CMPT 331 - Spring 2025 - Dr. Labouseur

INDIANA

A Legendary Language Based on the Epic Adventurer

Mitchell Levy

Mitchell.Levy1@Marist.edu

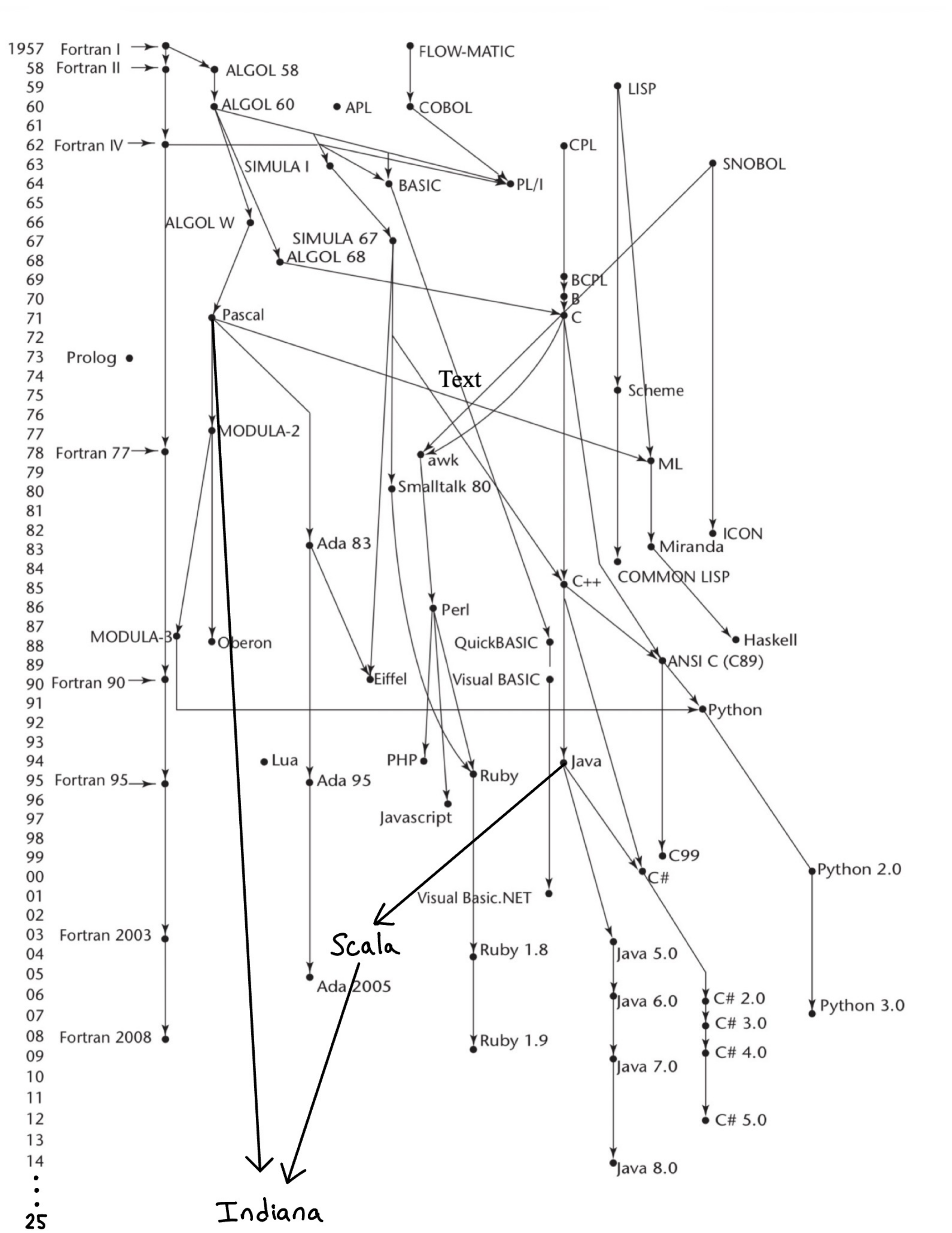
May 8, 2025

1 Introduction

Indiana is a modern, object-oriented, strongly typed programming language that supports both functional and imperative paradigms. The foundation of the language is Scala, but it also draws inspiration from Pascal. Uniquely, Indiana is themed around Indiana Jones and his quest for fortune and glory. The primary goal of Indiana is to make debugging as easy and intuitive as possible, while also embracing elegant syntax and thematic consistency. However, to fully embrace the theme, I will naturally have to sacrifice some readability and writability. This language is far from perfect, as, like Indiana Jones, "I'm making this up as I go." Compared to Scala, Indiana introduces the following changes:

1. Single-line comments begin with the keyword `NOTE`, while multi-line comments start with `JOURNAL` and end with `CLOSE`.
2. Recursive functions must be declared with the `rec` keyword to make recursion unmistakably clear to readers—an idea inspired by F#.
3. Type inference is unsupported in Indiana. All variables must be declared with an explicit type.
4. Semicolons are required at the end of every executable statement, just like in Java.
5. Several keywords are renamed to match the adventurous spirit of Indiana Jones, enhancing the language's identity and making it more fun to read and write.

1.1 Genealogy



1.2 Hello World

```
Artifact HelloWorld {
    adventure(): void = {
        reveal("Hello, archaeologist!");
    }
}
```

1.3 Program Structure

The key organizational concepts in Indiana are as follows:

1. Every program starts with an Artifact, which is the Indiana equivalent of a Scala class. All code must be placed within the braces of this Artifact. This enhances encapsulation and improves readability.

2. The `main` keyword is replaced with adventure. This serves as the entry point of every program.

3. Functions in Indiana are called `tools`, and their syntax closely follows Scala's style rather than Pascal's. A `tool` is public by default.

