(g)ROOT
Language Reference Manual

Samuel Russo Amy Bui Eliza Encherman
Zachary Goldstein Nickolas Gravel

February 27, 2022

Contents

1 Intro

Ocaml LRM

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 parenthesis (...) denote a grouping.

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.

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

2.3 Identifiers

Identifiers are sequences of letters, digits, and underscore characters (‘_’), starting with a letter. Letters consist of the 26 lowercase and 26 uppercase characters from the ASCII set.

⟨*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 operatprs 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 otherwise:

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

The following character sequence are also keywords:

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

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 Leaf

3.4 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 Groot 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 feet 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 not 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.

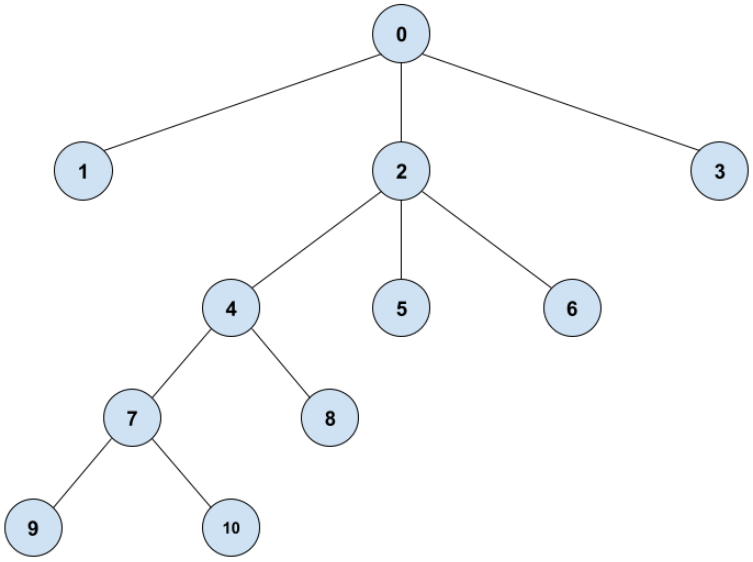
   Trees may be conveniently constructed in-place using the following construction syntax:

```
'(value left-tree right-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:



4 Names

4.1 Base Values

4.2 Functions

5 Constants

6 Expressions

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

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

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

6.1

6.2 Lambda Expression

7 Functions