



# Language Design & Principles for Ye

Version 1.0

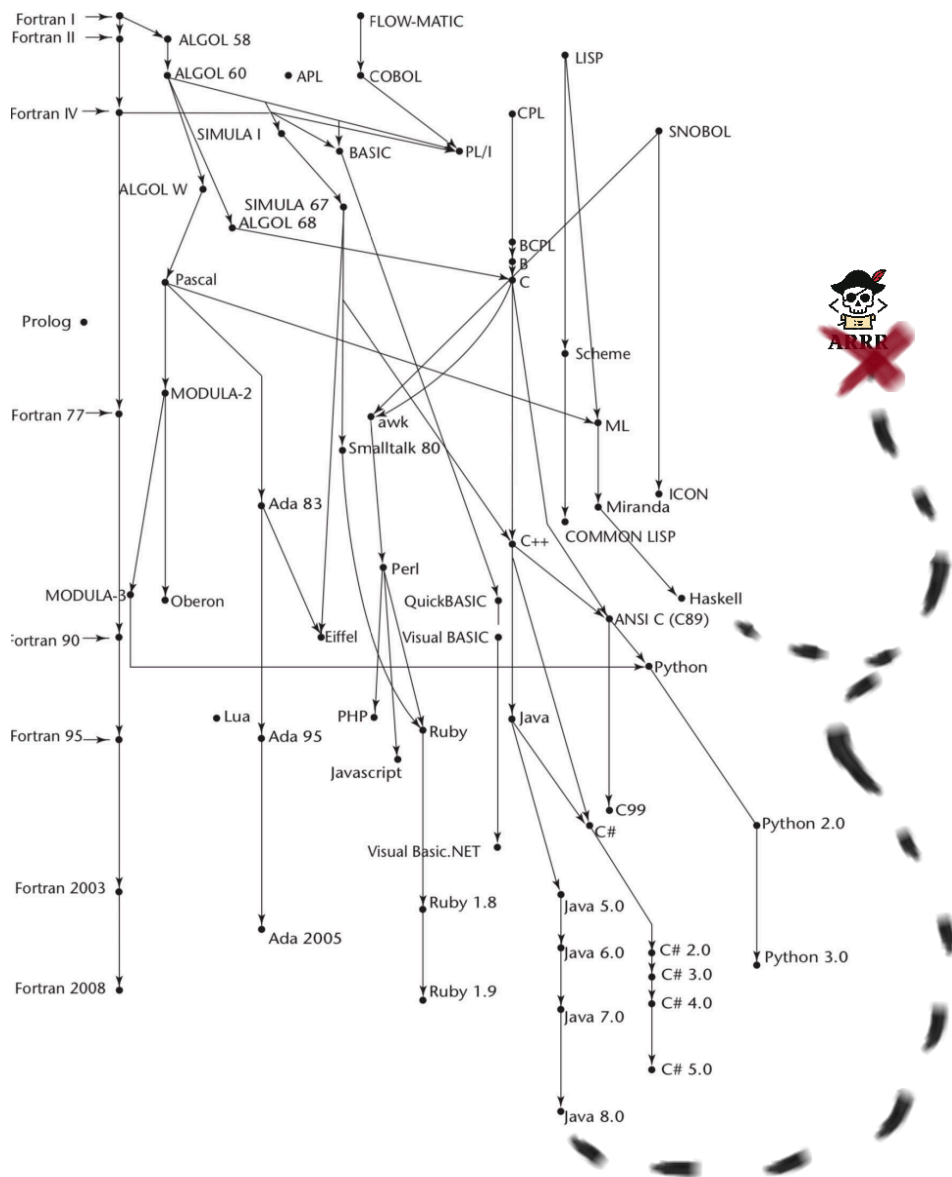
Designed by: Evan Brown

Final project for Theory of Programming Languages

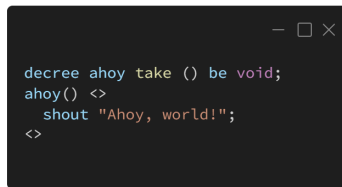
# 1. Introduction

Arrr (yes, like a pirate) is a strongly typed, safe by default, pirate-themed programming language intended to provide support to those who only speak using old pirate lingo and have needed a functional, object-oriented language. The language fuses the structural clarity of C++ and Java with the functional power of Haskell. Arrr emphasizes explicit typing, predeclared functions, expression-based control structures, and object-oriented modeling (hulls, cargo, decrees, shanties). Arrr supports pattern matching, sum types, and facilitates the use of both pure and impure functions.

