(g)ROOT
Language Reference Manual

Samuel Russo Amy Bui Eliza Encherman
Zachary Goldstein Nickolas Gravel

April 25, 2022

Contents

1 Intro

At its core, (g)ROOT is a general-purpose functional programming language. Its syntax stems from functional languages such as Scheme and other languages rooted in the Lisp family of programming languages. What sets Groot apart from other programming languages is its native employment of trees.

Trees are a widely-used, abstract data type that represents a hierarchical branching structure, with a root value and subtrees of children. In many general-purpose programming languages, trees are not a built-in feature. This forces the programmer to implement them from the ground up, which can be a tedious process for many beginner to intermediate programmers. Groot's built-in tree syntax abstracts this logic away from the programmer, significantly reducing the complexity inherent to tree development.

In the Groot programming language, trees are a primitive, immutable type consisting of an element, sibling and child or a *leaf*, a value representing an empty tree. This allows programmers to easily pass trees between functions, akin to passing lists in Lisp-style programming languages. Built-in functions are also supplied to accomplish many common tree operations. These language features alleviate the burden of implementing, operating, and maintaining a tree data structure from the programmer, allowing them to focus on solving more complex problems.

1.1 How to read manual

The syntax of the language will be given in BNF-like notation. Non-terminal symbol will be in italic font *like-this*, square brackets [...] denote optional components, curly braces { ... } denote zero or more repetitions of the enclosed component, and parentheses (...) denote a grouping. Note the font, as `[ ... ]` and `( ... )` are syntax requirements later in the manual.

2 Lexical Convention

2.1 Blanks

The following characters are considered as **blanks**: space, horizontal tab ('\t'), newline character ('\n'), and carriage return ('\r').

Blanks separate adjacent identifiers, literals, expressions, and keywords. They are otherwise ignored.

2.2 Comments

Comments are introduced with two adjact characters `(;` and terminated by two adjacent characters `;)`. Nested comments are currently not allowed. Multiline comments are allowed.

```
(; This is a comment. ;)

(; This is a
   multi-lined comment. ;)
```

2.3 Identifiers

Identifiers are sequences of letters, digits, and ASCII characters, starting with a letter. Letters consist of the 26 lowercase and 26 uppercase characters from the ASCII set. Identifiers may not start with an underscore character, and may not be any of the reserved character sequences.

⟨*ident*⟩ ::= *letter* (*letter* | *digit* | _)

⟨*letter*⟩ ::= `a...z` | `A...Z`

⟨*digit*⟩ ::= `0...9`

2.4 Integer Literals

An integer literal is a decimal, represented by a sequence of one or more digits, optionally preceded by a minus sign.

⟨*integer-literal*⟩ ::= [-] *digit* {*digit*}

⟨*digit*⟩ ::= `0...9`

2.5 Boolean Literals

Boolean literals are represented by two adjacent characters; the first is the octothorp character (`#`), and it is immediately followed by either the `t` or the `f` character.

⟨*boolean-literal*⟩ ::= `#` (`t` | `f`)

2.6 Character Literals

Character literals are a single character enclosed by two ' (single-quote) characters.

2.7 Operators

All of the following operators are prefix characters or prefixed characters read as single token. Binary operators are expected to be followed by two expressions, unary operators are expected to be followed by one expression.

⟨*operator*⟩ ::= (*unary-operator* | *binary-operator*)

⟨*unary-operator*⟩ ::= `!` | `-`

⟨*binary-operator*⟩ ::= `+` | `-` | `*` | `/` | `mod`
| `==` | < | > | ≤ | ≥ | `!=`
| `&&` | ||

2.8 Keywords

The below identifiers are reserved keywords and cannot be used except in their capacity as reserve keywords:

```
if      val         let
leaf    elm         tree
cld     sib         lambda
```

The following character sequence are also keywords:

```
==      +       &&      >       '
!=      -       ||      mod     #t
<=      *       !       (       #f
>=      /       <       )       anon
```

2.9 Syntax

See Definitions and Expression for concrete syntax for each definition and expressions, with examples.

3 Values

3.1 Base Values

3.1.1 Integer numbers

Integer values are integer numbers in range from -2^{32} to $2^{32}-1$, similar to LLVM's integers, and may support a wider range of integer values on other machines, such as -2^{64} to $2^{64}-1$ on a 64-bit machine.

