COMPILERS PROJECT PHASE 2 REPORT

DATA STRUCTURES & ALGORITHMS USED

CFG PARSER CLASS

The class responsible for eliminating left recursion and left factoring of the input grammar.

MEMBERS

- **`m_Grammar`**: The grammar after eliminating left recursion and left factoring.

METHODS

- **`eliminateLeftRecursion()`**: The method to eliminate immediate and non-immediate left recursion from the grammar according to the following algorithm:

```
order non terminals A_1, A_2,..., A_n
for i from 1 to n:
    for j from 1 to i - 1:
        replace each production A_i -> A_j gamma
             by A_i -> alpha_1 gamma | ... | alpha_k gamma
        where A_j -> alpha_1 | ... | alpha_k

        // eliminate immediate left recursion in A_i
        for production in non-recursive productions of A_i:
            production = production + A_i'

        for production in recursive productions of A_i':
            production = production + A_i'
        add epsilon production to A_i'
```

- **`leftFactor()`**: Left factor the grammar where the suffixes of all common prefixes of the productions of a non-terminal factored out and are considered the productions of a new non-terminal.
- **`getGrammar()`**: Returns **`m_Grammar`**.

PARSE TABLE GENERATOR CLASS

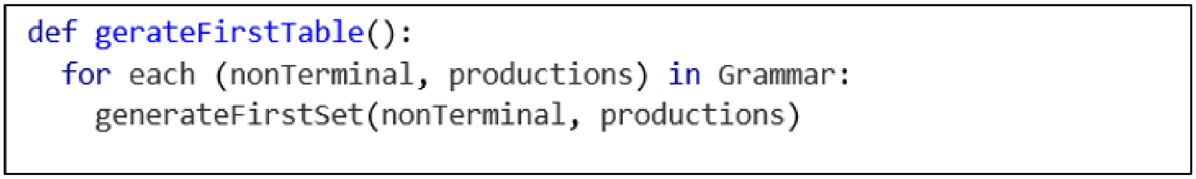
The class is responsible for generating the first set, and the following set of terminals is also responsible for generating the parse table.

MEMBERS

- `m_Grammar`:
    - **type:** `map<string, vector<vector<Symbol>>>`
    - **desc:** The grammar received from the CFG Parser class.
- `m_FirstTable`:
    - **type:** `unordered_map<string, unordered_set<string>>`
    - **desc:** a table containing each terminal with their first set.
- `m_FollowTable`:
    - **type:** `unordered_map<string, unordered_set<string>>`
    - **desc:** a table containing each terminal with their follow set.
- `m_ParseTable`:
    - **type:** `unordered_map<string,unordered_map<string,vector<Symbol>>>`
    - **desc:** a table containing the production rules used at each transition.
- `m_ProductionFirstTable`:
    - **type:** `map<vector<Symbol>, unordered_set<string>>`
    - **desc:** a table containing the first set of each production, used when generating the parse table.

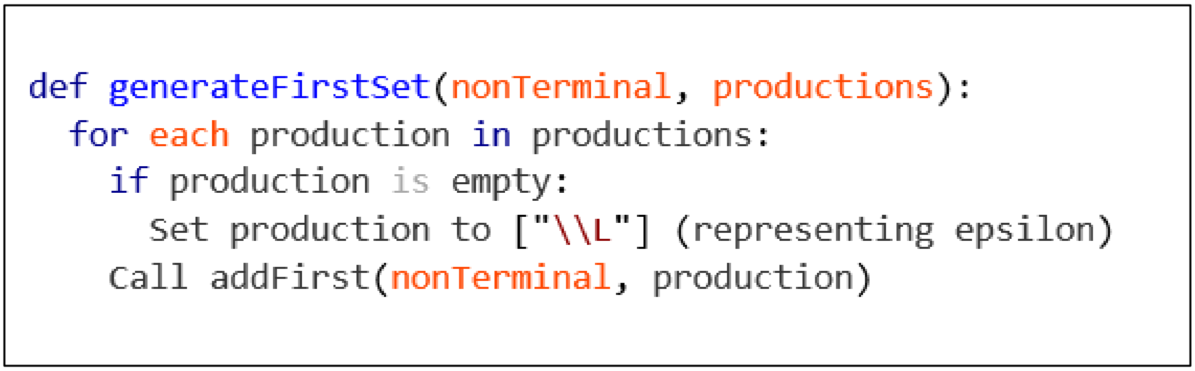
METHODS

- `generateFirstTable()`:
    - Iterates through all non-terminals in the grammar.
    - Calls generateFirstSet() to compute the FIRST set for each non-terminal.



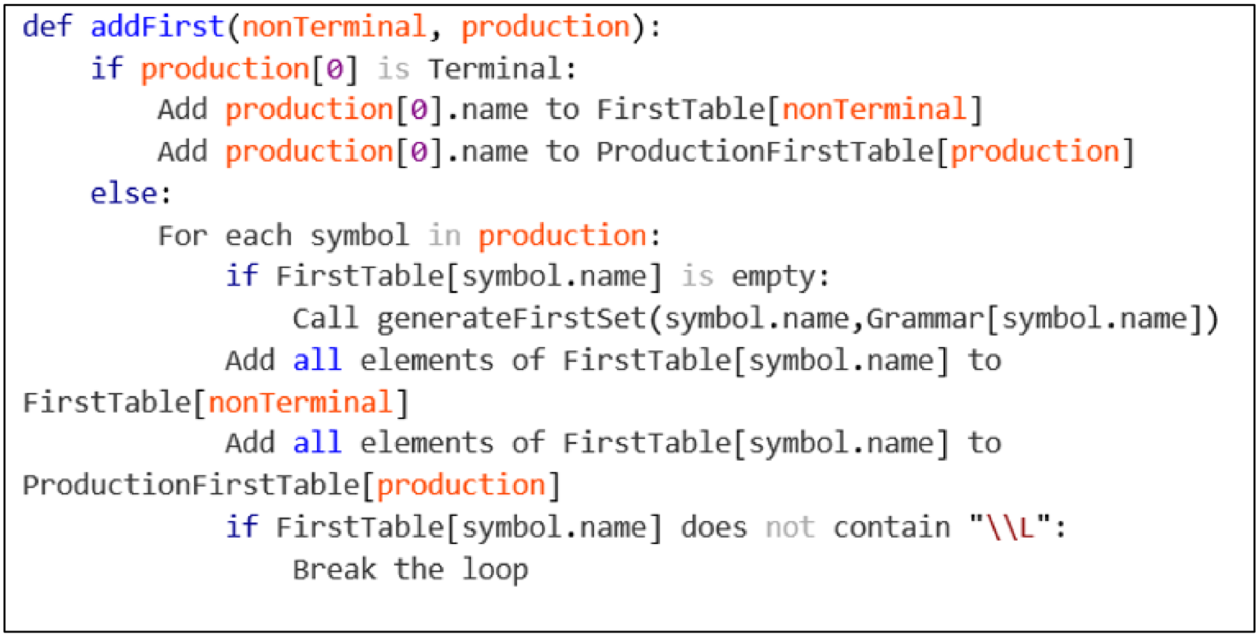
```
def gerateFirstTable():
  for each (nonTerminal, productions) in Grammar:
    generateFirstSet(nonTerminal, productions)
```

