Grammar

A general-purpose language is a programming language that can be used for a wide variety of tasks, from simple scripts to complex applications. Here is our sample grammar for a general-purpose language:

Grammar = (Vt, Vn, P, S) where:

Vt = { identifiers, literals, operators, punctuation }

identifiers: a set of alphanumeric strings starting with a letter or underscore
literals: a set of numeric, string, and boolean values
operators: a set of symbols representing arithmetic, logical, and bitwise operations
punctuation: a set of symbols used for syntax, such as parentheses, braces, and semicolons

Vn = { Program, Statement, Expression, AssignmentExpression, ConditionalExpression, LogicalExpression, BinaryExpression, UnaryExpression, CallExpression, MemberExpression, PrimaryExpression, FunctionExpression, ObjectExpression, ArrayExpression, VariableDeclaration, IfStatement, WhileStatement, ForStatement, FunctionDeclaration, ReturnStatement, BlockStatement, Identifier, Literal, NumericLiteral, StringLiteral, BooleanLiteral, Arguments, Property }

S = Program (the start symbol)

P = {
Program → Statement*
Statement → Expression | VariableDeclaration | IfStatement | WhileStatement | ForStatement | FunctionDeclaration | ReturnStatement
Expression → AssignmentExpression | ConditionalExpression | LogicalExpression | BinaryExpression | UnaryExpression | CallExpression | MemberExpression | Literal
AssignmentExpression → LeftHandSideExpression "=" Expression
ConditionalExpression → LogicalExpression "?" Expression ":" Expression
LogicalExpression → LogicalExpression "&&" LogicalExpression | LogicalExpression "||" LogicalExpression
BinaryExpression → Expression "+" Expression | Expression "-" Expression | Expression "" Expression | Expression "/" Expression | Expression "%" Expression | Expression "==" Expression | Expression "!=" Expression | Expression "<" Expression | Expression ">" Expression | Expression "<=" Expression | Expression ">=" Expression | Expression "<<" Expression | Expression ">>" Expression | Expression "&" Expression | Expression "|" Expression | Expression "^" Expression

UnaryExpression → "!" Expression | "-" Expression | "+" Expression | "~" Expression
CallExpression → MemberExpression Arguments
Arguments → "(" [Expression ("," Expression)] ")"
MemberExpression → PrimaryExpression ("." Identifier | "[" Expression "]")*
PrimaryExpression → Identifier | Literal | "(" Expression ")" | FunctionExpression | ObjectExpression | ArrayExpression
FunctionExpression → "function" [Identifier] "(" [Identifier ("," Identifier)] ")" BlockStatement
ObjectExpression → "{" [Property ("," Property)] "}"
Property → Identifier ":" Expression
ArrayExpression → "[" [Expression ("," Expression)] "]"
VariableDeclaration → "var" Identifier ["=" Expression] ";"
IfStatement → "if" "(" Expression ")" BlockStatement ["else" BlockStatement]
WhileStatement → "while" "(" Expression ")" BlockStatement
ForStatement → "for" "(" [VariableDeclaration | Expression] ";" Expression ";" Expression ")" BlockStatement
FunctionDeclaration → "function" Identifier "(" [Identifier ("," Identifier)] ")" BlockStatement
ReturnStatement → "return" [Expression] ";"
BlockStatement → "{" Statement* "}"
Identifier → /[a-zA-Z_][a-zA-Z0-9_]/
NumericLiteral → /[0-9]+(.[0-9]+)?/
StringLiteral → /"([^"\]|\.)"/
BooleanLiteral → "true" | "false"
}

In this grammar, Vt represents the terminal symbols (tokens) that can be recognized by a lexical analyzer or lexer, while Vn represents the nonterminal symbols (productions) that can be expanded into other symbols or productions. P represents the set of production rules that define how the nonterminal symbols can be expanded or replaced by other symbols or productions. Finally, S represents the start symbol of the grammar.

Some sample derivations:

1. Derivation for VariableDeclaration: VariableDeclaration → "var" Identifier ["=" Expression] ";" → "var" identifiers ["=" Expression] ";" → "var" "x" ["=" Expression] ";" Here, we have used "x" as an example identifier.

