FRY Language Reference

Tom DeVoe
tcd2123@columbia.edu

October 23, 2014

Contents

1 Introduction

This document serves as a reference manual for the **FRY** Programming Language. **FRY** is a language designed for processing delimited text files.

2 Lexical Conventions

2.1 Comments

Single line comments are denoted by the character, `#`. Multi-line comments are opened with `#/` and closed with `/#`.

```
# This is a single line comment

#/ This is a
multi-line comment /#
```

2.2 Identifiers

An identifier is a string of letters, digits, and underscores. A valid identifier begins with an letter or an underscore. Identifiers are case-sensitive and can be at most 31 characters long.

2.3 Keywords

The following identifiers are reserved and cannot be used otherwise:

```
int    str    float  bool    Layout
List   Table  if     else    elif
in     Sort   not    typeof  and
or
```

2.4 Constants

There is a constant corresponding to each Primitive data type mentioned in 4.1.1.

- **Integer Constants** - Integer constants are whole base-10 numbers represented by a series of numerical digits (0 - 9) and an optional leading sign character(+ or −). Absence of a sign character implies a positive number.

- **Float Constants** - Float constants are similar to Integer constants in that they are base-10 numbers represented by a series of numerical digits. However, floats must include a decimal separator and optionally, a fractional part. Can optionally include a sign character (+ or −). Absence of a sign character implies a positive number.

- **String Constants** - String constants are represented by a series of ASCII characters surrounded by quotation-marks (`" "`). Certain characters can be escaped inside of Strings with a backslash `'´`. These characters are:

Character	**Meaning**
`\n`	Newline
`\t`	Tab
`\\`	Backslash
`\"`	Double Quotes

- **Boolean Constants** - Boolean constants can either have the case-sensitive value *true* or *false*.

3 Syntax Notation

Borrowing from the *The C Programming Language* by Kernigan and Ritchie, syntactic categories are indicated by *italic* type and literal words and characters in `typewriter` style. Optional tokens will be underscored by $_{opt}$.

4 Meaning of Identifiers

4.1 Types

4.1.1 Basic Types

- `int` - 64-bit signed integer value
- `str` - An ASCII text value
- `float` - A double precision floating-point number
- `bool` - A boolean value. Can be either `true` or `false`

4.1.2 Compound Types

- `List` - an ordered collection of elements of the same data type. Every column in a *Table* is represented as a List. Lists can be initialized to an empty list or one full of values like so:
- `Layout` - a collection of named data types. Layouts behave similar to structs from C. Once a Layout is constructed, that layout may be used as a data type. An instance of a Layout is referred to as a *Record* and every table is made up of records of the Layout which corresponds to that table.
- `Table` - a representation of a relational table. Every column in a table can be treated as a *List* and every row is a record of a certain *Layout*. Tables are the meat and potatoes of **FRY** and will be the focus of most programs.

5 Conversions

Certain operators can cause different basic data types to be converted between one another.

5.1 Integer and Floating

5.2 Arithmetic Conversions

5.3 String Conversions

6 Expressions

An expression in **FRY** is a combination of variables, operators, constants, and functions. The list of expressions below are listed in order of precedence. Every expression in a subsection shares the same precedence (ex. Identifiers and Constants have the same precedence).

6.1 Primary Expressions

primary-expression :
  identifier
  constant
  (expression)

Primary Expressions are either identifiers, constants, or parenthesized expressions.

6.1.1 Identifiers

Identifiers types are specified during declaration by preceding that identifier by its type. Identifiers can be used for any primitive or compound data types and any functions.

6.1.2 Constants

Constants are either integer, string, float, or boolean constants as specified in 2.4

6.1.3 Parenthesized Expressions

Parenthesized expression is simply an expression surrounded by parentheses.

6.2 Postfix Expression

Operators in a postfix expression are grouped from left to right.

postfix-expression :
  primary-expression
  postfix-expression[expression(:expression)$_{opt}$]
  postfix-expression.{expression(:expression)$_{opt}$}
  postfix-expression(argument-list$_{opt}$)
  expression−−
  expression++

6.2.1 List Element Reference

A list identifier followed by square brackets with an integer-valued expression inside denotes referencing the element at that index in the List. For instance `MyLst[5]` would reference the 6^{th} element of the List, *MyLst*. Similarly, `MyLst [n]` would reference the $n - 1^{\text{th}}$ element of MyLst. The type of this element is the same as the type of elements the List you are accessing contains.
Sublists can be returned by *sliceing* the list. By specifying the optional colon (':') and indices before and/or after, the list is sliced and a sublist of the original list is returned. If there is an integer before the semi-colon and none after, then a sublist is returned spanning from the integer to the end of the list. If there

is an integer after the colon and none before, the a sublist is returned spanning from the beginning of the list to the integer index. If there is an integer before and after the colon, then a sublist is returned spanning from the first integer index to the second integer index.