3.1.2 Boolean values

Booleans have two values. `#t` evaluates to the boolean value `true`, and `#f` evaluates to the boolean value `false`.

3.1.3 Characters

Character values are 8-bit integers between 0 and 255, and follow ASCII standard.

3.2 Functions

Functional values are mappings from values to value.

3.3 N-Ary Tree Compound Type

A core feature of (g)ROOT is its n-ary tree compound value type. Every tree value consists of three components: an element, it's right-immediate sibling, and it's first child. This allows for the convenient implementation of robust recursive algorithms with an arbitrary branching factor.

Every tree value in (g)ROOT may be either a full tree instance with an element, sibling, and child, or a leaf. The leaf value in (g)ROOT represents the nullary, or empty, tree.

The tree type in (g)ROOT is modeled after a left-child right-sibling binary tree, where each node contains a reference to its first child and reference to its next sibling. This allows each node in (g)ROOT to have any number of children, while constraining the maximum number of fields per tree instance to three (element, sibling, and child).

The tree type is very similar in usage to the one-dimensional list type present in many other functional languages, but enforces two additional invariants that empower programmers to shoot themselves in their foot less often.

1. The element of a tree instance may not be itself a tree. An element may be a value of any other type. This enforces a consistent structure among all trees that could be created in (g)ROOT.

   Note: it is possible to circumvent this requirement by wrapping a tree instance in a no-args lambda closure. This is a reasonable means of achieving nested data structures

as it prevents the accidental creation of nested values; programmers who wrap tree instances in lambda closures likely did so with intention.

2. Every tree node must have a single immediate sibling and a single immediate child. This forces programmers to think in a purely recursive manner about their solutions.

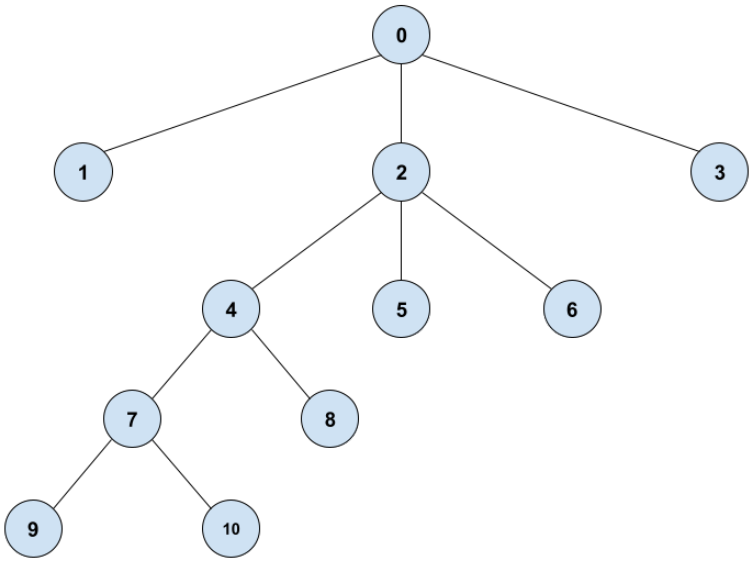
   Trees may be conveniently constructed in-place using the following construction syntax:

```
'(value sibling-tree child-tree)
```

   For example:

```
'(0 leaf (1 (2 (3 leaf leaf) (4 (5 (6 (leaf leaf)) leaf))) (7 (8 leaf leaf)
   (9 (10 leaf leaf) leaf))))
```

   represents the following tree (as drawn in normal binary tree form):



4 Types

5 Definitions and Expressions

⟨*def*⟩ ::= `(` `val` *ident* *expr* `)`
| *expr*

⟨*expr*⟩ ::= *literal*
| *ident*
| *unary-operator* *expr*
| `(` *binary-operator* *expr* *expr* `)`
| `(` *ident* {*expr*} `)`
| `(` `let` `(` `[`*ident* *expr*`]` {`[`*ident* *expr*`]`} `)` *expr* `)`
| `(` `if` *expr* *expr* *expr* `)`
| `(` `lambda` `(` {*arguments*} `)` *expr* `)`

⟨*literal*⟩ ::= *integer-literal* | *boolean-literal* | *character* | leaf

⟨*arguments*⟩ ::= ϵ
| *ident* :: *arguments*

Expressions are values or parenthetical expressions.

