

**SCHOOL OF ADVANCED TECHNOLOGY**

**ICT - Applications & Programming**

**Computer Engineering Technology – Computing Science**



A31

Model Definitions (RE/Automaton)

**Lab Professor / Lab Session:**

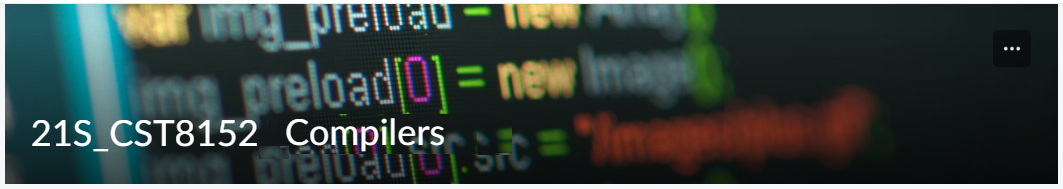
**[Pablo Sousa/ 311]**

**Team:**

**[Yen Huynh]** - Id: **[041 068 712] / [Mohammed Muhsin Shaik]** - Id: **[041 078 827]**

**Language Name [CoreX]**

***This template is suggested (not mandatory) to answer A31 Specification.***



**CoreX Language LANGUAGE SPECIFICATION**

**General View**

This document focusses on **CoreX** LS (Language Specification) that is based on combination between PLATYPUS language, originally created by Prof. Svillen Ranev for Algonquin College.

***Grammar, which knows how to control even kings . . .***

*—*Molière*, Les Femmes Savantes* (1672), Act II, scene vi

**Note 1**

Please change this template, replacing any “**CoreX LANGUAGE**” reference by your language name. Remember that this document is using the professor’s language and you need to adapt the BNF to your own language. This time, you just need to define the grammar, using **white boxes**.

**NOTE**: It is not necessary to solve problems such as **LR** (Left Recursion) and **LF** (Left Factoring) – shown in the **blue boxes**. You don’t need to define the FIRST set (that will be used later in the implementation).

A context-free grammar is used to define the lexical and syntactical parts of the **CoreX LANGUAGE** and the lexical and syntactic structure of a program.

1. **The CoreX LANGUAGE Lexical Specification**
   1. **White Space**

White spaceis defined as the ASCII space, horizontal and vertical tabs, and form feed characters, as well as line terminators. White space is discarded by the scanner.

**<white space>** → *one of* { SPACE, TAB, \t, \s or [:space], etc.}

* 1. **Comments**

**CoreX** LANGUAGE supports SLC and MLC all the text from the ASCII characters

**Single-line comment (SLC):** Open by # and close by moving to next line

**Multi-line comment (MLC):** Open by ## and close by ##

* 1. **Variable Identifiers**

The following variable identifier (VID) tokens are produced by the scanner: **VID\_T** will be our Variable Identifier for **String, Float, Int values.**

**<variable identifier>** → <VID\_T>

* 1. **Keywords**

The scanner produces a single token: **KW\_T**. The type of the keyword is defined by the attribute of the token (the index of the keywordTable[]). Remember that the list of keywords in **Sofia** language is given by:

**If, else, repeat, while, function, for, in, next, break, TRUE, FALSE, NULL, Inf, print**

* 1. **Integer Literals**

The scanner produces a single token: **INL\_T** with an integer value as an attribute.

**<integer\_literal>** → <INL\_T>

* 1. **Floating-point Literals**

**FPL\_T** token with a real decimal value as an attribute is produced by the scanner.

**<float\_literal>** → <FPL\_T>

* 1. **String Literals**

**STR\_T** token is produced by the scanner.

**<string\_literal>** → <STR\_T>

* 1. **Separators**

**<separator>** → *one of* {**( ), { }, ‘, “** }

Some different tokens are produced by the scanner - **LPR\_T**, **RPR\_T**, **LBR\_T**, **RBR\_T, SQ\_T, DQ\_T**.

* 1. **Operators**

A single token is produced by the scanner: **ART\_T**. The type of the operator is defined by the attribute of the token.

**<arithmetic operator>** → *one of* { +, -, \*, / }

A single token is produced by the scanner: **SC\_T**.

**<string concatenation operator>** → paste()

A single token is produced by the scanner: **LG\_T**. The type of the operator is defined by the attribute of the token.

**<logical operator>** → *one of* {**&&**, ||, !}

A single token is produced by the scanner: **AO\_T**.

