**Context-Free Grammars**

A context-free grammar (CFG) is a set of recursive rewriting rules (or *productions*) used to generate patterns of strings.

A CFG consists of the following components:

* a set of *terminal symbols*, which are the characters of the alphabet that appear in the strings generated by the grammar.
* a set of *nonterminal symbols*, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.
* a set of *productions*, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).
* a *start symbol*, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

To generate a string of terminal symbols from a CFG, we:

* Begin with a string consisting of the start symbol;
* Apply one of the productions with the start symbol on the left hand size, replacing the start symbol with the right hand side of the production;
* Repeat the process of selecting nonterminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all nonterminals have been replaced by terminal symbols.
* **Alphabets:**

**<alphabet>**🡪<**lower&upper**>

* <**lower&upper**>🡪<lowercase> | <uppercase>
* <**lowercase>🡪**a|b|c|d….|x|y|z
* <**uppercase>🡪**A|B|C|D….|X|Y|Z
* **Digits:**

<**digit**>🡪<**number**>

* <**number**>🡪0|1|2|3|4|5|6|7|8|9
* **Alphanumeric:**

<alphanumeric>🡪<**alpha-num**>

* <**alpha-num>🡪<**digit> | <alphabet>
* **Operators:**

**<operator>**🡪<arithmetic-op>|<relational-op>|<logical-op>|<assignment-op>|

**<**unary-op**>|<**binary-op**>**

* <**arithmetic-op>🡪<multiply-divide-modulus> | <plus-minus>**
* **<multiply-divide-modulus>🡪 \* | / | %**
* **<plus-minus>🡪 + | -**
* **<relational-op>🡪< | > | <= | >= | != | = =**
* **<logical-op>🡪<and> | <or> | <not>**
* **<and>🡪&&**
* **<or>🡪 ||**
* **<not>🡪!**
* **<assignment-op>🡪<simple> | <compound>**
* **<simple>🡪=**
* **<compound>🡪+= | - = | \*= | /=**
* **<binary-op>🡪 ++ | - -**
* **Punctuators:**

**<punctuator>🡪<punch>**

* **<punch>🡪; | , | ( | ) | . | { | } | [ | ]**
* **Constants:**

**<constant>🡪<int-const> |<float-const> | <string-const> | <bool-const>**

* **<int-const>🡪<sign><**digit><**int\_digit>**
* **<int\_digit>🡪<**digit><**int\_digit>**
* **<int\_digit>🡪 €**
* **<sign>🡪 + | -**
* **<float-const>🡪<**sign**><int\_digit**>**.**<digit><**int\_digit>** | **<int\_digit**>**.**<digit><**int\_digit>**
* **<string-const>🡪“<str-list>”**
* **<str-list>🡪<**digit**><str-list> | <**alphabet>**<str-list>** | < alphanumeric>**<str-list>** | <specialcharacter>**<str-list>** | €
* **<bool-const>🡪**true | false

