Bruciato

Sorin Macaluso

May 6, 2024

For context Bruciato is the Italian word for burnt.

It is also sometimes used as a adjective.

Specifically describing a person as having a burnt mind.

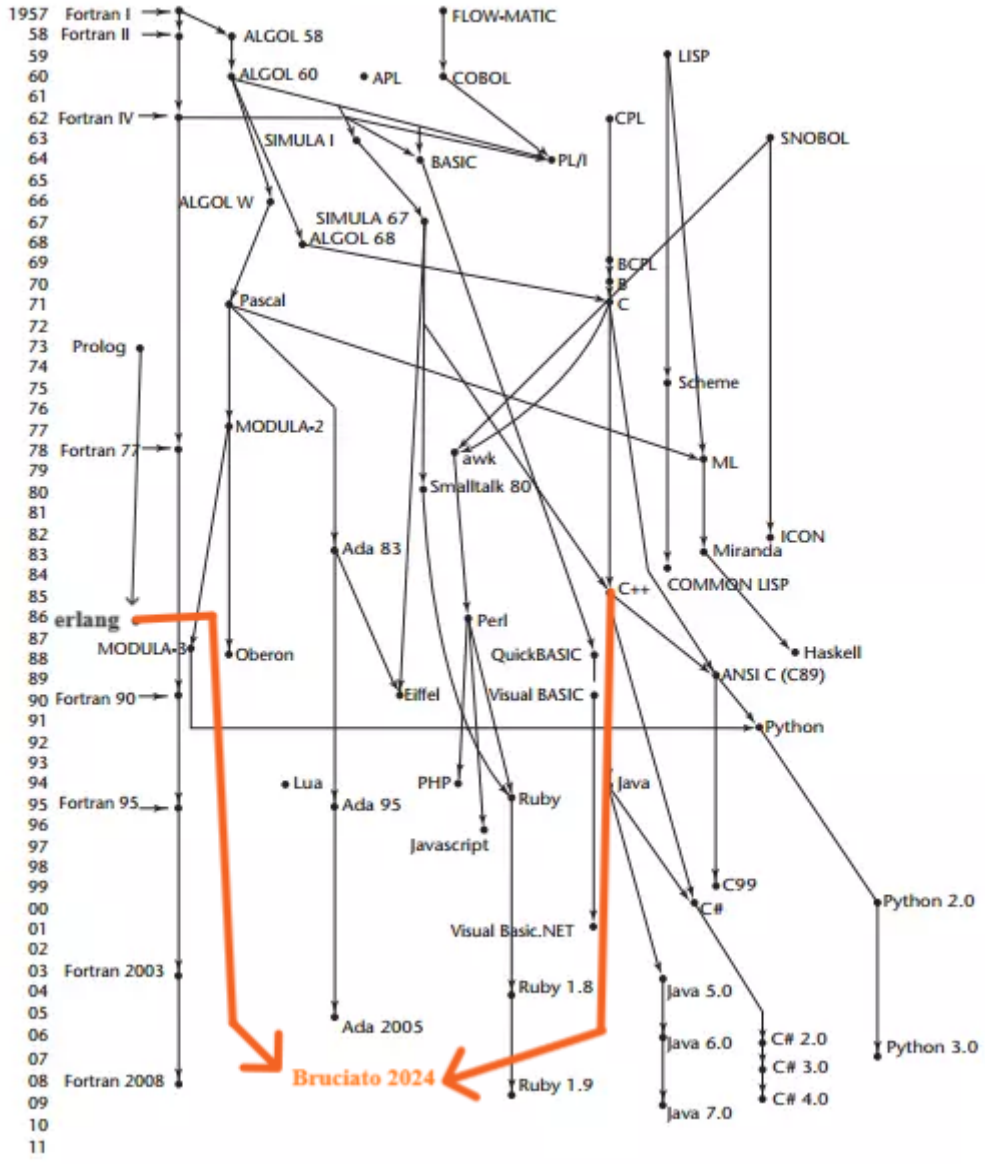
The goal of this programming language is to make you Bruciato.

