This document is part example of, and part instructions for, your final project in Theory of Programming Languages [150 points]. 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 “answer” and provide content for all the italized instructions.
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? Use a diagram like the one in our class two notes (shown below). Even better, find a higher resolution version of this one and modify it. Be sure 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)

```plaintext
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.

1.5 Statements Differing from YYY and YYZ

<table>
<thead>
<tr>
<th>Statement</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression statement</td>
<td>static void Main() &lt;br&gt;begin &lt;br&gt;    int i; &lt;br&gt;    i := 123; &lt;br&gt;    Put(i); &lt;br&gt;    inc(i); &lt;br&gt;    Put(i); &lt;br&gt;end Main</td>
</tr>
<tr>
<td>if statement</td>
<td>static void Main(string[] args) &lt;br&gt;begin &lt;br&gt;    if (args.Length = 0) &lt;br&gt;    Put(&quot;No arguments&quot;); &lt;br&gt;    else &lt;br&gt;    Put(&quot;One or more arguments&quot;); &lt;br&gt;end if &lt;br&gt;end Main</td>
</tr>
</tbody>
</table>

*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 where different from YYY and YYZ
Write your BNF grammar productions 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. Types

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

3.1 Value types (different from YYY and YYZ)

*Examples*

3.2 Reference types (differing from YYY and YYZ)

*Examples*
4. Example Programs

Illustrate your new language with six 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 this. Please include in your examples Caesar cipher encryption and decryption programs like those of our earlier 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.

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