5.1 Values

see Values.

5.2 Parenthetical expressions

Parenthetical expressions are always withing parentheses and include function application, lambda expressions, global and local definitions, binary and unary operations, and if-statements. In the above concrete syntax, the parentheses in this font, `( ... )`, are syntax requirements, rather than denoting a grouping which is given by (...)

5.2.1 Function application

Function application in (g)ROOT always returns a value, and is written as the function `ID` followed by a list of zero or more expressions, which are its arguments. The arguments are not separated from the ID by parentheses. (g)ROOT has first-class functions, therefore functions can be passed as arguments. Currying is allowed.

Example:

```
(foo)
(bar a b)
((baz x) y)
```

5.2.2 Lambda Expression

Lambda expressions are accomplished with the `lambda` keyword, a parentheses-enclosed list of 0 or more identifiers as formal arguments, followed by the expression that may use those arguments and/or any free variables. Nesting of lambda expressions is allowed.

Example:

```
(lambda () #t)
(lambda (x) x)
(lambda (x y) (+ x y))

(lambda (a) (add2 a b))

(lambda (n)
   (lambda (m)
      (add2 n m)))
```

5.2.3 Global definitions

Global definitions are accomplished using the `val` keyword, followed by an identifier, followed by the expression which is to be bound to that value.

Example:

```
(val x 4)
(val y (+ x 5))
(val foo (lambda (arg) (* arg arg)))
```

Calling a global definition with a preexisting identifier will re-bind that identifier to the new value - onlys allowed at the top level, and new definition must always of the same type as the previous definition.

5.2.4 Local definitions

Local definitions are found with the `let` expressions, which is the `let` keyword followed by the identifier(s) and the expression(s) to be bound to it, followed by the expression that local variable may be used. Let expressions must have at least one local binding.

Example:

```
(let ([x 4]) (+ 2 x))  (; return 6 ;)
(let ([x 4]) x)        (; return 4 ;)
(let () y)             (; not allowed! ;)
```

Variables defined within the let binding are not defined outside of it, while variables globally relative to the let can be accessed within it. Since let bindings are a type of expression, this allows for chained let bindings.

Example:

```
(let x 4
   (let y 5
      (let z 9
         (* x (* y z)))))
```

5.2.5 If-expression

If-expressions are the only form of control flow in (g)ROOT, and are always formed with the `if` keyword followed by three expressions (the *condition*, the *true case* and the *false case*). Omission of the false case is a syntax error, and the expressions are not separated by parentheses, brackets, or keywords.

Example:

```
(if #t 1 2)
(if (< 3 4)
   (+ x y)
   (- x y))
```

5.2.6 Unary operators

Unary operations (used for boolean or signed negation) must not be enclosed in parentheses. They are accomplished with a unary operator in front of the expression they negate.

Example:

```
-3
-(+ 3 4)
-(if #t 2 3)
-x

!#t
!x
!(expr)
```

5.2.7 Binary operators

The general use of binary operators is as follows: (*binary-operator* $expr_1$ $expr_2$)

The **arithmetic operators** (`+` , `-` , `*` , `/` , `mod`) take two expressions that evaluate to integers.

The **comparator operators** (`==` , $<$, $>$, $\leq$, $\geq$, `!=`) take two expressions that both evaluate to either integers or booleans.

The **boolean operators** (`&&` , `||`) take two expressions that evaluate to booleans.

6 Functions

6.1 Built-In Functions

Note: Throughout this section, the character pair `tr` is used as shorthand to represent a tree value, an identifier bound to a tree value.

There are four built-in functions which are integral to the tree data type. It is a checked-runtime error to call `elm, sib, cld` on a `leaf`:

1. `elm`

   Usage: `(elm tr)`

   Purpose: returns the element of the provided tree value `(tr)`

2. `sib`

   Usage: `(sib tr)`

   Purpose: returns the sibling tree of the provided tree value `(tr)`

3. `cld`

   Usage: `(cld tr)`

   Purpose: returns the child tree of the provided tree value `(tr)`

4. `leaf?`

   Usage: `(leaf?  tr)`

   Purpose: Returns boolean val `#t` iff `tr` is a leaf.