4. Each `tool` includes a `satchel` section, where all variables must be declared before any code is executed. This follows Pascal's structure and helps prevent hard-to-find bugs. The `adventure` (main) function does not require a `satchel`, similar to Pascal's approach.

5. Every file must be named according to its main Artifact, ending in .indy (e.g., Explorer.indy).

6. Indiana imports museums instead of packages—because museums are where priceless archaeological knowledge and tools are stored.

1.3.1 Example Program:

```
Artifact Crusader {

    NOTE Entry point of the program (main)
    tool adventure(): void = {
        val Person indy = new Person("Indiana", treasure);
        val Boolean[] cups = [trap, trap, treasure, trap];
        indy.choose(cups);
    }

    NOTE A tool (function) that allows a Person to choose a grail
    tool choose(self: Person, options: Boolean[]): void = {
        satchel {
            val numeral pick = 2;
        }

        if (options[pick] == treasure) {
            reveal("You chose... wisely.");
        } else {
            reveal("You chose... poorly.");
            self.alive = trap;
        }
    }

    NOTE Define a Person Artifact (like a class)
    Artifact Person {

        satchel {
            val Snake name;
            var Boolean alive;
        }

        NOTE Constructor for Person
        tool constructor(name: Snake, alive: Boolean): void = {
            satchel { }
            this.name = name;
            this.alive = alive;
        }
    }
}
```

This program simulates an Indiana Jones-like scenario where a character, represented by the Person Artifact, is tasked with choosing a cup from an array of grails. If the Indy selects the correct grail, they are deemed to have chosen wisely, but if they choose poorly, they "perish," with their status updated accordingly. The program demonstrates the use of objects, case statements, and basic logic flow.

1.4 Types and Variables

There are two kinds of types in Indiana: 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

By default, variables, functions (tools), and objects (Artifacts) are public, meaning they are accessible from anywhere. Indy needs all the help he can get to survive his dangerous adventures. However, the keyword `hidden` can be used to make a member private, restricting access to within the same Artifact or function where it is defined.

