



# BNF TASK FOR ASSIGNMENT 3.1

Team: [**Neil Kingdom**] - Id: [**040967309**]

/ [**Mohit Nargotra**] - Id: [**040817484**]

***Note:*** *This task is part of the Assignment 3 from Compilers Course.*

# PHEONIX LANGUAGE SPECIFICATION

|  |
| --- |
| **NOTE 1** |
| Change this file (starting with the **name** of your language) and check all BNF rules described here, adapting it to your language. Minimal requirements:   * *One method to use variables;* * *Inputs and outputs (including string messages);* * *Define mathematical expressions (using float-point variables).* |

### General View

This document focus on **Pheonix LS** (Language Specification) developed by Neil Kingdom and Mohit Nargotra.

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

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

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

1. **The Pheonix 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**

Single-line and multi-line comments are one in the same in Pheonix. Comments begin with a # as the start delimiter, and terminate with a semi-colon.

**<comments>**  # { sequence of ASCII chars } EOS\_T

* 1. **Variable Identifiers**

Pheonix does not have specific variable identifiers depending on data type. Instead, one global VID\_T is used to describe both method names, as well as variable names

* 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 Pheonix is given by: int, float, char, byte, bool, void, null, if, elif, else, while, for, function, return, print, read, true, false, main

* 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. **Boolean Literals**

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

**<boolean\_literal> ** BOL\_T

* 1. **Byte Literals**

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

**<byte\_literal> ** BYL\_T

* 1. **Char Literals**

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

**<char\_literal> ** CHL\_T

* 1. **String Literals**

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

**<string\_literal>**  STL\_T

* 1. **Separators**

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

Seven different tokens are produced by the scanner - **LPR\_T**, **RPR\_T**, **LBR\_T**, **RBR\_T**, **COM\_T**, **EOS\_T, COL\_T**.

* 1. **Operators**

**<arithmetic\_operator>**  *one of* { +, -, \*, /, ++, -- }

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

**<bitwise\_operator>**  *one of* { &,|, <<, >> }

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

**<assignment operator>**  =

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

1. **The Pheonix Syntactic Specification**
   1. **Pheonix Program**
      1. **Program**

Pheonix program is composed of function declarations or variable declarations. The final declaration must be for the main function.

**<program>**  <declarations><main>

* + 1. **Declarations**

As mentioned, declarations can either be function declarations or variable declarations. Therefore, general declarations can be defined as follows:

**<declarations>**  <function\_decls> | <variable\_decls> | ϵ

* + 1. **Function Declarations**

**Function Declarations**

**<function\_decls> ** <function\_decls><function\_decl> | <function\_decl>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

**Function Declaration**

Function declarations are composed of a function identifier (the “function” keyword in conjunction with a return type and the function name) followed by an optional parameter list and mandatory code block:

**<function\_decl>** <function\_id><opt\_param\_list><code\_block>

**Function Identifiers**

Function identifers are composed of the “function” keyword, return type, and a valid VID\_T:

**<function\_id>** KW\_T { function }<return\_type> VID\_T

**Return Types**

Valid return types include void, int, float, char, byte, and bool:

**<return\_type>** KW\_T { void, int, float, char, byte, bool, string }

**Optional Parameter List**

A sequence of data types that will determine the data types of LVID\_Ts (local variable identifiers):

**<opt\_param\_list> ** <parameters> | ϵ

**Parameters**

**<parameters> ** <parameters><parameter> | <parameter>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

**Parameter**

**<parameter> ** KW\_T { void, int, float, char, byte, bool, string }

* + 1. **Main Function Declaration**

Main is defined exactly the same as other functions, except for the fact that we use <main\_function\_id> instead of <function\_id>. The difference between them is that <function\_id> requires a valid VID\_T for the method name, whereas <main\_function\_id> uses the keyword “main”:

**<main>**  <main\_function\_id><opt\_param\_list><code\_block>

**Main Function Identifier**

As mentioned, the main function must contain the keyword main which is reserved exclusively for creating the main function in Pheonix.

**<main\_function\_id>** KW\_T { function }<return\_type><entry\_func\_name>

**Program Entry/Start Function**

In Pheonix LS, the entry/start function is simply called main.

**<entry\_func\_name>** -> KW\_T { main }

* + 1. **Variable Declarations**

**Variable Declarations:**

**<variable\_decls>**  <variable\_decls><variable\_decl> | <variable\_decl>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

**Variable Declaration:**

Variables in Pheonix can either be declared and initialized later, or they can be declared and initialized on the same line. Thus variable declarations can be defined as follows:

**<variable\_decl>**  <var\_decl\_and\_init> EOS\_T | <var\_decl\_no\_init> EOS\_T

* + 1. **Variable Declarations Without Initialization**

A variable declaration without initialization might simply look something like “int i;”. Therefore, <var\_decl\_no\_init> is simply any variable identifier.

