Jacob Graff - jag2302
Justin Walters - jw3043
Luis “Bert” Ramirez - lar2195
Shruti Kulkarni - sgk2118

PLTree: A Tree Programming Language

Language Reference Manual

Table of Contents

- Table of Contents
- Introduction
- Mechanics
- Lexical Conventions
  - Tokens
  - Reserved Keywords
  - Comments
  - Identifiers
- Types
  - Primitive Types
- Expressions
  - Variable Declaration
  - Type Inference
  - Operators
  - Operator Precedence
  - Operator Usage
  - Type Definition
  - Casting
  - File Input/Output
  - Import and Export
- Control Flow
  - Conditional Branching
    - If/Else
  - Loops
    - While
- Functions
  - Built-In Functions
  - User-Defined Functions
  - Nested Functions
- Program Structure and Scope
  - Structure
  - Scope Rules
  - Compilation
  - File Extension
- Standard Library
  - Built-In Functions
- Examples

References

Introduction

This manual describes the PLTree programming language as submitted on Monday, October 26, 2015. PLTree is a language for usage and manipulation of trees where the main data type is a tree. Every variable will be treated as a tree; for example, a string in our language would be a tree with leaves of characters. Every function is also a tree, whose children are statements, which themselves are trees. The language will make it easy to create and edit trees with functions such as adding a new item at a certain position in the tree or deleting items. It will also make it simple to manipulate trees with common tree functions such as pruning, grafting, finding the root, and searching for an item. A selection of relevant tree functions are provided, and user-defined functions may also be created for working with trees.

Mechanics

In the PLTree programming language, everything is a tree. Variables may be declared with one of the given primitive types, or as `void`, indicating that they have no data member. Every variable may have some number of branches coming off of them as well. (See also 'Expressions' section.) Functions are trees whose children are expressions, which are themselves trees, and can therefore have children of their own. Function definition can be thought of as creating a new type of tree with the same name as that function. (See also 'Functions' section.) Declaring a new variable of this functionName type and given the value of a particular tree will execute the body of this function.
(To see demonstrations of PLTree in use, please refer to the 'Examples' section.)

Lexical Conventions

Tokens

The classes of tokens are: identifiers, keywords, constants, literals, and operators.

Reserved Keywords

```
int
double
char
bool
if
ifelse
while
return
void
tree
string
width
```

```
typedef
import
file
filesystem
```

Comments

The characters /* introduce a comment, which terminates with the characters */. Comments do not nest, and they do not occur within a string or character literals. Comments can span multiple lines; multi-line comments are written in the same way as single-line comments.

Single line comments are written as follows:

```
/* this is a single-line comment */
```

Multi line comments are written similarly:

```
/* this is
a
multi-line comment */
```

Identifiers

An identifier is a sequence of letters and digits, where the underscore (_) is included as a letter. Identifiers must begin with a letter and may be of any length. Upper and lowercase letters are different.

Types

All elements, including primitive types and collections, are trees. When a new variable is declared, a new tree is created. The smallest unit of the language is a single node. A node has a data member (for example, the integer 5) and may have any number of children. All trees are built from these nodes. In this manual, we use node when we are not concerned with the children of that node, and tree elsewhere.

Primitive Types

Literals may be of the following primitive types:

Type	**Syntax**
boolean (`true, false`)	`bool`
character	`char`
integer	`int`
double	`double`

A primitive type may be the root of a tree

Expressions

Variable Declaration

When a new variable is declared, a tree of a single node is created containing the data that variable is assigned to hold, of the type that the variable is declared as.

Variables are declared in the following syntax:

```
(type name literal_value children)
```

where `type` is one of the primitive types or `void`, name is an optional identifier for this tree, `literal_value` is either a literal or an expression which evaluates to the appropriate type and `children` is an optional argument that takes the same form as that of a `typedef` structure. When a variable is declared to be of type `void`, if there is a `literal_value` and `children`, `literal_value` is ignored. If only one argument is present, it is interpreted as `children`.

For example:

```
(int a 5)
(char b 'h')
(double c 8.8)
(void t ()) /*empty tree*/
```

There are also unnamed variables, requiring just the type and value, declared as such:

```
(int 5)
(char 'B')
```

Type Inference

All primitive types can be written as is, without a type declaration or identifier.
For example:

`(int 5)` can be written as `5`
`(char 'a')` can be written as `'a'`
`(double 3.5)` can be written as `3.5`

In addition, strings, which are really trees of `char`, can be declared in the following manner:

`(void ('h' 'e' 'l' 'l' 'o'))` can be written as `"hello"`

Note the use of double quotes around strings and single quotes around `char`s.

