# Languages-beta: SL-Funcons *

The PLanCompS Project

`SL-Funcons.cbs` | PLAIN | PRETTY

OUTLINE



---

*Language* "SL"

[ *Funcon* sl-to-string
*Funcon* integer-add-else-string-append
*Funcon* int
*Funcon* bool
*Funcon* str
*Funcon* obj
*Funcon* fun
*Funcon* scope-closed
*Funcon* initialise-local-variables
*Funcon* local-variable
*Funcon* local-variable-initialise
*Funcon* local-variable-assign
*Funcon* initialise-global-bindings
*Funcon* override-global-bindings
*Funcon* global-bound
*Funcon* read-line
*Funcon* print-line ]

## SL-specific funcons

*Funcon* sl-to-string($V$ : sl-values) : $\Rightarrow$ strings
*Rule* sl-to-string(null-value) $\rightsquigarrow$ "`null`"
*Rule* sl-to-string($V$ : $\sim$ null-type) $\rightsquigarrow$ to-string($V$)

*Suggestions for improvement: `plancomps@gmail.com`.
Reports of issues: `https://github.com/plancomps/CBS-beta/issues`.

*Funcon* integer-add-else-string-append($V_1$ : sl-values, $V_2$ : sl-values) : ⇒ sl-values
⇝ else(
integer-add(int $V_1$, int $V_2$),
string-append(sl-to-string $V_1$, sl-to-string $V_2$))

## Abbreviations

*Funcon* int($V$ : sl-values) : ⇒ integers
⇝ checked cast-to-type($V$, integers)

*Funcon* bool($V$ : sl-values) : ⇒ booleans
⇝ checked cast-to-type($V$, booleans)

*Funcon* str($V$ : sl-values) : ⇒ strings
⇝ checked cast-to-type($V$, strings)

*Funcon* obj($V$ : sl-values) : ⇒ objects
⇝ checked cast-to-type($V$, objects)

*Funcon* fun($V$ : values) : ⇒ functions(_, _)
⇝ checked cast-to-type($V$, functions(_, _))

## Further funcons

Some of the funcons defined below might be sufficiently reuseful for inclusion in Funcons-beta.

### Binding

*Funcon* scope-closed($Env$ : envs, $X$ : ⇒ $T$) : ⇒ $T$
⇝ closed scope($Env$, $X$)

scope-closed($D$, $X$) evaluates $D$ in the current environment, then evaluates $X$ in the resulting environment. Note the difference between scope-closed($D$, $X$) and closed(scope($D$, $X$)): the latter is equivalent to closed(scope(closed $D$, $X$)), where $D$ cannot reference any bindings.

### Local variables

The local variable map is stored in a variable bound to "local-variables". Initialising a local variable updates the stored local variable map. Subsequent assignments to a local variable do not change the stored map.

*Funcon* initialise-local-variables : ⇒ environments
⇝ bind(
"`local-variables`",
allocate-initialised-variable(environments, map( )))

*Funcon* local-variable($I$ : ids) : ⇒ variables
⇝ checked lookup(assigned bound "`local-variables`", $I$)

```
Funcon  local-variable-initialise(I : ids, V : values) : ⇒ null-type
          ⇝ assign(
                bound "local-variables",
                map-override(
                  {I ↦ allocate-initialised-variable(values, V)},
                  assigned bound "local-variables"))

Funcon  local-variable-assign(I : ids, V : values) : ⇒ null-type
          ⇝ else(
                assign(local-variable I, V),
                local-variable-initialise(I, V))
```

## Global bindings

The global bindings map is stored in a variable bound to "global-bindings". Global declaration or redeclaration of an identifier involves updating the stored global environment.

```
Funcon  initialise-global-bindings : ⇒ environments
          ⇝ bind(
                "global-bindings",
                allocate-initialised-variable(environments, map( )))

Funcon  override-global-bindings(E : environments) : ⇒ null-type
          ⇝ assign(
                bound "global-bindings",
                map-override(E, assigned bound "global-bindings"))

Funcon  global-bound(I : ids) : ⇒ values
          ⇝ checked lookup(assigned bound "global-bindings", I)
```

## Composite input and output

```
Funcon  read-line : ⇒ strings
          ⇝ give(
                read,
                if-true-else(
                  is-eq(given, '\n'),
                  nil,
                  cons(given, read-line)))
```

read-line reads characters from the standard input until a linefeed character, giving the string formed from the sequence of characters excluding the newline. If the input ends before the end of the line, it fails.

```
Funcon  print-line(S : strings) : ⇒ null-type
          ⇝ print(S, "\n")
```