
    }
    # Interpret command, which should send an AppleEvent
    eval $command
  } else {
    # Interpret command which is a Unix like command line.
    # Exact role of appSig and appPath on Unix-like systems?
    if {$interact} {
      eval exec $appsig $command
    } else {
      eval exec $appsig $command &
    }
  }
}
```
⟨/tcl⟩

4 Examples

```
⟨*examples⟩
###### API examples #######
## "tex" is an API that uses a filename, a format and options.
## format defaults to "tex " and options to nothing
api::declare tex "Typeset a file with TeX" filename {format latex}\
                                                              {options ""}
## same for "pdftex"
api::declare pdftex "Typeset a file with pdfTeX" filename\
                                          {format pdflatex} {options ""}
## "dvips" is an API that uses a filename and options
api::declare dvips "Convert a DVI file to PostScript" filename\
                                                              {options ""}
## "ps2pdf is an API that uses a filename a paper size and options
## The default value of papersize is "a4"
api::declare ps2pdf "Distill a PostScript file into PDF" filename\
                                              {options ""} {papersize a4}

if {${alpha::platform} == "alpha"} {
  # Mac OS applications
  # Use CMacTeX for tex, pdftex and dvips
  api::registerApp tex CMacTeX *XeT {
```

```
    AEBuild $apiTarget CMTX exec  ----\
            [tclAE::build::TEXT "tex &$format [file tail $filename]"]  dest\
                        [tclAE::build::alis "[file dirname $filename]:"]
  }
  api::registerApp pdftex CMacTeX *XeT {
    AEBuild $apiTarget CMTX exec  ----\
         [tclAE::build::TEXT "pdftex &$format [file tail $filename]"]  dest\
                        [tclAE::build::alis "[file dirname $filename]:"]
  }
  api::registerApp dvips CMacTeX *XeT {
    AEBuild $apiTarget CMTX exec  ----\
                       [tclAE::build::TEXT "dvips $options \"$filename\""]
  }
  # Use macps2pdf for ps2pdf
  api::registerApp ps2pdf MacGhostView TMCA {
    set cmdline ""
    append cmdline "gs $options -dCompatibilityLevel=1.2\
                                                        -dMaxSubsetPct=100"
    append cmdline " -dNOPAUSE -dBATCH -sDEVICE=pdfwrite\
                                                  -sPAPERSIZE=$papersize"
    append cmdline "\
                    -sOutputFile=[file rootname [file tail $filename]].pdf"
    append cmdline " -c save pop -f [file tail $filename]"
    AEBuild $apiTarget CMTX exec \
     ---- [tclAE::build::TEXT $cmdline] \
     dest [tclAE::build::alis "[file dirname $filename]:"]
  }
  # We may also register OzTeX for tex
  api::registerApp tex OzTeX OTEX {
    sendOpenEvent noReply $apiTarget $filename
  }
} else {
  # Windows or Unix applications
  api::registerApp tex miktex csh {
    tex $options &$format $filename
  }
  api::registerApp pdftex miktex csh {
    pdftex $options &pdf$format $filename
  }
  api::registerApp dvips miktex csh {
    dvips $options $filename
  }
  api::registerApp ps2pdf miktex csh {
    gs $options -dCompatibilityLevel=1.2 -dMaxSubsetPct=100 -dNOPAUSE \
     -dBATCH -sDEVICE=pdfwrite -sPAPERSIZE=$papersize \
     -sOutputFile=[file rootname $filename].pdf \
     -c save pop -f $filename
  }
}
#
api::chooseAppFor tex CMacTeX
api::chooseAppFor pdftex CMacTeX
api::chooseAppFor dvips CMacTeX
api::chooseAppFor ps2pdf MacGhostView
```

```
429
430 api::invoke -foreground tex -format latex -filename\
                                                [file join [pwd] essai.tex]
431 api::invoke -foreground dvips -filename [file join [pwd] essai.dvi]
432 api::invoke -foreground ps2pdf -filename [file join [pwd] essai.ps]
433 ⟨/examples⟩
```

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition. By tradition there should also be a lot of numbers in roman that refer to the code lines where the entry is used, but unfortunately I don't have a convenient way of cross-referencing Tcl code to generate that information.

A

`applicationPath` (array), `api` NS
  {*app name*} entries 31
`applicationSigs` (array), `api` NS
  {*app name*} entries 31

C

`chooseAppFor` (proc), `api` NS *4*
`currentApp` (array), `api` NS
  {*api name*} entries 14

D

`declare` (proc), `api` NS *2*, 4
`describe` (proc), `api` NS *5*, 26
`description` (array), `api` NS
  {*api name*} entries 5

F

`forget` (proc), `api` NS *4*

G

`getCurrentAppFor` (proc), `api` NS . . . *4*, 70
`getSignatureFor` (proc), `api` NS *5*

I

`invoke` (proc), `api` NS *4*

R

`registerApp` (proc), `api` NS *2*
`registerAppPath` (proc), `api` NS *3*