BNF for C.jj

TOKENS

```
<DEFAULT> SKIP : {
" "
| "\t"
| "\n"
| "\r"
| <"//" (~["\n","\r"])* ("\n" | "\r" | "\r\n")>
| <"/*" (~["*"])* "*" ("*" | ~["*","/"] (~["*"])* "*")* "/">
| "#" : PREPROCESSOR_OUTPUT
}



<PREPROCESSOR_OUTPUT> SKIP : {
"\n" : DEFAULT
}



<PREPROCESSOR_OUTPUT> MORE : {
"\\\n"
| "\\\r\n"
| <~[]>
}



<DEFAULT> TOKEN : {
<INTEGER_LITERAL: <DECIMAL_LITERAL> (["l","L"])? | <HEX_LITERAL> (["l","L"])? | <OCTAL_LITERAL> (["l","L"])?>
| <#DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])*>
| <#HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+>
| <#OCTAL_LITERAL: "0" (["0"-"7"])*>
| <FLOATING_POINT_LITERAL: (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])? | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])? | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])? | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]>
| <#EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+>
| <CHARACTER_LITERAL: "\'" (~["\'","\\","\n","\r"] | "\\" (["n","t","b","r","f","\\","\'","\""] | ["0"-"7"] (["0"-"7"])? | ["0"-"3"] ["0"-"7"] ["0"-"7"])) "\'">
| <STRING_LITERAL: "\"" (~["\"","\\","\n","\r"] | "\\" (["n","t","b","r","f","\\","\'","\""] | ["0"-"7"] (["0"-"7"])? | ["0"-"3"] ["0"-"7"] ["0"-"7"] | ["\n","\r"] | "\r\n"))* "\"">
}



<DEFAULT> TOKEN : {
<CONTINUE: "continue">
| <DFLT: "default">
| <DOUBLE: "double">
| <SWITCH: "switch">
| <RETURN: "return">
| <WHILE: "while">
| <BREAK: "break">
| <FLOAT: "float">
| <ELSE: "else">
| <CASE: "case">
| <LONG: "long">
| <VOID: "void">
| <CHAR: "char">
| <GOTO: "goto">
| <FOR: "for">
| <INT: "int">
| <IF: "if">
| <DO: "do">
}



<DEFAULT> TOKEN : {
<IDENTIFIER: <LETTER> (<LETTER> | <DIGIT>)*>
| <#LETTER: ["$","A"-"Z","_","a"-"z"]>
| <#DIGIT: ["0"-"9"]>
}
```

NON-TERMINALS

```
TranslationUnit ::= ( ExternalDeclaration )+
ExternalDeclaration ::= ( FunctionDefinition | Declaration )
FunctionDefinition ::= ( DeclarationSpecifiers )? Declarator ( DeclarationList )? CompoundStatement
Declaration ::= DeclarationSpecifiers ( InitDeclaratorList )? ";"
DeclarationList ::= ( Declaration )+
DeclarationSpecifiers ::= TypeSpecifier ( DeclarationSpecifiers )?
TypeSpecifier ::= ( <VOID> | <CHAR> | <INT> | <LONG> | <FLOAT> | <DOUBLE> )
InitDeclaratorList ::= InitDeclarator ( "," InitDeclarator )*
InitDeclarator ::= Declarator ( "=" Initializer )?
SpecifierQualifierList ::= TypeSpecifier ( SpecifierQualifierList )?
EnumeratorList ::= Enumerator ( "," Enumerator )*
Enumerator ::= <IDENTIFIER> ( "=" ConstantExpression )?
Declarator ::= ( Pointer )? DirectDeclarator
DirectDeclarator ::= ( <IDENTIFIER> | "(" Declarator ")" ) ( "[" ( ConstantExpression )? "]" | "(" ParameterTypeList ")" | "(" ( IdentifierList )? ")" )*
Pointer ::= "*" ( Pointer )?
ParameterTypeList ::= ParameterList ( "," "..." )?
ParameterList ::= ParameterDeclaration ( "," ParameterDeclaration )*
ParameterDeclaration ::= DeclarationSpecifiers ( Declarator | ( AbstractDeclarator )? )
IdentifierList ::= <IDENTIFIER> ( "," <IDENTIFIER> )*
Initializer ::= ( AssignmentExpression | Initializer1 )
Initializer1 ::= "{" InitializerList ( comma )? "}"
comma ::= ","
InitializerList ::= Initializer ( "," Initializer )*
TypeName ::= SpecifierQualifierList ( AbstractDeclarator )?
AbstractDeclarator ::= ( Pointer | AbstractDeclarator1 )
AbstractDeclarator1 ::= ( Pointer )? DirectAbstractDeclarator
DirectAbstractDeclarator ::= ( "(" AbstractDeclarator ")" | "[" ( ConstantExpression )? "]" | "(" ( ParameterTypeList )? ")" ) ( "[" ( ConstantExpression )? "]" | "(" ( ParameterTypeList )? ")" )*
Statement ::= ( LabeledStatement | ExpressionStatement | CompoundStatement | SelectionStatement | IterationStatement | JumpStatement )
LabeledStatement ::= ( GotoLabel | CaseLabel | DefaultLabel )
GotoLabel ::= <IDENTIFIER> ":" Statement
CaseLabel ::= <CASE> ConstantExpression ":" Statement
DefaultLabel ::= <DFLT> ":" Statement
ExpressionStatement ::= ( Expression )? ";"
CompoundStatement ::= "{" ( DeclarationList )? ( StatementList )? "}"
StatementList ::= ( Statement )+
SelectionStatement ::= ( IfStatement | SwitchStatement )
IfStatement ::= <IF> "(" Expression ")" Statement ( <ELSE> Statement )?
