

Teko Language Specification

Version 0.2.0-alpha

Kevin Robert Stravers
email macocio@gmail.com

2017-07-01

1 Syntax

This section provides the grammar of the Teko programming language.

Grammar 1: Teko grammar

```
program = { expr } ;
expr    = ' ( ' { expr } ' ) '
        | atom ;
atom    = ? [ ^ [ : space : ] \ ( \ ) ] + ? ;
```

Grammar 1 uses ISO/IEC 14977 [1] Extended BNF with the special sequence ‘? ... ?’ defined as a POSIX Extended Regular Expressions [2].

2 Semantics

This section provides the semantics of the Teko programming language.

A program is represented as a list of $n \in \mathbb{N}$ expressions.

$$\text{program} = (\text{expr}_1, \text{expr}_2, \dots, \text{expr}_n)$$

An environment env_k is a set of atom-list pairs.

$$\text{env}_k = \{(a_1, L_1), (a_2, L_2), \dots\}, k \in \mathbb{N}, k \leq n + 1$$

A program’s output is defined.

$$\text{interpret}(\text{program}, \text{env}_1) = \text{env}_{n+1}.\text{return}$$

2.1 Core Semantics

Equations (1) to (8) define the core semantics of the Teko programming language.

Notation is specified in section 2.2.

$$\text{env}_{i+1} = \text{eval}(\text{expr}_i, \text{env}_i) \quad (1)$$

$$\begin{aligned} \text{env}_{i+1} &= \text{eval}(\text{atom}, \text{env}_i) \\ &= \text{env}_i[\text{return} = \text{env}_i.\text{atom}] \end{aligned} \quad (2)$$

$$\begin{aligned} \text{env}_{i+1} &= \text{eval}((\text{def atom expr}), \text{env}_i) \\ &= \text{env}_i[\text{atom} := \text{eval}(\text{expr}, \text{env}_i).\text{return}] \end{aligned} \quad (3)$$

$$\begin{aligned} \text{env}_{i+1} &= \text{eval}((\text{set! atom expr}), \text{env}_i) \\ &= \text{env}_i[\text{atom} = \text{eval}(\text{expr}, \text{env}_i).\text{return}] \end{aligned} \quad (4)$$

$$\begin{aligned} \text{env}_{i+1} &= \text{eval}((\text{fn params code}^+), \text{env}_i) \\ &= \text{env}_i[\text{return} = (\lambda \text{ params code})] \end{aligned} \quad (5)$$

$$\begin{aligned} \text{env}_{i+1} &= \text{eval}((\text{mo param code}^+), \text{env}_i) \\ &= \text{env}_i[\text{return} = (\tau \text{ param code})] \end{aligned} \quad (6)$$

$$\begin{aligned} \text{env}_{i+1} &= \text{eval}((\text{if test then else}), \text{env}_i) \\ &= \begin{cases} \text{eval}(\text{then}, \text{env}_i), & Q \text{ is not false} \\ \text{eval}(\text{else}, \text{env}_i), & Q \text{ is false} \end{cases} \\ \text{let } Q &= \text{eval}(\text{test}, \text{env}_i).\text{return} \end{aligned} \quad (7)$$

$$\begin{aligned} \text{env}_{i+1} &= \text{eval}((\text{expr args}^*), \text{env}_i) \\ &= \\ &\text{env}_i[\text{return} = \text{call}(\text{eval}(\text{expr}, \text{env}_i).\text{return}, \\ &\quad \text{env}_i[\text{params} \rightarrow (\text{args})])] \end{aligned} \quad (8)$$

2.2 Notation

1. env_i denotes the i -th element of the list env .
2. $\text{eval}(Q, P)$ calls the evaluate function with program Q and environment P .
3. $\text{env}_k[a_i := O_j]$ returns a new environment with $(a_i, (O_j))$ added to the k -th environment. It is an error if a_i already exists in env_k .
4. $\text{env}_k[a_i = O_j]$ mutates $L_{i,1}$ by assigning it O_j , returning env_k .
5. $\text{env}_k[a_i \rightarrow O_j]$ returns a new environment with O_j prepended to L_j .
6. $\text{env}_k.a_i$ returns the first element of L_i .
7. A^* and A^+ denote 0-or-more and 1-or-more respectively. A in equations (5), (6), (8) expands to $A_1, A_2, \dots, A_m$.
8. λ and τ are literals used to differentiate functions and macros, respectively.

2.3 Calling

Calling invokes a function (fn) or a macro (mo). Implementations must support an unbounded number of active tail calls.

A function call evaluates all elements in the list `envi.params` and binds each result to its respective symbol in the parameter list of the function to be called.

A macro call binds the list `envi.params` to the parameter name specified by the macro. The macro is then run and its results shall be a list which is evaluated in the previous context.

The behaviour is outlined in equation (9).

$$\begin{aligned} & \text{call}(C, \text{env}) \\ &= \begin{cases} \text{eval}(\text{code}(C), f), C \text{ is } \lambda \\ \text{eval}(\text{eval}(\text{code}(C), m).\text{return}, \text{env}), C \text{ is } \tau \end{cases} \\ & \text{let } f = \text{env}[C_{p1} \rightarrow \text{eval}(\text{env.params}_1, \text{env}), \dots] \\ & \text{let } m = \text{env}[C_p \rightarrow \text{env.params}] \end{aligned} \quad (9)$$

3 Environment Initialization

Initial environments require various macros, functions, and constants to be defined and immutable; this section specifies these values. Functions and macros behave as specified by Scheme[4].

3.1 Constants

The environment `env1` is to be initialized with all gaussian rationals such that atom lookups of numeric forms (e.g. `env1.‘3.14’`) return numeric objects (e.g. 3.14). Numeric symbol syntax is described by grammar 2. All numeric symbol-value pairs are immutable.

Grammar 2: Teko numeric symbols

number	=	[‘+’ ‘-’] enota
		[(‘+’ ‘-’) enota ‘i’]
		;
enota	=	numeral [‘e’ integer] ;
numeral	=	integer
		integer ‘.’
		‘.’ integer
		integer ‘.’ integer
		integer ‘/’ integer ;
integer	=	digit { digit } ;
digit	=	‘1’ ‘2’ ‘3’ ‘4’ ‘5’
		‘6’ ‘7’ ‘8’ ‘9’ ‘0’ ;

‘true’ and ‘false’ are unique and immutable objects in the environment. ‘pi’ and ‘e’ are to be stored with rational approximations π and e respectively.

3.2 Functions

The functions ‘append’, ‘string-append’, ‘take’, ‘drop’, ‘first’, ‘rest’, ‘+’, ‘-’, ‘*’, ‘/’, ‘%’, ‘exp’, ‘ln’, ‘sqrt’, ‘sin’, ‘cos’, ‘not’, ‘abs’, ‘even?’, ‘odd?’, ‘zero?’, ‘one?’, ‘read’, ‘write’, ‘ceiling’, ‘floor’, ‘round’, ‘read-line’, ‘read-string’, ‘eval’, ‘max’, ‘min’.

3.3 Macros

The macros ‘’, ‘<’, ‘<=’, ‘>’, ‘>=’, ‘=’, ‘equal?’, ‘and’, ‘or’, ‘xor’, ‘bitwise-and’, ‘bitwise-ior’, ‘bitwise-xor’,

3.4 Miscellaneous

Further initial macros, functions, and constants may be implemented provided all such symbols begin with ‘@’, which is reserved for the implementation.

The macro ‘;’ shall act as a comment; a program stripped off all ‘(; ...)’ forms shall behave exactly like the program containing the comments.

If the method of computation in section 2.1 is used as implementation, then ‘return’ and ‘params’ are to be renamed to a ‘@’-form.

4 Strings

” is the macro for generating strings; words are space-separated unless a list is betwixt. A list may contain a single integer, which is converted to a character and inserted. A list may contain an integer N and an expression. The expression is repeated N times and inserted. A list may contain an expression.

5 Comments

It can be seen from (4) that the Teko has dynamic scope and that it is latently typed.

Dynamic scoping may be beneficial to the user in terms of convenience[5], as well as being easy to implement. In addition to this one gets free parametrization as specified in SRFI-39[3].

References

- [1] ISO/IEC 14977. <https://www.iso.org/standard/26153.html>, 1996. Fetched Thu Jun 1 19:29:43 CEST 2017.
- [2] POSIX.1-2008. <http://pubs.opengroup.org/onlinepubs/9699919799/>, 2016. Fetched Thu Jun 1 19:29:00 CEST 2017.
- [3] Marc Feeley. Parameter objects. <https://srfi.schemers.org/srfi-39/srfi-39.html>, 2003. Fetched Thu Jun 1 19:49:49 CEST 2017.
- [4] Michael Sperber, R. Kent Dybvig, William Clinger, Jonathan Rees, Robert Bruce Findler, and Jacob Matthews. Revised6 report on the algorithmic language scheme. www.r6rs.org/final/r6rs.pdf, 2007. Fetched Thu Jun 1 19:28:13 CEST 2017.
- [5] Richard Stallman. Emacs: The extensible, customizable display editor. <https://www.gnu.org/software/emacs/emacs-paper.html#SEC15>, 1981. Fetched Thu Jun 1 19:38:44 CEST 2017.