# CamlTree Language Reference Manual

Haozhe Chen (Language Guru), Markov Chen (Manager), Yihan Shen (System Architect), Andrew Yang (System Architect), Xintong Yu (Tester), Vicky Zhou (Tester)

April 2023

## Table of Contents



## 1. Introduction

CamlTree is a programming language for running computations on trees. It enables users to define nodes, connections between nodes, and operations to expand and compute on the tree.

CamlTree consists of two sub-languages:

1. TML (Tree Markup Language) - a declarative, strongly-statically typed language with dynamic scoping and lazy evaluation for succinctly describing tree structures
2. TRS (Tree Script) - an imperative, strongly-statically typed language with static scoping and eager evaluation for defining operations and computations on the tree

CamlTree ensures memory safety through its strong static type system. It does not currently support concurrency or parallel execution.

## 2. Lexical Conventions

### 2.1. Tokens

There are five classes of tokens: identifiers, keywords, literals, operators, and separators. Whitespace is ignored but required to separate some tokens.

Comments are denoted between `/*` and `*/` and do not nest.

```
let whitespace = [' ' '\t' '\r' '\n']
let comment = "/*" { comment_body } "*/"
```

### 2.2. Identifiers

Identifiers are sequences of letters, digits, and underscores, starting with a letter. They are case-sensitive.

```
let alpha = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let id = alpha (alpha | digit | '_')*
```

Examples:

```
myTree
node1
child_2
```

### 2.3. Keywords

Keywords are reserved identifiers that cannot be used as ordinary identifiers.

```
root connect
create eval give
run check replace
def return
```

### 2.4. Literals

Literals include integers, floats, booleans, and strings. Characters are not used as literals, though we list a few characters we allow for strings.

```
let char = 'a'-'z' | 'A'-'Z' | '0'-'9' | '_' | '+' | '-' | '*' | '%' | '/' | '=' | '(' | ')' | '[' | ']' | '{' | '}' | '^'
let int = digit+
let float = int '.' int
let bool = "true" | "false"
let string = '\"' char* '\"'
```

Examples:

```
42
3.14
true
"hello"
```

### 2.5. Operators

```
+  -  *  /  %    (arithmetic)
=               (assignment)
== != < <= > >= (equivalence)
and or not      (logical)
```

### 2.6. Separators

```
( ) { } [ ] ; , -> <-
```

## 3. Types

### 3.1. Primitive Data Types

- `int` : 32-bit signed integers

  ```
  A a = 42
  ```

- `float` : 32-bit IEEE 754-2019 (most recent as of writing) floating-point numbers

  ```
  A a = 3.14
  ```

- `bool` : boolean values `true` and `false`

  ```
  A a = true
  ```

- `string` : sequences of UTF-8 characters

  ```
  A a = "hello world"
  ```

### 3.2. Non-Primitive Data Types

- `array` : fixed-size arrays of elements of the same type

  ```
  A a = [1, 2, 3, 4]
  A a = [0, 2 .. 10] /* [0, 2, 4, 6, 8, 10] */
  ```

### 3.3. Type Qualifiers

- `static` : provides global variable functionality, only one instance with fixed memory location

  ```
  A a = static 42
  ```

## 4. Operators

### 4.1. Arithmetic Operators

Binary operators `+`, `-`, `*`, `/`, `%` for addition, subtraction, multiplication, division, modulo.

Precedence: `*`, `/`, `%` > `+`, `-`. Left-associative.

```
A a = (1 + 2) * 3 /* 9 */
```

### 4.2. Assignment Operators

The `=` operator assigns an expression to a variable. Right-associative.

```
A a = 10
```

### 4.3. Equivalence Operators

Binary operators `==`, `!=`, `<`, `<=`, `>`, `>=` for equality, inequality, less than, less than or equal, greater than, greater than or equal.

Same precedence, lower than arithmetic. Left-associative.

```
A a = 2 < 3 /* true */
```

### 4.4. Logical Operators

Binary operators `and`, `or` and unary `not` for logical AND, OR, NOT.

Precedence: `not` > `and`, `or`. Left-associative.

```
if a and not b or c
```

## 5. Statements and Expressions

### 5.1. TML Statements

TML (Tree Markup Language) is a declarative language used to describe the initial structure of a tree in a `.tml` file. It has the following statements:

1. `root <type>`: Declares the root node of the tree and its type. There can only be one root node type per tree.

   ```
   root A
   ```

2. `node <type>`: Declares a new node type that can be used in the tree. Multiple node types can be declared.

   ```
   node B
   ```

3. `connect <parent_type> -> <child_type>`: Connects a parent and child node. The parent and child node types must be declared prior.

   ```
   connect A -> B
   ```

4. `<root_type> <root_name> = <forward_value> ' <backward_value>'` : Creates a root node of a particular node type and assigns a value. The node type must be declared prior. Only one root node may be created. The backward value (separated by ') is optional.

   ```
   A a = 0 ' 1
   ```

5. `<child_type> <child_name> <- <parent_name> = <forward_value> ' <backward_value>'` : Creates a node and links it to a parent node. The parent and child node types must be declared prior. The backward value is optional.

   ```
   B b1 <- a = 2 /* Forward value only */
   B b2 <- a = 3 ' 4 /* Forward and backward value */
   ```

### 5.2. TRS Statements

TRS (Tree Script) is an imperative language used to define operations and computations on the tree described in the associated `.tml` file. It has the following statements:

1. `open <tml_file> as <tree>`: Opens a `.tml` file and binds it to a variable.

   ```
   open tree.tml as Tree
   ```

2. `create <parent> -> <child> (): <<statements>@<number>>` : Defines a forward pass operation that creates a child node from a parent node. The statements specify how to compute the child node's forward value. The number specifies how many child nodes to create.

```
create A -> B ():
  make <parent + 1 @ 2> /* Create two children with values one larger than parent value. */
```

3. `eval <child>' -> <parent>' (): <statements>` : Defines a backward pass operation that aggregates information from child nodes to update the parent node's backward value. The statements specify how to compute the aggregated value.

```
eval B' -> A ():
  give sum(children)
```

4. `def <function_name>(<params>): <statements>` : Defines a function in the same way as Python. Adding `$` in front of an operation in a function indicates that the operation is done over all children forward/backward values (depending on whether it is currently a forward or backward pass).

```
def sum(children):
    return 0 $+ children
```

5. `<tree>.run()` : Executes the forward and backward pass operations on the tree.

```
Tree.run()
```

6. `<tree>.replace()` : Prepares the tree for the next run by replacing all forward pass values with the backward pass values.

```
Tree.replace()
```

7. `check <tree> == COMPLETE` : Checks if all parents have the right number children specified by `create` and all nodes have forward and backward values.

```
check Tree == COMPLETE
```

Special syntax:

- `$<operation><children_variable_name>` : Indicates that the operation should be repeated for all children.
- `@<number>` : Creates multiple child nodes with the same value.
- List comprehension: Can create a list based on a specified range. `[1..3] \* List [1, 2, 3] */`

### 5.3. Control Flow

CamlTree supports `if` - `else`

```
if x < y
  result = 0
else
  result = 1
```

It also supports `while` and `for` loops:

```
while n < 10:
  n = n * 2

for i in [0..5]:
  print(i)
```

## 6. Pre-Implemented Functions

- `rng(a, b)` - random integer between `a` and `b` inclusive
- `sum(arr)` - sum of array elements
- `len(arr)` - length of array
- `max(a, b)` - maximum of `a` and `b`
- `min(a, b)` - minimum of `a` and `b`
- `abs(x)` - absolute value of `x`
- `print(x)` - print `x` to standard output