The Stop Programming Language

Stop, a Simple funcTional Object-oriented Programming language

Jonathan Barrios (jeb2239), Jillian Knoll (jak2246), Lusa Zhan (lz2371), and James Stenger (jms2431)

February 9th, 2016

Abstract:

Stop is a general purpose programming language, syntactically similar to Scala, that will compile to the LLVM Intermediate Representation (LLVM IR).

Motivation and Language Description:

Functional languages have been steadily growing in popularity as general purpose languages. We propose to create a general purpose functional language with type inference capabilities.

As a general purpose language, Stop will have limitless possibilities for use cases. Users can harness the speed of a relatively expressive language compiled to bytecode, allowing users to combine Stop in a library with the language of their choosing. Stop can be utilized for a wide range of purposes, ranging from high level data computation to application building.

We believe that in order to allow for robust user created libraries, this language should be object-oriented. Every function and variable will be considered intrinsically as an object. This system will allow users of our language to easily create easily reusable inheritance structures to allow libraries to interact with additional languages. Stop will have limited standard libraries to allow for our team to focus on the compiler infrastructure.

One of our main challenges will be compiling a relatively expressive language to low-level bytecode: LLVM IR. The LLVM Core Libraries provide code generation support for a variety of computer architectures including x86, MIPS, and ARM. Compiling to this intermediate representation will prove pedagogically useful as it will require us to work with the LLVM Core Libraries and to gain a thorough understanding of compiler infrastructure.

By undertaking this challenge of compiling an expressive, functional, object-oriented programming language to low-level bytecode, we hope to learn about compiler infrastructure and wide ranging use capabilities of functional programming.

Language Syntax Overview:

The language will resemble Scala in appearance but will have optional type inference capabilities. Stop will have local and explicitly defined global variables for added security. We also would have templates which would basically generating code for each instance of the template.

Reserved words:

def	this is shorthand for declaring functions
final	like const in C
var	mutable variable declared locally
global	global variable declaration
Unit	like void in C
#	comment
method	declares a method
class	class declaration
if(){} else{}	if else expression
if(){}elseif(){} else{}	elseif , still needs to end with an else

Standard Library:

print,io primitive wrappers	prints strings
Array , List	collection objects

Int, Double	also objects, but they will map to their native counterparts, this is what GNU Smalltalk does.

Operators are just methods, for example:

```
1 + 2
#is the same as
1.operator+(2)
#could also get the same answer by doing
2.operator+(1)
```

Arrays for example:

```
final arr = Array<Int>({1,2,3,4,5}) #compiler sees we need an Int Array, so it will just
generate a class
var t = arr[0]
var k = arr.operator[](0)
#two ways of writing the same thing
```

Operators:

- arithmetic: % + - / & | >> << * () [] can be overloaded
- ==
- %
- !
- =
- logic : && ||

Sample Code:

```
def gcd = (var u:Int , var v:Int):Int {

        if(v!=0){
              gcd(v, u%v)
        }
        else{
```

```
                u
         }
#return last statement must else with every if
}




def count  = (var arr : Array<Int>): Unit {

        for(var a = 0 ; a<10 ;a++) {

                        print(arr[a])

        }
}
#the above is translated to
class count = {

        method operator() = (var arr: Array<int>):Unit {
                for(var a = 0 ; a<10 ;a++) {

                        print(arr[a])

           }
        }

}

#now if we want to call count
#we write
count()
#this is just going to be translated to
count.operator()()
#what about private data?
class Complicated = {

        def helper = (var arr : Array<Int>):Unit {
#need to write out function declaration with types

                print(arr[a])
        }

        #so this would be translated into a class with an apply method
        #this is only available in this code block
        #to make it callable by outside code
        method operator() = (var arr: Array<Int>):Unit { helper() }

        method just_call_it = (var arr: Array<Int>):Unit { helper() }

}


final arr_example = Array<Int>(4) #array of four zeros
#call complicated using .operator() short hand
Complicated(arr_example)
#call complicated using long hand
Complicated.operator()(arr_example)
#do the same thing but now via a regular user defined method
```

```
#there is no shorthand way of doing this
Complicated.just_call_it(arr_example)   #all these do the same thing



global a = 0    # a comment, btw this global and we make a new down there
                #all the following code blocks will capture this
def declaring = ():Unit {

        var a = 0   # local a is going to hide global a
        final b = 9   # this is immutable

}

# can't infer function arguments or return

def declaring_type_infer = (var a :Int , var b:Int):Unit {
        var r = 0 # r is inferred as an int

}

#what the int class could look like

class Int  = {
        #int_native is a keyword this logic may need some work
        var value: int_native = 0   #this is native load
        method operator= = (var a:int_native):Int {
                this.value=a

        }


}
```

Inspiration:

- Scala
  - About Scala Type System: http://lampwww.epfl.ch/~odersky/papers/mfcs06.pdf
  - http://lampwww.epfl.ch/~odersky/papers/icfp98.ps.gz
  - http://www.scala-lang.org/
- Go
  - We are the opposite of Go
  - https://golang.org/
- Rust
  - https://www.rust-lang.org/
- Dice
  - this language was from last semester, also compiles to llvm IR
  - http://www.cs.columbia.edu/~sedwards/classes/2015/4115-fall/reports/Dice.pdf