Concerto

Language Summary
and Example Programs
Version Tonic

Shannon Brady

Theory of Programming Languages

Professor Labouseur

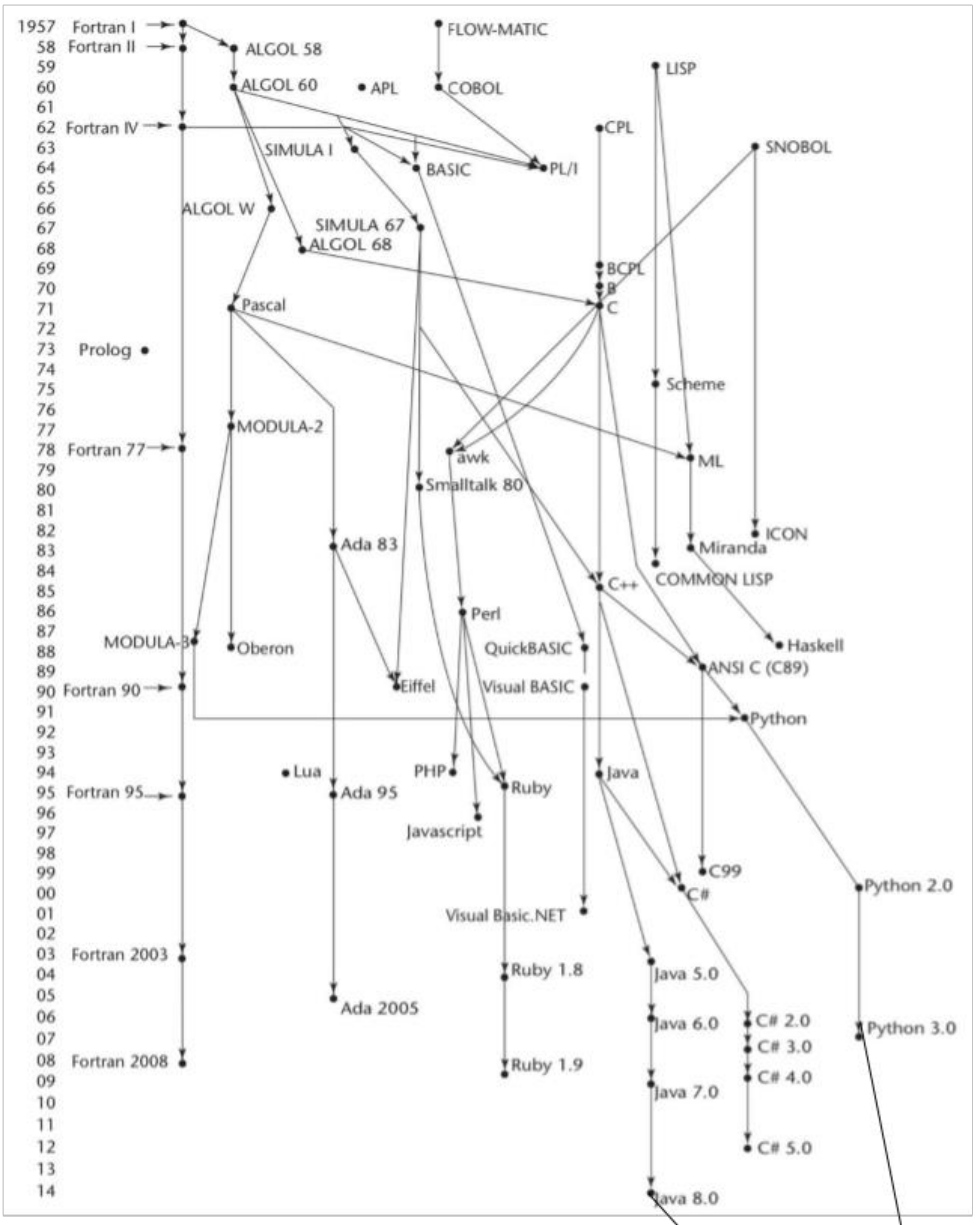
26 May 2021

1. Introduction

1. **Concerto** (pronounced "*kuhn • chehr • tow*") is a simple, modern, object-oriented, strongly-typed, and musically literate programming language. Based on predominantly on Scala with a slight hint of Python, **Concerto** differs in the following ways:

    a. **Concerto** is strongly typed. Variables must be one of the included data types (see Section 3).
    b. **Concerto** is compiled.
    c. **Concerto** is statically scoped.
    d. The end of every statement must be terminated with a semicolon. This eliminates **Concerto's** dependency on whitespace and newlines.
    e. Strings are stored in memory as arrays of characters, which allows for easier string manipulation as well as a diminished need for additional built-in functions.
    f. *Ensemble.playln()* provides console printing.
    g. Null values are denoted with *niente*.
        i. "niente" -- none or nothing
    h. Defining the main function is achieved by the following: *def void tutti()*
        i. "tutti" -- with all voices or instruments performing together
    i. All language elements emulate ensemble/orchestra functionality and its relevant musical attributes.

1.1 Genealogy



1.2 Hello World

```
performance helloWorld {

    def void tutti() {

        Ensemble.playln("Hello World!");
    }
}
```

1.3 Program structure

The key organizational concepts in **Concerto** are as follows:

1. The format of the code is determined by end of line semicolons and functions.
2. Execution of instructions is sequential.
3. Data types are declared first in variable declaration, which is immediately followed by a colon and then the variable name. This serves to enhance overall readability.
4. The main function is referred to as "tutti" (previously defined in Section 1.1).
5. Objects are referred to as *performances*.
6. Classes are referred to as *compositions*.
    a. "composition" -- refers to an original piece or work of music
7. Variables are assigned using "=>".
8. Instead of return statements, there are *rehearse* statements.
9. Test for equality is delimited by two equals signs "==". Conversely, the test for inequality is delimited by "!=".
10. Phrase (string) concatenation is achieved with "++".

1.4 Types and variables

Concerto uses two standard variable types, value types and reference types. All primitive variables like *beats* (integer/float) and *notes* (characters) are stored as their value in memory. More complex types like *phrases* (strings) and *performances* (objects) are stored as references to locations in memory. In regards to reference types, two variables may reference the same object; therefore, it is

possible for operations on one variable to affect the object referenced by the other variable.

1.5 Statements Differing from Scala and Python

Statement	Example

Expression Statement	`performance expression {`
	
	`  def void tutti() {`
	
	`  Phrase : myName => "Shannon Brady";`
	`  Ensemble.playln(myName);`
	
	`  }`
	`}`

For Statement	`  def void tutti() {`
	
	`  Phrase : exampleStr => "I love CMPT331";`
	
	`  for (i in range(len(exampleStr)))`
	`  Ensemble.playln(exampleStr[i]);`
	`  }`
	`}`

String as Array of Characters	`performance strAsChars {`
	
	`  def void tutti() {`
	
	`  Phrase : exStr => "Where words fail, music speaks";`
	`  Phrase : exSubstr;`
	`  Note : exChar;`
	
	`  # exSubstr is now "music speaks"`
	`  exSubstr => exStr[18, 29];`
	`  Ensemble.playln(exSubstr);`
	
	`  # exChar is now "W"`
	`  exChar => exStr[0];`
	`  Ensemble.playln(exChar);`
	
	`  # exStr is now "Whiri words fail, music spieks"`
	`  exStr => exStr.replace("e", "i");`
	`  Ensemble.playln(exStr);`
	`  }`
	`}`

If/Else If/ Else Block	

```
performance ifElse {

    def Beat getLargestNum(Beat : x, Beat : y) {
        Beat : largest => 0;

        if (x > y)
            largestNumber => x;
        else if (y > x)
            largestNumber => y;
        else largestNumber => Niente;

        rehearse largest;
    }

    def void tutti() {
        Beat : a => 10;
        Beat : b => 20;
        Beat : largestNum => getLargestNum(a, b);

        if (largestNum == Niente)
            Ensemble.playln("The numbers are equal");
        else Ensemble.playln("The largest number is " ++ largestNum);
    }
}
```

2. Lexical structure

2.1 Programs

A **Concerto** program consists of one or more *source files*. A source file is an ordered sequence of Unicode characters.