**<assignment operator>** → <-, ->, =

1. **The CoreX LANGUAGE Syntactic Specification**
   1. **CoreX LANGUAGE Program**
      1. **Program**

**CoreX** LANGUAGE does not have any start program.

**<program>** ®

<VID\_T> <AO\_T> <KW\_T(function)> <LPR\_T> <optional\_parameters><RPR\_T>

<LBR\_T>

<body>

<RBR\_T>

function\_body ->

<optional\_package\_session> (Optional, depends on which logistics we are going to use)

<data\_declaration>

* + 1. **DATA**

The first part is the body\_structure:

**body\_structure** ® <optional\_package\_session> <data\_declaration>

The second part is for the package session:

<optional\_package\_session> -> <KW\_T(library)> <LPR\_T> <package\_name> <RPR\_T>

The third part (**data**) is the place we declare the variables:

**data\_declaration** ® **<**STR\_L> | <STR\_L & INT\_L> <AO\_T> <STR\_L> | <INT\_L> | <FPL\_T>

**Variable Lists**

The optional variable list declarations are used to define several datatype declarations:

**<variable\_list>** ® **<**variable> | <variable\_list, variable>

**<variable>** → <VID\_T>

**Variable Declarations**

**For ‘<- and =’, we use:**

**<VID\_T & AO\_T> ->** <INL\_T | SQ STR\_T SQ | DQ STR\_T DQ | FPL\_T>

**For ‘->‘arithmetic only:**

<INT\_T> | <SQ STR\_T SQ> | <DQ STR\_T DQ> | <FPL\_T> -> <**AO\_T & VID\_T>**

* + 1. **Declaration of Lists:**

The variables list declaration is defined here:

**<declaration> ->**

**<VID\_T> 🡨 <**digit> | <digits> | <quotation string quotation> | <float>

* + 1. **List of Variables:**

The list of variables is defined here:

**Integers:**

**<variable\_list>** ® **<**variable> | <variable\_list>, <variable>

* + 1. **CODE session:**

The second part (CODE) is the place we have statements:

**<code\_session>** ® <opt\_statements>

**Optional Statements:**

**<opt\_statements>** → <statements> | ϵ

* + 1. **Statements**

**<statements>** → <statement> | <statements> <statement>

* 1. **Statement**

**<statement>** → <assignment statement> | <selection statement> | <iteration statement>

| <input statement> | <output statement>

* + 1. **Assignment Statement**

**<assignment statement>** → <assignment expression>

* + 1. **Assignment Expression**

**<assignment expression>** → <variable> = <arithmetic expression>

* + 1. **Selection Statement (if statement)**

**<selection statement>** → **if** (<conditional expression>)

{ <opt\_statements> }

<optional else statement>

**<optional else statement>** → **else** { <opt\_statements> } ; | ϵ

* + 1. **Iteration Statement (the loop statement)**

**<iteration statement>** → **while** (<conditional expression>)

**repeat** {<statements>}

**for** (<conditional expression>)

* + 1. **Input Statement**

**<input statement>** → **readline** (<parameter>);

**scan** () <- read the data from a vector or list

**read.csv** ()

**read.table** ()

**file** ()

**Ex for files:**

con <- file("input.txt", "r")

lines <- readLines(con)

close(con)

print(lines)

**Variable List:**

**<variable list>** → variable\_identifier | variable\_list, variable\_identifier

**Variable Identifier:**

**<variable identifier>** →<variables> | <variable>

* + 1. **Output Statement**

**<output statement>** → **print** (<opt\_variable list>) | <variable\_list> , <variable>

* **PROBLEM DETECTED: Left factoring – SOLVING FOR YOU:**

**<output statement>** → **print** (<output statement Prime>);

**<output statement Prime>** → <opt\_variable list> | STR\_T

**Optional Variable List:**

**<opt\_variable list>** →<variable list> | ϵ

* 1. **Expressions**
     1. **Arithmetic Expression**

**<arithmetic expression>** → <unary arithmetic expression>

| <additive arithmetic expression> **|** <multiplicative arithmetic expression> | <primary arithmetic expression>

**Unary Arithmetic Expression:**

**<unary arithmetic expression>** → - <primary arithmetic expression>

| + <primary arithmetic expression>

**Additive Arithmetic Expression:**

**<additive arithmetic expression>** →

<First operand> + <Second operand>

| <First operand> - <Second operand>

**Multiplicative Arithmetic Expression:**

**<multiplicative arithmetic expression>** →

<First operand> \* <Second operand>

| <First operand> / <Second operand>

**Primary Arithmetic Expression:**

**<primary arithmetic expression>** → <INL\_T> & (<arithmetic expression>)

* + 1. **String Expression**

**<string expression>** → <SC\_T> **🡨 Using paste() function to concatenate string**

* + 1. **Conditional Expression**

**<conditional expression>** → <logical OR expression> | <logical AND expression> | <logical NOT expression>

**Logical OR Expression:**

**<logical OR expression>** →

<boolean\_value\_1> **|** <boolean\_value\_2> <- for element-wise

<boolean\_value\_1> **||** <boolean\_value\_2> <- for short-circuit

**Logical AND Expression:**

**<logical AND expression>** →

<boolean\_value\_1> **&** <boolean\_value\_2> <- for element-wise

<boolean\_value\_1> **&&** <boolean\_value\_2> <- for short-circuit

**Logical NOT Expression:**

**<logical NOT expression>** → **!** <variable>

* + 1. **Relational Expression**

**<relational expression>** →

<variable\_1><relational\_expression><variable\_2>

**Relational Arithmetic Expression with Integer (INL\_T), String (STR\_T), Float (FPL\_T):**

**<relational\_expression>** →

<variable\_1> == | != | > | >= | < | <= <variable\_2>

**Good luck with Assignment 3.1!**