**<var\_decl\_no\_init>**  <integer\_id> | <float\_id> | <char\_id> | <bool\_id> | <byte\_id>|<string\_id>

* + 1. **Variable Identifiers**

Variable identifiers can be any valid <data\_type> followed by a valid VID\_T.

**Integer Identifier:**

**<integer\_id>**  KY\_W { int } VID\_T

**Float Identifier:**

**<float\_id>**  KY\_W { float } VID\_T

**Char Identifier:**

**<char\_id>**  KY\_W { char } VID\_T

**Byte Identifier:**

**<byte\_id>**  KY\_W { byte } VID\_T

**Bool Identifier:**

**<bool\_id>**  KY\_W { bool } VID\_T

**String Identifier:**

**<string\_id>**  KY\_W { string } VID\_T

* + 1. **Variable Declarations With Initialization**

Variable declarations with initialization get a bit complicated. Due to the fact that the syntax must follow <variable\_id> = <data><opt\_expr>; the solution is to create one kind of declaration for each data type.

**<var\_decl\_and\_init>**  <int\_initialization> | <float\_initialization> | <char\_initialization> | <bool\_initialization> | <byte\_initialization>| <string\_initialization>

* + 1. **Initialization**

As mentioned, each data type must have a unique initialization due to differing data type, as well as what sorts of data the data type could reasonably contain. Generally, most data types can accept literals, expressions containing literals, VID\_Ts and expressions containing VID\_Ts.

**Integer Initialization:**

Integers may be assigned an <integer\_literal>, <integer\_literal> followed by an arithmetic expression, VID\_T or VID\_T followed by an <arithmetic\_expr>.

**<int\_initialization>**  <integer\_id> ASS\_OP\_T <integer\_literal>

| <integer\_id> ASS\_OP\_T <integer\_literal><arithmetic\_expr>

| <integer\_id> ASS\_OP\_T VID\_T

| <integer\_id> ASS\_OP\_T VID\_T<arithmetic\_expr>

| <integer\_id> ASS\_OP\_T LVID\_T

| <integer\_id> ASS\_OP\_T LVID\_T<arithmetic\_expr>

**Integer Initialization:**

Floats may be assigned a <float\_literal>, <float\_literal> followed by an arithmetic expression, VID\_T or VID\_T followed by an <arithmetic\_expr>.

**<float\_initialization>**  <float\_id> ASS\_OP\_T <float\_literal>

| <float\_id> ASS\_OP\_T <float\_literal><arithmetic\_expr>

| <float\_id> ASS\_OP\_T VID\_T

| <float\_id> ASS\_OP\_T VID\_T<arithmetic\_expr>

| <float\_id> ASS\_OP\_T LVID\_T

| <float\_id> ASS\_OP\_T LVID\_T<arithmetic\_expr>

**Char Initialization:**

Chars are numeric in Pheonix, and therefore can also be assigned arithmetic expressions as well as <char\_literal>s.

**<char\_initialization>**  <char\_id> ASS\_OP\_T <char\_literal>

| <char\_id> ASS\_OP\_T <char\_literal><arithmetic\_expr>

| <char\_id> ASS\_OP\_T <integer\_literal>

| <char\_id> ASS\_OP\_T <integer\_literal><arithmetic\_expr>

| <char\_id> ASS\_OP\_T VID\_T

| <char\_id> ASS\_OP\_T VID\_T<arithmetic\_expr>

| <char\_id> ASS\_OP\_T LVID\_T

| <char\_id> ASS\_OP\_T LVID\_T<arithmetic\_expr>

**Byte Initialization:**

Bytes are a bit special in Pheonix because they are not considered to be numeric. Bytes may only be assigned <byte\_literal>s, or bitwise operations.

**<byte\_initialization>**  <byte\_id> ASS\_OP\_T <byte\_literal>

| <byte\_id> ASS\_OP\_T <byte\_literal><bitwise\_expr>

| <byte\_id> ASS\_OP\_T VID\_T

| <byte\_id> ASS\_OP\_T VID\_T<bitwise\_expr>

| <byte\_id> ASS\_OP\_T LVID\_T

| <byte\_id> ASS\_OP\_T LVID\_T<bitwise\_expr>

**Bool Initialization:**

Booleans are the most unqiue data type in terms of assignment. Arithmetic operations were considered to be valid expressions for bools, however, it’s been decided that relational and logical expressions are the only two expressions that are valid. Keywords true and false are also acceptable.

