**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the string, and replacing them with the right hand side of some corresponding production, until all nonterminals have been replaced by terminal symbols.

# Constants:

**<const>🡪** int-const | float-const | string-const | char-const

# Access-Modifier

**------<AM am>🡪**public {am=public} | private {am=private} | **€**

# Declaration:

**------**<**decl\_Init Ref> 🡪 <AM am> <decl, Ref,am>**

**------<init> 🡪 <init2> | € | assign <init2>**

**------<init2> 🡪 <const> | ID <init>**

**------<dec1 Ref,am> 🡪 DT {type=VP} <decl6 type, Ref,am>**

**------<decl6 type, Ref> 🡪 ID { name=VP } { insertCT(am, type, name, Ref) } <ass> {comp checkkaro} | [] {type+=[] } ID {name=VP] <dec5 am,type,name,Ref> | <list >**

**------**<**decl Ref,am**> 🡪 abstract {tm=abstract} **DT {type=VP} ID {name=VP} (<funct\_params paralist>) {insert(am,tmp,type,name,Ref)} <function\_body>**

**------<decl Ref,am> 🡪 {tm=””}** ID {type=VP} <**decl\_5** Ref,am,type>

**------**<decl\_5 Ref,am,type,tm>🡪 ID {name=VP}  **<ass2> {compcheck} | [] {type+=[]} ID {name=VP}** <decl5> | ( <funct\_params paralist > ) {type+=”->”+paralist} { insertCT(name, am, tm, type, Red) } <**function\_body** Ref>

**------<ass2> 🡪** ( **<funct\_params** paralist**>** ) {type+=”->”+paralist} {insertCT(am ,typr, name, tm, Ref)} **<function-body** Ref**> | <ass>** {insertCT(am ,tm, name, type, Ref)}

**------<ass> 🡪 assign <compcheck> <ass1>**

**------<ass1> 🡪 <dec2>** ter  **| <const>** ter

**------<dec2> 🡪** ID **<init> <list>**

**------<dec2> 🡪** new DT **( <params> )** ter

**------<dec5 am,type,name, Ref> 🡪** { insertCT(name,type,am,Ref) } assign **{compcheck} <ArrayInit> ter | ( <funct\_params** paralist**> ) {type+=”->”+** paralist **} {**insertCT(am,name,type,Ref) **} <function\_body** Ref**>**

**------<params> 🡪** €  **| <exp> <comma>**

**------<commas> 🡪** , **<params>**

**------<ArrayInit> 🡪** ID **| [<params>]**

**------<list> 🡪** ter **|** , ID **<init> <list>**

# Function

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

**------<R1>**🡪 **<exp>** ter **|**  ter

**------<funct\_params** paralist**> 🡪 ID {** paralist+=VP} **<array\_opt** paralist**> ID <list4** paralist**> |** €

**------<funct\_params> 🡪 DT <array\_opt> ID <list4> |** €

**------<array\_opt** paralist**> 🡪 [] {** paralist+=[] **} |** €

**------ <list4** paralist**> 🡪 , {** paralist+=”,” **} <funct\_params** paralist**>**

# Body:

**------**<**body** Ref**>🡪; | <SST** Ref**> |** { **<MST** Ref**>** }

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

**------**<**ID\_CONST**>🡪<**ID** > | <**const**>

# Increment-Decrement-Statement:

**------**<**INC\_DEC>🡪** inc **|** dec

# Loop-Statement:

**------**<**loop\_start** Ref>🡪<for-loop Ref> | <while-loop Ref> | <do-while-loop Ref>

* <**for-loop** Ref>🡪for ( <**assign\_call** Ref> <**exp**> ter **<cond3** Ref>)<**body**>

**<cond3** Ref**>**🡪 **ID {** lookupFT() **} <cond3’** Ref**> | <**INC\_DEC Ref>

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

**------<EXP t2> 🡪 <AND t2><OR’ t2>**

**------<OR’ t2> 🡪 || { opr=VP } <AND t2> <OR’ t2> | €**

**------<AND t2> 🡪 <ROP t2> <AND’ t2>**

**------<AND’ t2>🡪 && <ROP t2> <AND’ t2> | €**

**------<ROP t2> 🡪 <PM t2> <ROP’ t2>**

**------<ROP’ t2> 🡪 RO <PM t2> <ROP’ t2>|€**

**------<PM t2>🡪 <MDM t2><PM’ t2>**

**------<PM’ t2> 🡪** PM **<MDM t2><PM’ t2> | €**

**------<MDM t2> 🡪 <F t2> <MDM’ t2>**

**------<MDM’ t2> 🡪** MDM **<F t2><MDM’ t2> | €**

**------<F t2> 🡪** ID { t2=loopkupFT(VP) } <**DEC\_INC**>  **|** CONST **|** ( < **EXP >** ) | ! <**F**> | <**INC\_DEC**> ID

**------**<**DEC\_INC**> 🡪 <**INC\_DEC**> | | ( <**params**> ) **<DEC\_INC\_RE>** | . ID <**DEC\_INC**> | [ <Exp> ] **<DEC\_INC\_RE>**

**------**<DEC\_INC\_RE> 🡪 . ID <DEC\_INC\_RE\_1> |

**------**<DEC\_INC\_RE\_1>🡪 ( <params> ) <DEC\_INC\_RE> | [ <Exp> ] <DEC\_INC\_RE> | <DEC\_INC\_RE>

# If-else-Statement:

**<if-else-stmt>🡪 if (<**exp**>) <body> <list3>**

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

# Switch-Statement:

**<switch-stmt>🡪**switch (<**exp>) { <switch-body> }**

* **<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** Ref**>** 🡪 **<loop\_start** Ref**> | <if-else-stmt> | <switch> | <return> | break | continue | <assign\_call** Ref**>**

