Notes for `glide` users
4th edition – January 1989

Colin Runciman
Ian Toyn

Department of Computer Science
University of York

Introduction

This document contains notes for users of the interactive functional programming system `glide` that runs under UNIX. It is intended as a brief outline only. Tutorials in the Glide language are provided by taught courses. Reference information is available in the on-line manual, accessed by the `Help` command.

Getting started

Type `glide` as a shell command. After preliminary announcements, a prompt is given, starting the first cycle of a prompt-command-response loop. The prompt is usually `glide>`, but may change to indicate the presence of errors, as will be explained shortly.

```
$ glide
University of York Glider (built 21 Jan 1989)
Type Help for help.  Please notify any problems by "mail glide".
Towing... releasing tow rope...
glide>
```

There are various commands and responses; these will be described as we go along. To begin with, any expression in the functional language is acceptable as a command. If the expression has a defined value this will be displayed as the response. Usually the display will also be a legal expression (in which no further evaluation is possible); but if the whole result is a string, the displayed response is the corresponding series of characters.

```
glide> tail ["A", "B", "C"]
["B", "C"]
glide> "hack" :: "saw"
hacksaw
glide>
```

Hitting `return` normally indicates completion of a command. Although multi-line commands should be unnecessary, they are possible: precede each `return` by \, the backslash character.

Flocks, primitives and the standard library

Usually, programs require some explicitly defined functions in addition to the primitives. Some or all of these definitions may already have been written by the same or another programmer for a different application. If so, they can often be re-used without copying, because Glide supports the organization of definitions into groups called *flocks*, and allows the programmer to maintain a *search set* indicating which flocks are to be used.

```
glide> Edit Search
...enters an editor...
Search [/usr/glide/examples, listfuns]
...quit editor...
glide>
```

By default, `Edit` puts you into `vi`. If you want to use a different editor, before running `glide` set the environment variable `EDITOR` to that editor's name and export it.

As may be inferred from the above example, flock names are actually the names of UNIX directories, either absolute or relative to the *current flock* (directory). The `Flock` command, similar to UNIX's `cd`, should be used to change the current flock. Each flock has its own search set which always implicitly includes both itself and the flock of primitives `<primitive>`. Whenever an identifier is used in a flock, a definition for it is sought in that flock's search set. If more than one flock in the search set has a definition, `glide` uses type information to choose one. If there is no definition (or more than one) of suitable type `glide` complains.

Several functions are implicitly defined as primitives in `glide`, belonging to the flock `<primitive>`. For example the functions `tail` and `(::)` used earlier are primitive. Appendix A includes a list of all primitives.

The standard library `<standard>` is possibly the first flock you will wish to include in a search set. It contains definitions of frequently useful functions such as `reverse`.

```
glide> reverse ["racket", "hits", "ball"]
["ball", "hits", "racket"]
glide>
```

The actual directory names corresponding to `<standard>`, and other libraries (see on-line manual), may vary between machines; hence the need for a special notation. In all other respects, however, they are just like any other flock.

Creating explicit function definitions

The `Edit` command is also used to make a new definition in the current flock, or to modify an existing one there. Just as flocks correspond to directories, definitions correspond to files. In some cases `glide` maintains shared links to a definition file, and it makes various other assumptions about the files in a flock directory, so do *not* try to alter them using tools other than `glide`. Changing the name of a definition can be done using the `Edit` command (*don't* use `mv`) and definitions can be removed using the `UnDefine` command (*don't* use `rm`). If a new flock is required, one can be created using the command `NewFlock`, analogous to `mkdir`.

```
glide> Edit replicate
...enters an editor...
Define replicate x 0 -> []
   and replicate x (n+1) -> x : replicate x n
...quit editor...
glide> replicate "hello" 3
["hello", "hello", "hello"]
glide>
```

Each new or modified definition is checked for syntax and type errors. If there are any errors in the definition, the prompt changes to `glide E>`. Type `Edit` to edit the offending definition: error reports will have been inserted as comments. You may find it helpful to delete obsolete error reports as you fix the corresponding errors, but in any case `glide` automatically deletes existing error reports each time it reads a definition.

```
glide> Edit numbers
...enters an editor...
Define numbers -> "zero" : [1]
...quit editor...
Type error in /usr/glide/examples/numbers
glide E> Edit
...list entire file...
Define numbers -> "zero" : [1]
--?                          A A
--? A..A @ [num] but context @ [[char]]
...quit editor...
glide E>
```

Note that `Edit` without an argument automatically chooses a definition (if there is one) in which an error has been detected. If you make no change to this definition, a subsequent `Edit` with no argument will choose another erroneous definition (if there is another). Although this selection strategy is often convenient, you can always state explicitly which definition you wish to edit.

Within a single flock, an identifier can be given at most one definition. To determine what identifiers are already defined in the current flock, give the single word command `Define`.

Using the shell

It is sometimes convenient to execute shell commands within the context of `glide`. Following the convention adopted in other UNIX tools, the remainder of a command line starting `!` is passed to the shell.

Reading and writing Files

The primitive function `contents`, applied to a filename, returns the contents of the file as a string. The printed value of an expression *expr* can be directed to a file as follows.

```
[To "filename" (written expr)]
```

In place of `written` *expr* there could be any string-valued expression. The example shows a single item list explicitly expressed at the command level, but lists containing any number of `To` constructions may be built as the result of general computation and will be suitably interpreted. Use `ToEnd` to append to a file. The only other means by which a glide expression can communicate with the file system are the `access` and `stat` primitive functions for inspecting attributes of existing files. It is *not* possible to use the file system as updatable storage within a single computation: `To` and `ToEnd` have no effect until the computation terminates.