Conceptually speaking, a program is compiled using three main 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

This specification presents the syntax of the **Concerto** programming language where it differs from Scala and Python.

2.2.1 Lexical grammar where different from Scala and Python

The lexical grammar of **Concerto** is most closely compared to Scala with some flavors of Python.
Should identifiers contain one or more digits, those digits must appear at the end of the identifier.
Keywords or reserved words cannot be used as identifiers. Identifiers cannot be re-initialized in the same scope.

```
<assignment operator>    →    =>

<null value>             →    Niente

<end of line char>       →    ;

<concatenation sym>      →    ++

<identifier>             →    <character> <character list>
```

```
                              →      <character> <digit list>
                              →      <character>

<character list>              →      <character> <character list>
                              →      <character>

<digit list>                  →      <digit> <digit list>
                              →      <digit>

<digit>                       →      0,1,2,3,4,5,6,7,8,9

<character>                   →      a, b, ..., y, z
                              →      A, B, ..., Y, Z
```

2.2.2 Syntactic grammar where different from Scala and Python

```
<variable declaration>     →  <type>: <string> => <value>;

<performance declaration>  →  performance<identifier>

<composition declaration>  →  composition<identifier>

<function declaration>     →  def <type> <identifier> <parameter list>

<parameter list>           →  <parameter> <parameter list>
                           →  <parameter>

<parameter>                →  <type> <identifier>
```

2.3 Lexical analysis

2.3.1 Comments

Two forms of comments are supported by Concerto: 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 🎼>>. 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.

tokens:
identifier
keyword
integer-literal
real-literal
character-literal
string-literal
operator-or-punctuator

2.4.1 Keywords differing from Scala and Python

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the @ character.

New Keyword	Purpose
tutti	Used when declaring main function (ex: def void tutti())
composition	Class declaration
performance	Object declaration
phrase	String data type
beat	All-encompassing number data type
note	Character data type
niente	Null value
rehearse	Return statement
ensemble	Emulates Java's System class
playln play	Print statement
recordBeat recordNote recordPhrase	Read user input
void	Used when function has no rehearse (return) value

Removed Keywords:

println, print
readLine, readInt, read<remaining data types>
int

```
string
char
double
long
short
byte
val
var
null
class
object
```

3. Type System

Concerto, like Scala, uses a *strong static* type system. *Strong typing* means that type errors are caught and expressed to the programmer during compilation. Strongly-typed languages do not allow implicit conversions between unrelated types. *Static typing* means early binding compile-time type checking.

For example, a strong statically typed language like **Concerto** statically (at compile time) ensures that a certain value with type *Note* is correctly used throughout the program. At runtime, nothing else other than a *Note* can be held in that variable's memory location.

3.1 Type Rules

The type rules for **Concerto** are as follows:

Primitives:

Note *n*

$\vdash n$: Note

Phrase *p*

$\vdash p$: Phrase

Beat *b*

$\vdash b$: Beat

$\vdash$ sharp: Boolean

Boolean *bool*

$\vdash bool$: Boolean

$\vdash$ flat : Boolean

Assignment:

$S \vdash e_1 : T$
$S \vdash e_2 : T$
T is a primitive type

$S \vdash e_1 => e_2 : T$

Phrase Concatenation:

$S \vdash e_1$: Phrase
$S \vdash e_2$: Phrase

$S \vdash e_1 ++ e_2$: Phrase

Numerical Addition:

$S \vdash e_1$: Beat
$S \vdash e_2$: Beat

$S \vdash e_1 + e_2$: Beat

Comparison:

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 > e_2 : \text{Boolean}} \qquad \frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 <= e_2 : \text{Boolean}}$$

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 >= e_2 : \text{Boolean}} \qquad \frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 == e_2 : \text{Boolean}}$$

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 < e_2 : \text{Boolean}} \qquad \frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 != e_2 : \text{Boolean}}$$

Concerto types are divided into two main categories: *value types* and *reference types*.

