The CaPriCon Scripting Language Reference

Marc Coiffier

Contents

All the basic words are described below.

Stack manipulation

The environment of the interpreter consists mostly of a stack of values, that can be manipulated with the following words.

`dup` / `dupn` Duplicates the top element, or the nth top element of the stack.

- `dup` : x ... $\rightarrow$ x x ...
- `dupn` : n $x_0..x_n$ ... $\rightarrow$ x_n $x_0..x_n$ ...

`swap` / `swapn` Swaps the top element of the stack with the second, or the nth element.

- `swap` : x y ... $\rightarrow$ y x ...
- `swapn` : n x $y_0..y_n$ ... $\rightarrow$ y_n $y_0..y_{n-1}$ x ...

`shift` / `shaft` Shifts the nth element towards the top, or shaft the top to the nth place.

- `shift` : $n\ x_1..x_n\ ...\ \rightarrow\ x_n\ x_1..x_{n-1}\ ...$
- `shaft` : $n\ x_1..x_n\ ...\ \rightarrow\ x_2..x_n\ x_1...$

`pop` / `popn` Pops the top element, or the nth top element, off the stack.

- `pop` : $x\ ...\ \rightarrow\ ...$
- `popn` : $n\ x_0..x_n\ ...\ \rightarrow\ x_0..x_{n-1}\ ...$

`clear` Clears the stack.

- `clear` : $...\ \rightarrow$

`stack` / `set-stack` Pushes the current stack, as a list, on top of the current stack. In the second case, sets the top element of the stack as the new stack.

- `stack` : $Stack\ \rightarrow\ [Stack]Stack$
- `set-stack` : $[Stack]...\ \rightarrow\ Stack$

`pick` Picks the i-between-nth element of the stack, and discards all others. Can be useful for implementing arbitrary switch-like control-flow.

- `pick` : $i\ n\ x_0..x_i..x_{n-1}\ ...\ \rightarrow\ x_i\ ...$

Names and variables

`def` Sets the value of a variable.

- `def` : *value name* ... → ... in an environment where *value* is associated with the variable named *name*.

Examples :

```
'x 3 def 'y 7 def
x y x y + y * "(x + y) * y = %v; y = %v ; x = %v" printf
```

(x + y) * y = 70; y = 7 ; x = 3

`$` The inverse of `def`. Given the name of a variable at the top of the stack, this function produces the value of the corresponding variable in the current environment.

- `$` : $name\ ...\ \rightarrow\ \$name\ ...$

`vocabulary` / `set-vocabulary` Pushes the active dictionary, that contains all defined variables, on top of the stack. In the second case, make the top of the stack the current dictionary, redefining all variables at once.

`lookup` A more flexible version of `$`, where the environment is specified explicitly as a second argument (for example, from calling `vocabulary`).

First-class functions

`exec` Executes the value at the top of the stack, as if it were the meaning of a word. To illustrate, given a function, `'f $ exec` is equivalent to `f` itself. That is, evaluating a symbol is no different than looking it up in the current dictionary, and `exec`uting its value.

Lists

`[` Puts a "list beginning" (LB) marker on the stack

- `[` : ... $\rightarrow$ *LB* ...

`]` Creates a list of the elements on the stack until the next "list beginning" marker, and pushes it on the remaining stack.

- `]` : $x_0..x_n$ *LB* ... $\rightarrow$ $[x_n..x_0]$ ...

`each` Iterates over each element of its second argument, pushing it on the stack and running its second argument afterward.

Examples :

```
"Values: " print [ 1 2 3 ] { show pop } each
```

Values: 1 2 3

`range` Create a list of numbers from 0 to $n-1$, n being the top element of the stack.

- `range` : n ... $\rightarrow$ $[0..n-1]$ ...

Simple integer arithmetic

`+`, `-`, `*`, `div`, `mod` Performs the usual binary arithmetic operation on the top two elements of the stack, and replaces them with the result.

`sign` Computes the sign of the top stack element. If the sign is negative, produces -1, if positive produces 1, otherwise produces 0.

Strings

`format` Much like the `sprintf()` function in C, produces a string which may contain textual representations of various other values.

Examples :

```
"Some text" 1 "<p>%v: %s</p>" format show
```

"

1: Some text

"

`to-int` Tries to convert the top stack element to an integer, if possible.

Interacting with the environment

`exit` Exits the interpreter, immediately and unconditionally.

`print` Print the string at the top of the stack into the current document.

`source` Opens an external source file, and pushes a quote on the stack with its contents.