Contents

1 Introduction

The first thing that you will need to do for any program in Burciato is start the whole program off with the startProg command.The program will be ended with the endProg command. The language will give the user direct access to the pointers like C++ as well as have the Erlang. Bruciato is a strongly typed, object orientated language that has structurally equivalent types and has dynamic scope.

1.1 Genealogy



1.2 Hello World

This is what hello world would look like in Bruciato

```
* Hello world in Burciato *

startProg<>
    def var charR output[12] = ``Hello World!'';
    termOut<ouput>;
endProg<>
```

1.3 Program Structure

1.3.1 Organization

1. First every program will need to start with a the startProg function. This is unlike your normal start to a program. The startProg token is a function call to the startProg function with in the language.

2. startProg is a function that will shuffle whether or not true equals true or true equals false, same applies to the value of false.

3. The function works like this.

```
fun startProg<> :)
    def var num check = randomNum<>;
    * Will check if the randomNum is even or odd *
    may <check % 2 == 0> :)
        * if even then true = true and false = false*
        true = true;
        false = false;
    :) be :)
        * if odd then true = false and false = true *
        true = false;
        false = true;
    :)
:)
```

```
fun endProg<> :)
    true = true;
    false = false;
:)
```

   The user has a fifty percent chance to have true actually equal true or be equal to false, as well as false equalling false or false equalling true.

4. This only works since the program and the user has access to the pointers that are made to store data. This means that the user has direct access to the static pointer for the true and false value.

5. By default true will equal true. After the function is called the value of true and false may change.

6. If the boolean operations are switched in values then this will cause many problems in the functionality of the code as shown in the sample program.

7. endProg will reset the values of true and false back to there normal state.

1.3.2 Sample Program

```
1   * simple program to print out 0 to 9 *
2
3   startProg<>
4       termOut<true>
5
6       def var num ex = 0;
7
8       for <def var num i = 0, i < 10, i++> :)
9           termOut<ex>;
10          ex++;
11      :)
12  endProg<>
```

For this program if true does equal true. Then the for loop will check as any normal programming language would since i < 10 will always evaluate to true unless i is a number greater than or equal to 10. But if true equals false then any statement that was now previously true will become false. This will change the for loop to break out of the loop instantly since i < 10 will be false and then nothing will be printed out.

1.4 Types and Variables

There will be two types in this language. There will be the value type, this type will hold the direct data of what ever the variable may be. Then there will be the pointer type, this type will hold pointers to anything. This can range from pointers to where a function is, values being passed to functions, or even the location of objects from a class. For the variables the language are all single assignment like Erlang. There are some slight changes though. For loops are allowed and operate the same way as any other language.

1.5 Visibility

Just to make things as annoying as possible the language will be of Dynamic scope. This means that the time at which a something is added to the run time stack will change how the program will run. Since the language will have Dynamic Scope this means that it will be late binding, at run time, and again all about the time.

1.6 Differing from C++ and Erlang

Statements	Examples
if & else statements	If will be replaced with may and the else will be replaced with be. If you have a other decision to make instead of else if, it will be called if else. If will be called may, if else will be called may be, and else will be called be.
functions	functions will start with the fun key word and then be given a name then <> will be used after the name to store the arguments of the function. A function can return anything or nothing at all.
for & while loops	For loops will look syntactically the same as any c++ loop just that instead of the parenthesis it will have <> surrounding the arguments of the loop.
arguments	As stated in functions and for & while loops arguments will be encapsulated around the <> symbols. Like so < num age, num time >
printout	This will be handled by a very the termOut<> function built into the program.
Array Declaration	Same as declaring a variable must have def var and then the type then the name.
Variable Deceleration	When declaring a new variable in scope you must say the full incantation "def var" and then the type. The num type is for integers, a-z is for a single char, charR stands for char repeat (or a string), and 01 represents the boolean type.
Types	New name for the different types are num for integers. As well as a-z for single characters, the a-z represents the possible values of a single char. Then there is charR or char repeat, since a string is a char repeated. This also gives a hint to the typing, a-z are the same as charR since they are both of char type in the code. Finally 01 represents boolean values since a boolean can be true (1) or false (0).