5. `tree`

   Usage: `(tree ELEMENT SIBLING CHILD)`

   Purpose: constructs a tree value containing the value ELEMENT, with sibling SIBLING and child CHILD

6.2 Standard Library Functions

Note: Throughout this section, the character pair `tr` is used as shorthand to represent a tree value, an identifier bound to a tree value.

The first set of functions introduced in the standard library allow for programmers to use some of their beloved list functions present in other languages:

1. `(cons a b)`

   Creates a 1-ary tree approximating the functionality of single-dimensional list in other functional languages. `a` is any value not a tree, `b` may be `leaf` or `nil`.

2. `(car tr)`

   Extracts the first value in the pair.

3. `(cdr tr)`

   Extracts the second value in the pair.

4. `(null? tr)`

   A predicate function that evaluates to true iff the pseudo-list is empty.

5. `(val nil leaf)`

   A value `nil` that mirrors the built-in keyword leaf.

6. `(append xs ys)`

   Append the list of elements in `ys` to the end of `xs`.

7. `(revapp xs ys)`

   Append the list of elements in `ys` to the reverse of `xs`.

Now some useful tree functions!

1. `(graft oak fir)`

   Appends a sibling tree `fir` to some other tree `oak`. The result of calling this function is a new tree in which `fir` is the rightmost sibling of `oak`. This function is extremely useful when re-shaping trees.

2. `(level-flatten tr)`

   Flattens the provided tree to a 1-ary tree, arranging elements in level-order.

3. `(pre-flatten tr)`

   Flattens the provided tree to a 1-ary tree, arranging elements in pre-order.

4. `(post-flatten tr)`

   Flattens the provided tree to a 1-ary tree, arranging elements in post-order.

5. `(map f tr)` Maps a function `f` over every element of the provided tree `tr`.

6. `(filter p? tr)`

   Constructs a new tree containing only elements that satisfy the predicate function `p?`, which must return a boolean value.

7. `(level-fold fn base tr)`

   Folds a tree, visiting each element in a level-order traversal. Note that the accumulation function `fn` must take TWO arguments, the first of which is the current element, and second is the rest of the tree.

8. `(pre-fold fn base tr)`

   Folds a tree, visiting each element in a pre-order traversal. Note that the accumulation function `fn` must take TWO arguments, the first of which is the current element, and second is the rest of the tree.

9. `(post-fold fn base tr)`

   Folds a tree, visiting each element in a post-order traversal. Note that the accumulation function `fn` must take TWO arguments, the first of which is the current element, and second is the rest of the tree.

10. `(fold fn base tr)`

    Folds a tree, in a more intuitively tree-like manner. Note that the accumulation function `fn` must take THREE arguments: the value of the current node, the sibling accumulator, and the child.accumulator.

11. `(node-count tr)`

    Evaluates to the number of nodes in the provided tree.

12. `(height tr)`

    Evaluates to the height of the provided tree.

Higher-Order Functions:

1. `(curry f)`

   Allows for the partial application of a two-argument function.

2. `(uncurry f)`

   Uncurries a previously curried function such that both of its arguments must be provided at the same time.

3. `(o f g)`

   Given two single-argument functions, returns a single function which is the composition of `f` and `g`.

4. `(flip f)`

   Given a two argument function, reverses the order in which the arguments are evaluated and passed to the function.

5. `(flurry f)`

   Given a two argument function, reverses the order in which the arguments are evaluated and passed to the function, and allows for the partial application of that function. This function both flips and curries `f` (hence the fun name).

7 Appendix

```
(; We can define standard list functions with our tree data type! Fun! ;)

(define (cons a b) (tree a leaf (tree b leaf leaf)))
(define (car  tr) (elm tr))
(define (cdr  tr) (cld tr))
(define (nil   ) leaf)
(define (null? tr) (leaf? tree))

(; a list function! ;)
(define append (xs ys)
    (if (null? xs)
        ys
        (cons (car xs) (append (cdr xs) ys))
    )
)

(; another list function! ;)
(define revapp (xs ys) ; (reverse xs) followed by ys
    (if (null? xs)
        ys
        (revapp (cdr xs) (cons (car xs) ys))
    )
)

(; appends a sibling tree (fir) to tree (oak) ;)
(define (graft oak fir)
    (if (leaf? oak)
        fir
        (tree (elm oak) (graft (sib oak) fir) (cld oak))
    )
)

(; attempt 4
   this function flattens a tree, level-order ;)
(define (level-flatten tr)
    (if (leaf? tr)
        leaf
        (tree
            (elm tr)
            leaf
            (level-flatten (graft (sib tr) (cld tr)))
        )
    )
)
```

