Alan++

Language Design and Example Programs

Version 0.0.7

This document is part example of, and part instructions for, your final project [*150 points*] in Theory of Programming Languages. Read it fully and thoroughly before you begin.

Hints and Requirements:

- You are free to emulate my template style here, but you do not have to.
- Make the example code great. Syntax-highlight your code to make it easier to read and more impressive to look at.
- You **do** have to include all the sections here, including all heading and subheadings.
- I expect all of your content to be original. That means replacing the code fragments I've used here as examples with your own.
- Be sure that you address and provide content for all the italicized instructions and all sections.

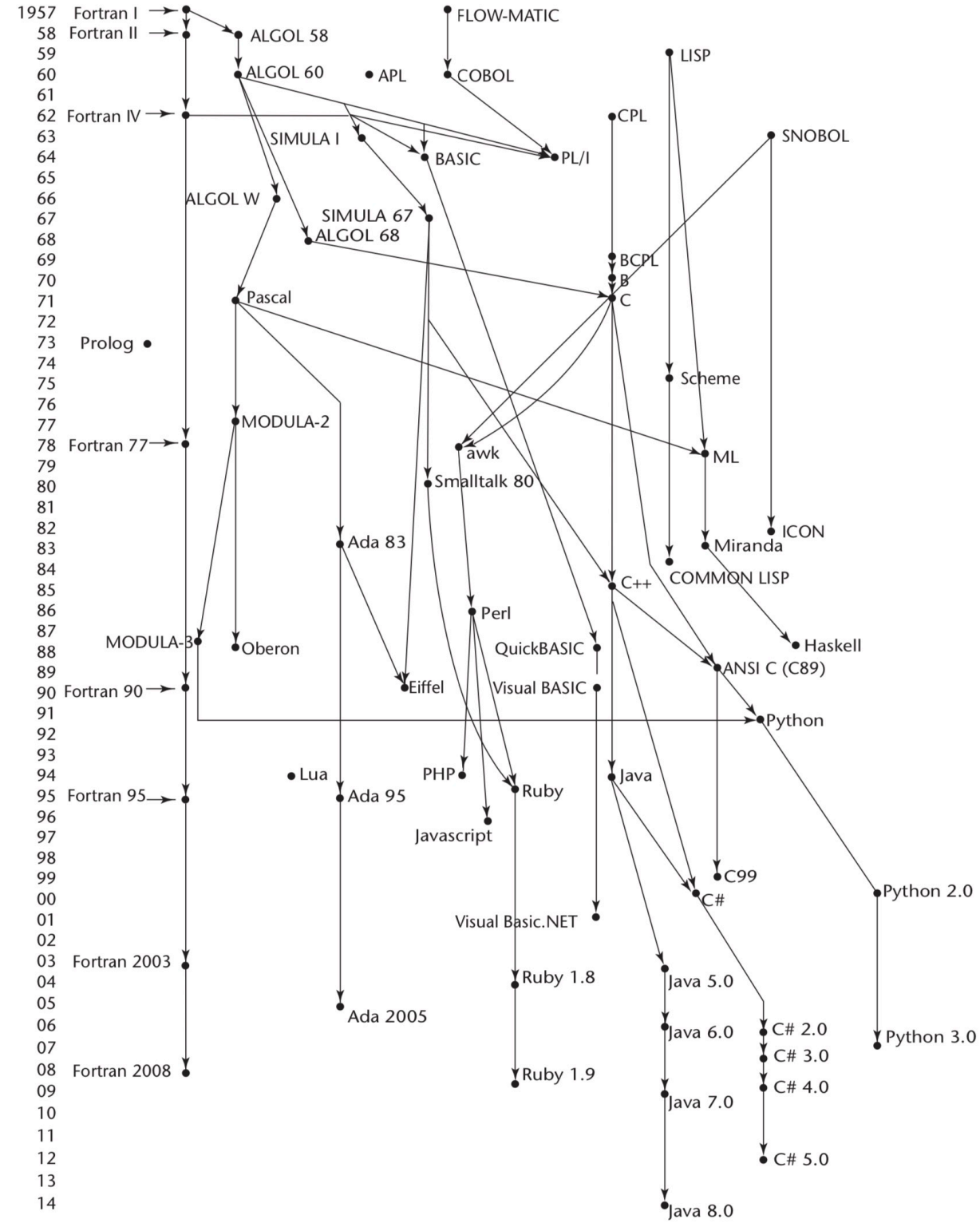
1. Introduction

Alan++ (pronounced “Alan plus plus”) is a simple, modern, object-oriented, and (strongly) type-safe programming language. Based on YYY and YYZ, but differing in the following ways:

1.
2.
3.