**<bool\_initialization>**  <bool\_id> ASS\_OP\_T <bool\_literal>

| <bool\_id> ASS\_OP\_T <bool\_literal><logical\_expr>

| <bool\_id> ASS\_OP\_T <integer\_literal><logical\_expr>

| <bool\_id> ASS\_OP\_T <float\_literal><logical\_expr>

| <bool\_id> ASS\_OP\_T <byte\_literal><logical\_expr>

| <bool\_id> ASS\_OP\_T <char\_literal><logical\_expr>

| <bool\_id> ASS\_OP\_T VID\_T

| <bool\_id> ASS\_OP\_T VID\_T<logical\_expr>

| <bool\_id> ASS\_OP\_T LVID\_T

| <bool\_id> ASS\_OP\_T LVID\_T<logical\_expr>

| <bool\_id> ASS\_OP\_T KW\_T { true, false }

**String Initialization:**

Strings cannot be assigned any sort of numeric, boolean, or binary value. Only other variables, or string literals suffice.

**<string\_initialization>**  <string\_id> ASS\_OP\_T STL\_T

| <string\_id> ASS\_OP\_T VID\_T

| <string\_id> ASS\_OP\_T LVID\_T

* + 1. **Code Blocks:**

Code blocks are where we put any sort of statements (optional).

**<code\_block>**  {

<opt\_statements>

}

**Optional Statements:**

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

* + 1. **Statements**

**<statements>**  <statements><statement> | <statement>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

* 1. **Statement**

**<statement>** 

<assignment\_statement> | <selection\_statement> | <iteration\_statement>

| <loop\_statement> | <input\_statement> | <output\_statement>

| <return\_statement> | <function\_call> | <variable\_decl>

* + 1. **Assignment Statement**

**<assignment\_statement>**  <assignment\_expression>EOS\_T

* + 1. **Assignment Expression**

**<assignment\_expr>**  VID\_T ASS\_OP\_T <numeric\_arithmetic\_expr>

| VID\_T ASS\_OP\_T <relational\_expr>

| VID\_T ASS\_OP\_T <logical\_or\_expr>

| VID\_T ASS\_OP\_T <bitwise\_byte\_expr>

| VID\_T ASS\_OP\_T <input\_statement>

| VID\_T ASS\_OP\_T <non\_string\_literals>

| VID\_T ASS\_OP\_T <string\_literal>

| VID\_T ASS\_OP\_T VID\_T

| VID\_T ASS\_OP\_T LVID\_T

| VID\_T ASS\_OP\_T KW\_T { true, false }

| LVID\_T ASS\_OP\_T <numeric\_arithmetic\_expr>

| LVID\_T ASS\_OP\_T <relational\_expr>

| LVID\_T ASS\_OP\_T <logical\_or\_expr>

| LVID\_T ASS\_OP\_T <bitwise\_byte\_expr>

| LVID\_T ASS\_OP\_T <input\_statement>

| VID\_T ASS\_OP\_T <non\_string\_literals>

| VID\_T ASS\_OP\_T <string\_literal>

| VID\_T ASS\_OP\_T VID\_T

| VID\_T ASS\_OP\_T LVID\_T

| LVID\_T ASS\_OP\_T KW\_T { true, false }

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

**<selection\_statement>** 

<primary\_selection><secondary\_selection><tertiary\_selection>

| <primary\_selection><secondary\_selection>

| <primary\_selection>

**Primary Selection**

**<primary\_selection> ** KW\_T { if } (<conditional\_expr>)

**Secondary Selection**

**<primary\_selection> ** KW\_T { elif } { <opt\_statements> }

**Tertiary Selection**

**<primary\_selection> ** KW\_T { else } { <opt\_statements> }

* + 1. **Iteration Statement (for loop)**

**<iteration\_statement>**  KW\_T { for }(<conditional\_for\_expr>) { <opt\_statements> }

**Conditional “For” Expression**

Pheonix attempts to take a more mainstream approach to for loop syntax. A variable must be supplied, followed by a colon, and three comma separated range values. The first range value represents the start index, the second being the end index, and the third being the step size. This would look something like the following: for(i : 1,10,1) { code }

**<conditional\_for\_expr> ** VID\_T COL\_T <range\_val> COM\_T <range\_val> COM\_T <range\_val>

**Range Values**

**<range\_val> ** <integer\_literal> | <float\_literal> | VID\_T | LVID\_T

* + 1. **Loop Statement (while)**

**<iteration\_statement> ** KW\_T { while }(<conditional\_expr>) { <opt\_statements> }

* + 1. **Input Statement**

**<input statement>**  KW\_T { read }EOS\_T

* + 1. **Output Statement**

**<output\_statement>**  KW\_T { print } <opt\_argument\_list>EOS\_T

**Optional Variable List:**

**<opt\_argument\_list>**  <argument\_list>| ϵ

**Argument\_List:**

**<argument\_list> ** <argument\_list><argument> | <argument>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

**Argument:**

**<argument> ** LVID\_T | VID\_T | <integer\_literal> | <float\_literal>

| <char\_literal> | <byte\_literal> | <bool\_literal> | <string\_literal>

* + 1. **Function Calls**

**<function\_call>** ****VID\_T <opt\_argument\_list>EOS\_T

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

**<arithmetic\_expr>**  <unary\_arithmetic\_expr> | <additive\_arithmetic\_expr>

**Unary Arithmetic Expression:**

**<unary\_arithmetic\_expr>**  - <primary\_arithmetic\_expr>

| + <primary\_arithmetic\_expr>

**Additive Arithmetic Expression:**

**<additive\_arithmetic\_expr>**  <additive\_arithmetic\_expr> + <multiplicative\_arithmetic\_expr>

| <additive\_arithmetic\_expr> - <multiplicative\_arithmetic\_expr>

| <multiplicative\_arithmetic\_expr>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

**Multiplicative Arithmetic Expression:**

**<multiplicative\_arithmetic\_expr>** 

<multiplicative\_arithmetic\_expr> \* <primary\_arithmetic\_expr>

| <multiplicative\_arithmetic\_expr> / <primary\_arithmetic\_expr>

| <primary\_arithmetic\_expr>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

**Primary Arithmetic Expression:**

**<primary\_arithmetic\_expression>**  VID\_T | LVID\_T | <float\_literal> | <integer\_literal> | <char\_literal> | (<arithmetic\_expr>)

* + 1. **Bitwise Expression**

**<bitwise\_expr> ** <left\_bitshift> | <right\_bitshift> | <bitwise\_and> | <bitwise\_or>

**Left Bitshift:**

**<left\_bitshift>** **** OP\_BSL <integer\_literal> | OP\_BSL VID\_T | OP\_BSL LVID\_T

**Right Bitshift:**

**<right\_bitshift>** **** OP\_BSR <integer\_literal> | OP\_BSR VID\_T | OP\_BSR LVID\_T

**Bitwise OR:**

**<bitwise\_or>** **** OP\_BWO <byte\_val>

**Bitwise AND:**

**<bitwise\_and>** **** OP\_BWA <byte\_val>

**Byte Value:**

**<byte\_val>** **** VID\_T | LVID\_T | <byte\_literal>

* + 1. **Conditional Expression**

**<conditional\_expr>**  <logical\_or\_expr>

**Logical OR Expression:**

**<logical\_or\_expr>**  <logical\_and\_expr> | <logical\_or\_expr> **OP\_OR** <logical\_and\_expr>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

**Logical AND Expression:**

**<logical\_and\_expr>**  <logical\_not\_expr> | <logical\_and\_expr> **OP\_AND** <logical\_not\_expr>

* **PROBLEM DETECTED: Left recursion:**

**New Grammar**

**Logical NOT Expression:**

**<logical\_not\_expr>**  **OP\_NOT** <relational\_expr> | <relational\_expr>

* + 1. **Relational Expression**

**<relational\_expr>** 

<inner\_conditional\_op> OP\_EQ <inner\_conditional\_op>

| <inner\_conditional\_op> OP\_NE <inner\_conditional\_op>

| <inner\_conditional\_op> OP\_LT <inner\_conditional\_op>

| <inner\_conditional\_op> OP\_GT <inner\_conditional\_op>

| <inner\_conditional\_op> OP\_LTE <inner\_conditional\_op>

| <inner\_conditional\_op> OP\_GTE <inner\_conditional\_op>

**Inner Conditional Operation:**

**<inner\_conditional\_op> ** LVID\_T | VID\_T | <non\_string\_literals> | <numeric\_arithmetic\_expr>

| <bitwise\_byte\_expr> **|** (<assignment\_expr>)

**Non String Literals:**

**<non\_string\_literals>**  <float\_literal> | <integer\_literal> | <char\_literal> | <bool\_literal> | <byte\_literal>

**Numeric Arithmetic Expressions:**

**<numeric\_arithmetic\_expr>**  <numeric\_val><arithmetic\_expr> | (<numeric\_val><artithmetic\_expr>)

**Numeric Values:**

**<numeric\_val>**  <float\_literal> | <integer\_literal> | <char\_literal> | LVID\_T | VID\_T

**Bitwise Byte Expression:**

***<bitwise\_byte\_expr>*** * <byte\_val><bitwise\_expr>* | (*<byte\_val><bitwise\_expr>)*