flattening functions!

```
(; this function flattens a tree, pre-order ;)
(define (pre-flatten tr)
    (if (leaf? tr)
        leaf
        (tree
            (elm tr)
            leaf
            (pre-flatten (graft (cld tr) (sib tr)))
        )
    )
)


(; this function flattens a tree, post-order ;)
(define (post-flatten tr)
    (if (leaf? tr)
        leaf
        (if (leaf? (cld tr))
            (tree
                (elm tr)
                leaf
                (post-flatten (sib tr)))
            (post-flatten (graft
                                (graft
                                    (cld tr)
                                    (tree (elm tr) leaf leaf))
                                (sib tr))
            )
        )
    )
)
```

Let's create some higher order functions! Yay!

```
(define (curry f)   (lambda (x)  (lambda (y) (f x y))))
(define (uncurry f) (lambda (x y) ((f x) y)))
(define (o     f g) (lambda (x) (f (g x))))
(define (flip  f)   (lambda (x)  (lambda (y) (f y x))))
```

function: (flurry func)

- precondition:

  func is a function that takes exactly two args

- evalutation:

evaluates to a curried form of func, where:

1. the first value passed to the curried function is treated as the second argument to (func).
2. the second value passed to the curried function is treated as the first argument to (func).

- note: flurry is a contraction of "flip & curry", and we're very proud of the wordplay there.

```
(define flurry (func)
    (curry (lambda (a b) (func b a)))
)



(; mapping function! ;)
(define (map f tr)
    (if (leaf? tr)
        leaf
        (tree
            (f (elm tr))
            (map f (sib tr))
            (map f (cld tr))
        )
    )
)



(; filter function! ;)
(define filter (p? tr)
    (if (leaf? tr)
        leaf
        (if (p? (elm tr))
            (tree (elm tr) (filter p? (sib tr)) (filter p? (cld tr)))
            (filter p? (graft (cld tr) (sib tr)))
        )
    )
)




(; folding functions! ;)

(; attempt 4
   this function folds a tree, level-order ;)
(define (level-fold fn base tr)
    (if (leaf? tr)
```

```
       base
       (fn
          (elm tr)
          (level-fold (graft (sib tr) (cld tr)))
       )
   )
)

(; this function folds a tree, pre-order ;)
(define (pre-fold fn base tr)
   (if (leaf? tr)
       base
       (fn
          (elm tr)
          (pre-fold (graft (cld tr) (sib tr)))
       )
   )
)

(; this function folds a tree, post-order ;)
(define (post-fold fn base tr)
   (if (leaf? tr)
       base
       (if (leaf? (cld tr))
          (fn (elm tr) (post-fold (sib tr)))
          (post-fold (graft
                        (graft (cld tr) (tree (elm tr) leaf leaf))
                           (sib tr)
                     )
          )
       )
   )
)

(; once we implement first-class functions, we
   should find that the behaviors of these three
   functions are exactly equivalent to their
   lower-class cousins ;)
(define (level-fold-flatten tr) (level-fold cons nil tr))
(define (pre-fold-flatten tr) (pre-fold cons nil tr))
(define (post-fold-flatten tr) (post-fold cons nil tr))

(; this function folds a tree, in a more intuitively
   tree-like manner
   fn must take in three params:
   - the current value
   - the sibling accumulator
```

```
   - the child accumulator ;)
(define (fold fn base tr)
    (if (leaf? tr)
        base
        (fn
            (elm tr)
            (fold fn base (sib tr))
            (fold fn base (cld tr))
        )
    )
)



(; some more fun folding functions! ;)
(define (node-count tr)
    (pre-fold (lambda (elem count) (+ 1 count)) 0 tr)
)

(; this uses the tree fold to calculate the height! ;)
(define (height tr)
    (fold
        (lambda (elem clds sibs) (+ 1 (max clds sibs)))
        0 tr
    )
)

(define (height-no-fold tr)
    (if (leaf? tr)
        0
        (max (height (sib tr))
            (+ 1 (height (cld tr)))
        )
    )
)
```