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

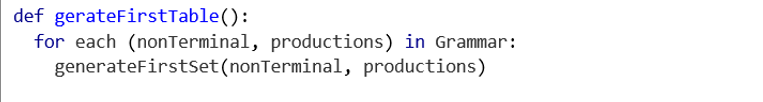
The class is responsible for generating the first set, and the following set of terminals is also responsible for generating the parse table.

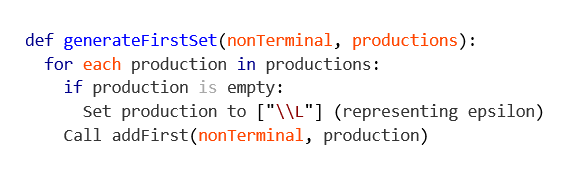
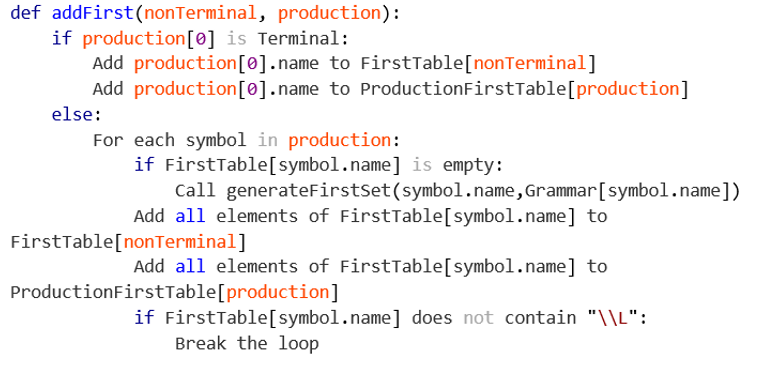
### members

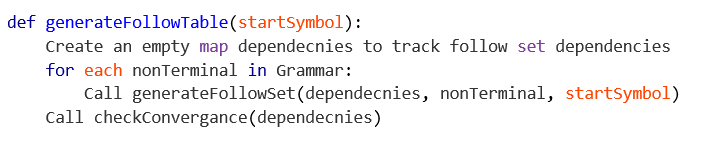
* m\_Grammar:
  + type: map<string, vector<vector<Symbol>>>
  + desc: The grammar received from the CFG Parser class.
* m\_FirstTable:
  + type: unordered\_map<string, unordered\_set<string>>
  + desc: a table containing each terminal with their first set.
* m\_FollowTable:
  + type: unordered\_map<string, 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

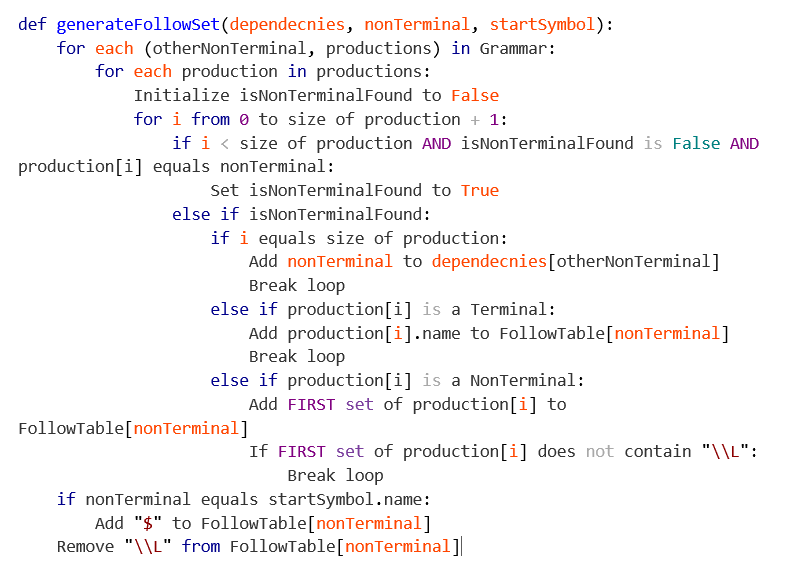
* generateFirstTable():
  + Iterates through all non-terminals in the grammar.
  + Calls generateFirstSet() to compute the FIRST set for each non-terminal.