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

You are morally obligated to write the "Hello World" program in your language here.

1.3.Program structure

The key organizational concepts in Alan++ are as follows:

1.
2.
3.

This example (*which should be replaced by an original one of your own*)

```
namespace Acme.Collections
begin
   public class Stack
   begin
      public Entry top;

      public void Push(object data)
      begin
         top := new Entry(top, data);
      end;

      public object Pop()
      begin
         if (top = null) then
            throw new InvalidOperationException();
         else
            object result := top.data;
            top := top.next;
            return result;
         end if
      end Pop


      class Entry
      begin
         public Entry next;
         public object data;

         public Entry(Entry next, object data)

         begin
            this.next := next;
            this.data := data;
         end;
      end class Entry;
   end class Stack;
end namespace Acme.Collections;
```

declares a class named `Stack` in a namespace called `Acme.Collections`. The fully qualified name of this class is `Acme.Collections.Stack`. The class contains several members: a field named `top`, two methods named `Push` and `Pop`, and a nested class named `Entry`. The `Entry` class further contains three members: a field named `next`, a field named `data`, and a constructor.

1.4.Types and Variables

There are two kinds of types in Alan++: ***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

Public, Private, Protected, Internal… more or less? How does your language handle this?

1.6.Statements Differing from YYY and YYZ

Statement	**Example**

Expression statement	`static void Main()`
	`begin`
	`  int i;`
	`  i := 123;`
	`  Put(i);`
	`  inc(i);`
	`  Put(i);`
	`end Main`

`if` statement	`static void Main(string[] args)`
	`begin`
	`  if (args.Length = 0)`
	`  Put("No arguments");`
	`  else`
	`  Put("One or more arguments");`
	`  end if`
	`end Main`

	Keep adding your examples. . . .

2. Lexical structure

2.1.Programs

A Alan++ ***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.

If there's anything different about your language in the regard, this is the place for it.

2.2.Grammars

This specification presents the syntax of the Alan++ programming language where it differs from YYY and YYZ.

2.2.1.Lexical grammar (tokens) where different from YYY and YYZ

Write your Regular Expressions for tokens here.

2.2.2.Syntactic ("parse") grammar where different from YYY and YYZ

Write your BNF grammar productions here.

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 characters `//` and extend to the end of the source line. ***Delimited comments*** start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines. Comments do not nest. (*Unless they do in your grammar. Be different. Specify something new and original.)*

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

List all valid tokens in your language.

2.4.1.Keywords different from YYY or YYZ

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:

```
begin       end         inc
```

Removed keywords:

```
do          goto     internal
```

3. Type System

Alan++ uses a **strong static** type system. (*This is nice, but feel free to use weak systems that are static or dynamic. Be sure to explain their details and document them with Type Inference diagrams.*) 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 Alan++ are as follows:

You have to specify type rules regardless of whether you are using a strong or weak type system. In fact, your type rules should explicitly reflect this choice. Write type inference rules in the style found in the "Scope and Type" slide deck on our web site, an example of which is given below. Write your own type rules; do not just include these verbatim. Be sure your type rules match the operators you chose for your language.

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

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

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

Alan++ types are divided into two main categories: ***Value types*** and ***Reference types***. (*Maybe you have some other thoughts here. I hope so.*)

3.2.Value types (different from YYY and YYZ)

Examples

3.3.Reference types (differing from YYY and YYZ)

Examples

4. Example Programs

Illustrate your new language with six (6) example programs that demonstrate its use; especially what's new and improved over current languages as well as YYY and YYZ, on which you based your design. Please include in your examples Caesar cipher encryption and decryption programs like those of our earlier class projects.

You must write example programs for the following:

1. Caesar Cipher encrypt
2. Caesar Cipher decrypt
3. Factorial
4. Sort (pick one: swap sort, bubble sort, merge sort, quicksort, or another one)

Write two more programs. Here are some ideas, but feel free to write whatever you like and think will be fun.

- More sorts
- Lambda functions (if possible in your language)
- Pattern matching
- Stack
- Queue
- Binary Tree
- Binary Search Tree
- List (single or doubly linked, circular)
- Text adventure game