Algonquin College Logo

# SCHOOL OF ADVANCED TECHNOLOGY

### ICT - Applications & Programming

### Computer Engineering Technology – Computing Science



A31

Language Grammar

Lab Professor / Lab Session:

[Lab Prof Name / Lab Session]

Team:

[Souber Daher] - Id: [041077888 Id] / [Johny Dawahiri] - Id: [04]

Language Name [Eazy]

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

Algonquin College Logo

A screenshot of a video game

Description automatically generated with medium confidence

# Eazy Language LANGUAGE SPECIFICATION

### General View

This document focusses on **Eazy** 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 “**Eazy 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**. It is not necessary to solve problems such as **LR** (Left Recursion) and **LF** (Left Factoring). 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 **Easzy LANGUAGE** and the lexical and syntactic structure of a program.

1. **The Eazy 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, FF, NL, CR, NLCR }

* 1. **Comments**

**Eazy** LANGUAGE supports only single-line comments: all the text from the ASCII characters **!!** to the end of the line is ignored by the scanner.

**<comments>** → # { sequence of ASCII chars } #

* 1. **Variable Identifiers**

The following variable identifier (Vid) tokens are produced by the scanner: two kinds of arithmetic tokens: **intVid\_T** (integer) and **fltVid\_T** (float-point numbers) and one kind of strings: **strVid\_T**.

**<variable identifier>** → IntVid \_T | fltVid \_T | strVid \_T

* 1. **Keywords**

The scanner produces a single token: **keywd\_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 **Eazy** language is given by:

**def, if, then, else, while, do, for, break, continue, return, as, or, try, catch, not , and, pin, pout**

* 1. **Integer Literals**

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

**<integer\_literal>** → int\_T

* 1. **Floating-point Literals**

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

**<flt\_literal>** → float\_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 - **LP\_T**, **RP\_T**, **LB\_T**, **RB\_T**, **Cmt\_T**, **EOS\_T**.

* 1. **Operators**

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

A single token is produced by the scanner: **art\_opr\_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: **scc\_opr\_T**.

**<string cocatenation operator>** → ++

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

**<relational operator>** → *one of* { >, <, ==, <> }

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

**<logical operator>** → *one of* { **.and.** , **.or.** , **.not.** }

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

**<assignment operator>** → =

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

**Eazy** LANGUAGE program is composed by one special function: “**main**” (Method name) defined as follows.

**<program>**  **main** {

<code\_session>

}

**Variable Lists**

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

**<opt\_varlist\_declarations>** <varlist\_declarations> | ϵ

**Variable Declarations**

**<varlist\_declarations>** → <varlist\_declaration>

| <varlist\_declarations><varlist\_declaration>

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

**New Grammar**

**<varlist\_declarations>** → <varlist\_declaration> <varlist\_declarationsPrime>

**<varlist\_declarationsPrime>** → <varlist\_declaration> <varlist\_declarationsPrime> | ϵ

Each variable declaration can be done as follows:

**<varlist\_declaration>** → <integer\_varlist\_declaration>

| <float\_varlist\_declaration>

| <string\_varlist\_declaration>

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

The variables list declaration is defined here:

**<integer\_varlist\_declaration>**  **int** <integer\_variable\_list>;

**<float\_varlist\_declaration>**  **float** <float\_variable\_list>;

**<string\_varlist\_declaration>**  **str** <string\_variable\_list>;

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

The list of variables is defined here:

**Integers:**

**<integer\_variable\_list>** <integer\_variable>

| <integer\_variable\_list>, <integer\_variable>

**<integer\_variable>** int\_T

**Float-points:**

**<float\_variable\_list>** <float\_variable>

| <float\_variable\_list>, <float\_variable>

**<float\_variable>** flt\_T

**Strings:**

**<string\_variable\_list>** <string\_variable>

| <float\_variable\_list>, <string\_variable>

**<string\_variable>** str\_T

* + 1. **CODE session:**

The (CODE) is the place we have statements:

**<code\_session>**  **code** {

<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>** → <integer\_variable> = <arithmetic expression>

| <float\_variable> = <arithmetic expression>

| <string\_variable>= <string 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>)

**do** { <statements>};

* + 1. **Input Statement**

**<input statement>** → **input** (<variable list>);

**Variable List:**

**<variable list>** → <variable identifier> | <variable list>,<variable identifier>

**Variable Identifier:**

**<variable identifier>** →<integer\_variable>

| <integer\_variable>

| <string\_variable>

* + 1. **Output Statement**

**<output statement>** → **pout** (<opt\_variable list>); | **pout** (str\_T);

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

**<output statement>** → **pout** (<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>

**Unary Arithmetic Expression:**

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

| + <primary arithmetic expression>

**Additive Arithmetic Expression:**

**<additive arithmetic expression>** →

<additive arithmetic expression> + <multiplicative arithmetic expression>

| <additive arithmetic expression> - <multiplicative arithmetic expression>

| <multiplicative arithmetic expression>

**Multiplicative Arithmetic Expression:**

**<multiplicative arithmetic expression>** →

<multiplicative arithmetic expression> \* <primary arithmetic expression>

| <multiplicative arithmetic expression> / <primary arithmetic expression>

| <primary arithmetic expression>

**Primary Arithmetic Expression:**

**<primary arithmetic expression>** → <integer\_variable>

| <float\_variable>

| flt\_T | int\_T

| (<arithmetic expression>)

* + 1. **String Expression**

**<string expression>** →

<primary string expression> | <string expression> ++ <primary string expression>

**Primary String Expression:**

**<primary string expression>** → <string\_variable> | str\_T

* + 1. **Conditional Expression**

**<conditional expression>** → <logical OR expression>

**Logical OR Expression:**

**<logical OR expression>** → <logical AND expression>

| <logical OR expression> **.or.** <logical AND expression>

**Logical AND Expression:**

**<logical AND expression>** → <logical NOT expression>

| <logical AND expression> **.and.** <logical NOT expression>

**Logical NOT Expression:**

**<logical NOT expression>** → **.not.** <relational expression>

| <relational expression>

* + 1. **Relational Expression**

**<relational expression>** →

<relational a\_expression> | <relational s\_expression>

**Relational Arithmetic Expression:**

**<relational a\_expression>** →

<primary a\_relational expression> == <primary a\_relational expression>

| <primary a\_relational expression> <> <primary a\_relational expression>

| <primary a\_relational expression> > <primary a\_relational expression>

| <primary a\_relational expression> < <primary a\_relational expression>

**Relational String Expression:**

**<relational s\_expression>** →

<primary s\_relational expression> == <primary s\_relational expression>

| <primary s\_relational expression> <> <primary s\_relational expression>

| <primary s\_relational expression> > <primary s\_relational expression>

| <primary s\_relational expression> < <primary s\_relational expression>

**Primary Arithmetic Relational Expression:**

**<primary a\_relational expression>** →<integer\_variable> | <float\_variable> | int\_T | flt\_T

**<primary s\_relational expression>** → <primary string expression>

**Good luck with Assignment 3.1!**