2 Lexical Structure

2.1 Programs

This language is purely made in order to be as annoying as possible. A program can contain many different files within a main file. The syntax will be similar to how C++ handles multiple files.

How the files are compiled is with these three steps.

1. First is that we need to have lexical analysis, this will make the typed up code into list of tokens to be processed further. This will check the validity of the words typed by the programmer.
2. Second is that we need to parse through the tokens. This will check the grammar and make sure that what was typed by the programmer is valid sentences in the language.
3. Third is then building a Abstract Syntax Tree and a symbol table. These will be used in order to make the machine code so that any operating system is able to run the code. At this point the output will not look like the original program anymore.
4. Fourth the generation of the machine code based on the Abstract Syntax Tree and scope table.

Disclaimer: I will only be going over the difference that Bruciato has compared to C++ and Erlang for parse and lexical analysis. If you want a more in depth look at the steps mentioned above. There is a class that meets on Mondays from 8am to 10:45am in Music room 3202.

2.2 Grammars

2.2.1 Lexical grammar (tokens) where different from C++ and Erlang

⟨*type*⟩ ⇒ ⟨*charR*⟩
| ⟨*a-z*⟩
| ⟨*01*⟩
| ⟨*num*⟩

⟨*charR*⟩ ⇒ A|B|C|D| ..... |X|Y|Z|
| a|b|c|d| ..... |x|y|z|
| ϵ

⟨*a-z*⟩ ⇒ A|B|C|D| ..... |X|Y|Z|
| a|b|c|d| ..... |x|y|z|

⟨*01*⟩ ⇒ true|false

⟨*num*⟩ ⇒ 1|2|3|4|6|7|8|9|0

⟨*closeBracket*⟩ ⇒ >

⟨*openBracket*⟩ ⇒ <

⟨*Begin Prog*⟩ ⇒ startProg()

⟨*End Prog*⟩ ⇒ endProg()

⟨*Begin Scope*⟩ ⇒ :)

⟨*End Scope*⟩ ⇒ :)

⟨*Single Line Comments*⟩ ⇒ *

2.2.2 Syntactic "parse") grammar where different from C++ and Erlang

⟨*var-decl*⟩ ⇒ def var ⟨*type*⟩ name

⟨*if*⟩ ⇒ if ⟨*openBracket*⟩ ⟨*bool-statement*⟩ ⟨*closeBracket*⟩

⟨*for*⟩ ⇒ for ⟨*openBracket*⟩ ⟨*statement;*⟩ ⟨*statement;*⟩ ⟨*end-statement*⟩ ⟨*closeBracket*⟩

⟨*while*⟩ ⇒ while ⟨*openBracket*⟩ ⟨*bool-statement*⟩ ⟨*closeBracket*⟩

⟨*function*⟩ ⇒ fun ⟨*name*⟩ ⟨*openBracket*⟩ ⟨*argument*⟩ ⟨*closeBracket*⟩
| ⟨*argument*⟩
| ϵ

⟨*Begin Prog*⟩ ⇒ startProg()

⟨*End Prog*⟩ ⇒ endProg()

⟨*Begin Scope*⟩ ⇒ :)

⟨*End Scope*⟩ ⇒ :)

2.3 Lexical Analysis

2.3.1 Comments

Comments in Bruciato are encapsulated around $*$ symbol. For example,

```
1     * This is a comment they are only single line *
```

They are only single line comments, just to really make it as annoying as humanly possible.

2.4 Tokens

There are many different kinds of tokens that are used in the Brucatio language.

1. Identifier Tokens: These tokens are used for naming things in the programming language. For example, naming variables and classes.
2. Keyword Tokens: These tokens are the reserved words in the programming language.
3. Integer-Literal Tokens: This is the token for the integer type.
4. Char Tokens: This is the token for the char type.
5. String Tokens: This is the token for the string type.
6. Boolean Tokens: This is the token for the boolean operator.