6.2.2 Layout Element Reference

A layout identifier followed by a dot and an expression in braces references an element of a layout. The expression in the braces must either be *(i)* the name of one of the member elements in the Layout you are accessing, such as `MyLyt.{elem_name}` or *(ii)* a integer reference to the n^{th} element of the Layout, i.e. `MyLyt.{2}` would access the 1^{st} member element. The type of the element returned will be the type that element was defined to be when the Layout was defined. If the member element you are accessing is itself a Layout, then the numeric and identifier references will both return a element of that Layout type. Sublayouts can be returned by *sliceing* the layout. Layout slicing syntax is mostly the same as the List slicing, except you can also specify element names as the indices on either side of the colon. This returns an instance of unamed layout type with unamed elements.

6.2.3 Function Calls

A function call consists of a function identifer, followed by parentheses with a possibly empty argument list contained. A copy is made of each object passed to the function, so the value of the original object will remained unchanged. Function declarations are discussed in 7.5.

6.2.4 $--$*expression*

The double minus sign ('-') decrements an integer value by 1. The type of this expression must be integer.

6.2.5 $++$*expression*

The double plus sign ('+') increments an integer value by 1. The type of this expression must be integer.

6.3 Unary Operators

Unary operators are grouped from right to left and include logical negation, incrementation, and decrementation operators.

unary-expression :
  postfix-expression
  `not` *unary-expression*
  `typeof`(*primary-expression*)

6.3.1 `not` *expression*

The `not` operator represents boolean negation. The type of the expression must be boolean.

6.3.2 `typeof(`*expression*`)`

The `typeof` operator returns the type of some identifier as a string.

6.4 Multiplicative Operators

These operators are grouped left to right.

multiplicative-expression :
    *multiplicative-expression*multiplicative-expression*
    multiplicative-expression/multiplicative-expression
    multiplicative-expression/multiplicative-expression

$*$ denotes mutltiplication, / denotes division, and % returns the remainder after division (also known as the modulo). The expressions on either side of these operators must be integer or floating point expressions. If the operand of / or % is 0, the result is undefined.

6.5 Additive Operators

These operators are grouped left to right.

additive-expression :
    multiplicative-expression
    additive-expression+additive-expression
    additive-expression-additive-expression

$+$ and $-$ denote addition and subtraction of the two operands respectively. Additionally the $+$ also denotes string concatenation. For $-$, the expressions on either side of the operators must be either integer or floating point valued. For $+$, the expressions can be integer, floating point or strings.

6.6 Containment Operators

containment-expression :
    additive-expression `in` *containment-expression*
    additive-expression `not in` *containment-expression*

The containment operators check whether an element is contained in a List. The right operand must be a List and the left operand must be the same type as the elements that List contains. Both operators return a boolean value. If the element is in the list, then `in` returns `true` and if the element is not in the list, it returns `false`. `not in` returns the opposite values.

6.7 Relational Operators

relational-expression :
  additive-expression
  relational-expression>*relational-expression*
  relational-expression>=*relational-expression*
  relational-expression<*relational-expression*
  relational-expression<=*relational-expression*

> represents greater than, >= represents greater than or equal to, < represents less than, and <= represents less than or equal to. These operators all return a boolean value corresponding to whether the relation is `true` or `false`. The type of each side of the operator should be either integer or floating point.

6.8 Equality Operators

equality-expression :
  relational-expression
  equality-expression==*equality-expression*
  equality-expression! =*equality-expression*

The == operator compares the equivalence of the two operands and returns the boolean value `true` if they are equal, `false` if they are not. ! = does the opposite, `true` if they are unequal, `false` if they are equal. The operands can be of any type - as long as they are both the same type.

6.9 Logical AND Operator

The logical AND operator is grouped left to right.

logical-AND-expression :
  equality-expression
  logical-AND-expression `and` *logical-AND-expression*

The logical *and* operator (`and`) only allows for boolean valued operands. This operator returns the boolean value true if both operands are true and false otherwise.

