D♭

Amanda Potenza

Theory of Programming Language Final Project

May 2025

Version 1.3.5

Now playing "Fur Elise"

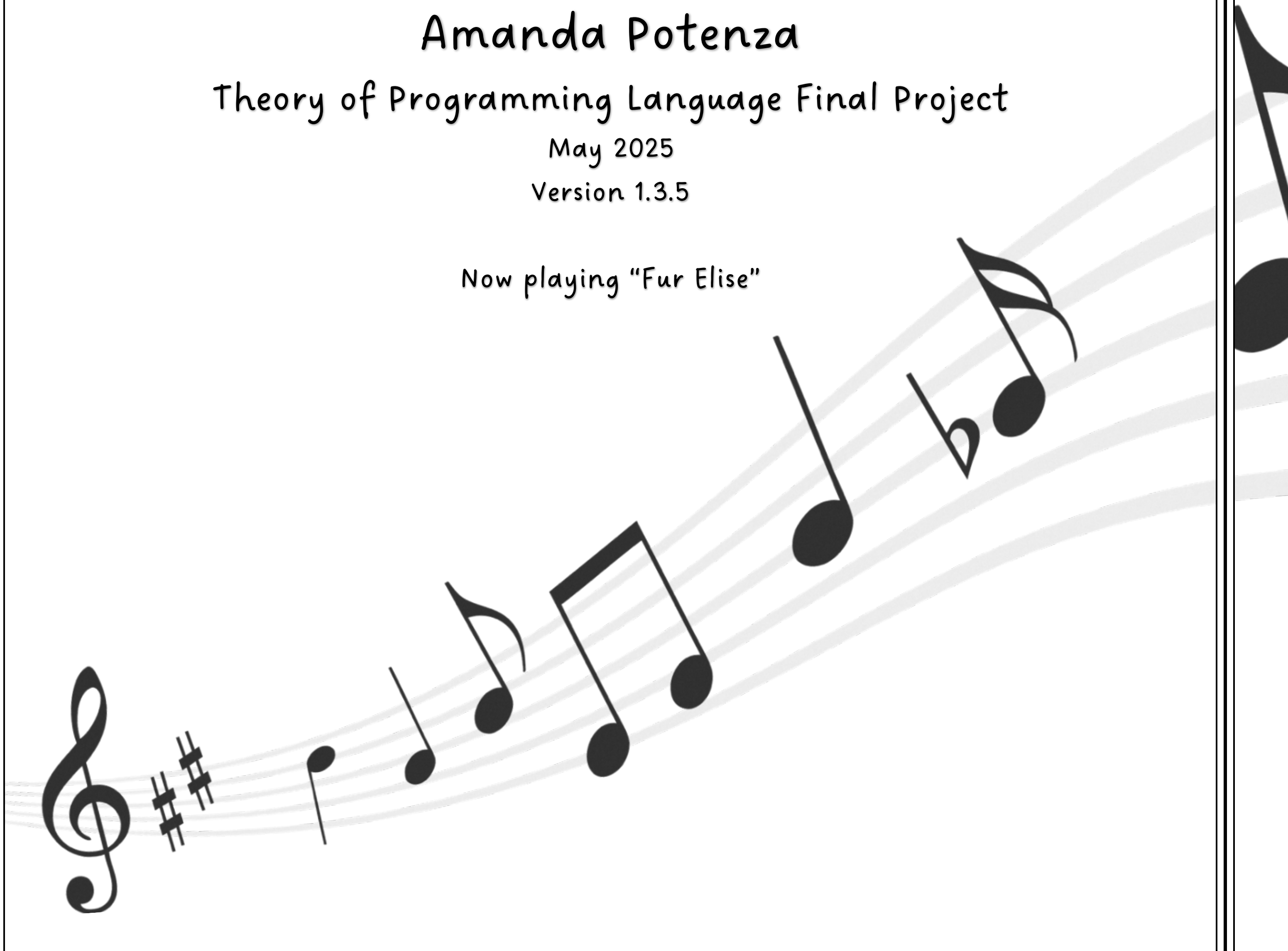


Table of Contents

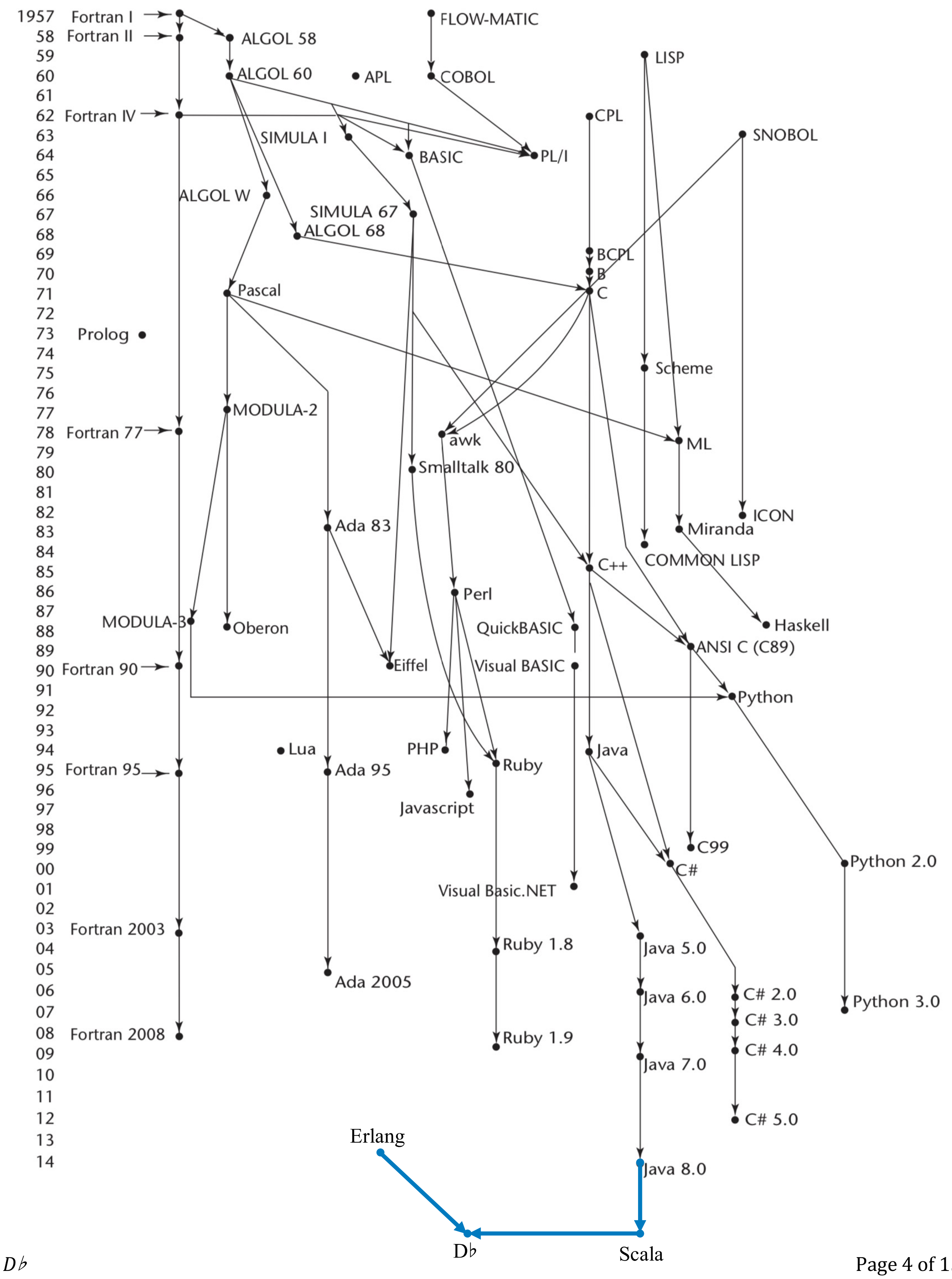
1. Introduction

I hope you practiced your major Scala(s)! “D♭” (pronounced “D flat”) is strongly typed, functional, and musical take on programming. The name “D♭” plays on the enharmonic equivalence of the note “C#,” another well-known programming language. Imagine if the movie “Pitch Perfect” had a computer science side quest… this is that! Despite the play on words, D♭ is not based on C#. The language takes heavy influence from languages with functional programming capabilities such as Scala and Erlang with some major differences:

1. Most keywords have a musical twist. Their meanings (to a degree) mirror their musical counterparts.
2. To home in on the functional programming capabilities of these languages, I removed some “procedural” belongings: no classes, no loops (let’s go recursion!), enforced immutability, and no void returning functions. In that sense, it is more like Erlang.
3. Utilizes Scala’s strong static type system.
4. Introduces scope using || instead of curly brackets (meant to mimic double bar lines).
    a. not to be confused with concatenation | (meant to mimic single bar lines).
5. Influenced by Scala’s higher-order functions that use lambdas directly.
6. Uses Erlang-style module exposure (‘publish’).
7. All functions require return type declarations.

