**PRACRICAL TWO:SYNTAX**

In this practical we take a look at the syntax of Python and Javascript using the BNF form.

**DEFINITIONS**

**BNF(Backus Naur form)** is a meta language, meaning it is a language used to describe other languages.

**Syntax:** Is a set of rules which dictate how programs should be written in a particular programming language.

It can also be defined as: The form or structure of expressions, statements and program units.

Below is an example of a general syntax of writing a person’s full name e.g. Mr. John Ndege generated in BNF form.

<fullname>::=<title><name>

<title>::=Mr|Mrs|Ms|Dr…

<name>::=<initial\_caps>|<name><name>

<initial\_caps>::=<uppercase><lower>

<uppercase>::=A|B|C|…..

<lowercase>::=a|b|c|…..

|<lower><lower>

**DEFINING THE SYNTAX OF JAVASCRIPT USING BNF**

Below is an example of the syntax of JavaScript using the Backus-Naur Form (BNF):

<program> ::= <statement\_list>

<statement\_list> ::= <statement> | <statement> <statement\_list>

<statement> ::= <variable\_declaration> ";" | <expression\_statement> ";"

<variable\_declaration> ::= "let" <identifier> "=" <expression>

<expression\_statement> ::= <expression>

<expression> ::= <assignment\_expression> | <binary\_expression> | <unary\_expression> | <primary\_expression>

<assignment\_expression> ::= <left\_hand\_side\_expression> "=" <assignment\_expression> | <conditional\_expression>

<binary\_expression> ::= <expression> <binary\_operator> <expression>

<unary\_expression> ::= <unary\_operator> <expression> | <primary\_expression>

<conditional\_expression> ::= <logical\_or\_expression> "?" <expression> ":" <expression>

<logical\_or\_expression> ::= <logical\_and\_expression> | <logical\_and\_expression> "||" <logical\_or\_expression>

<logical\_and\_expression> ::= <equality\_expression> | <equality\_expression> "&&" <logical\_and\_expression>

<equality\_expression> ::= <relational\_expression> | <relational\_expression> "==" <equality\_expression> | <relational\_expression> "!=" <equality\_expression>

<relational\_expression> ::= <additive\_expression> | <additive\_expression> "<" <relational\_expression> | <additive\_expression> ">" <relational\_expression> | <additive\_expression> "<=" <relational\_expression> | <additive\_expression> ">=" <relational\_expression>

<additive\_expression> ::= <multiplicative\_expression> | <multiplicative\_expression> "+" <additive\_expression> | <multiplicative\_expression> "-" <additive\_expression>

<multiplicative\_expression> ::= <unary\_expression> | <unary\_expression> "\*" <multiplicative\_expression> | <unary\_expression> "/" <multiplicative\_expression>

<left\_hand\_side\_expression> ::= <identifier> | <property\_access\_expression>

<property\_access\_expression> ::= <primary\_expression> "." <identifier>

<primary\_expression> ::= <identifier> | <literal> | "(" <expression> ")"

<identifier> ::= [a-zA-Z\_][a-zA-Z0-9\_]\*

<literal> ::= <number\_literal> | <string\_literal> | <boolean\_literal> | <null\_literal>

<number\_literal> ::= [0-9]+ | [0-9]+ "." [0-9]+

<string\_literal> ::= '"' <string\_characters> '"' | "'" <string\_characters> "'"

<string\_characters> ::= <string\_character> | <string\_character> <string\_characters>

<string\_character> ::= <any\_character\_except\_double\_quote\_or\_backslash> | "\" <escaped\_character>

<escaped\_character> ::= "\"" | "'" | "\\" | "b" | "f" | "n" | "r" | "t" | "u" <hex\_digit> <hex\_digit> <hex\_digit> <hex\_digit>

<boolean\_literal> ::= "true" | "false"

<null\_literal> ::= "null"

<binary\_operator> ::= "+" | "-" | "\*" | "/" | "==" | "!=" | "<" | ">" | "<=" | ">=" | "&&" | "||"

<unary\_operator> ::= "+" | "-" | "!" | "typeof"

<hex\_digit> ::= [0-9a-fA-F]

The above does not however comprehensively cover the whole syntax of Javascript but gives a general idea.

**EXPLANATION**

The provided code represents the syntax of JavaScript using the Backus-Naur Form (BNF). BNF is a notation system used to describe the grammar of a programming language. In this BNF representation, the code defines various production rules to express the structure and relationships between different elements of JavaScript.

The code starts with the <program> rule, which represents the entry point of a JavaScript program and consists of a <statement\_list>. The <statement\_list> rule allows for a sequence of one or more <statement>s, separated by semicolons. Each <statement> can be either a <variable\_declaration> or an <expression\_statement>. The <variable\_declaration> rule specifies the syntax for declaring variables using the let keyword, an <identifier>, an equals sign, and an <expression> to assign a value. The <expression\_statement> rule represents any expression followed by a semicolon.