- `generateFirstSet(nonTerminal, productions)`:
    - Processes all productions for a given non-terminal.
    - Ensures that the FIRST set is updated for each production by calling addFirst()



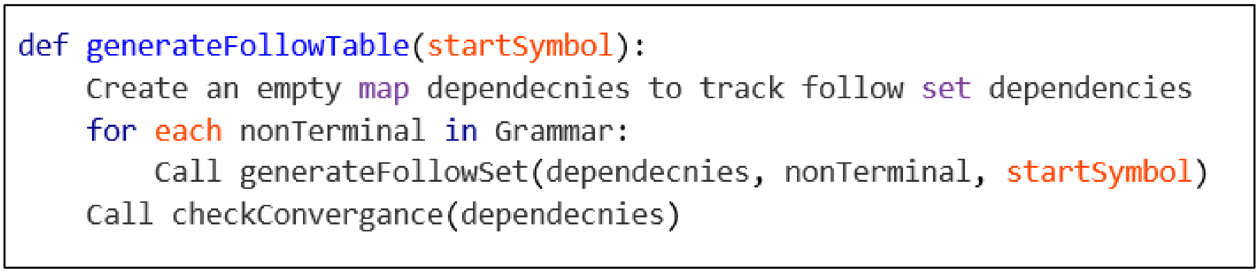
```
def generateFirstSet(nonTerminal, productions):
  for each production in productions:
    if production is empty:
      Set production to ["\\L"] (representing epsilon)
    Call addFirst(nonTerminal, production)
```

- `addFirst(nonTerminal, production):`
  - Determines the FIRST set for a single production.
  - If the first symbol in the production is a terminal, it is directly added to the FIRST set.
  - If the first symbol is a non-terminal, recursively computes its FIRST set and propagates it.
  - Handles the presence of \L (epsilon), ensuring the computation continues only if \L is in the FIRST set of the current symbol.



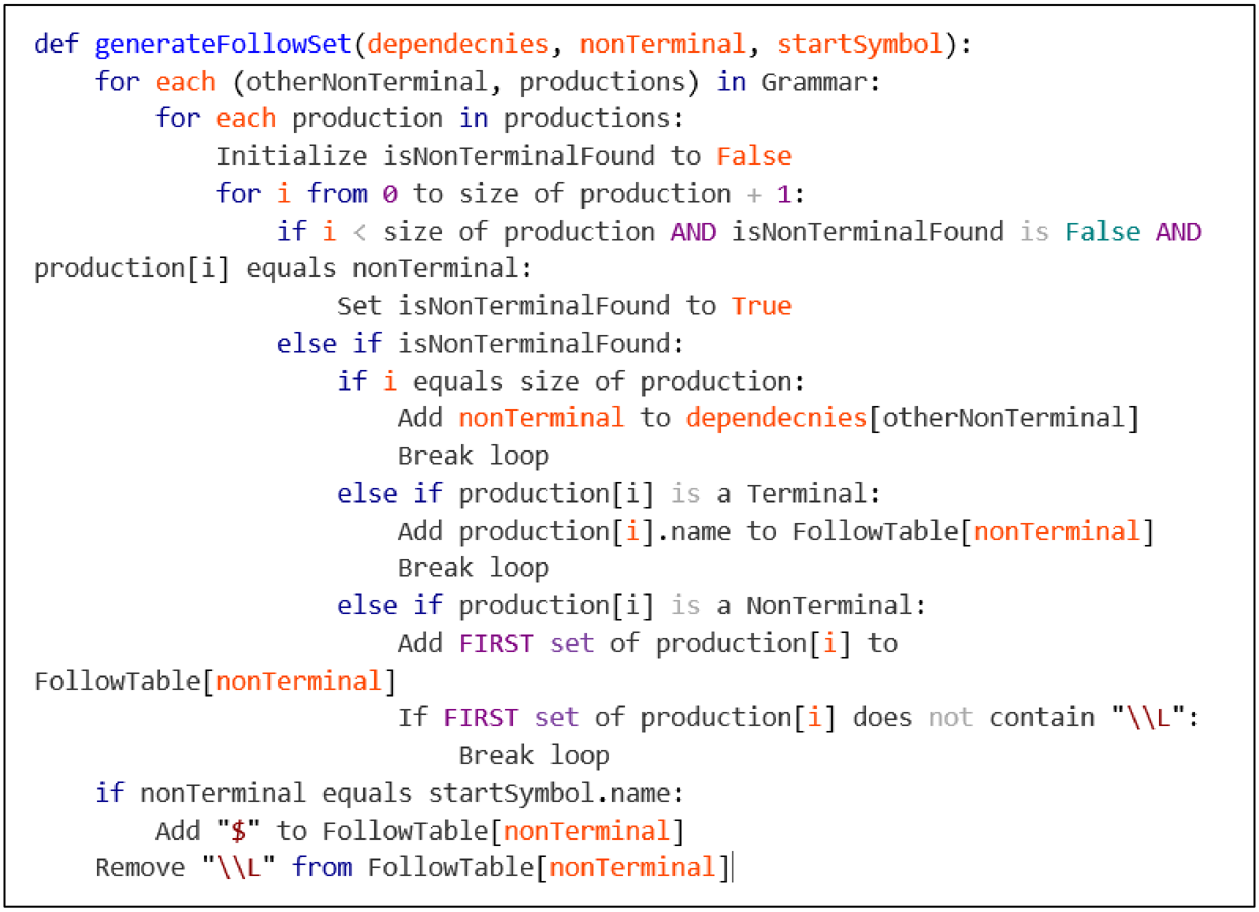
```
def addFirst(nonTerminal, production):
    if production[0] is Terminal:
        Add production[0].name to FirstTable[nonTerminal]
        Add production[0].name to ProductionFirstTable[production]
    else:
        For each symbol in production:
            if FirstTable[symbol.name] is empty:
                Call generateFirstSet(symbol.name,Grammar[symbol.name])
            Add all elements of FirstTable[symbol.name] to FirstTable[nonTerminal]
            Add all elements of FirstTable[symbol.name] to ProductionFirstTable[production]
            if FirstTable[symbol.name] does not contain "\\L":
                Break the loop
```

- `generateFollowTable(Symbol startSymbol):`
  - Orchestrates the computation of FOLLOW sets.
  - Initializes dependencies for FOLLOW set propagation.
  - Calls checkConvergance to ensure the FOLLOW sets stabilize.