## 1.1. Genealogy



## 1.2. Hello World



```

decrey ahoy take () be void;
ahoy() <>
  shout "Ahoy, world!";
<>

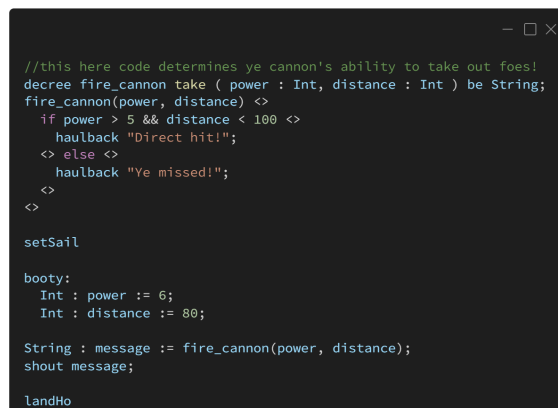
```

## 1.3. Program Structure

Arrr has a similar program structure to that of C++ and Java, and incorporates some Haskell-like formats in the way we can return values. Some key components of an Arrr program include:

- 1) In Arrr, all *decrey* and *shanty* functions must be declared before the setSail entry point. This mimics the structure of top-down languages like Java, and variable/function scope is easily predictable using this method.
- 2) Functions must also be predeclared with their input AND return types.
- 3) All code blocks are enclosed with “doubloons” (<>) to support effective readability.
- 4) Like many modern languages, lines of code are terminated with “;”. This allows the programmer to understand when a certain line of code is finished and multiline expressions are easily managed.
- 5) Variables are explicitly defined under the “booty” section, for all of our loot.
- 6) The main body of the program is located between “setSail” and “landHo”, to aid the sailor’s (programmer’s) journey through the Sea of Code.
- 7) Comments are made by using “//”, and multiline comments are made using “//>” paired with “<//”. Nested comments are allowed and explained in following sections.

Here is an example of a totally relevant problem in today’s world of warfare:



```

//this here code determines ye cannon's ability to take out foes!
decrey fire_cannon take ( power : Int, distance : Int ) be String;
fire_cannon(power, distance) <>
  if power > 5 && distance < 100 <>
    haulback "Direct hit!";
  <> else <>
    haulback "Ye missed!";
  <>
<>

setSail

booty:
  Int : power := 6;
  Int : distance := 80;

String : message := fire_cannon(power, distance);
shout message;

landHo

```

## 1.4. Types and Variables

Arrr is a statically typed language, where every variable and function declaration must be known at compile time. With that said, Arrr supports 5 main primitive value types:

- Int : integer values
- Bool : true or false
- Float : decimal values
- String : lists of characters
- Char : single characters

Furthermore, Arrr allows for a unique type declaration like Haskell, defined by “chest.” Chests can contain one or more user defined types.

Arrr also supports reference types (pointers). These types are defined with a scroll “~”, to signify the journey you may go on following a null abyss.

Variables are declared in the “booty” block as seen above for class level declaration. Limited scope variables are declared using “cargo.” See below for more info.

## 1.5. Visibility

In Arrr, not all treasure, or “booty”, is meant to be seen by other hulls. To manage this, Arrr implements visibility modifiers that can protect some loot if needed. In the booty heading, those variables remain public as they are always declared in the execution section of the program. In hulls, or objects, there are 3 modifiers that we use:

- commonfolk : anything can access
- private : only code inside the certain hull can access
- guarded : accessible from within a certain hull, or any hull that inherits this variable

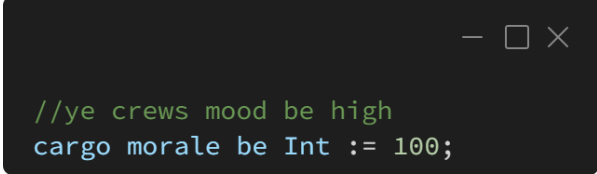
All three modifiers can be used on cargo (variables within a hull, private by default), decree and shanty (commonfolk by default), and hulls.

So, consider it like this. Each hull has its own cargo, and when sailing the seas we have an overview of the program's loot, or “booty.”

## 1.6. Unique Syntax and Statement Examples

In the examples I will list below, you will see several examples of variable declarations in hulls versus the program execution level. Also, Arrr allows for unique expression declarations using haulback in its structure (like return). This allows for a Haskell-like approach if needed.

