Friendly: A Beginner's Language

Ryan Anderson (ra2929)

Ahmed Alzubairi (afa2135)

Anne-Laure Razat (ar3607)

Michel Vazirani (mvv2114)

Professor Gu

COMS 4115 Programming Languages and Translators (Spring 2020)

May 14th, 2020

Table of Contents

1. Introduction

Friendly is a programming language that is user-friendly for children or first-time coders. We found that most current coding languages designed for kids, such as Scratch, are too simplistic or game-like, making them detached from real coding languages they might go on to learn. On the other hand, industry standard languages, like C or Java, often have jargon that might intimidate beginner programmers. We wanted to create an approachable language that will provide users with a simple but strong introduction to coding syntax and way of thought.

While the concrete skills of how to code are often taught, a big hurdle many struggle to overcome is perfecting their ability to write code with proper syntax. By making the language more closely resemble English, we hope to bridge this gap faced by users learning to code for the first time. We aim to help users think in a programming-oriented way that will prepare them for later languages, such as Java and Python, while still remaining an approachable first step to the coding world. We've defined enough functionality for a user to accomplish basic programming projects that will allow them to start their programming journey.

The syntax of *Friendly* moves away from abstract keywords and instead employs self-descriptive keywords. For example, instead of "print", we've defined "showWords"; to declare variables, we use the keywords "make_a", so it's clear variable creation is occurring. The common terminology breaks down concepts that are otherwise difficult to grasp for beginners in programming.

2. Language Tutorial

2.1 Setting Up Your Environment:

First, download and set up the latest version of LLVM, found at llvm.org, according to the instructions on their site. After downloading LLVM, download the *Friendly* project folder. Enter this folder in your terminal. You're ready to start writing and running your *Friendly* code!

2.2 Writing a Program

Create a new file that ends in the extension ".fr". Here is an example helloworld.fr program:

```
--------------

//helloworld.fr

1   make_a function called main outputting number using () does {

2     make_a sentence called a.

3     make a be "Hello World".

4     do showWords(a).

5     return 0.

6   }

--------------
```

2.3 Compiling

Make sure you're in the Friendly folder before you compile. In this example compilation, the file with our code is named "helloworld.fr". Run these commands in your terminal:

```
--------------

$     ocamlbuild -pkgs llvm friendly.native

$     ./friendly.native -l helloworld.fr > helloworld.out

$     lli helloworld.out

--------------
```

2.4 Class and Function Declaration and Creation

When creating a function, we begin with its definition, here is an example:

```
make_a function called f outputting number using (number called
a) does{ }
```

make_a declares that we are creating something; following it with the keyword **function** indicates we are making a function. Then we name our function, using the keyword **called** followed by the intended name; in this case, our function is named **f**. Then, we declare what we want to return in our function, saying **outputting** and following it with the type we want to return, in this case, we want to return a **number**. Then, we use the keyword **using** to indicate the upcoming (optional) arguments within the parenthesis; while having an argument is optional, including **using** and the following **()** is necessary, so even if no arguments are being passed, **using ()** will be in the definition. Here, we see our example argument is **number called a**. Finally, before we have the content of the function within the curly brackets, we have the keyword **does**, indicating that the following code is what this function **does**. Within the curly brackets of the function, declare and manipulate all of the variables, structs, and functions you wish, as shown below.

2.5 Variable Declaration and Creation

In *Friendly*, variables are declared using the key word **make_a.** For example if we wanted to make a new **number** variable and name it **x**, we would declare that variable as follows:

```
make_a number called x.
```

The type **number** is bound to **x** using the key work **called**. The line ends with a " **.** " to mark the end of the declaration. The **.** is similar to the end-of-the-line semicolon in other programming languages.