SwitchStatement ::= <SWITCH> "(" Expression ")" Statement
IterationStatement ::= ( WhileStatement | DoWhileStatement | ForStatement )
WhileStatement ::= <WHILE> "(" Expression ")" Statement
DoWhileStatement ::= <DO> Statement <WHILE> "(" Expression ")" ";"
ForStatement ::= <FOR> "(" ( Expression )? ";" ( Expression )? ";" ( Expression )? ")" Statement
JumpStatement ::= ( <GOTO> <IDENTIFIER> ";" | <CONTINUE> ";" | <BREAK> ";" | <RETURN> ( Expression )? ";" )
Expression ::= AssignmentExpression ( "," AssignmentExpression )*
AssignmentExpression ::= UnaryExpression AssignmentOperator AssignmentExpression
                       | ConditionalExpression
AssignmentOperator ::= ( "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | "&=" | "^=" | "|=" )
ConditionalExpression ::= LogicalORExpression ( ConditionalSubExpression )?
ConditionalSubExpression ::= "?" Expression ":" ConditionalExpression
ConstantExpression ::= ConditionalExpression
LogicalORExpression ::= LogicalANDExpression ( "||" LogicalORExpression )?
```

```
LogicalANDExpression ::= InclusiveORExpression ( "&&" LogicalANDExpression )?
InclusiveORExpression ::= ExclusiveORExpression ( "|" InclusiveORExpression )?
ExclusiveORExpression ::= ANDExpression ( "^" ExclusiveORExpression )?
ANDExpression ::= EqualityExpression ( "&" ANDExpression )?
EqualityExpression ::= RelationalExpression ( EqualitySymbols )?
EqualitySymbols ::= EqualityExpression1
                  | EqualityExpression2
EqualityExpression1 ::= "==" EqualityExpression
EqualityExpression2 ::= "!=" EqualityExpression
RelationalExpression ::= ShiftExpression ( RelationalSymbols )?
RelationalSymbols ::= RelationalExpression1
                    | RelationalExpression2
                    | RelationalExpression3
                    | RelationalExpression4
RelationalExpression1 ::= "<" RelationalExpression
RelationalExpression2 ::= ">" RelationalExpression
RelationalExpression3 ::= "<=" RelationalExpression
RelationalExpression4 ::= ">=" RelationalExpression
ShiftExpression ::= AdditiveExpression ( ShiftSymbols )?
ShiftSymbols ::= ShiftExpression1
               | ShiftExpression2
ShiftExpression1 ::= "<<" ShiftExpression
ShiftExpression2 ::= ">>" ShiftExpression
AdditiveExpression ::= MultiplicativeExpression ( AddSymbols )?
AddSymbols ::= AdditiveExpression1
             | AdditiveExpression2
AdditiveExpression1 ::= "+" AdditiveExpression
AdditiveExpression2 ::= "-" AdditiveExpression
MultiplicativeExpression ::= CastExpression ( MulSymbols )?
MulSymbols ::= MultiplicativeExpression1
             | MultiplicativeExpression2
             | MultiplicativeExpression3
MultiplicativeExpression1 ::= "*" MultiplicativeExpression
MultiplicativeExpression2 ::= "/" MultiplicativeExpression
MultiplicativeExpression3 ::= "%" MultiplicativeExpression
CastExpression ::= ( CastExpression1 | UnaryExpression )
CastExpression1 ::= "(" TypeName ")" CastExpression
UnaryExpression ::= ( UnaryExpression1 | UnaryExpression2 | UnaryExpression3 | UnaryExpression4 )
UnaryExpression1 ::= PostfixExpression
UnaryExpression2 ::= "++" UnaryExpression
UnaryExpression3 ::= "--" UnaryExpression
UnaryExpression4 ::= UnaryOperator CastExpression
UnaryOperator ::= ( "&" | "*" | "+" | "-" | "~" | "!" )
PostfixExpression ::= PrimaryExpression ( PostfixExpression1 | "(" ( ArgumentExpressionList )? ")" | PostfixExpression3 | PostfixExpression4 | PostfixExpression5 | PostfixExpression6 )*
PostfixExpression1 ::= "[" Expression "]"
PostfixExpression3 ::= "." <IDENTIFIER>
PostfixExpression4 ::= "->" <IDENTIFIER>
PostfixExpression5 ::= "++"
PostfixExpression6 ::= "--"
PrimaryExpression ::= ( <IDENTIFIER> | Constant | "(" Expression ")" )
ArgumentExpressionList ::= AssignmentExpression ( "," AssignmentExpression )*
Constant ::= <INTEGER_LITERAL>
           | <FLOATING_POINT_LITERAL>
           | <CHARACTER_LITERAL>
           | <STRING_LITERAL>
```