F.R.N.D.

Friendly, Radiant, and New Dialect Language Design and Example Programs

:)

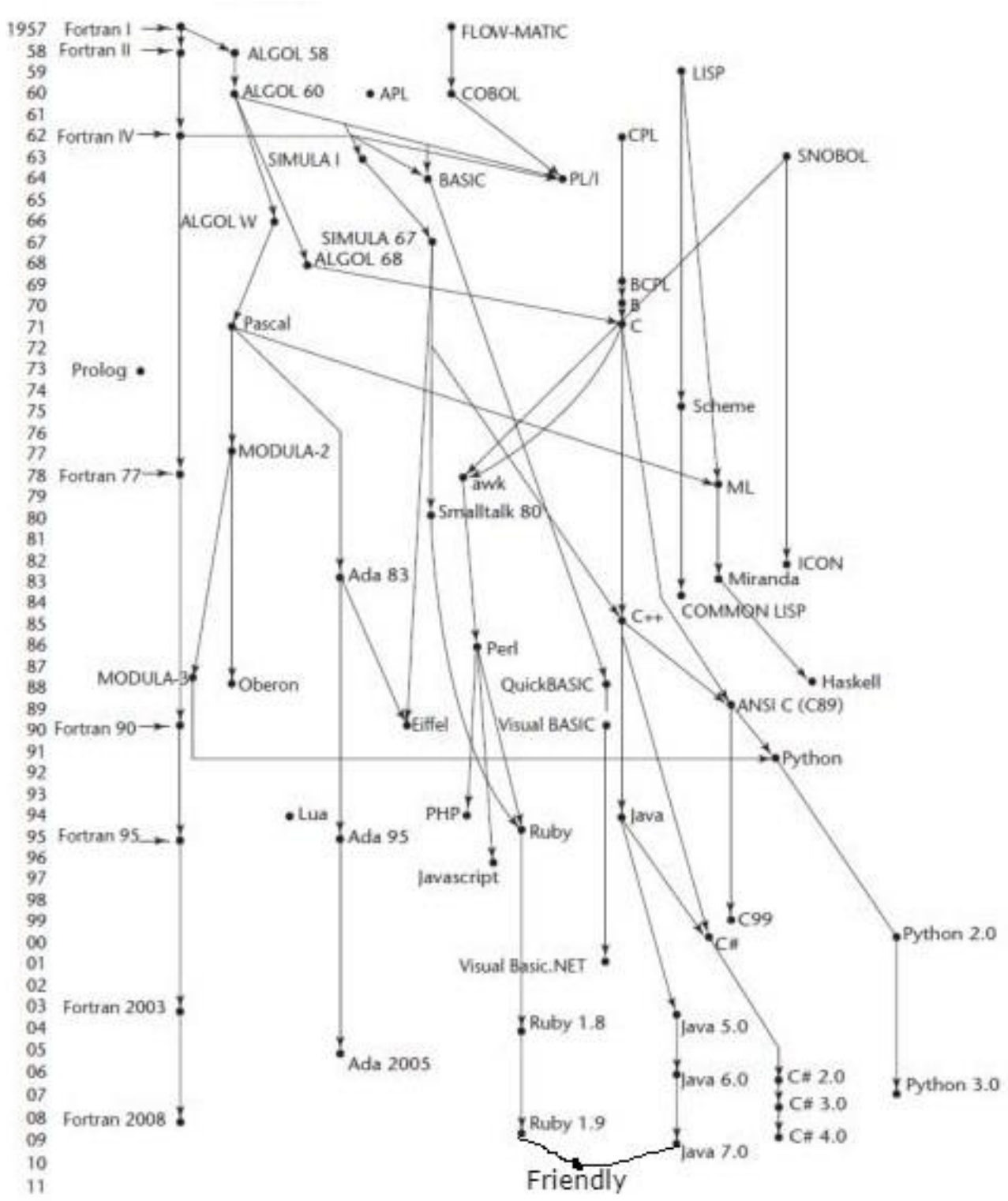
Version 2.4.6

1. Introduction

F.R.N.D. is a simple, well-mannered, modern, object-oriented, and (strongly) type-safe programming language. Inspired by the times I have cursed out my computer while coding and wanting to make a language that forces you to be friendly while coding so that less money goes into the swear jar. Also inspired by the idea of it being an educational language to simplify some of the syntax for Java. Based on Java and Ruby, but differing in the following ways:

1. Does not use a semicolon, instead statements are ended with “please”.
2. “Thankyou” must be present at the end of every class (friend).
3. Set variables with :=
4. Subtraction is still indicated with the symbol - but negative is now indicated with ~ to differentiate them.
5. Concatenation no longer is indicated with +, but instead _ to differentiate it from addition.
6. print/puts is replaced with say.
7. class is replaced with friend.
8. Ruby’s def is replaced with play.
9. return is replaced with borrow.
10. String is changed to Str and boolean is changed to bool because if integer can be shortened to int they should be shortened too.
11. Similar to Ruby, you can insert code and variables into Strings without the need of concatenation, however, it has been changed from #{ } to ${ } since I believe ‘#’ implies an int.
12. Java’s class, `Math`, which has methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions is renamed to `Smart`.
13. Arrays have been changed a little.
    a. Java declares arrays with:
        i. int nums[] = new int[10];
    b. F.R.N.D. declares arrays with:
        i. int[10] nums[] please
        ii. You can now set the size without having to declare it to a new int array of a set size

1.1. Genealogy



1.2. Hello world

```
friend HelloWorld
{
   play main()
   {
      say “Hello, World!” please
   }
}
Thankyou
```

1.3. Program structure

The key organizational concepts in F.R.N.D. are as follows:

1. All friends (classes) are public because it is rude to be embarrassed about other friends knowing your friend.
2. Similar to Java, there must be a main() function, but now it no longer needs the parameter of String args[]. Instead, the compiler will see main() and automatically infer the parameter of String args[]. You can type Str[] args into the parameters of main if you would like though.
3. Brackets MUST be on their own line when creating blocks of code.
4. Similar to Ruby, functions are defined very simply without the need of things like “public” or “static”, but the function must declare return type and parameter types like Java.
5. Thank you MUST be present on its own line directly after a friend’s (class’) closing bracket.

Example Program:

```
friend Ellipse
{
   whisper*main method*
   play main()
   {
      int width please
      int height please

      width := 5 please
      height := 7 please
```

```
        int ellipseArea := area(width, height)
        say. (“An ellipse with a width of ${width} and a height of
        ${height} has an area of ${ellipseArea}.”) please
    }

    whisper*method to calculate area*
    play int area (int w, int h)
    {
        borrow w * h * Smart.PI please
    }
}
Thankyou
```

declares a *friend(class)* named `Main.` `Main` contains several members: three fields named `width`, `height`, and `ellipseArea`, plays(methods) named `area` and `main`. The play `area` has the parameters for two integer values and it takes those two integer values and you borrow(return) the double for the two integer values multiplied together then multiplied by pi. The play `main` calls `area` passing `width` and `height` in order to get the area of the ellipse and set it to the field `ellipseArea`. Then says(prints) a statement using the three fields in `Main`.

1.4. Types and Variables

There are two kinds of types in F.R.N.D.: ***value types*** and ***reference types***. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable.

1.5. Visibility

All friends are public. It is much more friendly for friends to be public to each other.

1.6. Statements Differing from Ruby and Java

Statement	Example

Expression statement	play main()
	{
	int num please
	num := 123 please
	Str message please
	message := “Hello” please
	
	whisper*creates the int array nums of size 3*
	int[3] nums[] please
	nums := [0, 2, 4] please
	
	whisper* compiler will also infer the size based on
	what you initially put into the array.
	In this case the array will be size 3*
	int[] nums[] := [0, 2, 4] please
	}

if statement	play main()
	{
	whisper* \= means does not equal *
	if (x == 0 AND x \= 1)
	{
	say.("X is 0") please
	}
	elif(x > 0 OR x >= 1)
	{
	say.(“X is a positive number”) please
	}
	else
	{
	say.("X is a negative number") please
	}
	
	whisper* str=, str>, str< allows to compare strings
	alphabetically without using the annoying
	compareTo in Java.
	In this case, AA str> AB is true.
	*
	if (text1 str= text2)
	{
	say.(“They are the same string.”)
	}
	}

Comments	whisper*this is a comment* whisper*this comment is opened for multiple lines now it is closed after this line* whisper this is a one line comment

Print Statements	play main()
	{
	  whisper*These will all print the same statement*
	  say “Hello, World!” please
	  say (“Hello, World!”) please
	  say (“Hello, “ _ “ World!”) please
	
	  Str greeting := “Hello,” please
	  say “${greeting} World!” please
	  say (“${greeting} World!” please
	
	  whisper*to skip a line after printing*
	  say. “Hello, World!” please
	  say “This will print on the next line” please
	}

loops	play main()
	{
	  whisper* for loop *
	  loop for (int i := 0 | i < 5 | i++)
	  {
	  say. i please
	  }
	
	  whisper* while loop *
	  i := 0 please
	  loop while (i < 5)
	  {
	  say. i please
	  i++ please
	  }
	}

2. Lexical structure

2.1. Programs

A F.R.N.D. ***program*** consists of one or more ***source files***. A source file is an ordered sequence of (probably Unicode) characters.

Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

2.2. Grammars

This specification presents the syntax of the F.R.N.D. programming language where it differs from Java and Ruby.

2.2.1. Lexical grammar (tokens) where different from Java and Ruby

Regular Expressions for tokens.

<assignment operator> ➡ :=

<boolean operator> ➡ == | \= | <= | >= | < | > | str= | str\= | str<= | str>= | str< | str>

<print> ➡ say | say.

<String concatenation> ➡ _

2.2.2. Syntactic (“parse”) grammar where different from Java and Ruby

BNF grammar productions.

<class> ➡ friend <identifier>
{
<method>
}
Thank you

<method> ➡ play <identifier> (<parameters>)
➡ play <type> <identifier> (<parameters>)
➡ Є

<parameters> ➡ <parameter> <parameters>
<parameter>

<parameter> ➡ <type> <identifier>
➡ €

2.3. Lexical analysis

2.3.1. Comments

There are two forms of comments that are supported for F.R.N.D. : single-line comments and delimited comments.

Single-line comments start with the keyword `whisper` and extend to the end of the source line.

Delimited comments start with the keyword `whisper*` and end with the character `*`. Delimited comments may span multiple lines. Comments do not nest.

2.4. Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens where needed.

tokens:
identifier
keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator

2.4.1. Keywords different from Java or Ruby

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the `@` character.

New Keywords	
• `friend` • `play` • `Thankyou` • `please` • `say` • `whisper` • `borrow`	• `Str` • `bool` • `dec` • `loop` • `elif` • `AND` • `OR`

Removed Keywords
• `class` • `void` • `String` • `boolean` • `def` • `puts/print` • `double` • `float` • `real`

3. Type System

F.R.N.D uses a **strong static** type system. Strong typing means that type errors are caught and expressed to the programmer during compilation. Static typing means early binding compile-time type checking.

3.1. Type Rules

The type rules for F.R.N.D. are as follows:

Type Systems		
Assignment with Scope Context	$S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a primitive type $S \vdash e_1 := e_2 : T$	
Comparisons with Scope Context	$S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a primitive type (not Str/char) $S \vdash e_1 == e_2 : bool$	$S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a Str or char $S \vdash e_1$ str= $e_2 : bool$
	$S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a primitive type (not Str/char) $S \vdash e_1 \backslash= e_2 : bool$	$S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a Str or char $S \vdash e_1$ str\= $e_2 : bool$
	$S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a primitive type (not Str/char) $S \vdash e_1 >= e_2 : bool$	$S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a Str or char $S \vdash e_1$ str>= $e_2 : bool$
	$S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a primitive type (not Str/char) $S \vdash e_1 <= e_2 : bool$	$S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a Str or char $S \vdash e_1$ str<= $e_2 : bool$

	$S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a primitive type (not Str/char) $S \vdash e_1 > e_2 : bool$ $S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a primitive type (not Str/char) $S \vdash e_1 < e_2 : bool$	$S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a Str or char $S \vdash e_1$ str> $e_2 : bool$ $S \vdash e_1 : T$ $S \vdash e_2 : T$ T is a Str or char $S \vdash e_1$ str< $e_2 : bool$
Addition with Scope Context	$S \vdash e_1 : int$ $S \vdash e_2 : int$ $S \vdash e_1 + e_2 : int$ $S \vdash e_1 : dec$ $S \vdash e_2 : dec$ $S \vdash e_1 + e_2 : dec$ $S \vdash e_1 : int$ $S \vdash e_2 : dec$ $S \vdash e_1 + e_2 : dec$	$S \vdash e_1 : Str$ $S \vdash e_2 : Str$ $S \vdash e_1 \_ e_2 : Str$ $S \vdash e_1 : Str$ $S \vdash e_2 : int$ $S \vdash e_1 \_ e_2 : Str$ $S \vdash e_1 : Str$ $S \vdash e_2 : dec$ $S \vdash e_1 \_ e_2 : Str$ $S \vdash e_1 : Str$ $S \vdash e_2 : char$ $S \vdash e_1 \_ e_2 : Str$ $S \vdash e_1 : char$ $S \vdash e_2 : char$ $S \vdash e_1 \_ e_2 : Str$ $S \vdash e_1 : char$ $S \vdash e_2 : int$ $S \vdash e_1 \_ e_2 : Str$