SEE NEXT PAGE

Control Flow Statements	Example
Basic Expression	 <pre>//ye crews mood be high cargo morale be Int := 100;</pre>
If-Else (2 different ways)	 <pre>decree check_morale take () be String; check_morale() &lt;&gt;   if morale &gt;= 50 &lt;&gt;     haulback "The crew be hearty!";   &lt;&gt; else &lt;&gt;     haulback "They be grumblin'.";   &lt;&gt;</pre>  <pre>haulback if morale &gt;= 50 &lt;&gt;   "The crew be hearty!"; &lt;&gt; else &lt;&gt;   "They be grumblin'."; &lt;&gt;</pre>
For Loop (repeat)	 <pre>setSail  booty:   Int : strokes := 0;  repeat 3 times &lt;&gt;   shout "Heave!";   strokes := strokes + 1;   shout "Ho!" &lt;&gt;  landHo</pre>
While Loop (whilst)	 <pre>hull Cask be &lt;&gt;   cargo rum_supply be Int := 3;  commonfolk decree pour_rum take () be String; pour_rum() &lt;&gt;   String : log := "";    whilst rum_supply &gt; 0 &lt;&gt;     log := log ++ "Another one Matey!";     rum_supply := rum_supply - 1;   &lt;&gt;    haulback log;   &lt;&gt; &lt;&gt;</pre>

## 2. Lexical Structure

### 2.1. Programs

An Arrr program consists of one or more *source files*, defined by their “manifest <name>”, represented using lists of Unicode characters. Arrr is executed between the reserved entry point defined by “setSail” and “landHo,” and uses modern program approaches in having reserved hulls for specified classes and programs. Arrr is compiled in the following way:

- 1) Transformation - converting a file, or “manifest”, to Unicode so Arrr can be read regardless of its original encoding.
- 2) Lexical Analysis - the transformed character stream from step 1 is scanned and divided into a stream of tokens.
- 3) Syntactic Analysis - converts the stream of token from step 2 into executable code.

### 2.2. Grammars

Arrr uses many of the same tokens as C++, Java, and some Haskell identifiers. Below is a list of similar tokens, and altered names, as well as additions that I have added to the language in EBNF format.

#### 2.2.1. Lexical Grammar

<assignmentOperator>	::=	:=
<booleanOperator>	::=	==   !=   <=   >=   <   >
<endOfLineCharacter>	::=	;
<print>	::=	shout
<concatOperator>	::=	++
<blockDelimiters>	::=	< >
<returnKeyword>	::=	haulback
<comment>	::=	//   //>
<identifier>	::=	[a-zA-Z_][a-zA-Z0-9_]*
<variant>	::=	[A-Z][a-zA-Z0-9_]*
<intLiteral>	::=	[0-9]+
<floatLiteral>	::=	[0-9]+ "." [0-9]+
<charLiteral>	::=	'any single char'
<stringLiteral>	::=	"any sequence of char"
<arrayLiteral>	::=	[ <expression> { , <expression> } ]

#### 2.2.2. Syntactic Grammar

<program> ::= manifest <identifier> ; { <declaration> } setSail <statement>\* landHo

<block>	::= "<" <statement>* ">"
<statementEnd>	::= ";
<functionDecl>	::= decree <identifier> take (<params>) be <type> ;   shanty <identifier> take (<params>) ;
<entryPoint>	::= setSail ... landHo
<variableDecl>	::= <type> : <identifier> := <expression> ;
<fieldDecl>	::= <visibility>? cargo <identifier> be <type> := <expression> ;
<variableSection>	::= booty: { <variableDecl>* }
<returnStmt>	::= haulback <expression> ;
<matchExpr>	::= spyglass <expression> see "<" { <variant> -> <expression> ; } ">"
<repeatLoop>	::= repeat <expression> times "<" <statement>* ">"
<whilstLoop>	::= whilst <expression> "<" <statement>* ">"
<typeRef>	::= ~<type>
<classDecl>	::= hull <identifier> be "<" { <fieldDecl> <functionDecl>* } ">"
<enumDecl>	::= chest <identifier> = <variant> {   <variant> } ;

## 2.3. Lexical Analysis

### 2.3.1. Comments

In Arrr, we are allowed to type comments. Single-line comments are described using “//” which is common in many modern languages. Multi-line comments are described using “/\*>” for the beginning and “<\*/” for the end. Arrr does not allow for nested comments.

## 2.4. Tokens

Arrr includes several of the same types of tokens as C++, Java, and Haskell. Although many of the tokens remain the same, Arrr has introduced many new keywords to fit the theme and allow easier readability of the code. The tokens in Arrr include:

Tokens:

- identifier
- keyword
- integer-literal
- float-literal
- character-literal
- string-literal
- array-literal
- operator-or-punctuator

### 2.4.1 Keyword Differences

There are many new keywords in Arrr, as well as many that have been removed. Keywords are predefined and cannot be used as cargo or hull names.