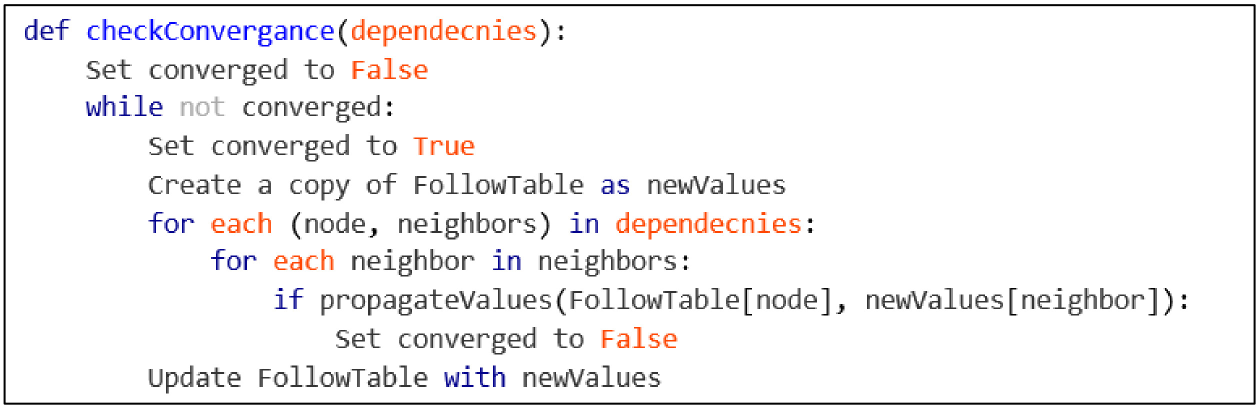
```
def generateFollowTable(startSymbol):
    Create an empty map dependecnies to track follow set dependencies
    for each nonTerminal in Grammar:
        Call generateFollowSet(dependecnies, nonTerminal, startSymbol)
    Call checkConvergance(dependecnies)
```

- `generateFollowSet(dependencies nonTerminal,startSymbol):`
  - Finds where the given nonTerminal appears in productions of other non-terminals.
  - Updates FOLLOW sets based on:
    - Terminals following the nonTerminal.
    - FIRST sets of subsequent non-terminals.
  - Tracks dependencies when the nonTerminal is at the end of a production.



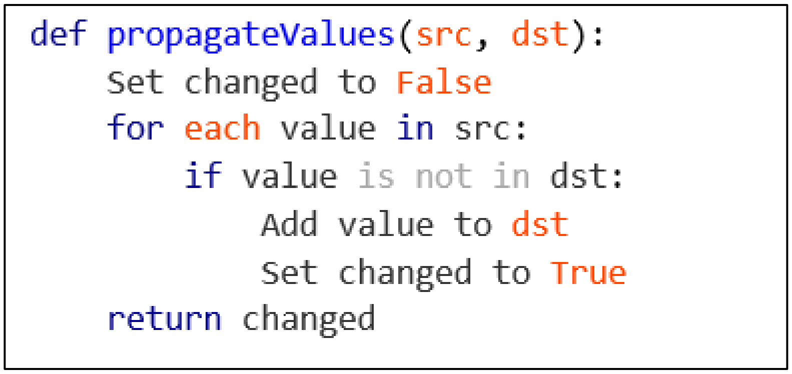
```
def generateFollowSet(dependecnies, nonTerminal, startSymbol):
    for each (otherNonTerminal, productions) in Grammar:
        for each production in productions:
            Initialize isNonTerminalFound to False
            for i from 0 to size of production + 1:
                if i < size of production AND isNonTerminalFound is False AND production[i] equals nonTerminal:
                    Set isNonTerminalFound to True
                else if isNonTerminalFound:
                    if i equals size of production:
                        Add nonTerminal to dependecnies[otherNonTerminal]
                        Break loop
                    else if production[i] is a Terminal:
                        Add production[i].name to FollowTable[nonTerminal]
                        Break loop
                    else if production[i] is a NonTerminal:
                        Add FIRST set of production[i] to FollowTable[nonTerminal]
                        If FIRST set of production[i] does not contain "\\L":
                            Break loop
    if nonTerminal equals startSymbol.name:
        Add "$" to FollowTable[nonTerminal]
    Remove "\\L" from FollowTable[nonTerminal]
```

- `checkConvergance(dependencies)`: Iteratively propagates FOLLOW set updates across dependent non-terminals until no changes occur (convergence).



```
def checkConvergance(dependecnies):
    Set converged to False
    while not converged:
        Set converged to True
        Create a copy of FollowTable as newValues
        for each (node, neighbors) in dependecnies:
            for each neighbor in neighbors:
                if propagateValues(FollowTable[node], newValues[neighbor]):
                    Set converged to False
        Update FollowTable with newValues
```

- `propagateValues(src,dst)`: Copies elements from one FOLLOW set to another and checks if any new elements were added.