1.1. Genealogy

Where does your language fit into the programming language genealogy? Add your language to this diagram to highlight your language and its ancestry.



1.2.Hello world

```
-compose(helloWorld);
-publish([main/0]);

# this is a “Hello World” script
notate main() ||
      play(“Hello World!”);
||
```

1.3.Program structure

The key organizational concepts in D♭ are as follows:

1. Every program begins with a module notated by the word “compose” followed by the word “publish” to mimic Erlang’s “export.”
    a. The content in the “compose” parentheses should reflect the filename.
2. Program blocks are indicated using || instead of curly brackets (meant to mimic double bar lines).
3. Single-line comments are notated using #, multi-line comments are notated using #b and b# (like flats and sharps)
4. Semicolons are not required in most instances, but they can avoid code ambiguity and some unexpected errors. Some examples of when to use them are after several declarations on one line, or if a line begins with (, [, +, -, /, or `. In optional scenarios, they can be used to visually terminate lines and increase readability.
5. D♭ defines functions using “def” like Scala and unlike Erlang’s “=>.”
6. Variables are assigned using “=”
7. The language does not support “for” or “while” loops, only recursion.
8. Type annotations are required for variables and return types.

```
-compose(reverseDemo);
-publish([main/0]);

#b
  This example demonstrates recursion and string handling
  in D♭. It reverses a string by recursively deconstructing
  the head and tail of the list.
b#

# reverse a list of characters
reverse([]) : []
reverse([H | T]) : reverse(T) ++ [H]

notate main() ||
  msg = "Amanda in D flat!";
  chars = explode(msg);     # turns string into char list
  rev = reverse(chars);
  out = implode(rev);       # turns char list back into string

  play("Reversed: " | out);
||
```

This snippet declares a module named reverseDemo using the compose keyword and exposes a single public function main using notate([main/0]). The module defines a recursive function reverse that operates on a list of characters. This function uses pattern matching to deconstruct the list into a head and tail: if the list is empty, it returns an empty list; otherwise, it appends the head element to the reversed tail. The main function assigns a string to the variable msg, then uses explode to convert the string into a list of characters. It applies the reverse function to the character list, then uses implode to reassemble the reversed list into a string. The result is output using the play function, printing the reversed string.

1.4.Types and Variables

There are two kinds of types in D♭: ***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. See Section 3 for details.

1.5.Visibility

Every module is private by default unless “publish” specifies otherwise. When using “publish,” the program “shares” the specified module(s) with the world (the program’s world!). Any function not in the -publish list is private and can only be called within the same module.

1.6.Statements Differing from Scala and Erlang

All types and expressions are static and checked at compile-time.

Statement	**Example**
Expression statement	`pages: Int = 100;`

solo/backup
(if/else) statement
	`notate main() ||`
	`  note : Int = 65;`
	`  solo (note > 60): ||`
	`    play("That's a high note!");`
	`  || backup: ||`
	`    play("That's a low note!");`
	`  ||`
	`||`

Function application	`notate square(n) ||`
	`  n * n`
	`||`
	
	`notate main()||`
	`  result: Int = square(4);`
	`  play("Square is: " ++ result);`
	`||`

Function chaining	`notate main()||`
	`  melody: Repertoire = [60, 62, 64, 65];`
	`  transposed = map(note -> note + 1, melody);`
	`  play(transposed);`
	`||`

Recursion loop	`# simulate a repeat-until behavior`
	`notate repeatPlay(note: Int, times: Int) : Unit ||`
	`  solo (times > 0): ||`
	`    play(note);`
	`    repeatPlay(note, times - 1);`
	`  ||`
	`||`

tunein (importing another module)	`# imagine importing a utility module for transformations`
	`tunein("transposeTools");`
	
	`notate main()||`
	`  melody : Repertoire = [60, 62, 64];`
	`  newMelody : Repertoire = transposeUp(melody, 2);`
	`  play(newMelody);`
	`||`

2.Lexical structure

2.1.Programs

A D♭ ***composition*** (aka 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

This specification presents the syntax of the D♭ programming language where it differs from Scala and Erlang.

