Specification for the Teko Programming Language

Conor Stuart Roe

December 26, 2018

1 Overview

Teko is a high-level, statically typed scripting language that is primarily interpreted. It follows an object-oriented and imperative programming paradigm. It is intended to offer a full type system with user-defined classes, class hierarchies, and generics, much like Java, and can be rigorously statically type checked. Its high-level architecture offers a variety of advanced data types, such as complex numbers, bytestrings, maps, and structs as primary types, and other features like asynchronous loops easily available. It follows a typical C family syntax; at a glance, Teko looks most like Java, with some syntactic features inspired by Javascript, C++, or Python.

Teko was created in 2018 by Conor Stuart Roe.

2 Lexical and Syntactic Structure

Teko lexical and syntactic structure are defined by the BNF descriptions below. := indicates definition of a nonterminal, pipe || indicates options, parentheses () indicate optional elements, and asterisk * indicates occurrence zero or more times. Angle brackets <> may be used to write English descriptions. Literal characters are written with `code formatting`. Teko is not whitespace-sensitive, except with regard to parsing strings and line comments.

2.1 Lexical Structure

Teko token capture is greedy, so if two adjacent tokens could be mistaken as comprising a single token, they must be separated by whitespace.

Label := Alphabetical (Alphanumeral)*

- Labels are any sequence of alphanumeric characters beginning with an alphabetical character, excluding the following reserved words: `true`, `false`, `if`, `else`, `for`, `while`, `each`, `begin`, `let`, `class` , `in` , `void` , `return` , `public` , `protected` , `private` , `readonly` , `typeof` .

INT := NUMERAL (NUMERAL)*

REAL := NUMERAL (NUMERAL)* `.` (NUMERAL)*

- REALs may end with a decimal point `.` , which implies a fractional part equal to zero.

BOOL := `true` ‖ `false`

CHAR := `'` CHARACTEROrESCAPE `'` ‖ `'"'` ‖ `'\''`

STRING := `"` (CHARACTEROrESCAPE ‖ `'` ‖ `\"`)* `"`

ALPHANUMERAL := ALPHABETICAL ‖ NUMERAL

CHARACTEROrESCAPE := ALPHABETICAL ‖ NUMERAL ‖ PUNCT ‖ <space> ‖ `\\` ‖ `\n` ‖ `\t`

- Teko characters and strings may not contain literal tab or newline characters.

ALPHABETICAL := (one of) `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_`

NUMERAL := (one of) `0123456789`

PUNCT := (one of) `!#$%&()*+,-./:;<=>?@]^{|}~`

2.2 Comment Structure

Teko support C-style line comments, which begin with a double slash `//` and end at the next newline.

2.3 Syntactic Structure

Each Teko file consists of a single CODEBLOCK.

CODEBLOCK := (LINE `;`)*

LINE := VARIABLEDECLARATION ‖ ASSIGNMENT ‖ EXPRESSION ‖ IFSTATEMENT ‖ WHILEBLOCK ‖ FORBLOCK

VariableDeclaration := `let` Declared SetInit (`,` Declared SetInit)* ‖ Type Declared (SetInit) (`,` Declared (SetInit))*

- Teko permits the declaration of variables using the declarator `let` in place of a type, but doing so requires that all variables declared in this way be set to a value on the same line, so that type can be inferred.

Declared := Label (Struct)

- The presence of a struct after the label indicates that the new variable's type is a function with given return type, rather than simply storing a value of the given type.

SetInit := `=` Expression ‖ `= {` CodeBlock `}`

- Initial setting of a variable's value differs from later assignments in that updating setters `+=`, `-=`, `*=`, `/=`, `^=`, and `%=` are not permitted.

Type := NamedType ‖ Type `{}` ‖ Type `<>` ‖ Type `[]` ‖ `{` Type `:` Type `}` ‖ `(` (Type, (`,` Type)*) `)` ‖ StructType

NamedType := `int` ‖ `real` ‖ `bool` ‖ `char` ‖ `str` ‖ `enum` ‖ `comp` ‖ `bits` ‖ <any label assigned to a type>

StructType := `(` (StructTypeParam (`,` StructTypeParam)*) `)`

StructTypeParam := Type Label (`?` Expression)

Assignment := Label Setter Expression ‖ Label `= {` CodeBlock `}`

Setter := `=` ‖ `+=` ‖ `-=` ‖ `*=` ‖ `/=` ‖ `^=` ‖ `%=`

Expression := Primitive ‖ Composite ‖ Label (Struct) (`.` Label (Struct))* ‖ Expression BinaryOp Expression ‖ Expression Comparison Expression ‖ Expression Converter ‖ Expression Slice ‖ `:` Expression) ‖ `(` Expression `)`

- Expressions are the only syntactic units which evaluate to a value; in fact, in all contexts, expressions must evaluate to an appropriately-typed value.

Primitive := Int ‖ Real ‖ Bool ‖ Char ‖ String

Composite := Array ‖ List ‖ Set ‖ Enum ‖ Map ‖ Tuple

Array := `[` (Expression (`,` Expression)*) `]`

List := `{` (Expression (`,` Expression)*) `}`

Set := `<` (Expression (`,` Expression)*) `>`

- In arrays, lists, and sets, collectively called the iterables, all contained expressions must evaluate to values of the same type, and the type of the iterable is derived from the type of the contained values.

Enum := `<` (Label (StructType) (`,` Label (StructType))*) `>`

- In an enum, all labels must be undeclared, and their inclusion in the enum declares them. The StructType after a label, if present, allows that enum element to have parameters, much like a paramaterized data type in Haskell.

Map := `{` (MapPair (`,` MapPair)*) `}`

MapPair := Expression `:` Expression

Tuple := `(` (Expression (`,` Expression)*) `)`

Struct := `(` (StructParam (`,` StructParam)*) `)`

StructParam := (Label `=`) Expression

- All parameters without equals `=` must occur before all parameters with equals.

BinaryOp := `+` ‖ `-` ‖ `*` ‖ `/` ‖ `^` ‖ `%` ‖ `<:`

Comparison := `==` ‖ `!=` ‖ `<` ‖ `>` ‖ `<=` ‖ `>=` ‖ `<:`

Converter := `.` ‖ `$` ‖ `[]` ‖ `{}` ‖ `<>`

Slice := `[` Expression (`:` Expression) `]`

IfStatement := `if (` Expression `) {` CodeBlock `}` (`else if (` Expression `) {` CodeBlock `}`)* (`else {` CodeBlock `}`)

WhileBlock := `while (` Expression `) {` CodeBlock `}`

ForBlock := `for (` Type Label `in` Expression `) {` CodeBlock `}`