CMPT 331L - Fall 2019 - Dr. Labouseur

Scaling

Oliver Fountain

Oliver.Fountain1@Marist.edu

December 16, 2019

Contents

1 Introduction

Scaling is a modified version of Scala that takes inspiration from Erlang. This led to the name, combining "Scala" and "Erlang" to make "Scalang". However, someone had my idea before and already used that name, so I simplified it to Scaling. Scaling also draws some inspiration from COBOL, this can be seen most obviously in the storage and procedure sections of each function. The goal of this language was to make something that is very easy to format and create readable code. The new features of Scaling are listed below:

1. The declaration of variables is changed so the type comes first. This was done to make the "storage" section of each function look neater and to make it easier for the programmer to categorize and index their variables.

2. Each function is broken up into two parts, storage and procedure. The storage section is where all variables must be declared. This was done to enforce best practices when writing code and help keep the programs organized. The procedure function comes next, this is where all of the code goes. Variables can be reassigned in the procedure but not declared.

3. The types Float, Long, Short and Byte have been removed and all fall under the identifier of int. This is done because the compiler of Scaling classifies the numeric variables itself behind the scenes and reduces the complexity to programmer.

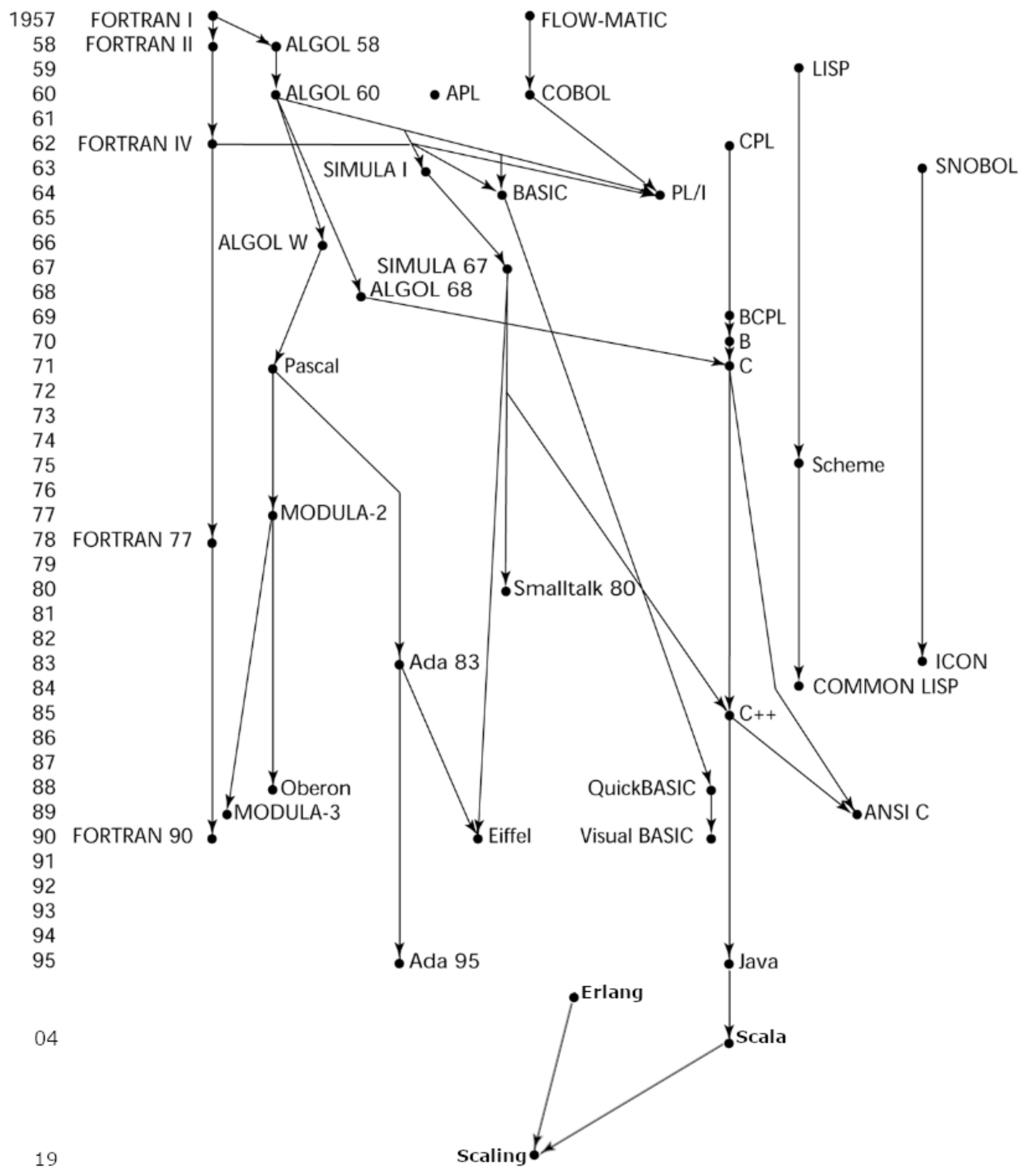
4. The type Double has also been put behind the scenes and is now represented by Float. When a Float is declared, the compiler decides the best way to store the number in memory.