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

<mathematical operators>	→	+, -, *, /, mod
<comparison operators>	→	==, !=, <, <=, >, >=
<assignment>	→	=
<begin/end block>	→	||
<single line comment >	→	#
<begin multi line comment>	→	#b
<end multi line comment>	→	b#
<concatenation>	→	|

2.2.2.Syntactic ("parse") grammar where different from Scala and Erlang

<program declaration>	→	-compose(<program name>);
<assignment operator>	→	-publish([main/0]);
<function declaration>	→	notate <function name>(<parameter list>) : <return data type>
<parameter>	→	*<identifier>* : *<data type>*
<identifier>	→	*A-Z, a-z,* ℝ
<data type>	→	*Int, Cadence, String, Float, Char, Repertoire, Unit*
<variable declaration>	→	*<identifier> : <data type> = <expression>*

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 character # and extend to the end of the source line. ***Delimited comments*** start with the characters #b and end with the characters b#. 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. This portion is heavily inspired by Scala.

```
tokens:
    identifier               # variable/function/module names
    keyword                  # reserved musical words (e.g., solo, play, Repertoire, Unit)
    integer-literal          # numeric values like 42
    real-literal             # float values like 3.14
    character-literal        # single chars like 'A'
    string-literal           # text like "Amanda in D flat!"
    list-literal             # Repertoire of values like [1, 2, 3]
    operator-or-punctuator   # +, -, *, /, ==, !=, <, >, mod, |, ||
```

2.4.1. Keywords different from Scala and Erlang

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 Keywords		Removed Keywords
• compose (module)	• crescendo (increment)	• Class
• publish (export)	• decrescendo (decrement)	• Object
• tunein (import)	• Repertoire (list)	• for
• || (curly brackets)	• Unit	• while
• Cadence(boolean)		• io:format
• harmonious (true)		• ok
• dissonant (false)		• begin
• solo/backup (if/else)		• end

3.Type System

D♭ 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

The type rules for D♭ are as follows:

concatenation:

$S \vdash e_1 : \text{string}$

$\underline{S \vdash e_2 : \text{string}}$

$S \vdash e_1 \mid e_2 : \text{string}$

integer addition:

$S \vdash e_1 : \text{integer}$

$\underline{S \vdash e_2 : \text{integer}}$

$S \vdash e_1 + e_2 : \text{integer}$

assignment:

$S \vdash e_1 : T$

$S \vdash e_2 : T$

$\underline{T \text{ is a primitive type}}$

$S \vdash e_1 = e_2 : T$

comparison:

$S \vdash e_1 : T$

$S \vdash e_2 : T$

$\underline{T \text{ is a primitive type}}$

$S \vdash e_1 == e_2 : \text{Cadence}$

$S \vdash e_1 : T$

$S \vdash e_2 : T$

$\underline{T \text{ is a primitive type}}$

$S \vdash e_1 \mathrel{!=} e_2 : \text{Cadence}$

$S \vdash e_1 : T$

$S \vdash e_2 : T$

$\underline{T \text{ is a primitive type}}$

$S \vdash e_1 > e_2 : \text{Cadence}$

$S \vdash e_1 : T$

$S \vdash e_2 : T$

$\underline{T \text{ is a primitive type}}$

$S \vdash e_1 < e_2 : \text{Cadence}$

$S \vdash e_1 : T$

$S \vdash e_2 : T$

$\underline{T \text{ is a primitive type}}$

$S \vdash e_1 = e_2 : \text{Cadence}$

D♭ types are divided into two main categories: ***Value types*** and ***Reference types.*** This difference separates simple, efficient values from more complex structures that need memory and recursion. Value types are passed directly and are fast.

3.2. Value types (different from Scala and Erlang)

Float: a floating point numeric value

- Float: exampleFloat := 101.01;

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 := 10;

Cadence: a type that represents either true or false.

- Cadence: exampleCadence := harmonious;

Char: A single character.