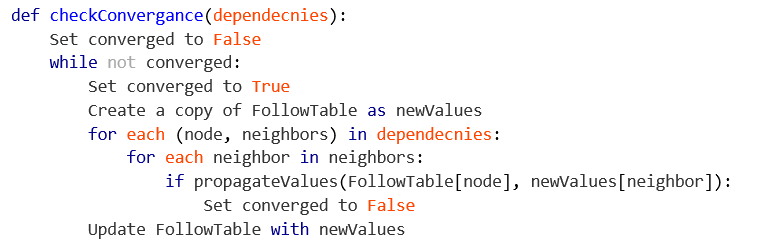
* generateFirstSet(nonTerminal, productions):
  + Processes all productions for a given non-terminal.
  + Ensures that the FIRST set is updated for each production by calling addFirst()
* addFirst(nonTerminal, production):
  + Determines the FIRST set for a single production.
  + If the first symbol 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.
* generateFollowTable(Symbol startSymbol):
  + Orchestrates the computation of FOLLOW sets.
  + Initializes dependencies for FOLLOW set propagation.
  + Calls checkConvergance to ensure the FOLLOW sets stabilize.

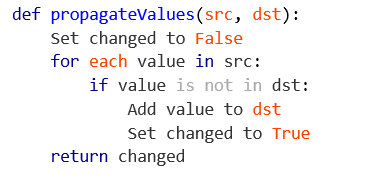


* generateFollowSet(dependencies nonTerminal,startSymbol):
  + Finds where the given nonTerminal appears in productions of other non-terminals.
  + Updates FOLLOW sets based on:
    - Terminals following the nonTerminal.
    - FIRST sets of subsequent non-terminals.
  + Tracks dependencies when the nonTerminal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



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

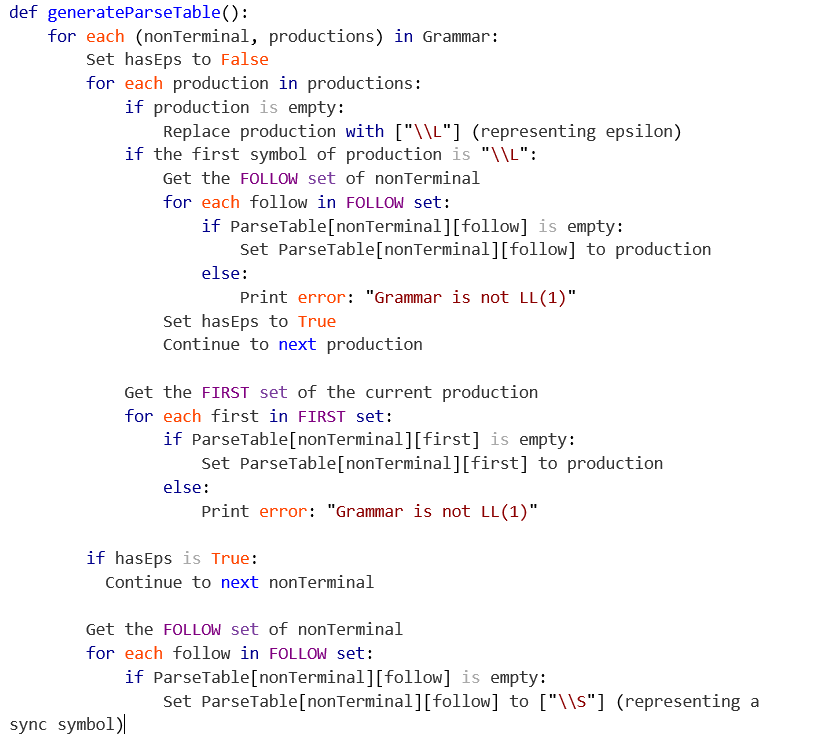


* propagateValues(src,dst): Copies elements from one FOLLOW set to another and checks if any new elements were added.
* generateParseTable():