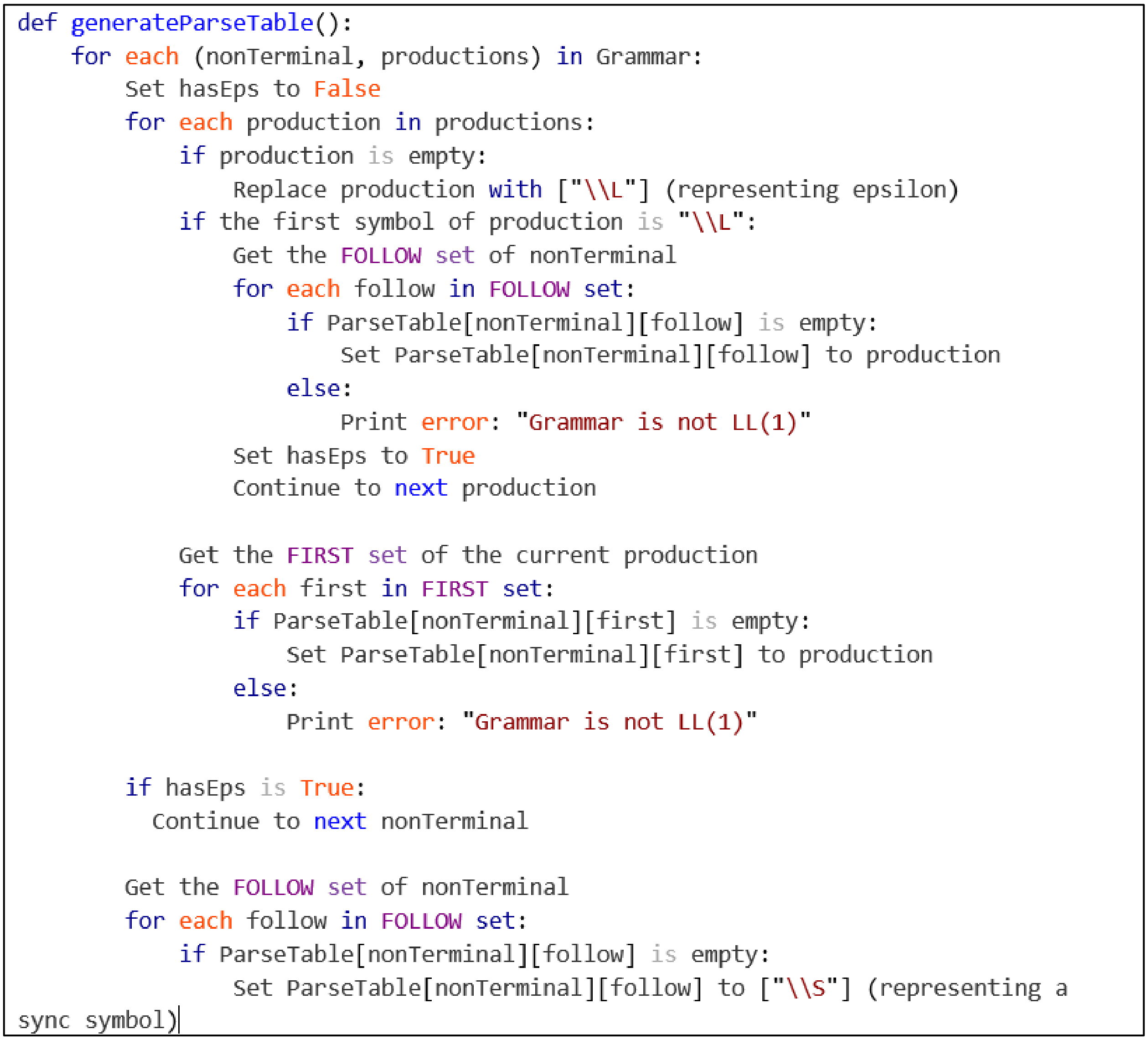
```
def propagateValues(src, dst):
    Set changed to False
    for each value in src:
        if value is not in dst:
            Add value to dst
            Set changed to True
    return changed
```

- `generateParseTable()`:

  This function generates the LL(1) Parse Table for the grammar by following these steps:

  1. Iterate Over Grammar: For each non-terminal and its productions:
     a. Handle empty productions by treating them as epsilon (\L).
     b. Check if epsilon is part of the production and update the parse table using the FOLLOW set.
  2. Update Parse Table Using FIRST Sets:
     a. For each production, add it to the parse table for all terminals in its FIRST set.
     b. If a conflict occurs (i.e., multiple entries for the same cell in the parse table), report an error indicating the grammar is not LL(1).
  3. Handle Synchronization (Sync) Symbols: If no epsilon exists for a non-terminal, populate the parse table using the FOLLOW set with a sync symbol (\S).

  It also Detects and reports conflicts when the grammar is not LL(1).



```
def generateParseTable():
    for each (nonTerminal, productions) in Grammar:
        Set hasEps to False
        for each production in productions:
            if production is empty:
                Replace production with ["\\L"] (representing epsilon)
            if the first symbol of production is "\\L":
                Get the FOLLOW set of nonTerminal
                for each follow in FOLLOW set:
                    if ParseTable[nonTerminal][follow] is empty:
                        Set ParseTable[nonTerminal][follow] to production
                    else:
                        Print error: "Grammar is not LL(1)"
                Set hasEps to True
                Continue to next production

            Get the FIRST set of the current production
            for each first in FIRST set:
                if ParseTable[nonTerminal][first] is empty:
                    Set ParseTable[nonTerminal][first] to production
                else:
                    Print error: "Grammar is not LL(1)"

        if hasEps is True:
          Continue to next nonTerminal

        Get the FOLLOW set of nonTerminal
        for each follow in FOLLOW set:
            if ParseTable[nonTerminal][follow] is empty:
                Set ParseTable[nonTerminal][follow] to ["\\S"] (representing a
sync symbol)
```

PARSER CLASS

The class responsible for parsing the tokens produced by the lexical analyser according to the parse table.

MEMBERS

- **`m_ParseTable:`** The parse table generated by the ParseTableGenerator class which contains a production rule to be done when receiving a terminal symbol when a non-terminal symbol is at the stack top.
- **`m_LexicalAnalyzer:`** The lexical analyser from which the terminal tokens are produced.
- **`m_Stack:`** The stack to hold the grammar symbols. Initially contains '$' and the starting symbol of the grammar.
- **`m_Outputs:`** The list of outputs that represent the left derivation of the input tokens.
- **`m_Finished:`** A boolean to indicate when the parser is done.

METHODS

- **`parseNextToken():`** The main method of the Parser class where the next token is extracted from the lexical analyser and is parsed according to the following algorithm:

```
if symbol at top of m_Stack is non-terminal:
      production = m_ParseTable[symbol][token]

      if production is empty:
            report error
            ignore token
            return

      if production is synch:
            report error
            pop off stack

      pop off stack
      push production onto stack in reverse
      replace the non-terminal in the previous output with production

else if symbol at top of m_Stack is terminal:
      if symbol is epsilon:
            pop off stack
      else if symbol is '$':
            pop off stack
            m_Finsihed = true
            return
      else if symbol matches token:
            pop off stack
            return
      else if symbol doesn't match token:
            report error
            insert symbol into input and match
            pop off stack
```

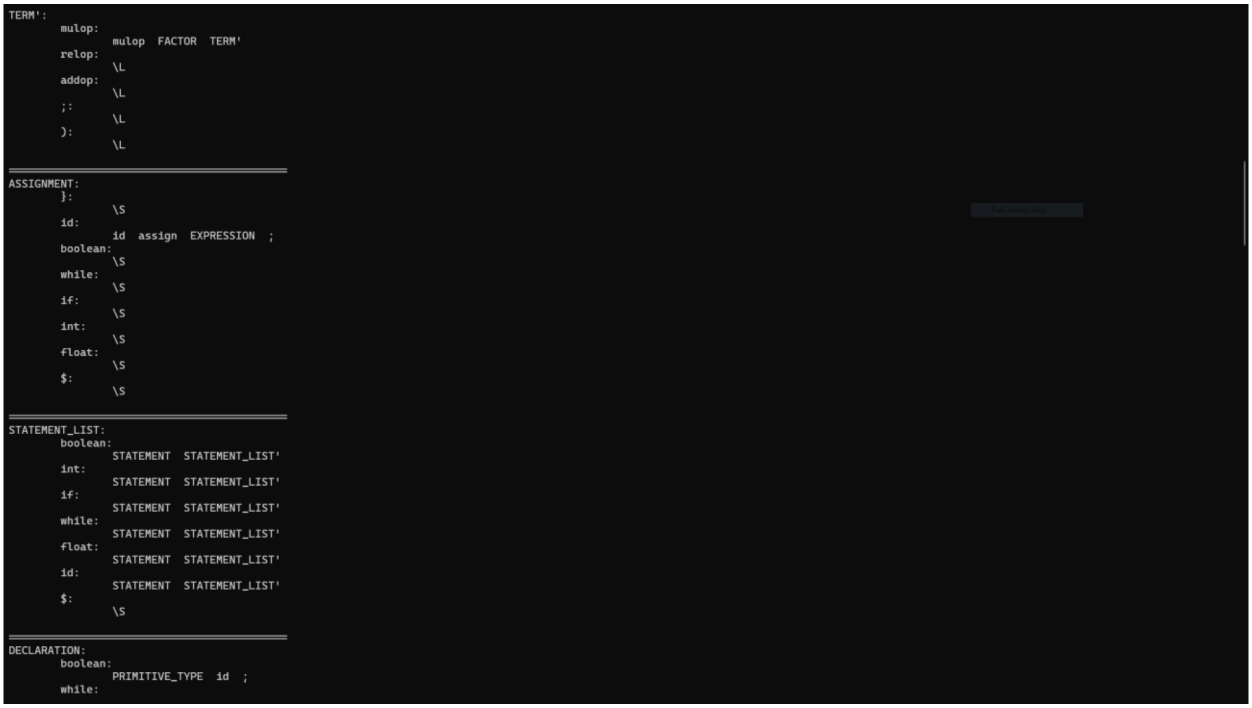
- `isFinished()`: Returns true if the parser is finished parsing the input from the lexical analyser.
- `getOutputs()`: Returns the left derivation of the input from the lexical analyser.

USAGE

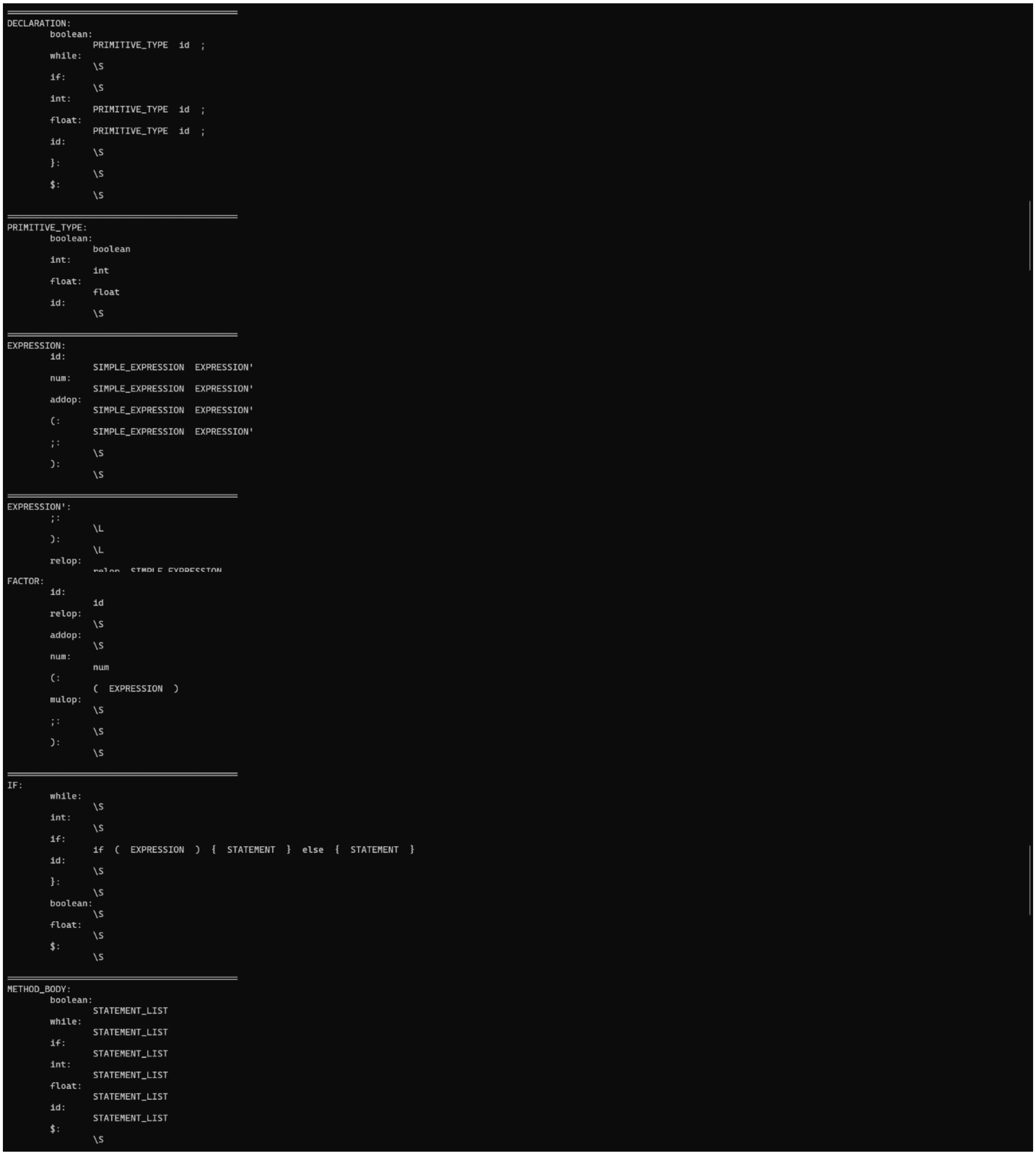
This class should be used as following:

```
while (!parser.isFinished())
       parser.parseNextToken();

std::vector<std::string> leftDerivation = parser.getOutputs();
```