3.2 Value types

Beat: A general purpose method of storing numbers; length of the datatype is automatically set by the compiler, which removes the need for individually defined Int, Float, Long, Short and Byte datatypes.

```
Beat : exNum => 12;
```

Boolean: A type that represents either sharp (true) or flat (false).

```
Boolean : exBool => sharp;
```

Note: A single (Unicode) character; an individual component of a phrase

Note : exChar => ‘♪’ ;

Niente: A null type representing no value

3.3 Reference types

Phrase: An array of notes (characters).

Phrase: exString => “Is it summer yet?” ;

Scale: An *array* (vector) that contains elements of all the same data type

Scale [Beat] : someNums => [1, 2, 3, 4, 5];
Scale [Note] : theNotes => [‘♩’ , ‘♫’ , ‘♪’];

Chord: A *list* (collection of items) that contains elements of multiple data types

Chord : stuff => [“apple”, 3.14159, flat, ‘%’];

4. Example Programs

4.1 Caesar Cipher (encrypt, decrypt and solve)

```
composition Caesar {

        Phrase : str => "";

        def Phrase encrypt(Beat : shftAmt) {
                Scale[Note] : charArray => str.toSharp().toNoteArray();
                Phrase : text => "";
        Beat : temp;

                for (i in range(len(charArray)) {
                        if (i == 32)
                                text += " ";
                        else {
                                temp = (i - 65 + shftAmt) % 26 + 65;
                                text += (temp).toNote;
                                }
                        }
                rehearse text;
        }

        def Phrase decrypt(Beat : shftAmt) {
                rehearse encrypt(26 - shftAmt);
        }

        def void solve() {
                for (i in range(26)) {
                        Ensemble.playln(i ++ " : " ++ encrypt(i));
                }
        }
}
```

4.2 Bubble Sort and Quick Sort

```
composition SomeSorts {

    def void bubbleSort(Scale[Beat] : myArray) {

        Beat : temp;

        for (i in range(len(myArray), -1, 0, -1)) {
            for (w in range(i)) {
                if (myArray[w] > myArray[w + 1]) {

                    temp => myArray[w];
                    myArray[w] => myArray[w +1 ];
                    myArray[w + 1] => temp;

                } #end if

            } #end for w
        } #end for i
    }
```

```
    def void quickSort(Scale[Beat] : myArray, Beat : min, Beat : max) {

        Beat : index;

        def Beat partition(Scale[Beat] : myArray, Beat : min, Beat : max) {

            Beat : i => min - 1;
            Beat : pivot => myArray[max];

            for (w in range(min, max)) {
                if (myArray[w] <= pivot) {

                    i += 1;
                    myArray[i], myArray[w] => myArray[w], myArray[i];

                } # end if

                myArray[i+1], myArray[max] => myArray[max], myArray[i+1];

            } # end for

            rehearse(i+1);
        }

        if (len(myArray == 1))
            rehearse myArray;

        if (min < max) {

            index => partition(myArray, min, max);
            quickSort(myArray, min, index-1);
            quickSort(myArray, index+1, max);

        } # end if
    }
}
```

```
performance testSorts {

    def void tutti() {

        Scale[Beat] : nums => [12, 6, 3, 22, 121];
        Scale[Beat] : moreNums => [55, 3, 8, 2, 11];
        SomeSorts : newSort => new SomeSorts();

        newSort.bubbleSort(nums);
        Ensemble.playln(nums);

        newSort.quickSort(moreNums);
        Ensemble.playln(moreNums);

    }
}
```

4.3 Factorial

```
performance factorial {

    def Beat calcFactorial(Beat : num) {
        Beat : temp => num;

        if (temp == 0)
            temp => 1;
        else temp => temp * calcFactorial(temp - 1);

        rehearse temp;
    }
}
```

4.4 Fibonacci Sequence

```
performance fibonacciSequence {

    def Scale[Beat] fibonacci(Beat : n) {

        Scale[Beat] : fibArray => [0,1];
        Beat : next;

        for (i in range(2, n+1)) {

            next => fibArray[-1] + fibArray[-2];
            fibArray.append(next);

        } #end for

        rehearse fibArray;

    }

    def void main() {
        Beat :  apex => 12;
        Scale[Beat] : outputArray;

        outputArray => fibonacci(apex);
        Ensemble.playln(outputArray);

    }
}
```