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:
   1. Handle empty productions by treating them as epsilon (\L).
   2. Check if epsilon is part of the production and update the parse table using the FOLLOW set.
2. **Update Parse Table Using FIRST Sets**:
   1. For each production, add it to the parse table for all terminals in its FIRST set.
   2. 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).



## 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

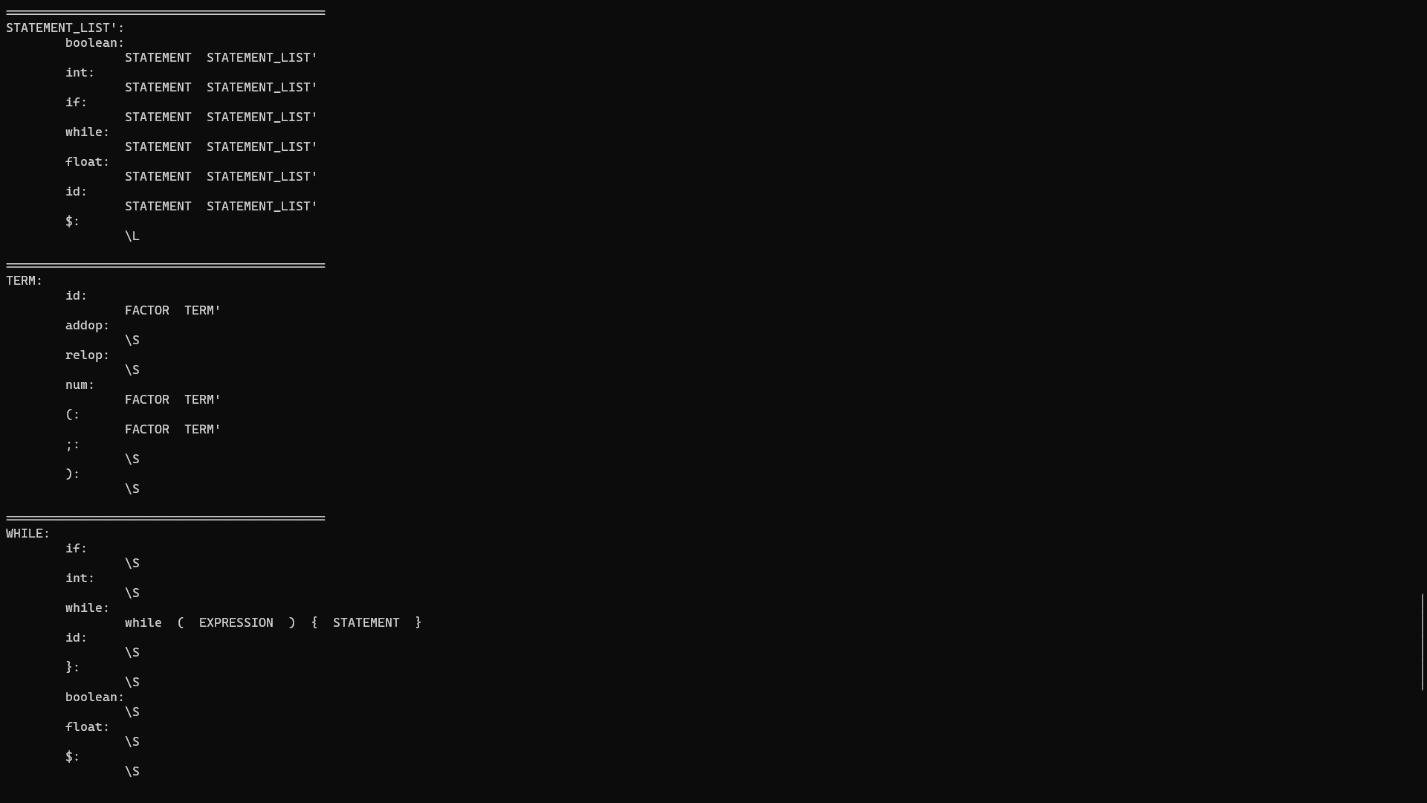
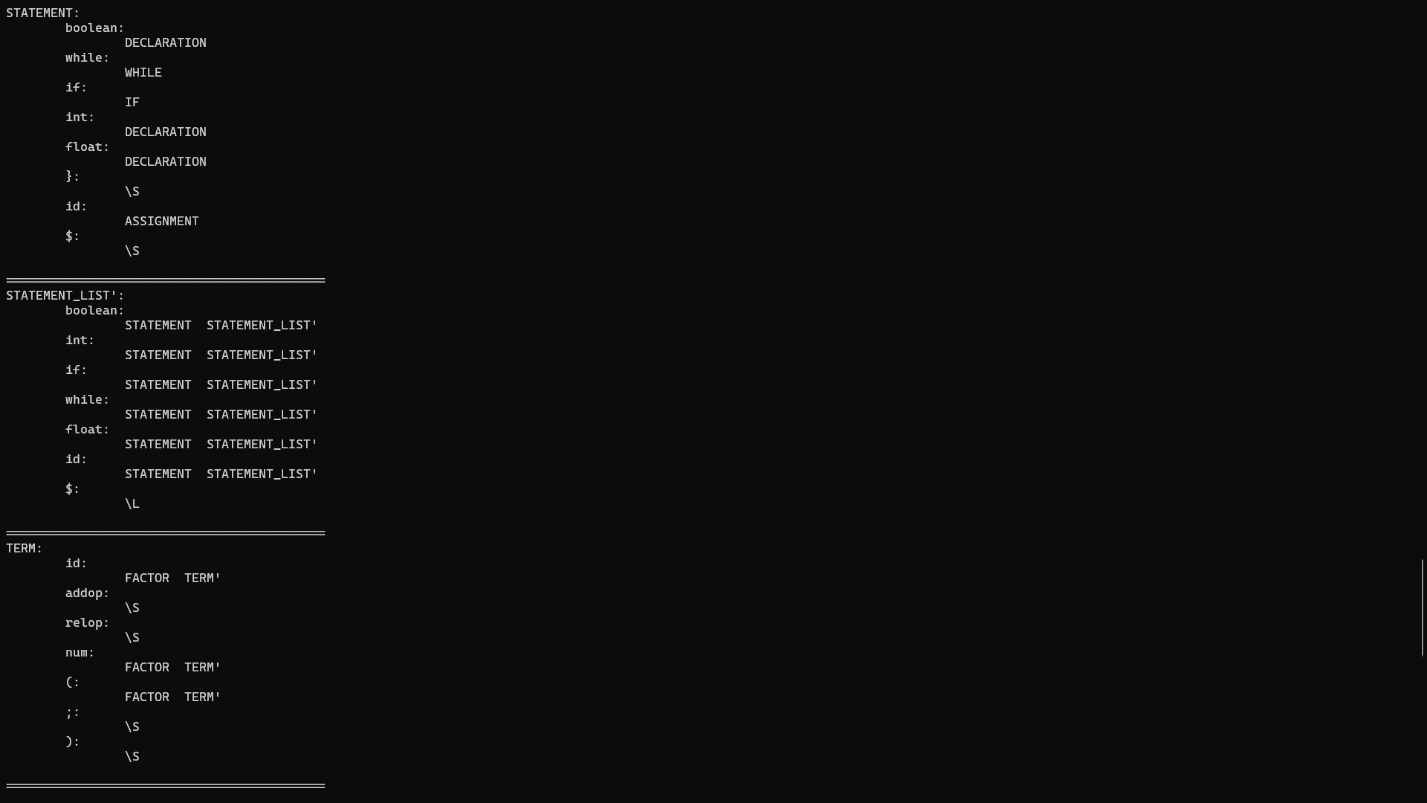