1.6 Statements Differing from Scala and Pascal

Statement	**Example**

Expression Statement	`Artifact expressions {`
	    `tool adventure(): void = {`
	        `val numeral a = 10;`
	        `val Snake message = "Uh Oh!";`
	        `var numeral b = 7;`
	        `var Boolean c = treasure;`
	        `val glyph d = 'j;`
	    `}`
	`}`

If Statement	`tool adventure(): void = {`
	    `reveal("Welcome to the Last Crusade!");`
	    `checkTreasure();`
	`}`
	
	`tool checkTreasure(): void = {`
	    `satchel {`
	        `val Boolean found = treasure;`
	    `}`
	    `if (found == treasure) {`
	        `reveal("You have found the treasure!");`
	    `} else {`
	        `reveal("It was a trap!");`
	    `}`
	`}`

Statement	Example
Recursion, Comments and Concatenation	

```
Artifact TombEscape {
    tool adventure(): void = {
        val numeral trapsRemaining = 5;
        disarm(trapsRemaining);
    }

    NOTE Recursive tool to disarm traps one by one
    tool rec disarm(n: numeral): void = {
        if (n == 0) {
            reveal("All traps disarmed. The idol is yours,
            ↪  Indy!");
        } else {
            reveal("Disarmed trap number " @ n @ "...");
            disarm(n - 1);
        }
    }
}
```

2 Lexical Structure

2.1 Programs

An Indiana 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 Indiana programming language where it differs from Scala and Pascal (The core inspiration for Indiana).

2.2.1 Lexical grammar (tokens) where different from Scala and Pascal

$< ArtifactDeclaration >$	$\rightarrow$ Artifact $< identifier >$
$< def >$	$\rightarrow$ tool
$< MathematicalOperators >$	$\rightarrow +\|-\|*\|/\|\%$
$< ComparisonOperators >$	$\rightarrow == \|!= \| < \| > \| <= \| >=$
$< SingleLineComment >$	$\rightarrow$ NOTE
$< MultilineComment >$	$\rightarrow$ JOURNAL . . . CLOSE
$< concatenation >$	$\rightarrow$ @
$< int, float, double >$	$\rightarrow$ *numeral*
$< BooleanLiteral >$	$\rightarrow$ treasure|trap
$< character >$	$\rightarrow$ glyph
$< String >$	$\rightarrow$ Snake

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

The grammar should only different in 3 distinct places:

1. When Declaring a tool/function, you must have a separate `satchel` for variables to be

declared at the start:

$$\begin{aligned}
<\textit{tooldeclaration}> &\rightarrow [<\textit{accmod}>]\text{"tool"} <\textit{id}> (<\textit{parameters}>) :<\textit{type}><\textit{block}> \\
<\textit{block}> &\rightarrow \{<\textit{satchel}><\textit{statement}> *\} \\
<\textit{satchel}> &\rightarrow \text{"satchel"}\{<\textit{variabledeclaration}> *\}
\end{aligned}$$

2. `Rec` must be in recursive functions:

$$<\textit{tooldeclaration}> \rightarrow [<\textit{accmod}>][\text{"rec"}]\text{"tool"} <\textit{id}> (<\textit{parameters}>) :<\textit{type}><\textit{block}>$$

3. Variable type must be specified:

$$<\textit{variabledeclaration}> \rightarrow [<\textit{var/val}>] <\textit{type}><\textit{id}> [\text{" : "} <\textit{type}>]\text{" = "} <\textit{expression}>$$

2.3 Lexical Analysis

2.3.1 Comments

Indiana supports two types of comments: single-line and multi-line (delimited) comments. Single-line comments begin with the keyword NOTE and continue to the end of the line—perfect for quick reminders Indy jots down to navigate treacherous traps. Delimited comments begin with JOURNAL and end with CLOSE, allowing for multi-line reflections essential to more elaborate quests, like the search for the Ark of the Covenant. 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` *(Used to name Artifacts and tools)*
- `keyword` *(Powerful and unique words)*
- `numeral-literal` *(Whole numbers/integers/floats/doubles)*
- `glyph-literal` *(Char)*
- `bool-literals` *(true/false)*
- `Snake-literal` *(Strings)*
- `operator-or-punctuator`

