CS335A (Compiler Design)

Assignment 0

Group Members

Name	Roll No	IITK Email
P Sughosh	14441	sughosh@iitk.ac.in
Sachin K Salim	14575	sachink@iitk.ac.in
Shivam Yadav	14655	shivamy@iitk.ac.in

T-Diagram

New rules

No new syntactic rules are added to the grammar since Javascript already has sufficient features. But few rules are deleted so that project can be completed in time.

Tools

We will be using PLY (Python Lex-Yacc) as a lexer and parser.

Source for BNF

https://github.com/antlr/grammars-v4/tree/master/javascript

Our Javascript EBNF

```
program
    : statement*
    ;

statement
    : block
    | ';'
    | variableStatement
    | expressionStatement
    | ifStatement
    | iterationStatement
    | continueStatement
    | breakStatement
    | returnStatement
    | withStatement
    | switchStatement
    | functionDeclaration
    ;

block
    : '{' statement* '}'
    ;

variableStatement
    : 'var' variableDeclarationList ';'
    ;

variableDeclarationList
    : variableDeclaration (',' variableDeclaration)*
    ;

variableDeclaration
    : (Identifier | arrayLiteral | objectLiteral) ('='
singleExpression)?
    ;

expressionStatement
    : expressionSequence ';'
    ;

ifStatement
    : 'if' '(' expressionSequence ')' statement ('else' statement)?
    ;
```

```
iterationStatement
    : 'do' statement 'while' '(' expressionSequence ')' ';'
    | 'while' '(' expressionSequence ')' statement
    | 'for' '(' expressionSequence? ';' expressionSequence? ';'
expressionSequence? ')' statement
    | 'for' '(' 'var' variableDeclarationList ';' expressionSequence?
';' expressionSequence? ')' statement
    | 'for' '(' singleExpression 'in' expressionSequence ')'
statement
    | 'for' '(' 'var' variableDeclaration 'in' expressionSequence ')'
statement
    ;

continueStatement
    : 'continue' (Identifier)? ';'
    ;

breakStatement
    : 'break' (Identifier)? ';'
    ;

returnStatement
    : 'return' (expressionSequence)? ';'
    ;

withStatement
    : 'with' '(' expressionSequence ')' statement
    ;

switchStatement
    : 'switch' '(' expressionSequence ')' caseBlock
    ;

caseBlock
    : '{' caseClauses? (defaultClause caseClauses?)? '}'
    ;

caseClauses
    : caseClause+
    ;

caseClause
    : 'case' expressionSequence ':' statement*
    ;

defaultClause
    : Default ':' statement*
    ;
```

```
functionDeclaration
    : 'function' Identifier '(' formalParameterList? ')' '{'
functionBody '}'
    ;

functionBody
    : sourceElements?
    ;

arrayLiteral
    : '[' ','* elementList? ','* ']'
    ;

elementList
    : singleExpression (','+ singleExpression)*
    ;

arguments
    : '('(
          singleExpression (',' singleExpression)*
    )?')'
    ;

expressionSequence
    : singleExpression (',' singleExpression)*
    ;

singleExpression
    : 'function' Identifier? '(' formalParameterList? ')' '{'
functionBody '}'
    | singleExpression '[' expressionSequence ']'
    | singleExpression '.' identifierName
    | singleExpression arguments
    | 'eval' '(' program ')'
    | 'new' singleExpression arguments?
    | singleExpression ('++' | '--')
    | 'delete' singleExpression
    | 'void' singleExpression
    | Typeof singleExpression
    | ('++'| '--' | '+' | '-' | '~' | '!') singleExpression
    | singleExpression ('+' | '-' | '*' | '/' | '%') singleExpression
    | singleExpression ('<<' | '>>' | '>>>') singleExpression    |
singleExpression ('<' | '>' | '<=' | '>=') singleExpression
    | singleExpression 'in' singleExpression
    | singleExpression ('==' | '!=' | '===' | '!==') singleExpression
    | singleExpression ('&' | '^' | '|' | '&&' | '||')
singleExpression
```

```
    | singleExpression '?' singleExpression ':' singleExpression
    | singleExpression '=' singleExpression
    | singleExpression assignmentOperator singleExpression
    | 'this'
    | 'undefined'
    | Identifier
    | literal
    | arrayLiteral
    | objectLiteral
    | '(' expressionSequence ')'
    ;

assignmentOperator
    : '*='
    | '/='
    | '%='
    | '+='
    | '-='
    | '<<='
    | '>>='
    | '>>>='
    | '&='
    | '^='
    | '|='
    ;

literal
    : NullLiteral
    | BooleanLiteral
    | StringLiteral
    | TemplateStringLiteral
    | DecimalLiteral
    ;

objectLiteral
    : '{' (propertyAssignment (',' propertyAssignment)*)? ','? '}'
    ;

propertyAssignment
    : propertyName (':' |'=') singleExpression
    | '[' singleExpression ']' ':' singleExpression
    | Identifier
    ;

propertyName
    : identifierName
    | StringLiteral
    | DecimalLiteral
```

```
    ;

identifierName
    : Identifier
    | reservedWord
    ;

reservedWord
    : keyword
    | NullLiteral
    | BooleanLiteral
    ;

Identifier
    : IdentifierStart IdentifierPart*;

IdentifierStart
    : UnicodeLetter
    | [$_]
    | '\\' UnicodeEscapeSequence
    ;
IdentifierPart
    : IdentifierStart
    | UnicodeCombiningMark
    | UnicodeDigit
    | UnicodeConnectorPunctuation
    | ZWNJ
    | ZWJ
    ;

StringLiteral
    : '"' DoubleStringCharacter* '"'
    | '\'' SingleStringCharacter* '\''
    ;

DoubleStringCharacter
    : ~["\\\r\n]
    | '\\' EscapeSequence
    | LineContinuation
    ;

SingleStringCharacter
    : ~['\\\r\n]
    | '\\' EscapeSequence
    | LineContinuation
    ;

TemplateStringLiteral
```

```
    : '`' ('\\`' | ~'`')* '`';

BooleanLiteral
    : 'true'
    | 'false';

NullLiteral
    : 'null';

DecimalLiteral
    : DecimalIntegerLiteral '.' [0-9]* ExponentPart?
    | '.' [0-9]+ ExponentPart?
    | DecimalIntegerLiteral ExponentPart?
    ;

DecimalIntegerLiteral
    : '0'
    | [1-9] [0-9]*
    ;

ExponentPart
    : [eE] [+-]? [0-9]+
    ;

keyword
    : 'break'
    | 'case'
    | 'console'
    | 'continue'
    | 'delete'
    | 'do'
    | 'else'
    | 'eval'
    | 'for'
    | 'function'
    | 'if'
    | 'in'
    | 'log'
    | 'new'
    | 'return'
    | 'switch'
    | 'this'
    | 'typeof'
    | 'undefined'
    | 'var'
    | 'void'
    | 'while'
    | 'with'
```