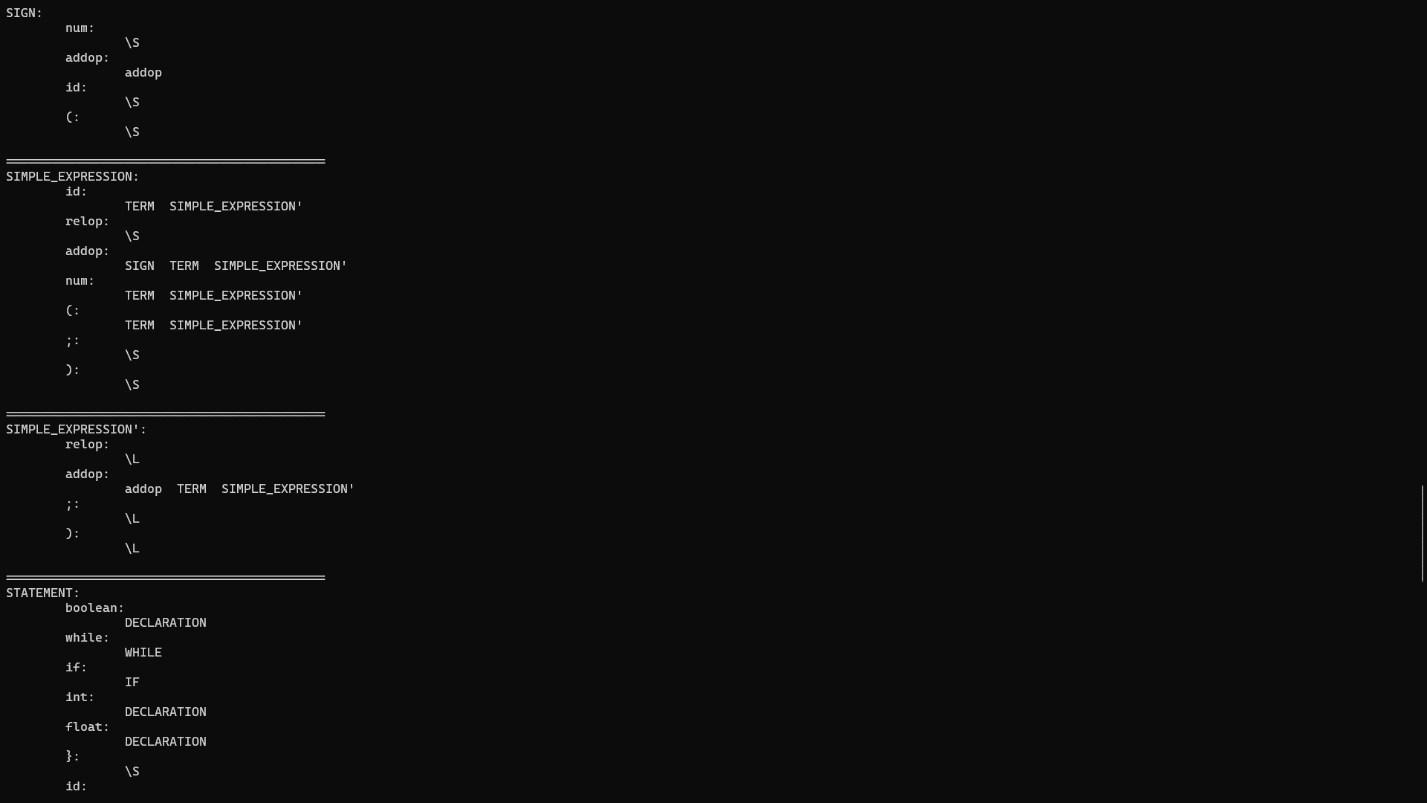
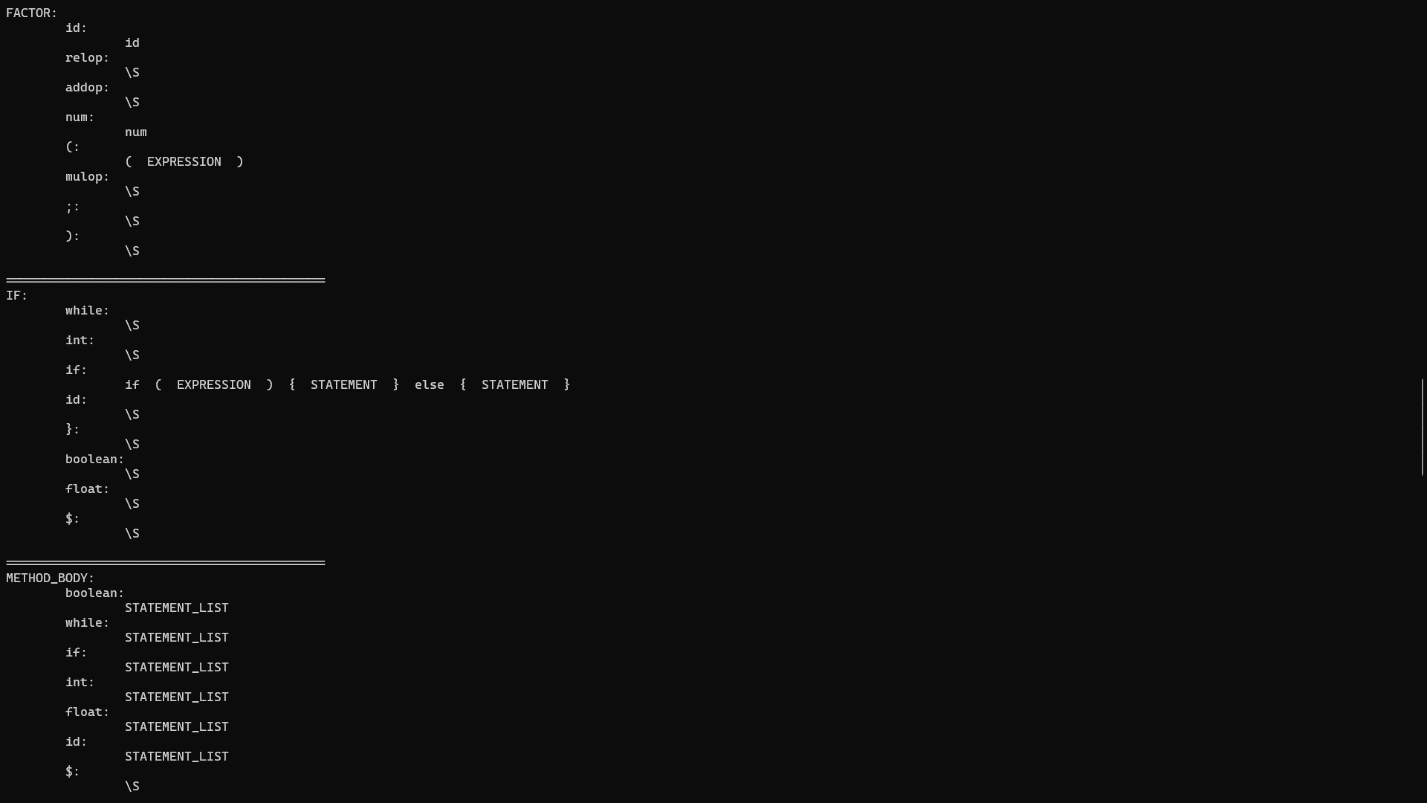
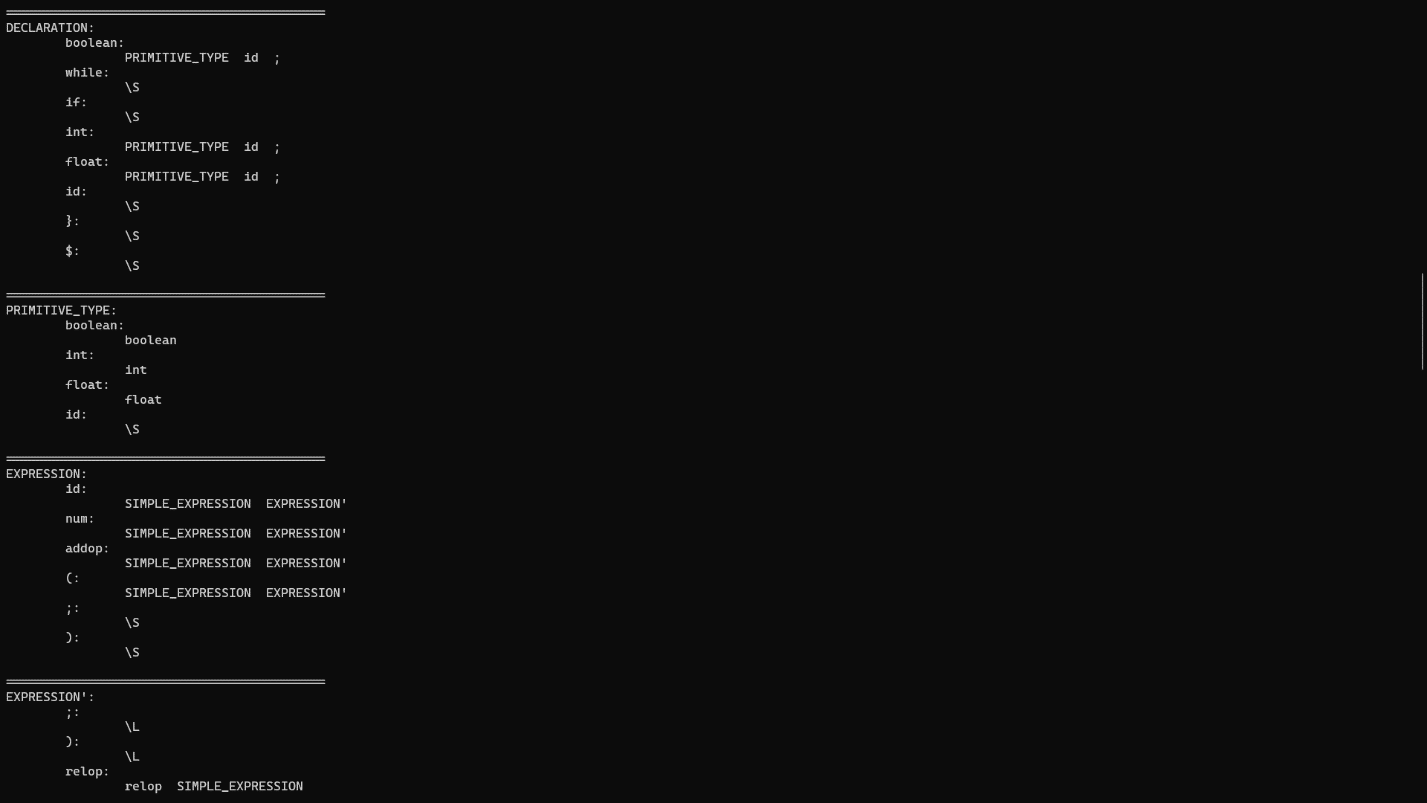
while (!parser.isFinished())

parser.parseNextToken();

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

This class should be used as following:

# resultant parse table



# resultant left derivation of the example program