Finding out what's going wrong

Sooner or later you will write a program that does not work as you intended. The computational fault will be one of three kinds:

- never reaching any result at all;
- reaching an undefined result;
- reaching a defined but wrong result.

Several commands are useful in various combinations; here are some suggestions to try first in each of the above cases.

An apparently unproductive computation should be interrupted using `^C`. Now give the `Trace` command: `glide` should respond with a schematic source-level view of the current state and derivation of the computation. This may confirm that an unintended infinite process is under way and make plain how it has arisen; but if, after all, the computation looks satisfactory, it can be continued by giving the command `Ok`, and can always be interrupted again later.

Faced with an undefined result, such as `tail []` which the typechecker will not have detected, again use `Trace` to see how it has arisen.

Well-defined but wrong results are often the most tricky to sort out. Try examining functions from the lowest level upward, observing each in turn until a fault shows up.

Finishing

To leave `glide` altogether, respond to the `glide>` prompt with an end-of-file character (probably `^Z` or `^D`).

Reporting problems

Although care has been taken in both design and implementation of `glide`, we are aware of defects and the system is an experimental one. If you suspect a fault, or have a suggestion for improvement, please `mail glide` as in the following example.

```
$ mail glide
I'm sad because glide won't accept my latest super ultra
very high order lazy interactive list-smashing function.
It's defined as follows ...

EOF
$
```

But do talk to someone else about the problem first – is it really a fault in the `glide` system or just an error you have made? Check the bugs entry in the on-line manual to see if the problem is already recognised as a fault.

Appendix A: Summary of Glide Programming Language

These syntactic definitions are written in Wirth's EBNF: each definition is of the form nonterminal = alternatives; the alternatives are separated by | characters; the form "token" is a terminal – the characters are written literally; the form [e] means 0 or 1 occurrences of e (e is optional); the form { e } means 0 or more occurrences (repetitions) of e.

```
expr    = expr expr
        | number | char | string | constructor | variable | "~"
        | expr binop expr
        | "[" [ expr { "," expr } ] "]"
        | expr "where" block
        | "let" block "in" expr
        | expr "@" type
        | "(" binop ")" | "(" "~" ")"
        | "(" expr binop ")" | "(" binop expr ")"
        | "(" expr ")" .

binop   = "|" | "&" | "=" | "<=" | "<" | ">" | ">=" | "~="
        | "::" | ":" | "^" | "+" | "-" | "*" | "/" | "\" | "o" .

pattern = number | char  | string | constructor | variable | "_"
        | "[" [ inner_pattern { "," inner_pattern } ] "]"
        | "(" inner_pattern ")" .

inner_pattern = inner_pattern "^" inner_pattern
        | inner_pattern ":" inner_pattern
        | variable "as" inner_pattern
        | inner_pattern "+" number
        | inner_pattern "@" type
        | pattern { pattern } .

block = [ declaration { "and" declaration } "with" ]
        definition { "and" definition } .

declaration = pattern "@" type .

definition = pattern "->" expr
           | clause { "and" clause } .

clause = variable pattern { pattern } "->" expr .

variable    = -- lower-case letter then letters, digits, _s, and trailing 's.
constructor = -- as variable, but with initial letter upper-case.
number      = -- one or more decimal digit characters (no sign).
char        = -- a character enclosed in single quotes, e.g. 'X'.
string      = -- zero or more chars enclosed in double quotes, e.g. "Boo!"
type        = -- see below.
```

Each character in a character or string literal can be a printable ASCII character other than \, ' and ", or a \ followed by an octal ASCII code-number, or a \ followed by any of n for newline, t for tab, b for backspace, r for return, f for form-feed, \ for \, ' for ', " for ".

Any text following a double dash -- and on the same line is treated as a comment. Although comments may occur almost anywhere within a definition, a recommended comment convention is to place a block of comment lines *before* each definition.

No identifier may appear more than once in the left-hand side of a single definition.

The associativities and precedences of operators are as follows (more tightly binding operators are lower in the table).

```
right     |
right     &
right     ~
non       = ~= <= >= < >
right     ::
right     :
right     ^
left      + -
left      * / \
right     o
left      @
left      function application
```

The remaining primitives use normal notation for application.

`access`	Would input of named file succeed
`chr`	Convert num to char
`consgas`	glide abstract syntax of named constructor
`contents`	Contents of named file
`error`	Display message and halt computation
`flip`	Exchange parameters of a function
`getenv`	The value of an environment variable
`getpid`	Process id of glide session
`head`	Select first item of a list
`if`	Choose one of two expressions based on a Boolean
`left`	Select component of a pair
`length`	Length of a list
`lstat`	Attributes of named symbolic link
`ord`	Convert char to num
`right`	Select component of a pair
`stat`	Attributes of named file
`tail`	Select all but first item of a list
`typegas`	glide abstract syntax of named type
`vargas`	glide abstract syntax of named variable
`writ`	Convert value to string without punctuation
`written`	Convert value to string with punctuation

Should you need to give type declarations, the syntax is as follows.

```
type     = typename argtype { argtype }
         | type "->" type
         | type "^" type
         | argtype .

argtype = typename
         | typevar
         | "[" type "]"
         | "(" type ")" .
```

`typenames` and `typevars` are like `variables`; if the initial letter is followed by only digits, then the name is a type variable, otherwise it denotes a particular type.