`cache` Given a resource name and a quote, does one of two things :

- if the resource already exists, try to open it as a CaPriCon object, ignoring the quote
- otherwise, run the quote and store its result in the resource for future use

After the builtin has run, the contents of the requested object can be found at the top of the stack.

`redirect` Given a resource name and a quote, executes the quote, redirecting its output to the resource.

String-Indexed Dictionaries

`empty` Pushes the empty dictionary onto the stack.

`insert` Given a dictionary `d`, a key `k` and a value `v`, inserts the value `v` at `k` in `d`, then pushes the result on the stack.

`delete` The reverse of `insert`. Given a dictionary `d` and a key `k`, produce a dictionary `d'` that is identical to `d`, without any association for `k`.

`keys` Given a dictionary `d`, pushes a list of all of `d`'s keys onto the stack.

Constructing typed terms

`universe` Produces a universe.

- `universe` : i ... $\rightarrow$ Set_i ...

`variable` Given a variable name, that exists in the current type context, produces that variable.

- `variable` : $name\ ...\ \rightarrow\ var(name)\ ...$

`apply` Given a function `f`, and a term `x`, produces the term `f x`.

- `apply` : $x\ f\ ...\ \rightarrow\ (f\ x)\ ...$

`lambda / forall` Abstracts the last hypothesis in context for the term at the top of the stack. That hypothesis is abstracted repectively as a lambda-abstraction, or a product.

- `lambda` : $(\Gamma, h : T_h \vdash x)\ ...\ \rightarrow (\Gamma \vdash (\lambda(h : T_h).x))\ ...$
- `forall` : $(\Gamma, h : T_h \vdash x)\ ...\ \rightarrow (\Gamma \vdash (\forall(h : T_h), x))\ ...$

`mu` Produces an inductive projection to a higher universe for the term at the top of the stack, if that term is of an inductive type.

- `mu` : $x\ ...\ \rightarrow\ \mu(x)\ ...$

`axiom` Given a combinatorial type (a type without free variables) and an associated tag, produce an axiom with that tag, that can serve as a proof of the given type.

- `axiom` : $tag\ T\ ...\ \rightarrow\ Axiom_{T,tag}\ ...$

Analysing typed terms

`type` Computes the type of the term at the top of the stack.

`match` Given a quote for each possible shape, and a term, executes the corresponding quote :

- $k_{Set}\ k_\lambda\ k_\forall\ k_{apply}\ k_\mu\ k_{var}\ k_{axiom}$ `match` :
- $|\ \Gamma \vdash (\lambda(x : T_x).y)\ ...\ \rightarrow\ k_\lambda(\Gamma, x : T_x \vdash\ x\ y\ ...)$
- $|\ \Gamma \vdash (\forall(x : T_x).y)\ ...\ \rightarrow\ k_\forall(\Gamma, x : T_x \vdash\ x\ y\ ...)$
- $|\ (f x_1..x_n)\ ...\ \rightarrow\ k_{apply}([x_1..x_n]\ f\ ...)$
- $|\ \mu(x)\ ...\ \rightarrow\ k_\mu(x\ ...)$
- $|\ x\ ...\ \rightarrow\ k_{var}(name(x)\ ...)$
- $|\ Axiom_{T,tag}\ ...\ \rightarrow\ k_{axiom}(tag\ T\ \ ...)$
- $|\ Set_n\ ...\ \rightarrow\ k_{Set}(n\ \ ...)$

`extract` Extract the term at the top of the stack into an abstract algebraic representation, suitable for the production of foreign functional code, such as OCaml or Haskell.

Managing the type context

`intro` Given a type T and a name H, adds a new hypothesis H of type T to the context. Alternately, you can give a second hypothesis name H', in which case the new hypothesis will be introduced before H'.

`intro` :
- $\mid \Gamma \vdash name(H)\ T\ ...\ \rightarrow \Gamma, H : T \vdash\ ...$
- $\mid \Gamma, H' : T_{H'}, \Delta \vdash name(H')\ name(H)\ T\ ...\ \rightarrow \Gamma, H : T, H' : T_{H'}, \Delta \vdash\ ...$

`extro-lambda / extro-forall` Clears the last hypothesis from the context. Every term that references that hypothesis is abstracted either as a lambda-expression, or as a product, depending on the variant that was called.

`rename` Renames a hypothesis. This function takes two parameters : a hypothesis name, and the new name to give it.

`substitute` Given a hypothesis name, and a term of the same type as that hypothesis, remove that hypothesis from the context by substituting all its occurences by the given term.

`hypotheses` Pushes a list of all the hypotheses' names in context, from most recent to the oldest.