FRY Language Reference

Tom DeVoe
tcd2123@columbia.edu

December 11, 2014

Contents

0.1 Introduction

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

0.2 Lexical Conventions

0.2.1 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 /#
```

0.2.2 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.

0.2.3 Keywords

The following identifiers are reserved and cannot be used otherwise:

```
int     str     float   bool     Layout
List    Table   if      else     elif
in      not     and     stdout
or      Write   Read    stderr   true
false
```

0.2.4 Constants

There is a constant corresponding to each Primitive data type mentioned in 0.4.1.1.

- **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*.

0.3 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 $_{opt}$.

0.4 Meaning of Identifiers

0.4.1 Types

0.4.1.1 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`

0.4.1.2 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.

0.5 Conversions

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

0.5.1 Integer and Floating

Integer and Floating point numbers can be converted between each other by simply creating a new identifier of the desired type and assigning the variable to be converted to that identifier. For example, to convert an integer to a floating point number:

```
int i = 5
float f = i
Write(stdout, f)
# 5.0
```

When converting a floating point number to an integer, any fractional part will be truncated:

```
float f = 5.5
int i = f
Write(stdout, i)
# 5
```

0.5.2 Arithmetic Conversions

For any binary operator with a floating point and an integer operator, the integer will be promoited to a float before the operation is performed.