Operators

Arithmetic operators:

Addition	+
Subtraction	-
Multiplication	*

Division	/
Modulo	%
Unary Minus (make int negative)	-

Numerical relational operators:

Is equal to	==
Is not equal to	!=
Is less than	<
Is less than or equal to	<=
Is greater than	>
Is greater than or equal to	>=

Logical operators:

And	&&
Or	||
Not	!

Other operators:

Accessor	@
Width	#
Branch Accessor	[i]

Operator Precedence

Operator precedence follows standard order of operations for all operators. Innermost parentheses always come first. Arithmetic operators take precedence over numerical relational operators. Numerical relational operators take precedence over logical operators.

Operator Usage

Operator functions are also available in addition to the standard operators. Operator functions are to be used for objects, while standard operators are to be used for primitive types.

All unary operators immediately precede their operand (e.g. -1, !true), except for the one referred to as Branch Accessor, which immediately follows its operand. Binary operators separate their operands (e.g 0 - 1, false && true). Operators that act on primitives automatically access the data member of the operand(s).
For all binary operators, both operands must be of the same type. All arithmetic operators expect Unary Minus are binary. They are all (including Unary Minus) to be used for either `ints` or `doubles`. The same type will be returned. Numerical relational operators are to be used for either `ints` or `doubles`. A boolean value will be returned. The logical operators may only be used for `bools`. A boolean value will be returned. The operator referred to as Not (!) is a unary operator. The other logical operators are binary. The operator referred to as Accessor (@) is a unary operator whose operand is a node. The node's data member, which may be of any type, will be returned. If the node has no data member, that is of type `void`, the empty character string (‘’) will be returned. Width (#) is a unary operator, shorthand for the function width. Its operand is a tree of any type. It returns an int equal to the number of branches, also known as the degree, of that tree. The Branch Accessor Operator ([i]) is used to access the ith branch of its operand where i is an integer. Branches are 0-indexed. Attempting to access a branch that does not exist will return the empty character string (‘’).

Type Definition

Type definition is declared in the following format:
`( typedef type_name structure )`
where `typedef` is a keyword, `type_name` is a valid identifier, and `structure` is made up of nodes and ranges. A node is designated by a pair of parentheses, and optionally contains a combination of nodes and ranges. A range is designated by an opening curly brace, an integer literal, a comma, an optional integer literal, and a closing curly brace. The first number represents a minimum, while the second represents a maximum, both inclusive.

A range immediately following a node defines the number of siblings that node can have, including itself. Otherwise, a range defines the depth of its containing node. A node can contain at most one of this type of range. If the second number of a range is omitted, there is no upper bound. The omission of a range is equivalent to defining a range with a lower bound of 0 and no upper bound.
`(typedef string ((char {0,}){0,}))`

Casting

Casting from one type to another type is done like this:
`((new-type) name-of-object)`

As all items in PLTree are trees, the original object is a tree of a particular type, and the new type that casting will provide also results in a tree of a particular type. Casting therefore changes, in essence, the format of the tree to another tree format.

Casting is required to pass a variable value of one type into a variable or function argument expecting another type. If casting is not done properly, an error occurs.

From/To	bool	char	int	double
bool	unchanged	‘t’ if true, ‘f’ otherwise	1 if true, 0 otherwise	1.0 if true, 0.0 otherwise
char	true if ‘t’, false otherwise	unchanged	convert to ASCII value as int	convert to ASCII value as double
int	false if 0, true otherwise	convert to ASCII character	unchanged	add decimal point: [int].0
double	false if 0.0, true otherwise	convert to ASCII character	remove decimal point and all digits after decimal point	unchanged

File Input/Output

The file system is always available as the variable `filesystem`, representing a pseudo-tree whose root is the same of that as the file system. Subdirectories and files are accessed in a syntax similar to that of branch access, except using double-quoted strings or trees whose leaves are of type `char` instead of `int`s. For example:
`filesystem[“hello”][“world”]` or `filesystem[“hello/world”]` or `filesystem[“hello”][(void (‘w’ ‘o’ ‘r’ ‘l’ ‘d’))]`

Accesses the file
“`/hello/world`” in UNIX.