2.4.1 Keywords different from C++ or Erlang

New Keyword	Old Keyword
num	int
01	boolean
a-z	char
charR	String
:)	{
termOut	System.out.print/cout
may	if
may be	else if
be	else
*	//

3 Type System

3.1 Type Rules

The type in Bruciato is a structurally equivalent language. This is taken from C++ which is also a structurally equivalent language.This means that even though the type that was assigned to the variable may be different, they can be still be the same as long as they are structurally the same.

This is shown in the a-z and charR type. A char is a part of a string and strings are made up of chars. So for that reason a-z and charR are the same thing since they are structural the same. Strings are made from chars to structurally the string type is just the char type repeated. With that in Bruciato you are allowed to compare, concatenate, and do any kind of comparison with a string and a char. This is also shown in arrays, since a-z and charR are the same type that means you can have a array of charR's with a-z in them or a array of a-z with charR's in them.

There are also other types like num and 01. These are basic types they represent int and boolean.

3.2 Value Types (different from C++ and Erlang)

Num ⇒ is a type that can be either int, float, or double.

01 ⇒ is a type that can be set to true or false. For the matching code it can either be 0 or 1

a-z ⇒ is a type that can be set to any single char

charR ⇒ is a type that can be set to multiple different char's

3.3 Pointer Types (differing from C++ and Erlang)

Pointers ⇒ These are very important types that can also be very dangerous. "With great power comes great responsibility". They are the direct access to the data that is stored in the machine code. Everything will work with pointer. Strings are a collection of chars with pointers to each char. Arrays are a pointer to the head of the list of items. For loops are pointers back to the head of the for loop. May, may be, and be are pointers to the different jumps that may or may not need to be taken based on the comparison. They are very important but can also cause lots of problems for security since there is a way to get direct access to the data in memory.

4 Example Programs

4.1 Caesar Cipher Encrypt

```
1   * Encrypt *
2
3   *now that is if true equals true*
4   *to effectively use this programming language you have to check*
5
6   startProg<>
7
8       * if true equals true *
9
10      may <true = true> :)
11          def var charR[5] word = ``sorin'';
12          def var num encrypt_char = 0;
13
14          for <def var num i = 0, i < word.length<>, i++> :)
15              encrypt_char = word[i].ascii<> - 65 + move % 26;
16
17              may <encrypt_char < 0> :)
18                  encrypt_char = encrypt_char + 26;
19              :) be :)
20                  encrypt_char;
21              :)
22
23              word[i] = encrypt_char.back_char<>;
24          :)
25
26      * if true equals false *
27
28      :) be :)
29          def var charR[5] word = ``sorin'';
30          def var num encrypt_char = 0;
31
32          for <def var num i = 0, i > word.length<>, i++> :)
33              encrypt_char = word[i].ascii<> - 65 + move % 26;
34
35              may <encrypt_char > 0> :)
36                  encrypt_char = encrypt_char + 26;
37              :) be :)
38                  encrypt_char;
39              :)
40
41              word[i] = encrypt_char.back_char<>;
42          :)
43      :)
44  endProg<>
```

For this program the "basic" Caesar cipher encrypt now become something that you have to make two versions of. One version if the startProg<> function makes true equal to true or if true is equal to false. Pretty much you have to flip the symbols around for if true equals false. Since true equals false now you need to check for the the false statement until it becomes true. Since true now equals false the false statement will now become true until the condition is meet. The the check will become false and jump out of the loop. For example, i > word.length<> will be false until i is greater than word.length<> (or 5). If true equals false then i > word.length<> will be true until i > word.length<> in which it will be false.

Big Error: greater than works differently than less than 5 is less than 5 but 5 is not greater than 5. Might have a error where things happen one less time.