```
float f = 5.25
int i = 2
```

```
Write(stdout, f*i)
# 10.50
Write(stdout, f-i)
# 3.25
```

0.5.3 String Conversions

String conversions are automatically performed when a non-string variable is concatenated with a string variable.

0.6 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 subsubsection shares the same precedence (ex. Identifiers and Constants have the same precedence).

0.6.1 Primary Expressions

primary-expression :
identifier
literal
(expression)

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

0.6.1.1 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.

0.6.1.2 Literal

Literals are either integer, string, float, or boolean constants as specified in 0.2.4

0.6.1.3 Parenthesized Expressions

Parenthesized expression is simply an expression surrounded by parentheses.

0.6.2 Set Builder Expressions

set-build-expression:
primary-expression
[*return-layout* | *identifier* <- *set-build-expression* ; *expression*]

A set-build-expression consists of a *return-layout*, which is the format of the columns which should be returned, an identifier for records in the table identifier specified by *set-build-expression*, and an *expression* which is a boolean expression.

The Set-builder notation evaluates the boolean expression for every record in the source table. If the boolean expression is true, then the *return-layout* is returned for that record. The Set Builder expression finally returns a table composed of all of the records which passed the boolean condition, formatted with the *return-layout*.

return-layout:
identifier
{ *layout specifier*$_{\text{opt}}$ *layout-instance-list* }

The *return-layout* must be a Layout type, and can be either a Layout *identifier* or a *Layout-instance-list* as described in 0.7.3.

0.6.3 Postfix Expression

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

postfix-expression :
set-build-expression
postfix-expression[slice-opt]
postfix-expression.{*expression*$_{opt}$}
expression−−
expression++

slice-opt :
:expr
expr:
expr:expr
expr

0.6.3.1 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 *slicing* 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.

0.6.3.2 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.

0.6.3.3 *expression*−−

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

0.6.3.4 *expression*++

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

0.6.4 Prefix Expressions

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

prefix-expression :
postfix-expression
`not` *unary-expression*

0.6.4.1 `not` *expression*

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

0.6.5 Multiplicative Operators

These operators are grouped left to right.

multiplicative-expression :
unary-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.

0.6.6 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. Both operands must be strings or both operands must be float/int. You cannot mix string operands with numeric operands.

0.6.7 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.

0.6.8 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. This operator compares the value of the identifier, not the reference for equivalence. The operands can be of any type, but operands of two different types will never be equivalent.

0.6.9 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.

0.6.10 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.

0.6.11 Assignment Expressions

Assignment operators are grouped right to left.

assignment-expression :
logical-OR-expression
identifier=*assignment-expression*

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

0.6.12 Function Calls

func-call :
assignment-expression
assignment-expression(argument-list$_{opt}$*)*

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 0.7.4.

0.6.13 List Initializers

list-initializer :
  func-call
  [*list-intializer-list*]
  [*func-call* `to` *func-call*]

list -initializer-list :
  func-call
  list-intializer-list,func-call

A list initializer generates a list containing a range of values. The first form creates a list containing the values specified in the *list-initializer-list*. Every element of the *list-initializer-list* needs to be of the same type or an exception is thrown at compile time. The second form takes two integer values and returns an inclusive list containing the values from the first integer to the second. The first integer must be smaller than the second integer.

0.6.14 Layout Initializer

layout-initializer :
  list-initializer
  `Layout` *identifier layout-initializer-list*

layout-initializer-list :
  list-initializer
  list-intializer-list,list-initializer

A layout initializer creates an instance of the layout type specified. The type of each layout initializer field needs to match those defined in the layout.

0.6.15 Table Initializer

table-initializer :
  layout-initializer
  `Table` ($full\text{-}type_{opt}$)

A Table initializer, initializes a table and optionally associates a layout with that table. *full-type* is described in detail in 0.7.1

0.6.16 Expressions

expression :
  table-initializer

0.7 Declarations

0.7.1 Type Specifiers

The different type specifiers available are:

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

full-type :
    type-specifier
    type-specifier `List`
    `Layout` *identifier*

0.7.2 Variable Declarations

variable-declaration :
    full-type declarator

declarator :
    identifier
    identifier `=` *expr*

When assigning a value in a variable declaration, the type of the identifier in *full-type* must match that of the *expr* which it is assigned.

0.7.3 Layout Declarations

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

layout-declaration :
    `Layout` *identifier* `=` { *layout-declaration-list* }

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.

layout-declaration-list :
    layout-element
    layout-declaration-list, layout-element

layout-element :
    full-type : $identifier_{opt}$

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 0.6.3.2.

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

Layout creations have a few special rules:

- Layout delcarations are treated as special statements in that they are evaluated out of order versus other statements. For example, you can create layouts at the end of your program, and reference that layout type as though it were created in the beginning. However, if that layout declarations have order respective to each other.
- Layouts are only allowed to be declared on the top level of scoping.

0.7.4 Function Declarations

Function declarations are created along with their definition and have the following format:

function-declaration :
full-type identifier (*parameter-list*$_{opt}$) { *statement-list* }

The full-type in the beginning of the function declaration specifies what type is returned by that function. The identifier that follows is the name of the function and will be referenced anytime that function should be called.

Then there is a *parameter-list*, i.e. a list of arguments, inside of parentheses.

parameter-list :
type-specifier identifier
parameter-list, type-specifier identifier

These arguments must be passed with the function whenever it is called.

After the arguments comes the function definition inside of curly braces. The definition can contain any number statements, expressions, and declarations. The one caveat is the definition must contain a *ret* statement for the return type indicated. If the function does not need to return a value, it is a best practice to return an int as the error code. You can **overload** functions, meaning you can have multiple functions with the same name, so long as they have different signatures.

0.8 Statements

Unless otherwise described, statements are executed in sequence. Statements should be ended by a semi-colon(";"). Statements can be broken up into the following:

statement :
    expression-statement
    return-statement
    statement-block
    conditional-statement
    iterative-statement
    variable-declaration
    layout-declaration
    function-declaration

Statements are separated by newlines and a series of statements will be called a *statement-list*.

statement-list :
    statement
    statement-list `\n` *statement*

0.8.1 Expression Statements

Expressions statements make up the majority or statements:

expression-statements :
    expression

An expression statement is made up of one or more expressions as defined in 0.6. After the entire statement is evaluated and all effects are completed, then the next statement is executed.

0.8.2 Return Statement

return-statements :
  `ret` *expr*

A return statement is included in a function and indicates the value to be returned by that function. This expr needs to have the same type as defined in the function declaration.

0.8.3 Statement Block

statement-block :
    { *statement-list* }

A statement block groups multiple statements together. Any variable declared in a statement block is only in scope until that statement block is closed.

0.8.4 Conditional Statements

Conditional statements control the flow of a program by performing different sets of statements depending on some boolean value.

conditional-statements :
`if (`*expression*`)` *statement elif-list*
`if (`*expression*`)` *statement elif-list* `else` *statement*

elif-list :
`elif (`*expression*`)` *statement*
elif-list `elif (`*expression*`)` *statement*

The expression in the parentheses after `if`, `elif`, and `else` must be boolean-valued. If it is true, execute the corresponding statement and jump out of the conditional expression. If it is false, do not execute the statement and evaluate the next expression after an `elif` or `else`.

0.8.5 Iterative Statements

iterative-statements:
`for` *identifier* `<-` *expression statement*
`while (` *expression* `)` *statement*

0.8.5.1 for loop

The type of the expression following the left arrow ("¡-") must be a list. A *for* loop executes the *statement* once for each elements in that List.

0.8.6 while loop

The *expression* inside of the parentheses of a while loop must be boolean-valued. The while loop repeatedly executes the *statement-list* as long as the value of the expression is `true`.

0.9 Scope

Scope is handled simply in **FRY**, a variable cannot be referenced outside of the code block it was declared inside. In most cases, this block is denoted by curly braces. One exception is the *elements-of* subsection of a Set-builder statements 0.6.2, the scope for these variables are only inside the Set Builder statement (i.e. inside the square brackets). Any variable delcared outside of any code block is considered a global variable and can be referenced anywhere in the program.