6.10 Logical OR Operator

The logical OR operator is grouped left to right.

logical-OR-expression :
  logical-AND-expression
  logical-OR-expression `or` *logical-OR-expression*

The logical *or* operator (`or`) only allows for boolean valued operands. This operator returns the boolean false if both operands are false and true otherwise.

6.11 Assignment Expressions

Assignment operators are grouped right to left.

assignment-expression :
  logical-OR-expression
  primary-expression=assignment-expression

Assignment operators expect a variable identifier on the left and a constant or variable of the same type on the right side.

7 Declarations

Declarations give the ability to define identifiers type and value.

declarations :
  type-specifier declarator
  type-specifier declarator = initializer

7.1 Type Specifiers

The different *type-specifiers* available are:

type-specifiers :
  `int`
  `str`
  `float`
  `bool`
  `List`
  `Layout`
  `Table`

Exactly one type-specifier must be provided during a declaration. These types are described in more detail in 4.1.

7.2 List Declarations

A List is an ordered collection of elements of the same type.

List-intializers :
  `List` *identifier*
  `List` *identifier* `= [` *List-declaration-list* `]`

A List declaration consists of the keyword `List` followed by an identifier for the list and optionally followed by an assignment from a *List-declaration-list* surrounded by square brackets.

List-declaration-list :
  constant-or-identifier
  List-declaration-list, constant-or-identifier

constant-or-identifier :
  identifier
  constant

The *List-declaration-list* is a comma-separated list of identifiers or constants which define the values contained in that list. The associated type of the list is determined by the first element in the *List-declaration-list*. It is invalid to add elements of a different type than the first element to that list. Declaring a List with no *List-declaration-list* initializes an empty list with no associated type. The first element added to that list sets that List's type.

For example,

```
List l_int = [1,2,3,4]
List l_empty
List l_str = ["This", "is", "a", "list"]
```

declares a list containing the first for natural integers, an empty list, and a list of strings.

7.3 Layout Declarations

A Layout is a collection of optionally named members of various types.

Layout-intializers :
  `Layout` *identifier* = { *Layout-declaration-list, Print-specifier*$_{opt}$ }

A Layout declaration consists of the keyword `Layout` followed by an identifier and then an assignment from a *Layout-declaration-list* surrounded by curly braces. It is also optional to define a *Print-specifier*, which is a string that defines how this Layout should be formatted when printed.

Layout-declaration-list :
  Layout-element
  Layout-declaration-list, Layout-element

Layout-element :
  type-specifier : *identifier*$_{opt}$

Print-specifier :
  `Print`: *string-identifier-or-constant*

The *Layout-declaration-list* is a comma-separated list of *Layout-elements* which defines the members of the Layout being declared. If no identifier is provided for an element, it can be accessed using the numeric Layout element reference as described in 6.2.2. If no *Print-specifier* is defined, the elements are printed in the same order they were defined with any specified delimiter.

An instance of an already created layout is created using similar syntax to the declaration:

Layout-instance-creation :
  `Layout` *identifier identifier*
  `Layout` *identifier identifier* `=` { *Layout-instance-list* }

The first identifier is the identifier of the Layout you are creating an instance of and the second is the identifier of this layout instance.

Layout-instance-list :
  constant-or-identifier
  Layout-instance-list **,** *constant-or-identifier*

The types of each element in the *Layout-instance-list* must match the types of the members of the Layout type. Each member of the Layout instance declared is assigned the value of the *constant-or-identifier* at that member's position. A mismatch between the member type and the constant or identifier type is considered an error.

Some examples of Layout declaration and creation are

```
Layout date = { int: mon, int: day, int: year \
                Print:mon+"-"+day+"-"+year }
Layout date today = {10, 23, 2014}
Write(stdout, today)
# 10-23-2014
Layout date nextweek = { today.mon, today.day+7, \
                               today.year}
Layout userinfo = {str: Fname, str: Lname, \
                        Layout date:bday}
```

In this example, date is defined as a Layout with three integer fields, for month, day, and year, and has a *Print-specifier* defined. An instance of the date Layout is created with value 10-23-2014 and written to stdout using the Write function (part of the standard library). The date Layout instance `nextweek` is computed from today's date. The `userinfo` layout is created with the date Layout nested inside of it.

7.4 Table Declarations

7.5 Function Declarations

8 Statements

9 Scope