- Char: exampleChar := ‘a’;

Unit: A type with a single value that represents “no meaningful return.” Equivalent to void in imperative languages.

- Unit: exampleUnit := ();

3.3. Reference types (differing from Scala and Erlang)

Repertoire: an immutable collection of data set in length.

- Repertoire(Char): exampleList := [1,2,3,4,5];

String: an array of characters.

- String: exampleString := “Hello”;

4.Example Programs

4.1.Caesar Cipher encrypt

```
-compose(CaesarCipherEncrypt);
-publish([main/0]);

#b Encrypt a string using Caesar cipher b#
notate caesarEncrypt(text: String, shift: Int) : String ||
  alphabet = explode("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
  rotate = map(c -> alphabet[((ord(c) - 65 + shift) mod 26)], explode(text));
  implode(rotate);
||

notate main() ||
  message: String = "HELLO";
  shift: Int = 3;
  encrypted: String = caesarEncrypt(message, shift);
  play("Encrypted: " | encrypted);
||
```

4.2.Caesar Cipher decrypt

```
-compose(CaesarCipherDecrypt);
-publish([main/0]);

#b Decrypt using Caesar cipher b#
notate caesarDecrypt(text: String, shift: Int) : String ||
  caesarEncrypt(text, 26 - (shift mod 26));
||

notate main() ||
  secret: String = "KHOOR";
  shift: Int = 3;
  original: String = caesarDecrypt(secret, shift);
  play("Decrypted: " | original);
||
```

4.3. Factorial

```
-compose(Factorial);
-publish([main/0]);

rec def factorial(n: Int) : Int ||
  solo(n == 0): ||
    1;
  || backup: ||
    n * factorial(n - 1);
  ||
||

notate main() ||
  result: Int = factorial(5);
  play("Factorial: " | result);
||
```

4.4. Quicksort

```
-compose(Quicksort);
-publish([main/0]);

notate quicksort(lst: Repertoire(Int)) : Repertoire(Int) ||
  solo(lst == []): ||
    [];
  || backup: ||
    pivot = head(lst);
    tail = tail(lst);
    left = filter(x -> x < pivot, tail);
    right = filter(x -> x >= pivot, tail);
    quicksort(left) ++ [pivot] ++ quicksort(right);
  ||
||

notate main() ||
  nums: Repertoire(Int) = [5, 2, 8, 3];
  sorted: Repertoire(Int) = quicksort(nums);
  play("Sorted: " | sorted);
||
```

4.5.Binary Search Tree

```
-compose(BSTree);
-publish([main/0]);

#b Each tree node is [value, leftSubtree, rightSubtree]
   An empty tree is simply: []
b#

notate insert(tree: Any, value: Int) : Any ||
  solo(tree == []): ||
    [value, [], []];
  || backup: ||
    root = tree[0];
    left = tree[1];
    right = tree[2];

    solo(value < root): ||
      [root, insert(left, value), right];
    || backup: ||
      [root, left, insert(right, value)];
    ||
  ||
||

notate inOrder(tree: Any) : Repertoire(Int) ||
  solo(tree == []): ||
    [];
  || backup: ||
    root = tree[0];
    left = tree[1];
    right = tree[2];

    inOrder(left) ++ [root] ++ inOrder(right);
  ||
||

notate main() ||
  tree = [];
  tree = insert(tree, 5);
  tree = insert(tree, 3);
  tree = insert(tree, 7);
  tree = insert(tree, 4);

  result: Repertoire(Int) = inOrder(tree);
  play("In-order: " | result);
||
```

4.6.Stack

```
-compose(Stack);
-publish([main/0]);

#b stack is a Repertoire(Int), LIFO b#

notate push(stack: repertoire(Int), value: Int) : repertoire(Int) ||
  [value] ++ stack;
||

# returns [top, rest] as a repertoire
notate pop(stack: repertoire(Int)) : repertoire(Any) ||
  solo(stack != []): ||
    [stack[0], stack[1:]];
  || backup: ||
    ["empty", []];  # return string flag if empty
  ||
||

notate main() ||
  s: Repertoire(Int) = [];
  s = push(s, 1);
  s = push(s, 2);
  s = push(s, 3);

  popped = pop(s);
  top = popped[0];
  rest = popped[1];

  play("Top: " | top);
||
```