Now that we've created x, we want to assign a value to it:

```
make x be 5.
```

This assigns the value **5** to the variable **x**. We can similarly do this with strings, known as sentence in *Friendly*:

```
make_a sentence called a.

make a be "Hello World".
```

With `number` and `decimal` variables, simple math can be done, including addition, subtraction, multiplication, and division. Numbers and all binary operators need to be separated by spacing in order to show that there is an expression taking place or a value assignment such as a - to represent a negative number value. Say we have these variables

```
make_a number called a.

make_a number called b.

make_a decimal called c.

make_a number called d.

make a be 10.

make b be 2.

make c be -2.5.
```

We could perform these mathematical actions:

Addition	Subtraction	Multiplication	Division

`make d be (a + b)`
`do showNumber(d)`
`----------`
`$ 12`
	`make d be (a - c)`
	`do showNumber(d)`
	`----------`
	`$ 12.5`
		`make d be (a * b)`
		`do showNumber(d)`
		`----------`
		`$ 20`
			`make d be (a / c)`
			`do showNumber(d)`
			`----------`
			`$ -4`

We can also perform comparisons and other boolean operations on these same variables:

< or >	<= or >=

`do showTruth(a < b).`
`-----`
`$ 0  // False, 10 is not less than 2`
	`do showTruth(b <= a)`
	`-----`
	`$ 1   //True, 2 is <= 10`

==	!=
`do showTruth(a == a).`	`do showTruth (a != a).`

----- $ 1 //True, 10 == 10	----- $ 0 //False, a==a

We also have a boolean type, `truth`, that these operations can similarly be performed upon.

With `sentences` (strings), `sentence` concatenation and comparison (using the above operands: `<, >, <=, >=, ==, !=`) can be done.

Concatenation	Comparison
make_a sentence called a. make a be (“hello” + “world”). do showWords(a). -------------- $ helloworld	make_a sentence called a. make_a sentence called b. make a be (“hello”) make b be (“world”) do showTruth(a < b) -------------- $ 1 //True, ”hello” comes before // “world” lexicographically

2.6 Built In Functions

As seen above, the way we print out variables is by using several built in functions: `showWords(), showNumber(), showTruth(), showDecimal()`

Function:	`showWords()`	`showNumber()`	`showDecimal()`	`showTruth()`
Use for variable:	`sentence`	`number`	`decimal`	`TrueFalse`
Example:	make_a sentence called a. make a be “hello”.	make_a number called a. make a be 10. showNumber(a).	make_a decimal called a. make a be 10.0. showDecimal(a).	make_a TrueFalse called a. make a be 1. showTruth(a).

	showWords(a). ------------ $ hello	------------ $ 10	-------- $ 10.0	---------- $ 1

2.7 Structs

Friendly has a struct feature known as `chunk`.
You can create a chunk using **`make_a chunk`** :

```
make_a chunk called charizard{
  make_a decimal called health.
}
```

You can then create `functions`, as we've learned about in section 2.4, to manipulate, initialize, and access your `chunk`. Specifically, using the `name-of-the-chunk : variable-in-the-chunk`, one is able to access variables from within the chunk. For example, accessing a variable called `health` from chunk `c` would be done by doing "`c:health`":

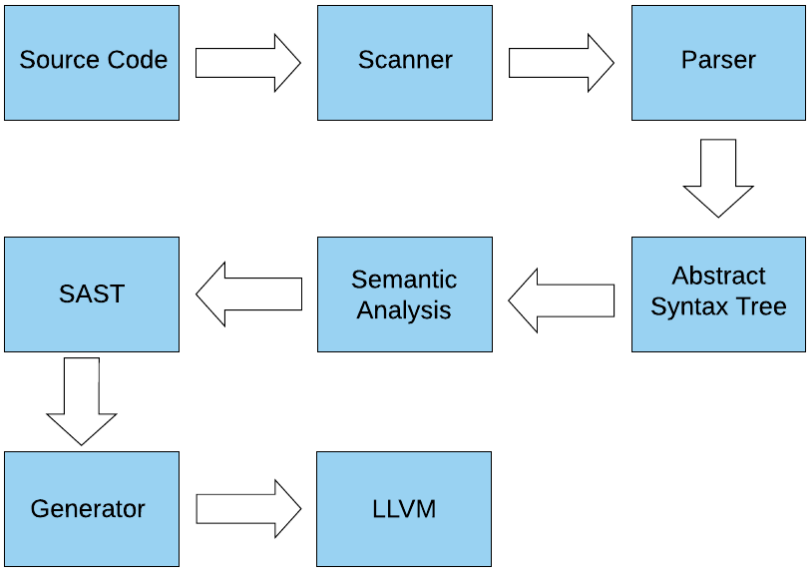
```
make_a function called init_charizard outputting chunk charizard
using (decimal called h) does {
  make_a chunk charizard called c.
  make c:health be h.
  return c.
}

make_a function called main outputting number using () does {

  make_a chunk charizard called pete.
  make pete be do init_charizard(10.0).
  do showWords("Charizard health is").
  do showDecimal(pete:health).
  return 0.
}

-------
$ 10.0 //Pete's health
```

