Cappy

Language Design
& Example Programs

Calista Phippen

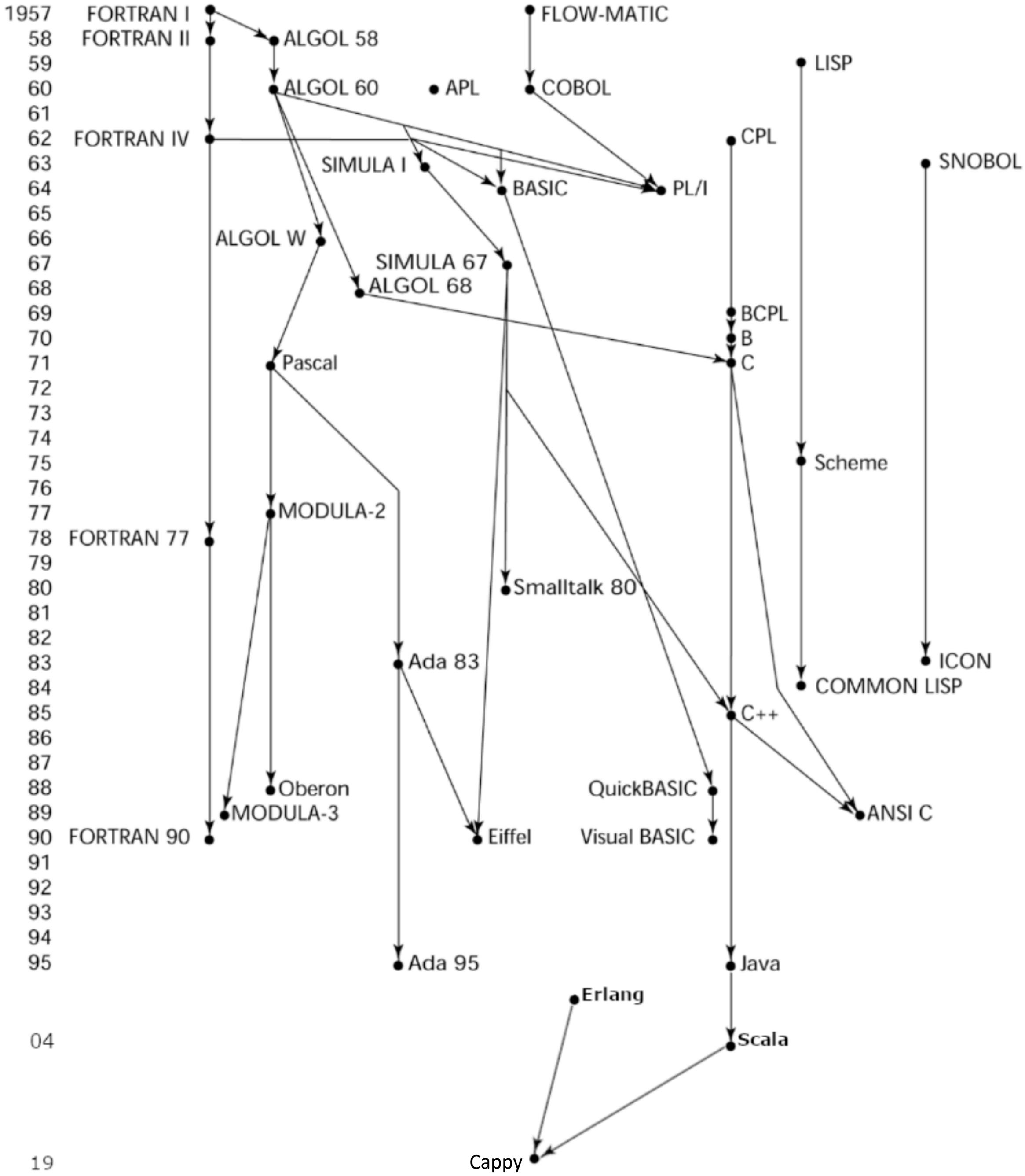
Version 0.0.1

1. Introduction

Cappy is a modern, functional, type-safe programming language. It is based on Erlang and Scala, but also incorporates things that just make sense to me, both applicably and aesthetically. Cappy differs from Erlang and/or Scala in the following ways:

1. Cappy is a purely functional programming language. Therefore, there are no classes or objects, variables are immutable, and there are no loops – only recursion.
2. Cappy is strong typed, so all variables must be one of the specified data types.
3. Unlike Erlang, but like Scala, Cappy is statically typed, so data types are **NOT** inferred. Data types must be declared by the programmer.
4. Cappy is compiled by the CAM virtual machine.
5. Unlike Erlang, but like Scala, brackets are used to indicate the start and end of a block.
6. There are no if statements, only case statements. However, case statements can be used in one of two ways: a case statement with a sequence of guards or a case statement with a sequence of patterns utilizing an optional guard (see 1.6 for more).
7. Functions are called in the cap command line: c(module_name), module_name:function_name(params)

1.1 Genealogy



1.2 Hello world

println(s”hello world!”)

1.3 Program structure

The key organizational concepts in Cappy are as follows:

1. Cappy code is divided into modules. A module consists of a sequence of attributes and function declarations. The ***module*** statement is like adding a namespace to any programming language. The module name needs to be the same as the file name minus the extension .cappy.
2. The predefined module attribute ***export*** specified which of the public functions defined within the module will be visible from outside the module.
3. Private and public functions must be declared in their respective division.
4. Variables are immutable.

Structural Example:

```
-module(bakery)
-author(“calista”)
-export([start/0])

-public
def start() : unit = {
   var baked_good : tuple = (“donut”, 1.00)
   var priced_good : double = price(baked_good)
   // tuple elements can also be accessed with ._position
   println(s”baked good: ${baked_good._1}, price:
$${priced_good}”).
}
-private
// an example of pattern matching in Cappy. Gets the price from
the tuple.
def price(item : tuple) : double = {
   var (name : string, price : double) = item
   price
}
```

This example declares a module named bakery. The module contains two functions, start, which is public, and price, which is private. This example displays two ways to access the elements of a tuple. The price function utilizes pattern matching to return the price of the baked good. The print statement utilizes position to access the name of the baked good.

1.4 Types and Variables

There are two kinds of types in Cappy: ***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

Cappy is very strict about visibility when it comes to functions. Public and private functions must be organized in their respective divisions (-public, -private). Public functions that will be called from the command line when running the module must be exported. Variable visibility is optional.

1.6 Statements Differing from Scala and Erlang

<table>
<tr><th>Statement</th><th>Example</th></tr>
<tr><td>Expression statement</td><td><pre>
-module(expression_stmt)
-author(“calista”)
-export([start/0])

-public
def start() : unit = {
   var x : int = 1
   var y : int = 5
   var sum : int = x + y

   println(s”${x} + ${y} = ${sum}”)
}
</pre></td></tr>
<tr><td>Case statement: sequence of guards</td><td><pre>
-module(case_stmt)
-author(“calista”)
-export([start/0])

-public
def start(x, y) : unit = {
   case x of
        x > y -> println(s“${x} > ${y}”)
        x < y -> println(s“${x} < ${y}”)
        _   -> println(s“${x} = ${y}”)
}
</pre></td></tr>
<tr><td>Case statement: sequence of patterns with an optional guard</td><td><pre>
-module(case_stmt)
-author(“calista”)
-export([start/0])

-public
def start(x, y) : unit = {
   case x >= y
       true -> println(s“${x} >= ${y}”)
       false -> println(s“${x} < ${y}”)
}
</pre></td></tr>
</table>

<table>
<tr>
<td>Recursive function declaration which also displays public and private functions.</td>
<td><pre>
-module(recursive_fn)
-author(“calista”)
-export([start/0])

-public
def start() : unit = {
   println(s“${sum(10)}”)
}

-private
rec def sum(num : int) : int = {
   case num of
        num == 1 -> 1
        _   -> sum(sum-1) + num
}
</pre></td>
</tr>
</table>

2. Lexical structure

2.1 Programs

All Cappy programs must be ***compiled*** to object code. CAM (Cappy Abstract Machine) is a virtual machine that is part of the Cappy Run-Time System (CRTS), which compiles Cappy source code into bytecode, which is then executed on the CAM. CAM bytecode files have the .cam file extension.

Conceptually speaking, a Cappy program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of bytecode characters.
2. Lexical analysis, which translates a stream of bytecode 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 Cappy programming language where it differs from Scala and Erlang.

2.2.1 Lexical grammar (tokens) where different from Scala and Erlang:

The lexical grammar of Cappy is essentially Erlang and Scala combined. Therefore, what is different from one is familiar to the other. The Cappy lexical grammar is as follows:

```
<Assignment operator>           →  =
<Mathematical operators>        →  + | - | * | /
<Comparison operators>          →  == | != | <= | >=
<keyword>                       →  language defined
                                   variable defined
<begin block>                   →  {
<end block>                     →  }
<single line comment >          →  //
<begin multi line comment>      →  /*
<end multi line comment>        →  */
<concatenation>                 →  ++
```

2.2.2 Syntactic (“parse”) grammar where different from Scala and Erlang

Again, the syntactic grammar for Cappy is a combination of Scala and Erlang. Some of the syntactic grammar for Cappy is as follows:

```
<identifier>              → A-Z, a-z, ℝ
<data type>               → Int, Boolean, String, Long, Double, Char, Unit, List, Tuple
<module declaration>      → -module(<identifier>)

<variable declaration>    → var <identifier> : <data type>
```

<parameter> → <identifier> : <data type>

<function> → def <identifier> (<parameter(s)>) : <return data type>

<recursive function> → rec def <identifier> (<parameter(s)>) : <return data type>

2.3 Lexical analysis

2.3.1 Comments

Two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the characters `//` and extend to the end of the source line. ***Delimited comments*** start with the characters `/*` and end with the characters `*/`. 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. Cappy's tokens are strongly based on Scala's tokens.

Cappy Tokens: identifier, keyword, integer-literal, double-literal, long-literal, character-literal, string-literal, boolean-literal, operator-or-punctuator

2.4.1 Keywords different from Scala and/or Erlang

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier ever.

New keywords:

```
rec, def, author, export, private, public, var, and, or, when, unit
```

Removed keywords:

```
Class, Object, val, if, else, else if, io:fwrite,io:fread, ok, end, begin,
foreach, for, while, do, extends, new, yield, any, float
```

3. Type System

Unlike Erlang, but like Scala, Cappy 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

Some of the type rules for Cappy are as follows:

String Concatenation:

$\vdash$ e1 : string
$\vdash$ e2 : string

____
$\vdash$ e1 ++ e2 : string

List Concatenation:

$\vdash$ e1 : list
$\vdash$ e2 : list

____
$\vdash$ e1 ++ e2 : list

Assignment with Scope Context:

S $\vdash$ e1 : T
S $\vdash$ e2 : T
T is a primitive type

____
S $\vdash$ e1 = e2 : T

Integer Literals with Scope Context:
S $\vdash$ e1 : integer
S $\vdash$ e2 : integer

____
S $\vdash$ e1 + e2 : integer

Comparisons with Scope Context:
S $\vdash$ e1 : T
S $\vdash$ e2 : T
T is a primitive type

____
S $\vdash$ e1 != e2 : boolean

Cappy types are divided into two main categories: ***value types*** and ***reference types***. I considered not using reference types due to Cappy's immutable nature, but lists, strings, and tuples are implemented in Scala using references, so I decided against it and kept the reference types. Also, for the sake of aesthetics, types begin with a lowercase letter. I don't care what problems this causes theoretically; it just looks nicer in Cappy.

3.2 Value types (Based on Scala)

Data Type	***Description***
Int	32-bit signed value. Range -2147483648 to 2147483647
Double	64-bit IEEE 754 double-precision float
Long	64-bit signed value. -9223372036854775808 to 9223372036854775807
Char	16-bit unsigned Unicode character. Range from U+0000 to U+FFFF
Boolean	Either the literal true or the literal false
Unit	Corresponds to no value

3.3 Reference types (Based on Scala and Erlang)

Data Type	***Description***
String	A sequence of Chars
List	a compound data type with a variable number of terms.
Tuple	a compound data type with a fixed number of terms

4. Example Programs

4.1 & 4.2 Caesar Cipher Encrypt & Decrypt

```
-module(encrypt)
-author(“calista”)
-export([caesar/2])

-public
def caesar(str : string, shift : int) : unit = {
     var encrypted_str : string = encrypt(str.to_upper, shift)
     println(s”encrypted: ${encrypted_str}”)

     var decrypted_str : string = decrypt(encrypted_str, shift)
     println(s”dcrypted: ${decrypted_str}”)
}

-private
def encrypt(str : string,  shift : int) : unit = {
     str.to_list.map(x -> shift_char(x, shift)).to_string
}

def decrypt(encrypted_str : str,  shift: int) : string = {
     encrypt(encrypted_str, 26 - shift)
}

def shift_char(chr: char, shift: int) : char = {
     var ascii_val : int = char.to_int
     case ascii_val of
          32  -> (32).to_char
           _     -> (((ascii + shift  - 65) % 26) + 65).to_char
}
```

4.2 Factorial

```
-module(factorial)
-author(“calista”)
-export([factorial/1])

-public
rec def factorial(num : int) : int = {
     case num of
          num <= 1 -> 1
          _        -> num * factorial(num-1)
}
```

4.3 Recursive Bubble Sort based on Erlang’s recursion methods.

```
-module(factorial)
-author(“calista”)
-export([bubble_sort/2])

-public
// This works like Erlang’s recursion.
rec bubble_sort(l: list) when l.length =< 1 : list = {
     l
}
rec bubble_sort(l : list) : list = {
    SL = bubble_sort_p(l: list)
    bubble_sort(lists:sublist(sl,1,(sl.length)-1)) ++
[lists:last(sl)]

}

rec bubble_sort_p([]: list) : list = {
   []
}
rec bubble_sort_p([f]: list) : list = {
  [f]
}

rec bubble_sort_p([f,g|t] : list) when f > g : list = {
    [g|bubble_sort_p([f|t] : list)]
}

rec bubble_sort_p([f,g|t] : list) : list = {
    [f|bubble_sort_p([g|t] : list)]
}
```

4.4 Mutual Recursion

```
-module(factorial)
-author(“calista”)
-export([is_even/1, is_odd/1])

-public
def is_even(num : int) : boolean = {
   case num of
        n == 1 -> true
        n == 0 -> false
        _           -> is_odd(n-1)
}

def is_odd(num : int) : boolean = {
   case num of
        n == 1 -> false
        n== 0  -> true
        _           -> is_even(n-1)
}
```

4.5 Filters

```
-module(factorial)
-author(“calista”)
-export([start/0])

-public
def start() : unit = {
   var l : list = [1, 2, 3, 4, 5, 6, 7]
   // l[2, 4, 6]
   var even_l : list = l.filter((n : int) -> n % 2 == 0)
   // l[1, 3, 5, 7]
   var odd_l : list = l.filter((n : int) -> n % 2 != 0)

   println(s”even: ${even_l}”)
   println(s”odd: ${odd_l}”)

}
```