;

Deleted rules

```
program
    : sourceElements? EOF
    ;

sourceElements
    : sourceElement+
    ;

sourceElement
    : Export statement
    ;

statement
    : labelledStatement
    | throwStatement
    | tryStatement
    | debuggerStatement
    | classDeclaration
    ;

varModifier  // let, const - ECMAScript 6
    : Let
    | Const
    ;

labelledStatement
    : Identifier ':' statement
    ;

throwStatement
    : Throw {notLineTerminator()}? expressionSequence eos
    ;

tryStatement
    : Try block (catchProduction finallyProduction? |
finallyProduction)
    ;

catchProduction
    : Catch '(' Identifier ')' block
    ;

finallyProduction
    : Finally block
```

```
    ;

debuggerStatement
    : Debugger eos
    ;

classDeclaration
    : Class Identifier classTail
    ;

classTail
    : (Extends singleExpression)? '{' classElement* '}'
    ;

classElement
    : Static? methodDefinition
    ;

methodDefinition
    : propertyName '(' formalParameterList? ')' '{' functionBody '}'
    | getter '(' ')' '{' functionBody '}'
    | setter '(' formalParameterList? ')' '{' functionBody '}'
    | generatorMethod
    ;

generatorMethod
    : '*'? Identifier '(' formalParameterList? ')' '{' functionBody
'}'
    ;

formalParameterList
    : formalParameterArg (',' formalParameterArg)* (','
lastFormalParameterArg)?
    | lastFormalParameterArg
    | arrayLiteral                              // ECMAScript 6:
Parameter Context Matching
    | objectLiteral                             // ECMAScript 6:
Parameter Context Matching
    ;

formalParameterArg
    : Identifier ('=' singleExpression)?      // ECMAScript 6:
Initialization
    ;

lastFormalParameterArg                          // ECMAScript 6: Rest
Parameter
    : Ellipsis Identifier
```

```
    ;

elementList
    : singleExpression (','+ lastElement)
    | lastElement
    ;

lastElement                         // ECMAScript 6: Spread Operator
    : Ellipsis Identifier
    ;



propertyAssignment
    : getter '(' ')' '{' functionBody '}'             # PropertyGetter
    | setter '(' Identifier ')' '{' functionBody '}' # PropertySetter
    | generatorMethod                                 # MethodProperty
    ;

lastArgument                                    // ECMAScript 6: Spread
Operator
    : Ellipsis Identifier
    ;

singleExpression
    : Class Identifier? classTail
# ClassExpression
    | singleExpression Instanceof singleExpression
# InstanceofExpression
    | Super
# SuperExpression
    | arrowFunctionParameters '=>' arrowFunctionBody
# ArrowFunctionExpression   // ECMAScript 6
    ;

arrowFunctionParameters
    : Identifier
    | '(' formalParameterList? ')'
    ;

arrowFunctionBody
    : singleExpression
    | '{' functionBody '}'
    ;

literal
    : numericLiteral
    ;
```

```
numericLiteral
    : HexIntegerLiteral
    | OctalIntegerLiteral
    | OctalIntegerLiteral2
    | BinaryIntegerLiteral
    ;

keyword
    : Instanceof
    | Catch
    | Finally
    | Debugger
    | Default
    | Throw
    | Try
    | Class
    | Enum
    | Extends
    | Super
    | Const
    | Export
    | Import
    | Implements
    | Let
    | Private
    | Public
    | Interface
    | Package
    | Protected
    | Static
    | Yield
    ;

getter
    : Identifier{p("get")}? propertyName
    ;

setter
    : Identifier{p("set")}? propertyName
    ;

eos
    : EOF
    | {lineTerminatorAhead()}?
    | {closeBrace()}?
    ;
```