3. Architectural Design

3.1 Overview of Architecture

The compiler of *Friendly* starts with the first line of code and is passed through to the scanner; once the code reaches the scanner it is then tokenized and passed through a parser. In the parser the Abstract Syntax Tree (AST) is built and then sent to the semantic checker to check for issues in the semantics of our program and build the Semantically-checked Abstract Syntax Tree (SAST). After the code is semantically checked, the irgen phase traverses the SAST to generate LLVM code in the .out executable.

Figure 3.1: *Friendly* Architecture



3.2 Architecture Roles

Hello World	Michel/Ahmed/Anne-Laure/Ryan
Variable Declaration	Michel/Ahmed/Anne-Laure/Ryan
Functions	Michel/Ahmed/Anne-Laure/Ryan
Simple Math	Michel/Ahmed/Anne-Laure/Ryan
Strings	Michel/Ahmed/Anne-Laure/Ryan
String concatenation	Michel/Ahmed/Anne-Laure/Ryan
Built-in functions (ie showWords())	Michel/Ahmed/Anne-Laure/Ryan
Chunks	Michel/Ahmed/Anne-Laure/Ryan

3.3 Language Architecture

Our architecture for *Friendly* consists of a scanner and parser that build up our abstract syntax tree for each file.

3.4 Scanner, Parser and Abstract Syntax Tree (AST)

The Scanner works with the parser to tokenize our syntax for the Abstract Syntax Tree. The parser is the component that reads through the source code of our program breaking the program into chunks that are then sent to the Scanner and AST. The AST is responsible for ensuring name binding and specifying a return type.

3.5 Semantic Analysis

Semantic Analysis checks the expressions, variables and over all semantics of the *Friendly* syntax by ensuring the types are in the Abstract Syntax Tree. In this section of the program, there are both the built in native types and the *Friendly* defined types.

3.6 Semantically checked AST (SAST)

After the code is semantically checked, the irgen phase traverses the SAST to generate LLVM code in the .out executable.

3.7 Code Generation

Code Generation translates the semantically checked Abstract Syntax Tree to produce LLVM IR code.

3.8 LLVM files and .out files

The LLVM IR files are used to produce .out files. The .out file type is what is then executed using lli.

4. Test Plan

Test Plan: Show two or three representative source language programs along with the target language program generated for each, What kind of automation was used in testing [within 1 page].