5. Nested comments have also been added to allow for easier debugging and to encourage the use of comments when writing code.

6. The end of every statement in Scaling must be terminated with a semicolon. This was done to remove this languages dependency on whitespace and newlines to preemptively avoid any annoying issues in the code.

7. In Scaling, Strings are stored in memory as arrays of characters. This allows for easier string manipulation as well as reducing the need for extra built in functions.

1.1 Genealogy



1.2 Hello World!

```
object HelloWorld {

    def main(){

        storage {
            String: HelloWorld = "Hello World!";
        }

        procedure {
            printLine(HelloWorld);
        }
    }
}
```

1.3 Program structure

```
object Example {

    def main() {

        storage {

            int: number := 5;
            String: word := "Hello";
            bool: yesno := true;

        }

        procedure {

            printLine("Scaling is an easy to use lanaguage the follows a strict form");

            case (number == 5) {

                true {

                    printLine("Cases are just as easy to use as if statemets!");

                }

                false {

                    printLine("This is impossible");

                }

                //I can write comments too!

                /* even /* nested /* ones */ */ */

            }
        }
    }
}
```

- All code must be inside the object block that is named after the filename of the program.

- The format of the code is determined by end of line semicolons and functions, storage and procedures are terminated by closing parenthesis. Scaling does not make use of whitespace to determine end of lines or functions.

- Like COBOL, Scaling has two parts to every function the storage and the procedure. Variables can only be declared in the storage section. This is done to encourage best practices and help organize the code in a more readable way. Directly after the storage section is the procedure section. This is where all of the code belongs, variables can be reassigned in this section but not declared.

- Variables are assigned using the ":=" sign.

- The type of the variable in its declaration is specified first, this allows for better formatting of the lines. After the type a colon is used to indicate that the variable name is coming next.

- To test for equality in Scaling, "==" is used and "!=" is used for inequality.

1.4 Types and variables

Scaling uses the two standard variables types, value and reference. All primitive variables like ints and chars are stored as their value in memory. More complex types like objects and Strings are stored as references to locations in memory.

1.5 Example Statements

Statement	**Example**

Basic Expression	`object basicExpression{`
	
	`def main(){`
	
	`        storage {`
	`            int: x := 1;`
	`            int: y;`
	`        }`
	
	`        procedure {`
	`            y := 3;`
	`            printLine(x + y);`
	`        }`
	`    }`
	`}`

Statement | **Example**

Nested Comments

```
object nestedComments{

//Single line comment
    def main(){

        storage {
            String: hello := "Hello";
        //  String: test := "Test";

        }

        procedure {      //I can start them mid line too

            printLine(hello);

            /*I dont want to print test right now
            but I can keep my comments nested inside this one

                /*these statements print test two times */
                printLine(test);
                printLine(test);

            */
        }
    }
}
```

Strings as Arrays of Characters

```
object stringsasArrayofChars{

    def main() {

        storage {
            String: examplestring := "Hello World";
            String: examplesubstring;
            char: examplechar;
        }

        procedure {

            //examplestring is now equal to "Hello"
            examplesubstring := examplestring(0, 5)

            //examplechar is now equal to 'H'
            examplechar := example(0);

        }
    }
}
```

If Else

```
object ifElse{

    def main(){

        storage {
            int: x := 1;
            int: y := 2;
        }

        procedure {

            case (x < y){

                true {
                    printLine("X is less than Y");
                }

                false {
                    printLine("Y is less than X");
                }
            }
        }
    }
}
```

Statement	Example

Multipart if Else	`object multipartIfElse{`
	
	`    def main(){`
	
	`        storage {`
	`            int: x := 1;`
	`        }`
	
	`        procedure {`
	
	`            case (x){`
	
	`                (x >= 5) {`
	`                    printLine("X is greater than or equal to five");`
	`                }`
	
	`                (x <= 3) {`
	`                    printLine("X is less than or equal to three");`
	`                }`
	
	`                (x == 4) {`
	`                    printLine("X is equal to four");`
	`                }`
	
	`                () {`
	`                    printLine("Else");`
	`                }`
	`            }`
	`        }`
	`    }`
	`}`

2 Lexical Structure

2.1 Programs

A Scaling 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

Although Scaling is based on Scala, it has several differences that I will outline below.

2.2.1 Lexical grammar where different from Scala

<assignmentOperator>	->	:=
<booleanOperator>	->	== | != | <= | >= | <| >
<endOfLineCharacter>	->	;

<assignmentOperator>	->	:=
<print>	->	print | printLine
<concatarrays>	->	++

2.2.2 Syntactic grammar where different from Scala

<program>	->	object <string>{ <function>}
<function>	->	def <string>(<input>) { <storage><procedure>} <function>
	->	ϵ
<storage>	->	storage{ <declaration>}
<procedure>	->	procedure{ <statement>}
<declaration>	->	<type>: <string>= <value>; <declaration>
	->	ϵ
<ifElseStatement>	->	case(<expr>) { <ifBool>}
<ifBool>	->	<boolExpr>{ <statement>} <ifBool>
	->	ϵ

2.3 Lexical analysis

2.3.1 Comments

Scaling supports two types of comments, single-line and multi-line. To use a single line comment simply type // everything after the two slashes will be commented out until a newline is found. To begin a multi-line comment, the characters /* are used, to terminate the comment use */. Scaling does have support for nested comments. This was done to allow for easier debugging and encourage commenting of the code as the program is being written.

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 Scala

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

- New keywords: storage, procedure, case, printLine, const, void

- Removed keywords: extends, if, else, charAt, substring, println, Double, long, short, byte, val, var

3 Types

Scaling types are divided into two main categories: Value types and Reference types.

3.1 Value Types

Float: a floating point numeric value, the compiler can determine when it is appropriate to use float or double.
 - Float: exampleFloat := 99.99;

Int: A method of storing numbers, the compiler automatically sets the length of the datatype, removing the need for Float, Long, Short and Byte datatypes.
 - Int: exampleInt := 99;

Boolean: A type that represents either true or false.
 - Boolean: exampleBool := true;

Char: A single character.
 - Char: exampleChar := ‘a’;

3.2 Reference Types

List: An immutable collection of data set in length.
 - List(Char): exampleList := [1,2,3,4,5];

Array: A mutable collection of data that has a variable length .
 - Array(Char): exampleArray := [1,2,3,4,5]

String: An array of characters.
 - String: exampleString := “Hello”;

4 Example Programs

4.1 Caesar Cipher Encrypt

```
object Encrypt {

    //Encrypt fucntion that takes in a string as plaintext as well as
    //a cipher number to return the encrypted text
    def encrypt String: ( String: plaintext, int: ciphernum) {


        storage {

            String: ciphertext;
            int: tempciphernum := ciphernum;

        }

        procedure {

            //Case for wrap around, if the character goes past z,
            //subtract 26.  Else, do nothing
            case (((plaintext(0)).toInt) + ciphernum) > 90) {

                true {
                    tempciphernum := tempciphernum - 26;
                }

                false {

                }
            }

            //Case for finding spaces in the plaintext
            //If a space is found add it to the ciphertext
            //If not, shift the character
            case () {

                (plaintext.length() > 1 && plaintext(0) != ' ') {
                    ciphertext := ((((plaintext(0)).toInt) + tempciphernum).toChar);
                    ciphertext := ciphertext ++ (encrypt(plaintext(1, plaintext.length()), ciphernum));
                }

                (plaintext.length() > 1 && plaintext.charAt(0) == ' ') {
                    ciphertext := ' ';
                    ciphertext := ciphertext ++ (encrypt(plaintext(1, plaintext.length()), ciphernum));
                }

                //Else the length of the string is one, do not
                //recursivly call the fucntion
                () {
                    ciphertext := ((((plaintext(0)).toInt) + tempciphernum).toChar);
                }

            }

            return(ciphertext);
        }
    }
}
```

4.2 Caesar Cipher Decrypt

```
object Decrypt {

    //Decrypt fucntion that takes in a string as ciphertext as well as
    //a cipher number to return the decrypted text
    def decrypt String: ( String: ciphertext, int: ciphernum) {


        storage {

            String: plaintext;
            int: tempciphernum := ciphernum;

        }

        procedure {

            //Case for wrap around, if the character goes past a,
            //add 26.  Else, do nothing
            case ((((ciphertext(0)).toInt) - ciphernum) < 65) {

                true {
                    tempciphernum := tempciphernum + 26;
                }

                false {

                }
            }

            //Case for finding spaces in the ciphertext
            //If a space is found add it to the plaintext
            //If not, shift the character
            case () {

                (ciphertext.length() > 1 && ciphertext(0) != ' ') {
                    plaintext := ((((ciphertext(0)).toInt) - tempciphernum).toChar);
                    plaintext := plaintext ++ (decrypt(ciphertext(1, ciphertext.length()), ciphernum));
                }

                (ciphertext.length() > 1 && ciphertext.charAt(0) == ' ') {
                    plaintext := ' ';
                    plaintext := plaintext ++ (decrypt(ciphertext(1, ciphertext.length()), ciphernum));
                }

                //Else the length of the string is one, do not
                //recursivly call the function
                () {
                    plaintext := ((((ciphertext(0)).toInt) + tempciphernum).toChar);
                }

            }

            return(plaintext)
        }
    }
}
```