2.4.1 Keywords different from Scala or Pascal

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

Indiana Word	**Replaces / Represents**
`Artifact`	`class`
`tool`	`function / method`
`satchel`	Start-of-function variable declarations
`rec`	Explicit recursion marker
`treasure`	`true`
`trap`	`false`
`reveal`	`print` / `System.out.println`
`NOTE`	Single-line `// comment`
`JOURNAL ... CLOSE`	Multi-line `/* comment */`
`museum`	`package`
`adventure`	`main` function
`Snake`	`String`
`numeral`	`int/float/double`
`glyph`	`character`

3 Type System

3.1 Type Rules

The type rules for Indiana are as follows:

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 = e_2 : T} \qquad \frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 == e_2 : \text{boolean}}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 > e_2 : \text{boolean}} \qquad \frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 \mathrel{!=} e_2 : \text{boolean}}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 < e_2 : \text{boolean}} \qquad \frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 + e_2 : T}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 - e_2 : T} \qquad \frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 * e_2 : T}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 \mathbin{/} e_2 : T} \qquad \frac{\begin{array}{l} S \vdash e_1 : \text{Snake} \\ S \vdash e_2 : \text{Snake} \end{array}}{S \vdash e_1 \mathbin{@} e_2 : \text{Snake}}$$

This language is strongly typed with static, early binding to promote safety and efficiency. By enforcing type constraints at compile time, it helps catch errors early in development. Early binding ensures that function and variable references are resolved before runtime, improving performance and reducing ambiguity.

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

3.2 Value types (different from Scala and Pascal)

- **treasure** (Boolean): Represents the `true` value.
- **trap** (Boolean): Represents the `false` value.
- **numeral** (Int/Float/Double): A unified type for all numeric values, designed to seamlessly handle both integers and floating-point numbers. It simplifies arithmetic operations by automatically managing conversions, making programming more intuitive. For example, `1 + 3.14` becomes `1.0 + 3.14`. A special shout-out to Andrew Hatch's liteRoast project for inspiring this clever idea!
- **glyph** (Character): Represents a single character.

3.3 Reference types (different from Scala and Pascal)

- **Snake** (String): An array of characters.

- **Artifact** (class): Used for making an object/class.
- **museum** (package): Same thing as a package in Java.

4 Example Programs

4.1 Caesar Cipher Encrypt and Decrypt

```
Artifact Cipher {
    tool encrypt(input: Snake, shift: numeral): Snake = {
        satchel {
            var Snake newLetter = "";
            var Snake upperI = input.toUpperCase();
            var numeral fshift = shift % 26;
            var Snake result = "";
        }

        NOTE Format the shift
        if (fshift < 0) {
            fshift = (shift % 26 + 26) % 26;
        }

        if (input.isEmpty()) {
            return "";
        } else {
            var numeral firstGlyph = upperI.glyphAt(0).toNumeral();
            if (firstGlyph >= 'A'.toNumeral() && firstGlyph <= 'Z'.toNumeral()) {
                result = (((firstGlyph - 'A'.toNumeral() + fshift + 26) % 26) +
                 ↪  'A'.toNumeral()).toGlyph().toSnake();
            } else {
                result = input.glyphAt(0).toSnake();
            }

            NOTE We use @ for concatenation, because it looks like Indy's whip!
            return result @ encrypt(input.tail(), shift);
        }
    }

    tool decrypt(input: Snake, shift: numeral): Snake = {
        satchel {}
        NOTE Decrypt is just encrypt with a negative shift
        return encrypt(input, -shift);
    }
}
```