4.1 Test 1

Test 1 Code: `pokebattle.fr`

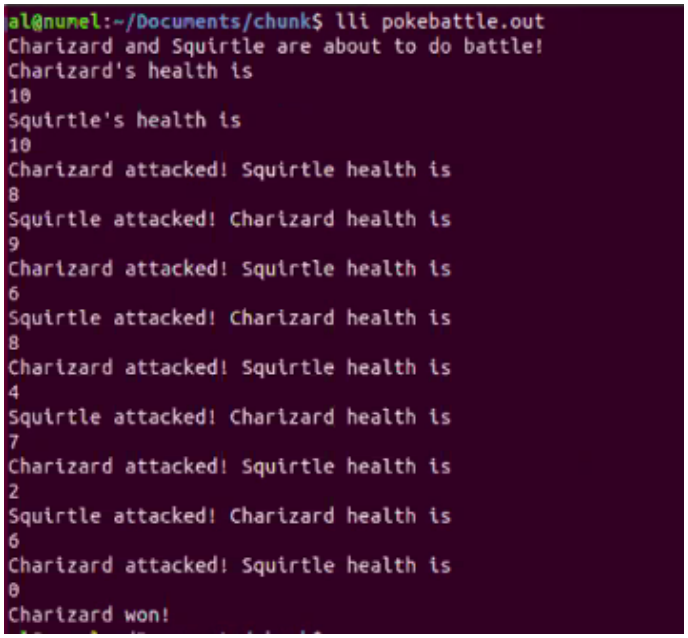
```
make_a chunk called pokemon{
  make_a number called health.
  make_a number called damage.
  make_a sentence called name.

}

make_a function called init_pokemon outputting chunk pokemon using (number called h,
number called d, sentence called n) does {
  make_a chunk pokemon called p.
  make p:health be h.
  make p:damage be d.
  make p:name be n.
  return p.

}

make_a function called attack outputting chunk pokemon using (chunk pokemon called
p1, chunk pokemon called p2) does {
  make p2:health be p2:health - p1:damage.
  do showWords(p1:name + " attacked! " + p2:name + " health is").
  do showNumber(p2:health).
  return p2.
}

make_a function called battle outputting chunk pokemon using (chunk pokemon called
p1, chunk pokemon called p2) does{

  make_a chunk pokemon called winner.

  do showWords(p1:name + " and " + p2:name + " are about to do battle!").
  do showWords(p1:name + "'s health is").
  do showNumber(p1:health).
  do showWords(p2:name + "'s health is").
  do showNumber(p2:health).
```

```
  while (p1:health > 0 and p2:health > 0){
    make p2 be do attack(p1, p2).
    if(p2:health > 0){
      make p1 be do attack(p2, p1).
    } else {}

    if(p2:health <= 0){
      do showWords(p1:name + " won!").
      make winner be p1.
    } else {}



    if(p1:health <= 0){
      do showWords(p2:name + " won!").
      make winner be p2.
    } else {}
  }
  return winner.
}

make_a function called main outputting number using () does {

  make_a chunk pokemon called charizard.
  make_a chunk pokemon called squirtle.

  make charizard be do init_pokemon(10, 2, "Charizard").
  make squirtle be do init_pokemon(10, 1, "Squirtle").

  do battle(charizard, squirtle).

  return 0.
}
```

Test 1 Compilation:

```
$     ocamlbuild -pkgs llvm friendly.native

$     ./friendly.native -l pokebattle.fr > pokebattle.out

$     lli pokebattle.out
```

Test 1 Output:



Test 1 Explanation:

In order to test Friendly and the characteristics of the language we made a Pokemon battle program that tests the functionality of the language. In the Pokemon.fr file above, a chunk is made to represent each of the Pokemon in the battle. Each chunk is given health and damage attributes that we are then able to manipulate using the `battle` and `attack` function.

4.2 Test 2

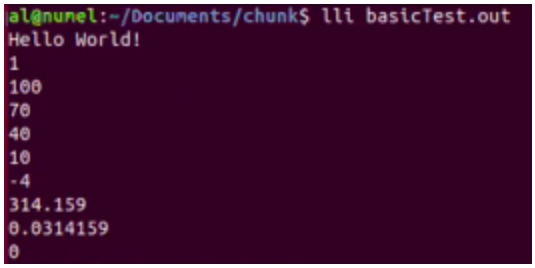
Test 2 Code: `basicTest.fr`

```
make_a function called main outputting number using () does {


  make_a sentence called s.
  make_a number called n.
  make_a decimal called d.
  make_a truth called t.

  do showWords("Hello World!").

  make s be "Hello World!".
  do showTruth("Hello World!" == s).

  make n be 100.

  while(n > 0) {
    do showNumber(n).
    make n be n - 30.
  }

  do showNumber( -4 ).

  make d be 3.14159.
  do showDecimal(d * 100.0).
  do showDecimal(d / 100.0).

  make t be n > 0.
  do showTruth(t).


  return 0.


}
```

Test 2 Compilation:

```
$     ocamlbuild -pkgs llvm friendly.native

$     ./friendly.native -l basicTest.fr > basicTest.out

$     lli basicTest.out
```

Test 2 Output:



Test 2 Explanation:

This example code shows the ability of *Friendly* to create a variety of variables (`sentences, numbers, decimals, TrueFalse`), manipulate, and output them.

5. Summary

5.1 Official Roles

Anne-Laure Razat (Manager), Michel Vazirani (System Architect), Ryan Anderson (Tester), Ahmed Alzubairi (Language Guru)

5.2 Who did what

Overall, we worked together on creating *Friendly*, using the screen sharing feature on Zoom to edit and test our code together. However, individually, Ahmed and Michel were able to get certain features to work on their own time, such as string manipulation/comparison and chunks.

5.3 Reflections

Ahmed's Reflection: I remember the first two weeks of this course when Ocaml and lambda calculus was introduced. I thought that I would need to drop the course because no matter how hard I tried, I couldn't understand it. Before, I thought functional programming just meant you use recursion, I didn't even know what imperative programming was nor the fact that it was what I have been doing all these years. Now, with the end of the class, thinking in a "functional programming" mindset has been second nature. I went from dreading the homework assignments to loving them. Being one of the main programmers of the project, I also learned a great deal of how a programming language works and how a compiler works. I thought that this project was going to be unimaginably hard, but with the help of my team, we were able to complete it in piece by piece till we completed the project. I was amazed how we went from saying how the heck would we even start this thing to it becoming easier and easier to implement features into the project. I also owe a great deal of my understanding to the awesome TA's. Each one helped me a great deal learn the contents I struggled with. In conclusion, I am glad I took the course when I did. The online format changed everything, but I had an awesome team and I learned how to build a programming language!

Anne-Laure's Reflection: Overall, this project has made me realize everything it takes to truly create and implement a programming language. While I felt I understood the abstract building blocks of programming languages that we discussed in class, this hands-on experience helped me to truly understand the extent of what it takes to make a PL. While initially I was worried about the continuation of this project when school moved online, I think we really found our groove as a team because of it. Group coding sessions on zoom allowed for efficient development of our code as we were able to each bring our own perspective to the edits we were making.

Michel's Reflection: I appreciated the project for the opportunity to implement a programming language from top to bottom. It helped me understand each module in depth and how the

different modules feed into each other. I was excited to see how implementing new features became easier and easier as we got more comfortable with manipulating the modules and understood the gist of how features are implemented. For instance, when we had to implement “chunks” for our language, it was much easier having already done a lot of manipulation for syntax and having studied the implementation of functions. In addition, I appreciated the opportunity for a group coding project, as coding projects have always been individual in other classes. It was efficient to be able to assign people different components to work on, and it was very productive whenever we all came together to look at the code, as we were all able to simultaneously give ideas and input on implementation and debugging.

Ryan’s Reflection: The project has made me more mindful of programming and understanding the process behind the code. Before this class I had an idea of how a compiler worked but now I have a better understanding of how source code is read by the computer. I think the most important thing with the project is time management and good communication with your team. For our group we were unable to meet in person and were able to adapt to the circumstances and organize zoom calls to plan and produce the language.

5.4 Advice For Future Teams:

As a team, something we found incredibly helpful was coding our language together, using screen sharing on zoom. This made for efficient code writing, checking, and problem solving, as everyone was able to chime in and contribute, preventing one person from coding a lot on their own and getting stuck on a problem. It also kept all group members up to date with the functionality of the code, eliminating the need for us to catch each other up on new functionalities and code capabilities.