		$S \vdash e_1$: char $S \vdash e_2$: dec $S \vdash e_1 \_ e_2$: Str
Subtraction with Scope Context	$S \vdash e_1$: int $S \vdash e_2$: int $S \vdash e_1 - e_2$: int $S \vdash e_1$: dec $S \vdash e_2$: dec $S \vdash e_1 - e_2$: dec $S \vdash e_1$: int $S \vdash e_2$: dec $S \vdash e_1 - e_2$: dec	
Multiplication with Scope Context	$S \vdash e_1$: int $S \vdash e_2$: int $S \vdash e_1 * e_2$: int $S \vdash e_1$: dec $S \vdash e_2$: dec $S \vdash e_1 * e_2$: dec $S \vdash e_1$: int $S \vdash e_2$: dec $S \vdash e_1 * e_2$: dec	
Division with Scope Context	$S \vdash e_1$: int $S \vdash e_2$: int $S \vdash e_1 / e_2$: dec $S \vdash e_1$: dec $S \vdash e_2$: dec $S \vdash e_1 / e_2$: dec	

	$S \vdash e_1 : int$ $S \vdash e_2 : dec$ $S \vdash e_1 / e_2 : dec$

F.R.N.D. types are divided into two main categories: ***Value types*** and ***Reference types***.

3.2. Value types (differing from Java and Ruby)

Examples

- `bool`
  - Can either return true or false.
  - ex) bool isReady := true please

- `dec`
  - Used to represent floating point numbers.
  - The compiler will determine whether float, double, or real is appropriate for the specific situation
  - ex) dec := 123.456 please

3.3. Reference types (differing from Java and Ruby)

Examples

- `Str`
  - An object that represents a sequence of chars
  - ex) Str greeting := “Hello!” please

- `(Value Type)[]`
  - Array: A sequence of same typed variables with a set size.
  - ex) int[] numList[] := [0, 1] please whisper* size will be 2 *
  - ex) int[5] numList[] please

4. Example Programs

Program	Example

Caesar Cipher encrypt	play Str encrypt (Str myString, int shiftAmount)
	{
	int i please
	whisper* .toBytes is Rubys .bytes which returns an array of ascii codes *
	int[] charNums[] := myStr.toBytes please
	Str newString please
	loop for (i := 0 | i < charNums.length | i++)
	{
	if (charNums[i] == 32)
	{
	newString := newString _ " " please
	}
	else
	{
	newString := newString _ ((((charNums[i] – 65 + shiftAmount)% 26) + 65).toChar) please
	}
	}
	borrow newString please
	}

Caesar Cipher decrypt	play Str decrypt (Str myString, int shiftAmount)
	{
	Str newString please
	newString := encrypt(myString, shiftAmount * ~1) please
	borrow newString please
	}

Factorial	play int factorial (int num)
	{
	if (num == 0)
	{
	borrow 1 please
	}
	else
	{
	borrow num * factorial(num - 1) please
	}
	}

Insertion Sort

```
play insertionSort(int[] array[])
{
       int n := array.length please
       loop for (int i := 1 | i < n | i++)
       {
              int key := array[i] please
              int j := i - 1 please
              loop while (j >= 0 AND array[j] < key)
              {
                     array[j+1] := array[j] please
                     j := j - 1 please
              }
              array[j+1] := key please
       }
}
```

QuickSort

```
friend QuickSort
{
       play swap(int[] array[], int a, int b)
       {
              int temp := array[a] please
              array[a] := array[b] please
              array[b] := temp please
       }

       play int partition(int[] array[], int min, int max)
       {
              int pivot := array[max] please

              int i := (min - 1) please

              loop for (int j := min | j <= max - 1 | j++)
              {
                     if (array[j] > pivot)
                     {
                            i++ please
                            swap(array, i, j) please
                     }
              }

              swap(array, i + 1, max) please
              borrow i + 1 please
       }
```

	play quickSort(int[] array[], int min, int max)
	{
	if (min < max)
	{
	int partLocation := partition(array, min, max) please
	
	quickSort(array, min, partLocation - 1) please
	quickSort(array, partLocation + 1, max) please
	}
	}
	}
	Thankyou

Linear Search	play linearSearch(Str[] array[], Str target)
	{
	int i := 0 please
	bool found := false please
	int result := -1 please
	
	loop while (i < array.length AND result \= 0)
	{
	if(array[i] str= target)
	{
	say. "The item, ${target}, was found at index ${i}" please
	found = true please
	}
	i++ please
	}
	
	if (found == false)
	{
	say. "The item, " _ target _ ", was not found" please
	}
	}