Import and Export

All PLTree source code files are importable. They may be imported like this:
`import filesystem[“path”][“to”][“name-of-file.tree”]`

Libraries may be imported like this:
`import name-of-library`

Control Flow

Support for conditional branching in the form of if-statements and if-else statements, as well as loops in the form of while loops, are included.

Conditional Branching

If

If statements are written like this:

```
(if (condition) (
            /* code here */
      )
)
```

If/Else

If/else statements are written like this:

```
(ifelse (condition) (
            /* condition met; do something */
      )
      (
            /* condition not met; do something */
      )
)
```

Loops

While

While loops are written like this:

```
(while (condition) (
            /* condition met; do something */
      )
)
```

Functions

Functions are actually a specialized type of tree. Execution of a function always proceeds in depth-first search order through the function tree.

Built-In Functions

Built in functions are provided to all files by the standard library.

User-Defined Functions

All functions accept as an argument either zero or one trees, which may itself contain additional arguments as its children. A function is declared in the following syntax:

```
(type name (void t) (
      /* do something */
      (return value)
))
```

Where `type` is the return type, `name` is the name of the function, `(void t)` is the input argument, and the last parentheses `( /*do something */ )` surround an execution statement consisting of any number of statements. A function may return any valid type. A function must always return a value of the appropriate type. If a function never returns, its return type should be declared as `void`.

Nested Functions

Functions may be defined within functions; these are known as nested functions. As functions are trees in PLTree, nested functions further extend the tree with additional branching.

Program Structure and Scope

Structure

A PLTree program may exist entirely within a single source file or within multiple source files. A source file may include and link with files from existing libraries or other files. By convention import statements are typically to be included in the header of a source code file.

Scope Rules

If there is data in the node that you are in and it is above you, then you can see that. You can see your parents'/ancestors' data on the branch that you are on. Same scoping rules as C.

Compilation

A compiler is provided for source code written in PLTree. Source code compiles to LLVM.

File Extension

File extension for all PLTree programs shall be `.tree`

Standard Library

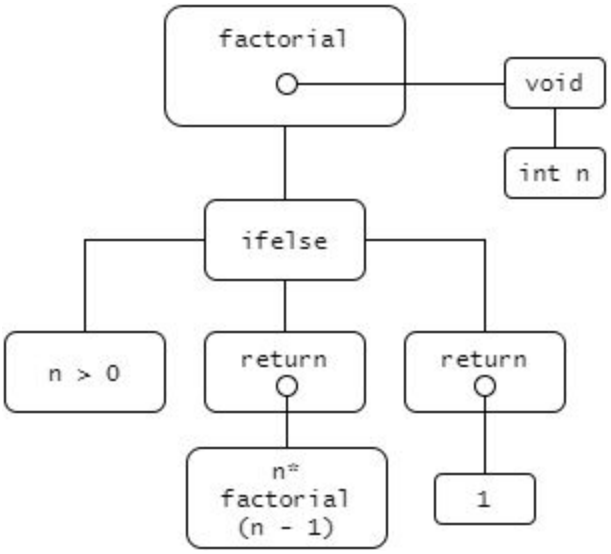
Built-In Functions

[Please refer to the Standard Library document for a current list.]

Examples

```
/* factorial function */
(int factorial (void {int n}) (
     (ifelse (n > 0)
          (return (n * (factorial n - 1)))
          (return 1)
     )
))
```

```
/* diagram of factorial function tree
   -  not part of code or output */
```




```
/* print implementation */

/*type name   argument  */
(void print (void instring) (
     /* variable declaration */
     (int i 0)

     /*loop     comparison */
     (while (i < (width instring)) (
          /* variable declaration */
          (void node instring[i])

          /* if     condition */
          (ifelse (isleaf node)
               /* if branch */
               (
                    (putchar node)
               )

               /* else branch */
               (
```

```
                    (print node)
                )
            )

            /* increment */
            (i (i+1))
        ))
))

/* an example of type inference */
(print “Hello, World!\n”)
```

References

B.W. Kernighan and D.M. Ritchie. "Appendix A - Reference Manual," in *The C Programming Language, 2nd edition*. Murray Hill, NJ: AT&T Bell Laboratories.