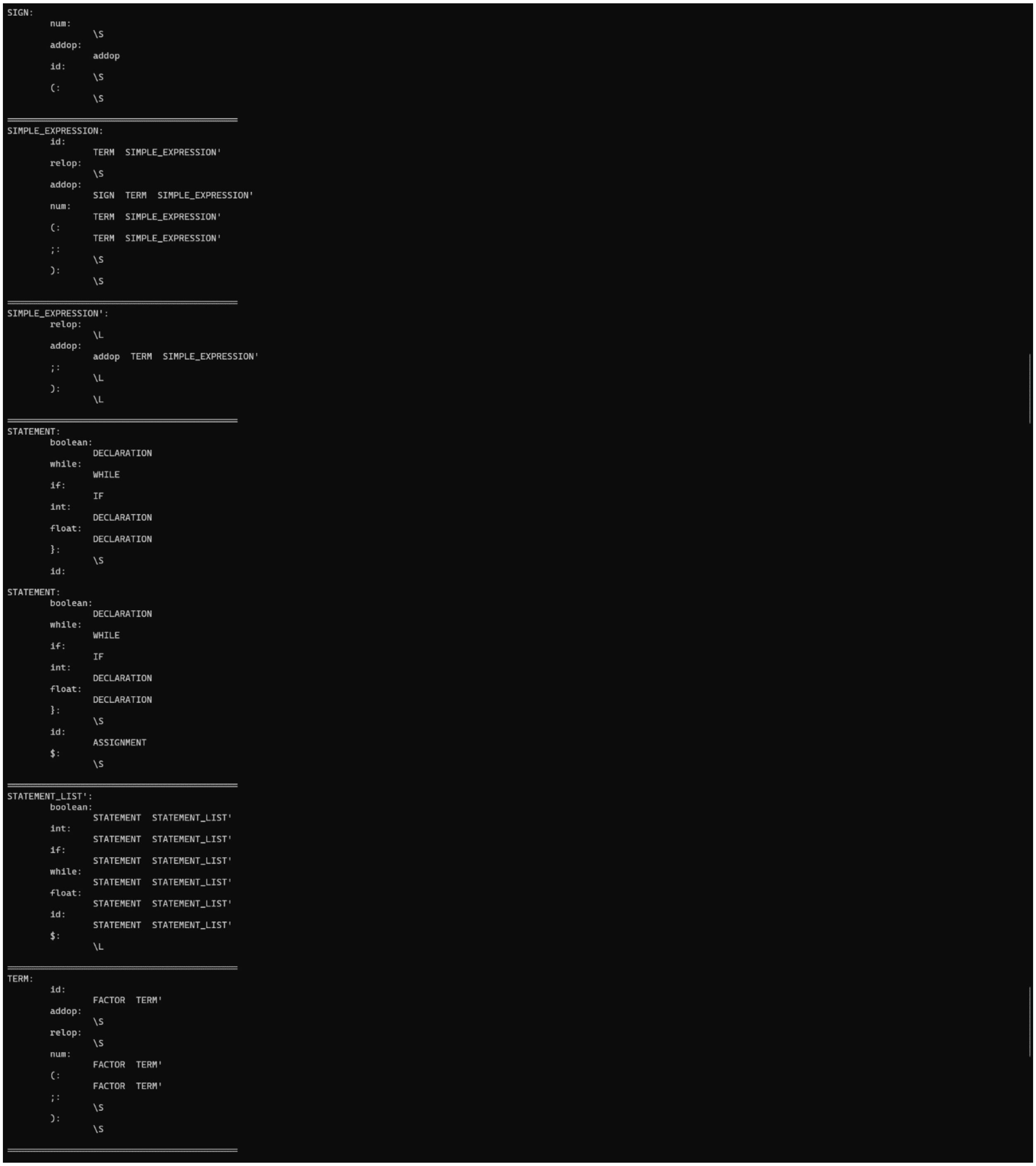
RESULTANT PARSE TABLE



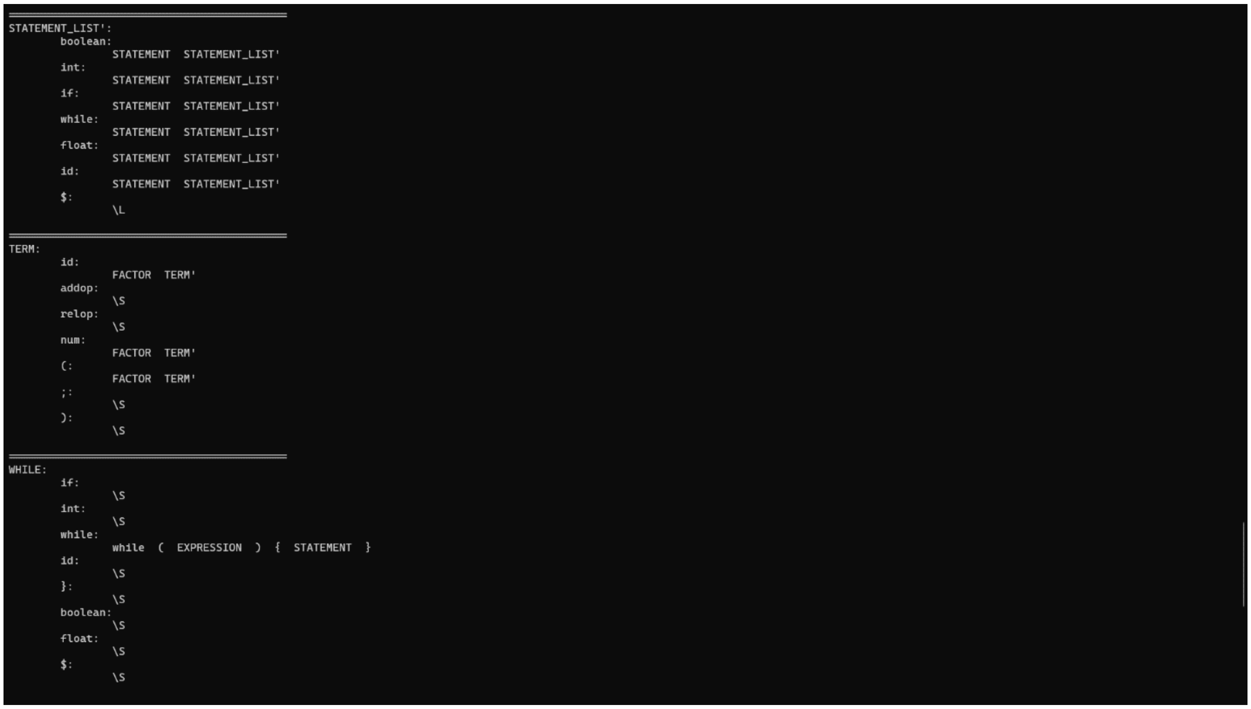
```
TERM':
        mulop:
                mulop  FACTOR  TERM'
        relop:
                \L
        addop:
                \L
        ;:
                \L
        ):
                \L

════════════════════════════════════════════
ASSIGNMENT:
        }:
                \S
        id:
                id  assign  EXPRESSION  ;
        boolean:
                \S
        while:
                \S
        if:
                \S
        int:
                \S
        float:
                \S
        $:
                \S

════════════════════════════════════════════
STATEMENT_LIST:
        boolean:
                STATEMENT  STATEMENT_LIST'
        int:
                STATEMENT  STATEMENT_LIST'
        if:
                STATEMENT  STATEMENT_LIST'
        while:
                STATEMENT  STATEMENT_LIST'
        float:
                STATEMENT  STATEMENT_LIST'
        id:
                STATEMENT  STATEMENT_LIST'
        $:
                \S

════════════════════════════════════════════
DECLARATION:
        boolean:
                PRIMITIVE_TYPE  id  ;
        while:
```



```
═══════════════════════════════════════════
DECLARATION:
        boolean:
                PRIMITIVE_TYPE  id  ;
        while:
                \S
        if:
                \S
        int:
                PRIMITIVE_TYPE  id  ;
        float:
                PRIMITIVE_TYPE  id  ;
        id:
                \S
        }:
                \S
        $:
                \S

═══════════════════════════════════════════
PRIMITIVE_TYPE:
        boolean:
                boolean
        int:
                int
        float:
                float
        id:
                \S

═══════════════════════════════════════════
EXPRESSION:
        id:
                SIMPLE_EXPRESSION  EXPRESSION'
        num:
                SIMPLE_EXPRESSION  EXPRESSION'
        addop:
                SIMPLE_EXPRESSION  EXPRESSION'
        (:
                SIMPLE_EXPRESSION  EXPRESSION'
        ;:
                \S
        ):
                \S

═══════════════════════════════════════════
EXPRESSION':
        ;:
                \L
        ):
                \L
        relop:
                relop  SIMPLE_EXPRESSION
FACTOR:
        id:
                id
        relop:
                \S
        addop:
                \S
        num:
                num
        (:
                (  EXPRESSION  )
        mulop:
                \S
        ;:
                \S
        ):
                \S

═══════════════════════════════════════════
IF:
        while:
                \S
        int:
                \S
        if:
                if  (  EXPRESSION  )  {  STATEMENT  }  else  {  STATEMENT  }
        id:
                \S
        }:
                \S
        boolean:
                \S
        float:
                \S
        $:
                \S

═══════════════════════════════════════════
METHOD_BODY:
        boolean:
                STATEMENT_LIST
        while:
                STATEMENT_LIST
        if:
                STATEMENT_LIST
        int:
                STATEMENT_LIST
        float:
                STATEMENT_LIST
        id:
                STATEMENT_LIST
        $:
                \S
```