# Data-type:

**<dt>🡪 int | float | string | char**

# Access-Modifier

**<AM>🡪**public | private

# Identifier:

**<ID>🡪< alphabet ><ID\_1>< alphanumeric > | \_ < ID\_1 >< alphanumeric >**

* **<list>🡪<**alphabet> | \_<alphabet> |\_<alphanumeric>|

\_<alphabet>\_ | \_<alphanumeric>\_

* **Declaration:**

<**decl\_Init> 🡪 <access\_mod> <decl>**

**<init> 🡪 <init2> | €**

**<init2> 🡪 <const> | ID <init>**

**<dec1> 🡪 DT ID <ass>**

<**decl\_funct\_abst**> 🡪 abstract **DT ID (<params>) <function-body>**

**<ass> 🡪 = <dec2> ter | € | (<params>) <function-body>**

**<decl> 🡪 DT <dec4>**

**<dec3> 🡪 ID <ass>**

**<decl> 🡪 ID ID <ass>**

**<dec2> 🡪 ID <init> <list>**

**<dec2> 🡪 new DT ( <params )**

**<dec4> 🡪 <dec4> | <dec5>**

**<dec5> 🡪 [index] ID = <ArrayInit>**

**<params> 🡪 € | <id\_const><comma> | <exp><comma>**

**<commas> 🡪 , <params>**

**<ArrayInit> 🡪 ID | [<params>]**

**<list> 🡪 ; | , ID <init> <list>**

* **Function**

**<function-body>**🡪 **; | { <mst> return <R1>}**

**<R1>**🡪 **<exp> |**  € **|** <**cond**>

**<function-call>🡪 ID<assign\_list> (<params>)**

* **Body:**

<**body>🡪; | <SST> | <MST>**

<**return\_stmt>**🡪 return <**exp>ter**

* **Assignment-Statement:**

<**assign**>🡪 ID <**assign\_list>** <assignment-op> <**exp**>

<**assign\_list> 🡪 .** ID <**assign\_list> | €**

<**ID\_CONST**>🡪<**ID** > | <**constant**>

* **Increment-Decrement-Statement:**

<**incdec-stmt>🡪 ++ | - -**

* **Loop-Statement:**

<**loop-stmt**>🡪<for-loop> | <while-loop> | <do-while-loop>

* <**for-loop**>🡪for ( <**exp**> **;** <**exp**> **;** cond3)<**body**>

**<cond3>**🡪 **ID <cond3’> | <**incdec-stmt>

* **<cond3’>🡪<assignment-op><exp> | ++**
* <**while-loop>🡪**while(<cond2>)<body>
* <**do-while-loop**>🡪do <body>while(<cond2>) ter

**<EXP> 🡪 <AND> <OR’>**

**<OR’> 🡪 || <AND> <OR’> | €**

**<AND> 🡪 <ROP> <AND’>**

**<AND’>🡪 && <ROP> <AND’> | €**

**<ROP> 🡪 <PM> <ROP’>**

**<ROP’> 🡪 RO <PM> <ROP>|€**

**<PM>🡪 <MDM><PM’>**

**<PM’> 🡪 <PM><MDM><PM’> | €**

**<MDM> 🡪 <F> <MDM’>**

**<MDM’> 🡪 <MDM><F><MDM’> | €**

**<F> 🡪** ID <**DEC\_INC**> **|** CONST **|** ( < **OR >** ) | ! <**F**> | <**INC\_DEC**> ID

<**DEC\_INC**> 🡪 <**INC\_DEC**> | | ( <**params**> ) | . ID <**DEC\_INC**>

* **If-else-Statement:**

**<if-else-stmt>🡪 if (<**cond2**>) <list3>**

* **<list3>🡪**<body> | <body>**else <**body> | <body>**else < if-else-stmt>**

# Class-Statement:

<**class-str >🡪**class <**extend-stmt>{<multiple\_stmt> }**

**< extend-stmt>🡪**extends <**ID >**

**<am-stmt>🡪< am > | €**

**< abstract-stmt>🡪**abstract | €

* **Switch-Statement:**

**<switch-stmt>🡪**switch(<**switch-cond>)(<switch-body>)**

* <**switch-cond>🡪 <cond> | <exp>**
* **<switch-body>🡪** € | <**case-st><switch-body> | <default-st> <switch2>**
* **<switch2>🡪** € | <case-st> <switch2>
* **<case-st>🡪** case <ID-Const> : <**mst>** break;
* **<default-st>🡪 default :<mst>** break;
* **SST:**

**<sst>** 🡪 **<loop\_start> | <condIfelse> | <switch> | <decl> | return | break | continue |**

**<mst> 🡪 <sst><mst> | €**

* **Class\_Start:**

**<class\_str> 🡪 <abst> class ID <extends> <body> <class\_re>**

**<abst>** 🡪 **abstract | €**

**<extend>** 🡪 **extends ID | €**

**<class\_re> 🡪 <class\_str> | €**

* **Starting Point:**

**<S> 🡪 <class\_str>$**