4.2 Caesar Cipher Decrypt

```
1   * Decrypt *
2
3   *now that is if true equals true*
4   *to effectively use this programming language you have to check*
5
6   startProg<>
7
8       * if true equals true *
9
10      may <true = true> :)
11          def var charR[5] word = ''sorin'';
12          def var num encrypt_char = 0;
13
14          for <def var num i = 0, i < word.length<>, i++> :)
15              encrypt_char = word[i].ascii<> - 65 - move % 26;
16
17              may <encrypt_char < 0> :)
18                  encrypt_char = encrypt_char + 26;
19              :) be :)
20                  encrypt_char;
21              :)
22
23              word[i] = encrypt_char.back_char<>;
24          :)
25
26      * if true equals false *
27
28      :) be :)
29          def var charR[5] word = ''sorin'';
30          def var num encrypt_char = 0;
31
32          for <def var num i = 0, i > word.length<>, i++> :)
33              encrypt_char = word[i].ascii<> - 65 - move % 26;
34
35              may <encrypt_char > 0> :)
36                  encrypt_char = encrypt_char + 26;
37              :) be :)
38                  encrypt_char;
39              :)
40
41              word[i] = encrypt_char.back_char<>;
```

```
            :)
        :)
    endProg<>
```

4.3 Factorial

```
    *global*

    fun factorial <num fac> :)
         may <true = true> :)
            may < fac == 0 || fac < 0 > :)
                return 0;
            :) be :)
                return <fac * factorial<fac-1>;
            :)
        :) be :)
            may < fac != 0 || fac > 0 > :)
                return 0;
            :) be :)
                return <fac * factorial<fac-1>;
            :)
        :)
    :)

    startProg<>
        may <true = true> :)
            def var num fac = 5;

            factorial <fac>;
        :) be :)
            def var num fac = 5;

            factorial <fac>;
        :)
    endProg<>
```

4.4 Selection Sort

```
1   *global*
2
3   def var charR[5] test = ``hello'';
4
5   startProg<>
6       may <true = true> :)
7           for <def var num i = 0, i < test.lenght<>, i++> :)
8
9               def var char place = test[i]
10
11              for <def var num k = i+1, k < test.length<>, k++> :)
12
13                  may <test[k] < place> :)
14
15                      test[i] = test[k]
16
17                  :)
18
19              :)
20          :)
21      :) be :)
22          for <def var num i = 0, i > test.lenght<>, i++> :)
23
24              def var char place = test[i]
25
26              for <def var num k = i+1, k > test.length<>, k++> :)
27
28                  may <test[k] > place> :)
29
30                      test[i] = test[k]
31
32                  :)
33
34              :)
35          :)
36      :)
37  endProg<>
```

4.5 Bogo Sort

```
1   import selection_sort.bru
2   *global*
3
4   def var charR[5] test = ``hello'';
5
6   startProg<>
7       may <true = true> :)
8           for<def var num i = 0, i < test.length, i++> :)
9               def var num random = randomNum<> % 5;
10
11              def var char temp = test[i];
12              test[random] = temp;
13          :)
14
15          def var charR selection = selection_sort<test>;
16
17          may <selection == test> :)
18              termOut(``Solved'');
19          :) be :)
20              termOut(``Not Solved'');
21      :) be :)
22          for<def var num i = 0, i > test.length, i++> :)
23              def var num random = randomNum<> % 5;
24
25              def var char temp = test[i];
26              test[random] = temp;
27          :)
28
29          def var charR selection = selection_sort<test>;
30
31          may <selection != test> :)
32              termOut(``Solved'');
33          :) be :)
34              termOut(``Not Solved'');
35      :)
36  endProg<>
```

4.6 Bozo Sort

```
import selection_sort.bru
*global*

def var charR[5] test = ``hello'';

startProg<>
    may <true = true> :)
        for<def var num i = 0, i < test.length, i++> :)
            def var num random1 = randomNum<> % 5;
            def var num random2 = randomNum<> % 5;

            def var char temp = test[random1];
            test[random1] = test[random2];
            test[random2] = temp;
        :)

        def var charR selection = selection_sort<test>;

        may <selection == test> :)
            termOut(``Solved'');
        :) be :)
            termOut(``Not Solved'');
    :) be :)
        for<def var num i = 0, i > test.length, i++> :)
            def var num random1 = randomNum<> % 5;
            def var num random2 = randomNum<> % 5;

            def var char temp = test[random1];
            test[random1] = test[random2];
            test[random2] = temp;
        :)

        def var charR selection = selection_sort<test>;

        may <selection != test> :)
            termOut(``Solved'');
        :) be :)
            termOut(``Not Solved'');
    :)
endProg<>
```

5 References

Sort