2. Derivation for LogicalExpression: LogicalExpression → LogicalExpression "&&" LogicalExpression → LogicalExpression "&&" Expression → LogicalExpression "&&" UnaryExpression → UnaryExpression "&&" UnaryExpression → "!" UnaryExpression "&&" UnaryExpression → "!" Expression "&&" UnaryExpression → "!" PrimaryExpression "&&" UnaryExpression → "!" Identifier "&&" UnaryExpression Here, we have used the "!" operator and an example identifier "x" to derive the logical expression.
3. Derivation for ArrayExpression: ArrayExpression → "[" [Expression ("," Expression)] "]" → "[" Expression ("," Expression) "]" → "[" Identifier ("," Expression) "]" → "[" Identifier "," Expression "]" Here, we have used an example identifier "arr" and an example expression to derive an ArrayExpression.
4. Derivation for FunctionDeclaration: FunctionDeclaration → "function" Identifier "(" [Identifier ("," Identifier)] ")" BlockStatement → "function" "foo" "(" [Identifier ("," Identifier)] ")" BlockStatement Here, we have used "foo" as an example function identifier.
5. Derivation for Program: Program → Statement* → VariableDeclaration ";" Statement* → "var" Identifier ["=" Expression] ";" Statement* → "var" "x" ["=" Expression] ";" Statement* → "var" "x" ["=" Expression] ";" Expression ";" Statement* → "var" "x" ["=" Expression] ";" UnaryExpression ";" Statement* → "var" "x" ["=" Expression] ";" "!" UnaryExpression ";" Statement* → "var" "x" ["=" Expression] ";" "!" Identifier "&&" UnaryExpression ";" Statement* → "var" "x" ["=" Expression] ";" "!" Identifier "&&" PrimaryExpression ";" Statement* → "var" "x" ["=" Expression] ";" "!" Identifier "&&" "(" Expression ")" ";" Statement* → "var" "x" ["=" Expression] ";" "!" Identifier "&&" "(" "x" ")" ";" Statement* → "var" "x" ["=" Expression] ";" "!" Identifier "&&" "(" "x" ")" ";" VariableDeclaration ";" Statement* → "var" "x" ["=" Expression] ";" "!" Identifier "&&" "(" "x" ")" ";" FunctionDeclaration ";" Statement* → "var" "x" ["=" Expression] ";" "!" Identifier "&&" "(" "x" ")" ";" FunctionDeclaration ";" ReturnStatement ";" Statement* → "var" "x" ["=" Expression] ";" "!" Identifier "&&" "(" "x" ")" ";" FunctionDeclaration ";" ReturnStatement ";" VariableDeclaration ";" Statement* Here, we have used a sequence of different statements to derive a Program.

Theoretical example of the grammar within a host language:

```
grammar Language;

program: statement*;
```

```
statement:
      expression
    | variableDeclaration
    | ifStatement
    | whileStatement
    | forStatement
    | foreachStatement
    | switchStatement
    | functionDeclaration
    | returnStatement
    | breakStatement
    | continueStatement
    | tryStatement
    | throwStatement;

expression:
      assignmentExpression
    | conditionalExpression
    | logicalExpression
    | binaryExpression
    | unaryExpression
    | callExpression
    | memberExpression
    | literal;

assignmentExpression:
      leftHandSideExpression '=' expression;

conditionalExpression:
      logicalExpression '?' expression ':' expression;

logicalExpression:
      logicalExpression '&&' logicalExpression
    | logicalExpression '||' logicalExpression;

binaryExpression:
      expression '+' expression
    | expression '-' expression
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
    | expression '==' expression
```

```
    | expression '!=' expression
    | expression '<' expression
    | expression '>' expression
    | expression '<=' expression
    | expression '>=' expression
    | expression '<<' expression
    | expression '>>' expression
    | expression '&' expression
    | expression '|' expression
    | expression '^' expression;

unaryExpression:
       '!' expression
    | '-' expression
    | '+' expression
    | '~' expression;

callExpression:
     memberExpression arguments;

arguments:
     '(' expression (',' expression)* ')';

memberExpression:
     primaryExpression ('.' identifier | '[' expression ']')*;

primaryExpression: identifier
    | literal
    | '(' expression ')'
    | functionExpression
    | objectExpression
    | arrayExpression;

functionExpression:
     'function' (identifier)? '(' (identifier (',' identifier)*)?
')' blockStatement;

objectExpression:
     '{' (property (',' property)*)? '}';

property:
     identifier ':' expression;

arrayExpression:
```

```
      '[' (expression (',' expression)*)? ']';

variableDeclaration:
      'var' identifier ('=' expression)? ';';

ifStatement:
      'if' '(' expression ')' blockOrStatement ('else'
blockOrStatement)?;

whileStatement:
      'while' '(' expression ')' blockOrStatement;

forStatement:
      'for' '(' (variableDeclaration | expression)? ';' expression
';' expression ')' blockOrStatement;

foreachStatement:
      'foreach' '(' identifier 'in' expression ')' blockOrStatement;

switchStatement:
      'switch' '(' expression ')' '{' switchCases '}' ;

switchCases:
      switchCase* defaultCase?;

switchCase:
      'case' expression ':' blockOrStatement;

defaultCase:
      'default' ':' blockOrStatement;

functionDeclaration:
      'function' identifier '(' (identifier (',' identifier)*)? ')'
blockOrStatement;

returnStatement:
      'return' expression? ';';

breakStatement:
      'break' ';';

continueStatement:
```

```
      'continue' ';';

tryStatement:
      'try' blockOrStatement catchClause;

catchClause:
      'catch' '(' identifier ')' blockOrStatement;

throwStatement:
      'throw' expression ';';

blockOrStatement:
      Block

identifier:
      /[a-zA-Z_][a-zA-Z0-9_]*/;

literal:
      numericLiteral
    | stringLiteral
    | booleanLiteral;

numericLiteral:
      /[0-9]+(\.[0-9]+)?/;

stringLiteral:
      /"([^"]|\.)+"/;

booleanLiteral:
      'true' | 'false';

WS:
      [ \t\n\r]+ -> skip;
```

The WS rule is used to skip whitespace in the input.