**------<assign\_call** Ref**> 🡪** ID { t1=lookupFT(VP) } **<assign1 t1,Ref>**

**------<assign1 t1, Ref>🡪 <assign\_list t1,Ref> <assign\_call2 t1,Ref> |** ID { name=VP } <assign\_call4 t1, Ref, Name>

**------**<assign\_call4 t1,Ref,Name >🡪 assign <assign\_call3 t1, Ref, name> | {insertFT(name,type,scope) } ter

**------**<assign\_call3 Ref, t1>🡪 new ID { if(type==VP) } ( <params paralist> ) {checktoMatch(t1, paralist)} { insertFT(t1,name,scope) } ter | <exp t2> { compability(t1, t2) } {insertFT(t1,name,scope) } ter

**------<assign\_call Ref> 🡪** DT { t1=VP } ID { name = VP } <assign\_call5 t1, name ,Ref>

**------**<assign\_call5 t1, name, Ref>🡪assign  **<assign\_call1 name,t1,Ref > |** { insertFT(name, scope, t1) } **ter**

**------<assign\_call1 t1,name,Ref> 🡪 <Exp t2> { compability(t1, t2,”=”)} |** new DT ( <params> ) ter

**------<assign\_call2> 🡪 (<params>) ter | < assignment-op> <call>**

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

**------< assignment-op> 🡪** assign | compAss

**------<call> 🡪** ID **<assign\_list> <call1>**

**------<call1> 🡪 (params)** ter **|** <**init\_list**> exp ter

**------**<**init\_list**> 🡪 assign ID <**init\_list**> | €

**------**<assign\_re> 🡪 . ID <assign\_re1> | ter

**------**<assign\_re1> 🡪 ( <params> ) <assign\_re> | [ <Exp> ] <assign\_re> | ter

**------<call> 🡪** <**const**> ter

**------<MST> 🡪 <sst><mst> |** €

# Class\_Start:

**------ <class\_str> 🡪 <abst** category **><sealeds** category **> class {type=CP} ID {name=VP} <extends {parent}> {** Ref=insert(name, type, category, parent) } **<class\_body** Ref**> <class\_re>**

**------<abst** category**>** 🡪abstract {category= abstract} **| €**

**------<extend** parent **>** 🡪extends ID {parent=VP} | €

**------<sealed category > 🡪** sealed {category=sealed} | € {category=general}

**------<class\_body** Ref**> 🡪 { <decl\_init** Ref**> <class\_body’** Ref**> } | ter**

**------<class\_body’ Ref> 🡪 <decl\_init Ref> <class\_body’ Ref> |** €

**------<class\_re> 🡪 <class\_str> | €**

# Starting Point:

**<S> 🡪 <class\_str>$**

# First And Follow Set:

|  |  |  |
| --- | --- | --- |
| Non-Terminal | First | Follow |
| Class\_str | abstract, sealed, class | $ |
| Abstract | Abstract, € | Sealed, class |
| Sealed | sealed, € | class |
| Extend | extend, € | { |
| Class\_body | {, ter | abstract, extend, sealed, class |
| Class\_body’ | Public, private, DT, ID, abstract | } |
| Class\_re | abstract, sealed, class, € | $ |
| Decl\_init | Public, private, DT, ID, abstract | Public, private, abstract, DT, ID |
| Decl | DT, ID, abstract | Public, private, abstract, DT, ID |
| Ass | Assign | (, ID, new |
| Ass1 | New,id,( |  |
|  |  |  |
| Dec3 | ID | Public, private, abstract, DT, ID |
| Dec4 | ID, [ | Public, private, abstract, DT, ID |
| Dec2 | New, Id | Ter |
| Dec5 | [ | Public, private, abstract, DT, ID |
| Params | ||, &&, PM, RO, MDM, ROP, ID, int\_const, float\_const, string\_const, char\_const, (, !, inc,dec, € | ), ] |
| Commas | , | ), ] |
| ArrayInit | ID, [ |  |
| Init | int\_const, float\_const, string\_const, char\_const,€ | ter , , |
| List | ter, , |  |
| Function\_body | Ter, { |  |
| R1 | ||, &&, PM, RO, MDM, ROP, ID, int\_const, float\_const, string\_const, char\_const, (, !, inc,dec, ter | } |
| <const> | int\_const, float\_const, string\_const, char\_const | Repeating |
| <int\_const> | Int\_const | Repeating |
| <float\_const> | Float\_const | Repeating |
| <str\_const> | String\_const | Repeating |
| <char\_const> | Char\_const | Repeating |
| <AM> | Public, private, € | DT, ID, abstract |
| Body | ter , For, while, do, if, switch, Public, private, DT, ID, abstract, return, break, continue, ID, { | Repeating |
| Return\_stmt | Return | ||, &&, PM, RO, MDM, ROP, ID, int\_const, float\_const, string\_const, char\_const, (, !, inc,dec |
| <assign> | ID | Repeating |
| INC\_DEC | inc, dec | Repeating |
| Loop\_stmt | For, while, do | Repeating |
| Exp | ||, &&, PM, RO, MDM, ROP, ID, int\_const, float\_const, string\_const, char\_const, (, !, inc,dec | Repeating |
| If\_else\_stmt | If | Repeating |
| Switch\_stmt | Switch | Repeating |
| SST | For, while, do, if, switch, Public, private, DT, ID, abstract, return, break, continue, ID | Repeating |
| MST | For, while, do, if, switch, Public, private, DT, ID, abstract, return, break, continue, ID | Repeating |