4.3 Factorial

```
object Factorial {

    //Factorial function that takes in a number
    //and return its factorial
    def factorial int: ( int: number) {

        storage {

            int: tempnum := number;

        }

        procedure {

            //Case for if the number is greater than 0
            //if it is, multiply it by one less than itself
            //recursivly.  If it is 0 then muliply by one
            case (temp > 0) {

                true {

                    tempnum := tempnum * factorial(tempnum - 1);

                }

                false {

                    return(1);

                }
            }
        }
    }
}
```

4.4 Bubble Sort

```
object BubbleSort {

    //Main method to create the unsorted array, call bubblesort and
    //print the result
    def main() {

        storage {

            Array( int ): sortarray := [9,6,17,2,6,12,10,7,5,22];

        }

        procedure {

            bubblesort(sortarray, sortarray.size);

            printarray(sortarray);

        }


    }

    //BubbleSort fucntion that takes in an array and a pointer to return null
    def bubblesort void: (Array( int ): sortarray: , int: pointer) {

        storage {

            int: temp;

        }

        procedure {

            //If the pointer is at position one, the whole
            //list has been sorted and the fucntion can stop
            case (pointer == 1) {

                true {

                    return;

                }

                //If the pointer is not at one, run a for
                //loop and run through the array, swapping
                //digits where appropriate
                false {

                    for(i <- 1 to (pointer - 1)){

                        //Case to find if a number is larger
                        //than its neighbor, if it is, swap
                        //them.  Else, do nothing
                        case (sortarray(i) > sortarray(i-1)) {

                            true {

                                temp := sortarray(i);
                                sortarray(i) := sortarray(i - 1);
                                sortarray(i - 1) := temp;

                            }

                            false {

                            }
                        }
                    }
                }
            //Recursivly call bubblesort but move the pointer
            bubblesort(sortarray, pointer - 1);

        }
    }

    //Fucntion to run through the array one by one and print
    //the result on each line
    def printarray(Array( int ): sortedarray) {

        storage {

        }

        procedure {

            for(i <- 0 to (sortedarray.size - 1)){

                printLine(sortedarray(i));

            }
        }
    }
}
```

4.5 Fibonacci

```
object Fibonacci {

    //Main method to create the basic fibonacci array
    //and call the fucntion
    def main() {

        storage {

            Array( int ): fibarray := [0,1];

        }

        procedure {

            printarray(fibonacci(fibarray, 15));

        }

    }

    //Fibonacci fucntion that takes in the array as well as the
    //number of iterations to run and returns the fibonacci
    //sequence as an array
    def fibonacci Array( int ): (Array( int ): fibarray, int: cap) {

        storage {

            Array ( int ): temparray := fibarray;
            int: temp;

        }

        procedure {

            //If the cap has not been reached, add the previous
            //two values and add them to the sequence. then
            //recursivly call the fucntion
            case (fibarray.size <= cap) {

                true {

                    temp := (fibarray(fibarray.size - 2)) + (fibarray(fibarray.size - 1));

                    temparray := temparray ++ temp;
                    fibonacci(temparray, cap);

                }

                //When the cap is reached, return the finished
                //sequence
            //sequence
            false {

                return temparray;

            }
        }
    }
}

//Fucntion to run through the array one by one and print
//the result on each line
def printarray(Array( int ): sortedarray) {

    storage {

    }

    procedure {

        for(i <- 0 to (sortedarray.size - 1)){

            printLine(sortedarray(i));

        }
    }
}
}
```

4.6 Distance Between Two Points

```
object DistanceBetweenPoints {

    //Main method to create the two points as lists and
    //call the fucntion that finds the distance of the
    //line between them
    def main() {

        storage {

            List( int ): point1 := [4,3];
            List( int ): point2 := [3,-2];

        }

        procedure {

            printLine(distancebetween(point1, point2));

        }

    }

    //This fucntion takes in the two points as lists and
    //returns a float that is the distance between them
    def distancebetween Float: (List( int ): point1, List( int ): point2) {

        //Assign all the varibales from the input
        storage {

            Float: x1 := point1(0);
            Float: y1 := point1(1);

            Float: x2 := point2(0);
            Float: y2 := point2(1);

            Float: distance;

        }

        //Apply the distance formula to the points and return
        //the result
        procedure {

            distance := Math.sqrt((x2 - x1) * (x2 - x1)) + ((y2 - y1) * (y2 - y1));

            return distance;

        }
    }
}
```