New/Changed Keywords: manifest, setSail, landHo, booty, cargo, hull, decree, shanty, haulback, spyglass, chest, whilst, repeat, shout

Removed Keywords: do, goto, switch, class, function, def, internal

## 3. Types

### 3.1. Type Rules

#### Assignment

$$S \vdash e1 : T$$

$$S \vdash e2 : T$$

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


---


$$S \vdash e1 := e2; : T$$

#### Operations

$$S \vdash e1 : T$$

$$S \vdash e2 : T$$

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


---


$$S \vdash e1 (+, -, /, *, \%) e2; : T$$

#### Comparisons

$$S \vdash e1 : T$$

$$S \vdash e2 : T$$

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


---


$$S \vdash e1 == e2 : \text{Boolean}$$

$$S \vdash e1 : T$$

$$S \vdash e2 : T$$

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


---


$$S \vdash e1 != e2 : \text{Boolean}$$

$$S \vdash e1 : T$$



$$\frac{S \vdash e2 : T \quad T \text{ is a primitive type}}{S \vdash e1 > e2 : \text{Boolean}}$$

$$\frac{S \vdash e1 : T \quad S \vdash e2 : T \quad T \text{ is a primitive type}}{S \vdash e1 < e2 : \text{Boolean}}$$

$$\frac{\text{haulback} \quad S \vdash e : T}{S \vdash \text{haulback } e ; : T}$$

Arrr types are categorized into two main sections: Value types & Reference types.

### 3.2. Value Types

There are two kinds of value types in Arrr: primitive & composite.

- **Primitive:**
  - Int – whole numbers
  - Float – decimal numbers
  - Bool – true or false
  - Char – single character
  - String – text data (immutable)
- **Composite:**
  - List<T> – a sequence of values (e.g., [1, 2, 3])
  - Array<T> – indexable collection (e.g., [1, 2, 3])
  - chest types – user-defined sum/enum types (like in Haskell)
    - Example: chest Loot = Gold | Rum | Maps;

### 3.3. Reference Types

In Arrr, we have reference types for arrays and lists if needed. By default however, arrays will be passed by value to keep the data immutable for simple calculations. For more advanced manipulation of data, we can use reference versions of these types (using “~”). These reference types include: (See next page)

- ~T – reference to any type T
- ~List<T> – reference to a list

- ~Array<T> – reference to an array
- ~String – reference to a string

This allows the programmer to experience the ease of Java-like data sets on the surface level, but if dealing with graphs that need node classes, Arrr allows for those types to be passed by reference.

## 4. Example Programs

For some of these programs, I am using a utility function “length” that will be added at the end of these examples for more clarity. All programs would have been named using a manifest, and run in the setSail-landHo block.

### 4.1. Caesar Cypher Encrypt Function

```

decre caesar_encrypt take (msg : String, shift : Int) be String;
caesar_encrypt(msg, shift) <>
  String : result := "";
  repeat length(msg) times take (i : Int) <>
    Char : c := msg[i];
    Char : shifted := if c >= 'A' && c <= 'Z' <>
      ((c - 'A' + shift) % 26) + 'A';
    <> else if c >= 'a' && c <= 'z' <>
      ((c - 'a' + shift) % 26) + 'a';
    <> else <>
      c;
    <>
    result := result ++ shifted;
  <>
  haulback result;
<>

```

## 4.2. Caesar Cypher Decrypt Function

```

decrey caesar_decrypt take (msg : String, shift : Int) be String;
caesar_decrypt(msg, shift) <>
    haulback caesar_encrypt(msg, -shift);
<>

```

## 4.3. Factorial

```

decrey factorial take (n : Int) be Int;
factorial(n) <>
    if n <= 1 <>
        haulback 1;
    <> else <>
        haulback n * factorial(n - 1);
    <>
<>

```

## 4.4. BubbleSort

```

decrey swap_sort take (arr : ~Array<Int>) be void;
swap_sort(arr) <>
    Int : n := length(arr);
    repeat n times take (i : Int) <>
        repeat n - 1 times take (j : Int) <>
            if arr[j] > arr[j + 1] <>
                Int : temp := arr[j];
                arr[j] := arr[j + 1];
                arr[j + 1] := temp;
            <>
        <>
    <>
<>

```

## 4.5. Pattern Matching with Chest Types

```
chest Weather = Sunny | Cloudy | Rainy;

decree outfit_recommendation take (w : Weather) be String;
outfit_recommendation(w) <>
  spyglass w see <>
    Sunny -> haulback "Hoist yer sails and wear yer shades!";
    Cloudy -> haulback "Keep a cloak handy, the skies be brooding.";
    Rainy -> haulback "Don yer raincoat, sailor, a storm brews ahead!";
  <>
<>
```

## 4.6. Utility Library

### 4.6.1. Length()

```
decree length take (lst : List<T>) be Int;
length(lst) <>
  if lst == [] <>
    haulback 0;
  <> else <>
    haulback 1 + length(lst[1:]);
  <>
<>
```