4.2 Factorial

```
Artifact Recursion {
    tool rec factorial(n: numeral): numeral = {
        satchel {}
        NOTE Handle negative inputs
        if (n < 0) {
            reveal("Error: Cannot compute factorial of a negative number.");
            return -1;
        }

        NOTE Base case: 0! and 1! are both 1
        if (n == 0 || n == 1) {
            return 1;
        } else {
            return n * factorial(n - 1);  NOTE Recursive call
        }
    }

    tool adventure(): void = {
        val numeral number = 5;
        val numeral result = factorial(number);

        if (result != -1) {
            reveal("The factorial of " @ number @ " is " @ result);
        }
    }
}
```

4.3 Bubble Sort

```
Artifact SortTemple {
    tool bubbleSort(arr: numeral[]): void = {
        satchel {
            val numeral n = arr.length();
            var Boolean swapped = treasure;
        }

        while (swapped == treasure) {
            swapped = trap;
            for (i: numeral = 0; i < n - 1; i = i + 1) {
                if (arr[i] > arr[i + 1]) {
                    NOTE Swap the elements
                    var numeral temp = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = temp;
                    swapped = treasure;
                }
            }
            n = n - 1;
        }
    }

    tool adventure(): void = {
        var numeral[] relics = [5, 3, 8, 2, 1];

        bubbleSort(relics);
        for (i: numeral = 0; i < relics.length(); i = i + 1) {
            reveal("Relic " @ i @ ": " @ relics[i]);
    }
}
```

4.4 Fibonacci Sequence

```
Artifact Fibonacci {
  tool adventure(): void = {
    val numeral n = 10;  NOTE Calculate the first 10 Fibonacci numbers
    reveal("Fibonacci series up to " @ n @ ":");

    for (i: numeral = 0; i < n; i = i + 1) {
      reveal(fibonacci(i).toSnake());
    }
  }

  tool rec fibonacci(n: numeral): numeral = {
    satchel {}
    if (n <= 1) {
      return n;  NOTE If n is 0 or 1, return n
    } else {
      return fibonacci(n - 1) + fibonacci(n - 2);
    }
  }
}
```

4.5 Text Adventure Game

```
Artifact TempleTrivia {
  tool adventure(): void = {
    NOTE Welcome the user to the game
    reveal("Let's see if you have what it takes to claim the Ark of the Covenant!");

    val Snake[] treasureWords = ["T", "t", "treasure"];
    val Snake[] trapWords = ["F", "f", "trap"];

    var numeral score = 0;

    reveal("What is your name, adventurer?");
    val Snake userName = getInput();

    reveal("Welcome, " @ userName @ "! Your answers must be 'treasure' (true) or 'trap'
    ↪  (false).\n");

    NOTE Question 1
    reveal("Indiana Jones is a professor of anthropology.");
    if (isTrap(getInput(), trapWords)) {
      score = score + 1;
    }
    NOTE Question 2
    reveal("Indiana is named after his dog.");
    if (isTreasure(getInput(), treasureWords)) {
      score = score + 1;
    }
    NOTE Question 3
    reveal("Indy's greatest fear is snakes.");
    if (isTreasure(getInput(), treasureWords)) {
      score = score + 1;
    }
    NOTE Question 4
    reveal("Temple of Doom is a prequel to Raiders of the Lost Ark.");
    if (isTreasure(getInput(), treasureWords)) {
      score = score + 1;
    }
    NOTE Question 5
    reveal("Indy teaches at Marshall College.");
    if (isTreasure(getInput(), treasureWords)) {
      score = score + 1;
    }

    reveal("\nYou have completed the Temple of Trivia, " @ userName @ "!");
    reveal("You scored " @ score @ " out of 5!");
  }

  tool isTreasure(answer: Snake, treasureWords: Snake[]): fate = {
    satchel {}
    if (treasureWords.contains(answer)) {
      return treasure;
    } else {
      return trap;
    }
  }

  tool isTrap(answer: Snake, trapWords: Snake[]): fate = {
    satchel {}
    if (trapWords.contains(answer)) {
      return treasure;
    } else {
      return trap;
    }
  }
}
```