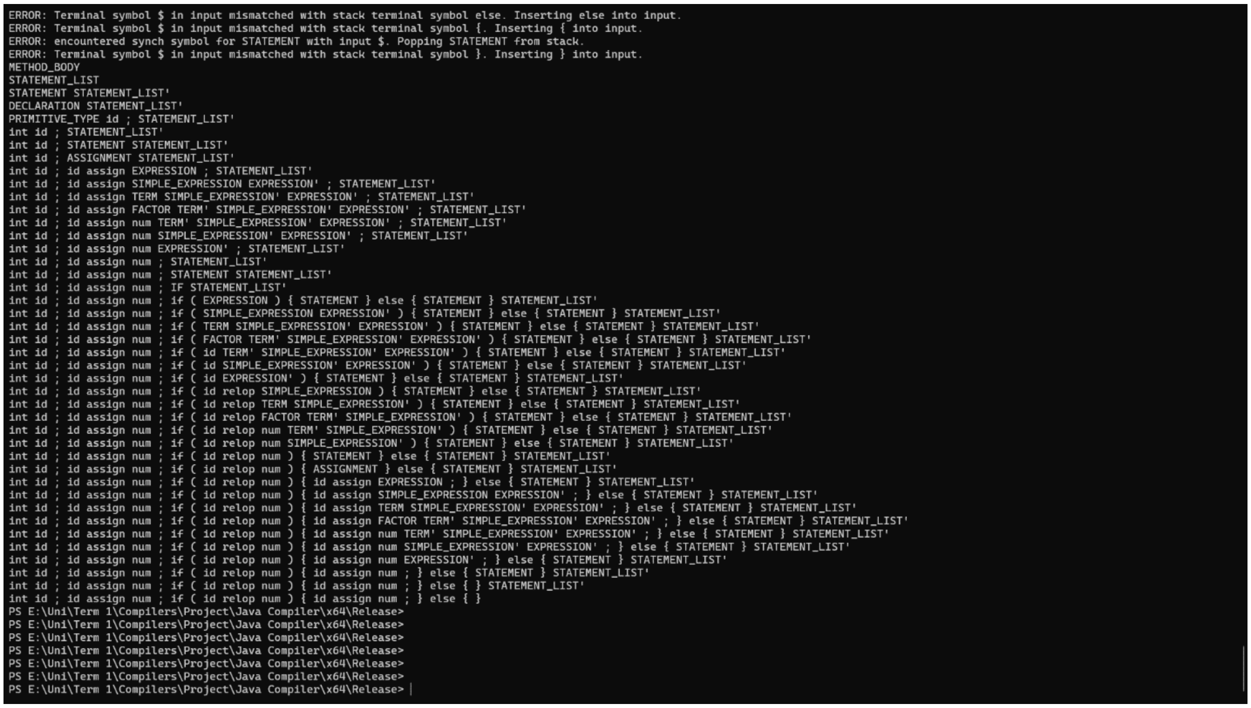


```
SIGN:
        num:
                \S
        addop:
                addop
        id:
                \S
        (:
                \S
═══════════════════════════════════════════
SIMPLE_EXPRESSION:
        id:
                TERM  SIMPLE_EXPRESSION'
        relop:
                \S
        addop:
                SIGN  TERM  SIMPLE_EXPRESSION'
        num:
                TERM  SIMPLE_EXPRESSION'
        (:
                TERM  SIMPLE_EXPRESSION'
        ;:
                \S
        ):
                \S
═══════════════════════════════════════════
SIMPLE_EXPRESSION':
        relop:
                \L
        addop:
                addop  TERM  SIMPLE_EXPRESSION'
        ;:
                \L
        ):
                \L
═══════════════════════════════════════════
STATEMENT:
        boolean:
                DECLARATION
        while:
                WHILE
        if:
                IF
        int:
                DECLARATION
        float:
                DECLARATION
        }:
                \S
        id:
STATEMENT:
        boolean:
                DECLARATION
        while:
                WHILE
        if:
                IF
        int:
                DECLARATION
        float:
                DECLARATION
        }:
                \S
        id:
                ASSIGNMENT
        $:
                \S
═══════════════════════════════════════════
STATEMENT_LIST':
        boolean:
                STATEMENT  STATEMENT_LIST'
        int:
                STATEMENT  STATEMENT_LIST'
        if:
                STATEMENT  STATEMENT_LIST'
        while:
                STATEMENT  STATEMENT_LIST'
        float:
                STATEMENT  STATEMENT_LIST'
        id:
                STATEMENT  STATEMENT_LIST'
        $:
                \L
═══════════════════════════════════════════
TERM:
        id:
                FACTOR  TERM'
        addop:
                \S
        relop:
                \S
        num:
                FACTOR  TERM'
        (:
                FACTOR  TERM'
        ;:
                \S
        ):
                \S
═══════════════════════════════════════════
```



```
===========================================
STATEMENT_LIST':
        boolean:
                STATEMENT  STATEMENT_LIST'
        int:
                STATEMENT  STATEMENT_LIST'
        if:
                STATEMENT  STATEMENT_LIST'
        while:
                STATEMENT  STATEMENT_LIST'
        float:
                STATEMENT  STATEMENT_LIST'
        id:
                STATEMENT  STATEMENT_LIST'
        $:
                \L

===========================================
TERM:
        id:
                FACTOR  TERM'
        addop:
                \S
        relop:
                \S
        num:
                FACTOR  TERM'
        (:
                FACTOR  TERM'
        ;:
                \S
        ):
                \S

===========================================
WHILE:
        if:
                \S
        int:
                \S
        while:
                while  (  EXPRESSION  )  {  STATEMENT  }
        id:
                \S
        }:
                \S
        boolean:
                \S
        float:
                \S
        $:
                \S
```

RESULTANT LEFT DERIVATION OF THE EXAMPLE PROGRAM



```
ERROR: Terminal symbol $ in input mismatched with stack terminal symbol else. Inserting else into input.
ERROR: Terminal symbol $ in input mismatched with stack terminal symbol {. Inserting { into input.
ERROR: encountered synch symbol for STATEMENT with input $. Popping STATEMENT from stack.
ERROR: Terminal symbol $ in input mismatched with stack terminal symbol }. Inserting } into input.
METHOD_BODY
STATEMENT_LIST
STATEMENT STATEMENT_LIST'
DECLARATION STATEMENT_LIST'
PRIMITIVE_TYPE id ; STATEMENT_LIST'
int id ; STATEMENT_LIST'
int id ; STATEMENT STATEMENT_LIST'
int id ; ASSIGNMENT STATEMENT_LIST'
int id ; id assign EXPRESSION ; STATEMENT_LIST'
int id ; id assign SIMPLE_EXPRESSION EXPRESSION' ; STATEMENT_LIST'
int id ; id assign TERM SIMPLE_EXPRESSION' EXPRESSION' ; STATEMENT_LIST'
int id ; id assign FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION' ; STATEMENT_LIST'
int id ; id assign num TERM' SIMPLE_EXPRESSION' EXPRESSION' ; STATEMENT_LIST'
int id ; id assign num SIMPLE_EXPRESSION' EXPRESSION' ; STATEMENT_LIST'
int id ; id assign num EXPRESSION' ; STATEMENT_LIST'
int id ; id assign num ; STATEMENT_LIST'
int id ; id assign num ; STATEMENT STATEMENT_LIST'
int id ; id assign num ; IF STATEMENT_LIST'
int id ; id assign num ; if ( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( SIMPLE_EXPRESSION EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( TERM SIMPLE_EXPRESSION' EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id TERM' SIMPLE_EXPRESSION' EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id SIMPLE_EXPRESSION' EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop SIMPLE_EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop TERM SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop FACTOR TERM' SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num TERM' SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { ASSIGNMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign EXPRESSION ; } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign SIMPLE_EXPRESSION EXPRESSION' ; } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign TERM SIMPLE_EXPRESSION' EXPRESSION' ; } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION' ; } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num TERM' SIMPLE_EXPRESSION' EXPRESSION' ; } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num SIMPLE_EXPRESSION' EXPRESSION' ; } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num EXPRESSION' ; } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; } else { } STATEMENT_LIST'
int id ; id assign num ; if ( id relop num ) { id assign num ; } else { }
PS E:\Uni\Term 1\Compilers\Project\Java Compiler\x64\Release>
PS E:\Uni\Term 1\Compilers\Project\Java Compiler\x64\Release>
PS E:\Uni\Term 1\Compilers\Project\Java Compiler\x64\Release>
PS E:\Uni\Term 1\Compilers\Project\Java Compiler\x64\Release>
PS E:\Uni\Term 1\Compilers\Project\Java Compiler\x64\Release>
PS E:\Uni\Term 1\Compilers\Project\Java Compiler\x64\Release>
PS E:\Uni\Term 1\Compilers\Project\Java Compiler\x64\Release>
```