The remaining rules define various expressions and operators, such as <binary\_expression>, <unary\_expression>, and <primary\_expression>. These rules specify the syntax for mathematical and logical operations, including assignments, comparisons, and logical operators like && and ||. The BNF representation also includes rules for literals, identifiers, and other language constructs.

Overall, the provided code in BNF form represents the hierarchical structure and syntax rules of JavaScript, defining how different language elements can be combined to form valid JavaScript programs.

**DEFINING THE BNF SYNTAX USING THE BNF FORM**

Below is the syntax of some few programs in python

Functions

<function\_definition> ::= "def" <function\_name> "(" [ <parameter\_list> ] ")" ":" <statement\_list>

<function\_name> ::= <identifier>

<parameter\_list> ::= <parameter> [ "," <parameter\_list> ]

<parameter> ::= <identifier>

<statement\_list> ::= <statement>

| <statement> <statement\_list>

<statement> ::= ...

The `<function\_definition>` rule represents the syntax for defining a function in Python. It starts with the keyword "def", followed by the `<function\_name>` (which is an `<identifier>` representing the name of the function), parentheses that enclose an optional `<parameter\_list>`, a colon (":"), and a `<statement\_list>` that represents the body of the function.

The `<function\_name>` is an `<identifier>`, which is a placeholder representing any valid identifier name in Python. An identifier consists of letters, digits, and underscores, but cannot start with a digit.

The `<parameter\_list>` represents the comma-separated list of parameters that the function takes. Each `<parameter>` is an `<identifier>` representing the name of theparameter.

The `<statement\_list>` represents the body of the function, consisting of one or more statements. The specific syntax for `<statement\_list>` and `<statement>` is not included in the above BNF snippet, as it can vary and depend on the statements within the function.

Here's an example of a function definition in Python:

def greet(name):

print("Hello, " + name + "!")

In the above example, we define a function named `greet` that takes one parameter `name`. When called, the function prints a greeting message with the provided name.

**A SIMPLE PROGRAM TO SUM TWO NUMBERS**

# This program adds two numbers

num1 = 1.5

num2 = 6.3

# Add two numbers

sum = num1 + num2

# Display the sum

print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))

**BNF SYNTAX OF THE PROGRAM**

<addition> ::= <number> "+" <number>

<number> ::= <digit> { <digit> }

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

This BNF represents a simple addition operation with two numbers. `<addition>` is the top-level rule, which consists of two `<number>` values separated by a plus sign ("+"). The `<number>` rule represents a sequence of one or more `<digit>` characters.

**SYNTAX OF PYTHON CONDITIONAL EXPRESSIONS**

**IN BNF**

<conditional\_expression> ::= <expression> "?" <expression> ":" <expression>

The `<expression>` represents any valid expression in Python. The conditional expression consists of three parts separated by the question mark (`?`) and the colon (`:`). The first expression before the question mark represents the condition. If the condition is true, the second expression after the question mark is evaluated and returned. Otherwise, if the condition is false, the third expression after the colon is evaluated and returned.

**PROGRAM EXAMPLE**

x = 10

y = 5

Example usage:

python

x = 10

y = 5

max\_value = x if x > y else y

print(max\_value) # Output: 10

In the above example, the conditional expression `(x if x > y else y)` is used to determine the maximum value between `x` and `y`. If `x` is greater than `y`, the value of `x` is assigned to `max\_value`. Otherwise, if `x` is not greater than `y`, the value of `y` is assigned to `max\_value`. In this case, since `x` is 10 and `y` is 5, `max\_value` is assigned the value of `x`, which is 10.

**USING ONLINE GENERATING TOOLS TO GENERATE PARSE TREES**

**A tree** is a data structure used to Store data in a hierarchical structure. It can have an arbitrary number of child nodes.

Each node stores a value. They are commonly used to store data such as file systems.

In languages, parse trees are used.

**Example of a parse tree**

Statement is; A=B +CONSTANT

<program>

<statement>

<variable> = <expression>

<term> + <term>

<variable>

A B CONSTANT

A parse tree can be generated by use of online tools, such as

* BNF Generator by Antr Works
* Online Syntax Tree Generator
* ANTLR
* BNF Visualizer
* Online BNF parse Generator etc.

**BNF Visualizer** allows you to input your BNF grammar and then generates a tree diagram based on the rules defined in the grammar. This tool also provides various customization options to modify the appearance of the generated tree.

The **Online BNF Parser Generator,** which not only generates the tree diagram based on the BNF grammar but also allows you to input a sample string and see how it is parsed according to the grammar rules.

For example;

To use the BNF Generator,

1.Go to the BNF Generator,

Use the following link: <http://mshang.ca/syntree/>

2.Input your BNF grammar into the text box provided on the website. For example, let's say our BNF grammar is as follows:

**NP^ Alice]**

**[NP [N Alice] and [N Bob]]**

**[S[NP[N Alice]][VP[V is][NP[N'[N a student][PP^ of physics**

**[S [X\_a Movement] [Y example <a>]]**

The BNF Generator will process the grammar and generate a parse tree such as shown below:

