Alexandria University- Faculty of Engineering

Computer and Systems Engineering Department

Phase 2 : Parser Generator

Name	ID
Mohamed Mostafa Ibrahim	19016506
Abdelrahman Elsayed Ahmed	19015893
Engy Ibrahim Mahmoud	19015478
Esraa Hassan Mokhtar	19015407

Objective:

This phase of the assignment aims to practice techniques for building automatic parser generator tools.

Description:

1. Your task in this phase of the assignment is to design and implement an LL (1) parser generator tool.

2. The parser generator expects an LL (1) grammar as input. It should compute First and Follow sets and use them to construct a predictive parsing table for the grammar.

3. The table is to be used to drive a predictive top-down parser. If the input grammar is not LL (1), an appropriate error message should be produced.

4. The generated parser is required to produce some representation of the leftmost derivation for a correct input.

5. If an error is encountered, a panic-mode error recovery routine is to be called to print an error message and to resume parsing.

6. The parser generator is required to be tested using the given context free grammar of a small subset of Java. Of course, you have to modify the grammar to allow predictive parsing.

7. Combine the lexical analyzer generated in phase1 and parser such that the lexical analyzer is to be called by the parser to find the next token. Use the simple program given in phase 1 to test the combined lexical analyzer and parser.

a) A description of the used data structure:

- For Grammar Parser:

  1. Maps:

     - productions: Maps each non-terminal to its corresponding set of production rules represented as a vector of vectors of strings.

     - formattedProds: Maps non-terminals to their formatted production rules, where each production is represented as a vector of strings.

  2. Vectors:

     - terminals: Stores the terminal symbols of the grammar

     - nonTerminals: Contains the non-terminal symbols of the grammar.

     - rule: is a vector used to represent individual production rules (sequences of symbols) within the grammar.

     - newAiRules, newNonTerminalRules, formattedRules: are vectors used for different purposes within the context of manipulating grammar rules

  3. Strings:

     - string startSymbol: Stores the starting symbol of the grammar.

  4. Sets:

     - set<string> terminalsSet, derivedNonTerminalsSet, usedNonTerminalsSet: Sets used for efficiently storing unique terminal symbols, nonterminal symbols, derived non-terminals, and used non-terminals, respectively.

b) Explanation of all algorithms and techniques used

1. Parsing the grammar file:

- The parse method tokenizes the input grammar file, splitting it into individual symbols, rules, and non-terminals. It recognizes special tokens like '#' for new productions, '=' for rule definitions, and '|' for alternatives.
- It recognizes non-terminals, terminals, and production rules based on delimiters and special tokens.
- it populates data structures (productions, terminalsSet, nonTerminalsSet, etc.) with the extracted information.

2. Data Structures Initialization:

- Sets (terminalsSet, nonTerminalsSet, etc.) are used to maintain unique terminal and non-terminal symbols encountered in the grammar.
- Maps (productions, formattedProds) are employed to represent the grammar rules associated with each non-terminal symbol.

3. Left Recursion Elimination:

- The eliminateLeftRecursion method detects and eliminates left recursion in the grammar.
- It iterates through non-terminals and checks for left recursion using a substitution technique.
- Creates new non-terminals to eliminate left recursion while maintaining the equivalent grammar.

4. Left Factoring:

- The leftFactor method addresses left-factoring in the grammar
- It identifies common prefixes among production rules for a non-terminal and refactors them to introduce new non-terminals.
- Helps in reducing redundancy and ambiguity in the grammar rules.

5. **Verification of Grammar:**

- Completeness Check: The verifyCompleteGrammar method ensures that all non-terminals used in the grammar have their corresponding rules defined.
- Verification of Derived Non-Terminals: It verifies that the non-terminals derived during transformations are properly tracked.

6. Formatting Productions:

- The formatProductions method converts the internal representation of rules into a more human-readable format (strings).
- Improved Readability: Provides a more readable format for displaying the production rules of the grammar.

7. Helper Methods:

- Substitution: The substitute method facilitates the substitution of recursive rules with non-recursive alternatives by modifying production rules.
- Terminal Identification: isTerminal determines if a symbol is a terminal or not.

8. Processing symbols and Rules:

- Rule Manipulation: Iterating over rules allows for the identification of specific patterns and their modification to transform the grammar's structure.
- Symbol Processing: Symbols are processed to perform substitutions or transformations based on predefined rules and conditions.

a) <u>A description of the used data structure:</u>

- For First and Follow:

    1. Maps:

        - first:

            1. This map stores the FIRST sets for each non-terminal symbol.
            2. Each non-terminal symbol (key) maps to a vector of strings (value) representing the set of terminal symbols that can start a string derived from that non-terminal.

        - follow:

            1. This map stores the FOLLOW sets for each non-terminal symbol.
            2. Each non-terminal symbol (key) maps to a vector of strings (value) representing the set of terminal symbols that can appear immediately after occurrences of that non-terminal in any derivation.

        - first_complemntary:

            1. used to store additional information related to the FIRST sets to help us in the parsing table.
            2. It stores the production rules that cause changes in the FIRST set

b) Explanation of all algorithms and techniques used

The provided code seems to be implementing the computation of First and Follow sets for a given context-free grammar (CFG). The First set of a non-terminal represents the set of terminals that begin the strings derivable from that non-terminal, while the Follow set represents the set of terminals that can follow that non-terminal in some derivation

1. Compute First Sets:

- Initialization : Initialize all First sets for non-terminals as empty.
- Iteration and Processing:
  - ➔ Iterate over each production rule of the form X -> α.
  - ➔ For each symbol in the production:
    - Check if the symbol is a terminal or non-terminal.
    - Apply rules:
      1. If the symbol is a terminal, add it to the First set of the corresponding non-terminal.
      2. If the symbol is a non-terminal, consider its First set:
         - Add terminals from First(Yi) to First(X) for each Yi in the production.
         - Check if ε (epsilon) is in all First(Yj) for j=1 to i-1. If so, add ε to First(X).
- Iteration Continues: Repeat until no changes are made in any First set.

2. **Compute Follow Sets:**

- Initialization : Initialize all Follow sets for non-terminals as empty. Add '$' to the Follow set of the start symbol.

- Iteration and Processing:
  - ➔ Iterate over each production rule of the form A -> αBβ.

  - ➔ For each non-terminal B in a production::
    - Apply rules:
      1. Compute First(β), and add everything except ε to Follow(B).

      2. Check if all symbols in β derive ε. If so, add Follow(A) to Follow(B).

- Iteration Continues: Repeat until no changes are made in any Follow set.

Techniques used:

- Iteration over Productions: The code iterates over the productions of each non-terminal, considering each symbol and applying the respective rules.

- Checking Terminals/Non-terminals: The functions isTerminal and isNonTerminal help in identifying terminal and non-terminal symbols.

- Handling Epsilon: Epsilon ('ε') represents an empty string. The code considers ε for various conditions in calculating First and Follow sets.

- Tracking Changes: The boolean flag changes are used to determine if any changes have been made to the sets, allowing iteration until no more changes occur.

- Set Operations: Operations like addition of elements to First and Follow sets based on certain conditions are core to these algorithms.

a) <u>description of the used data structure:</u>

- For Parsing Table:

  1. Maps:

     - table:

       Map that uses a pair of strings as keys and vectors of strings as values. It stores the actual parsing table where the keys are pairs of non-terminal symbols and terminal symbols, and the values are vectors of strings containing production rules or 'sync' for error handling in LL(1) parsing.

b) **<u>Explanation of all algorithms and techniques used:</u>**

For the construction of the parsing table:

1. Initialization:
   - The constructor initializes the valid flag as true
   - It initializes a boolean variable sync.

2. Parsing Table Construction:
   - It iterates through each non-terminal symbol in the first set.

   - For each non-terminal symbol:
     1. Initializes sync as true.

     2. Iterates through each terminal symbol in the first set of the current non-terminal.

     3. If the terminal symbol is not an empty string:
        - Adds an entry to the parsing table for the pair (non-terminal, terminal) with the corresponding value from comp

     4. If the terminal symbol is an empty string (ε-production):
        - Sets sync to false.

        - For each terminal symbol in the follow set of the current non-terminal:
          - ➔ If the parsing table entry for (non-terminal, terminal) is not empty:
            - If the value is not "sync", marks the grammar as not LL(1) (valid = false), clears the table, and returns.

            - Otherwise, update the value in the table with the corresponding value from comp.

          - ➔ If the parsing table entry is empty, adds the epsilon value to the table

     5. If all terminals in the first set were processed without encountering an ε-production:
        - For each terminal symbol in the follow set of the current non-terminal:
          - ➔ If the parsing table entry for (non-terminal, terminal) is empty, add "sync" to the table.

a) <u>description of the used data structure:</u>

- **For Parser output:**

  1. Stacks:

     - inputs:

       This stack holds the input tokens that need to be processed by the parser.

     - st:

       This stack represents the parser's stack, containing terminals, non-terminals, and parsing actions during the parsing process

  2. Vectors:

     - out:

       This vector stores the output or the sequence of left derivation taken by the parser during parsing.

b) Explanation of all algorithms and techniques used:

1. Function getStackValues:

- This function retrieves the values from a stack st and constructs a space-separated string containing these values.
- it initializes an empty string out and a stack cs.
- The while loop runs until there is no elements in the stack
  - ➔ It takes the top element curr from the stack st, removes it using st.pop(), appends it to the out string followed by a space, and pushes curr onto the stack cs.
- After this loop, it reverses the elements from stack cs back to stack st while constructing the output string.
- Finally, it returns the constructed string out.

2. Function run:

- It retrieves the top elements from two stacks: st and inputs.

- If st_f (top element of stack st) is found in terminals, it checks for specific conditions:
    - ➔ If st_f and in_f (top element of stack inputs) are both equal to "$" (denoting the end of input), then the current state is accepted and return , indicating successful parsing.

    - ➔ If st_f and in_f are equal but not "$", so match them and pop the elements from both stacks, update the matched string, and call getNextInput() to fetch the next input.

    - ➔ If st_f and in_f are not equal, it reports an error and pushes st_f back to inputs.

- If st_f is not found in terminals, it checks the map mp for a corresponding entry:
    - ➔ If found, it processes based on the value retrieved from the map:
        - If the value is "sync", it reports an error and pops the top element from st.

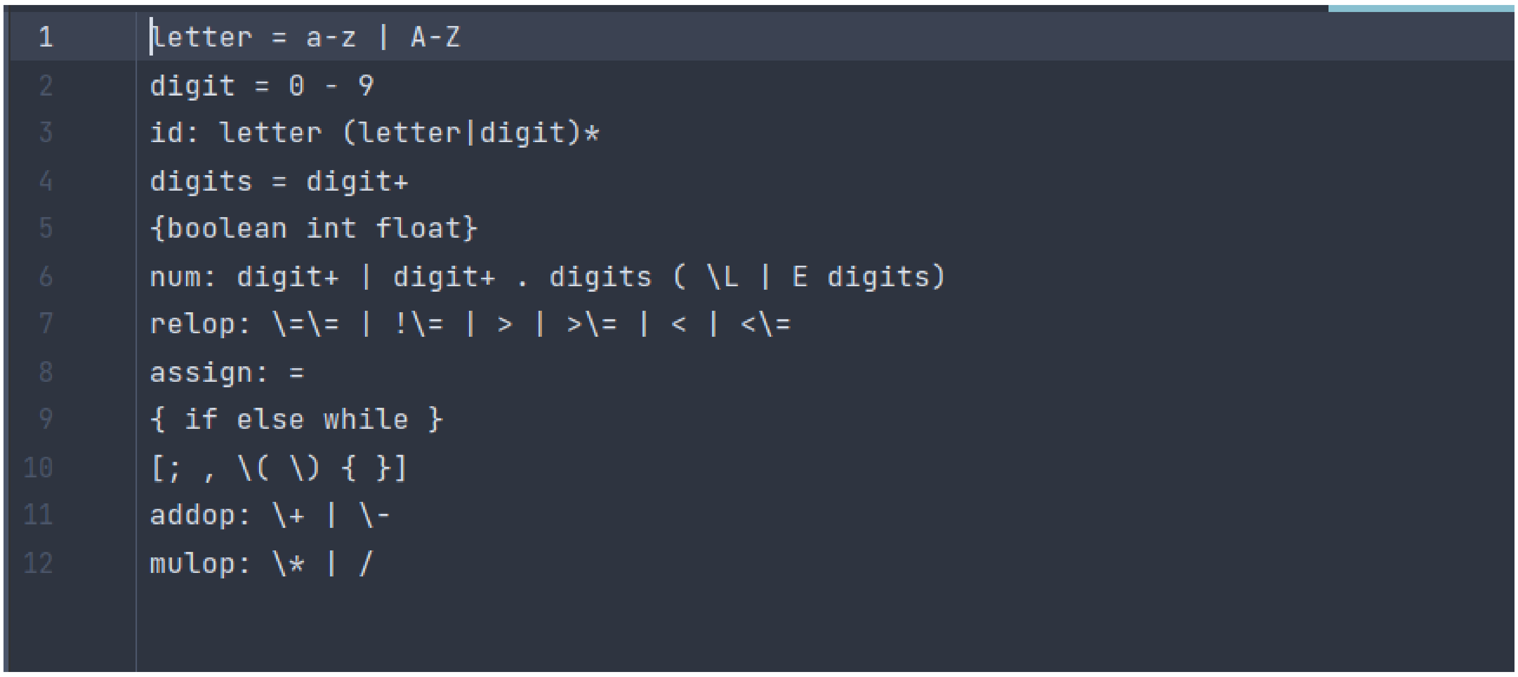
        - Otherwise substitute the top of stack by its value on the table using the top of input stack

    - ➔ if st_f is not found in mp, it reports an error, discards the top element from inputs, and retrieves the next input if it is not "$".

- The function then appends the constructed string from getStackValues() to the matched string, writes to csvFile, and recursively calls itself (run()) again, effectively continuing the parsing process.

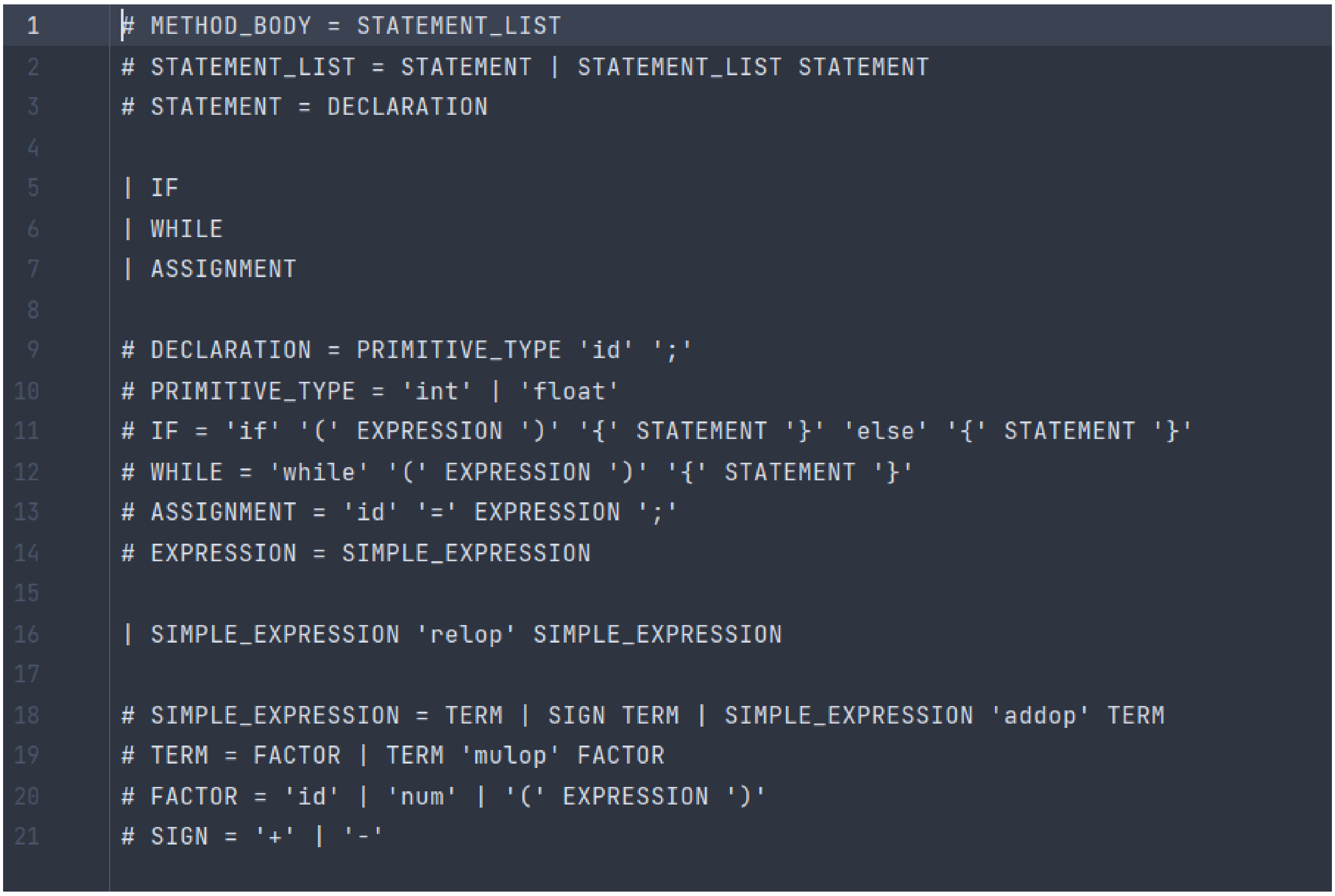
The Input files we have:

- The lexical rules file:



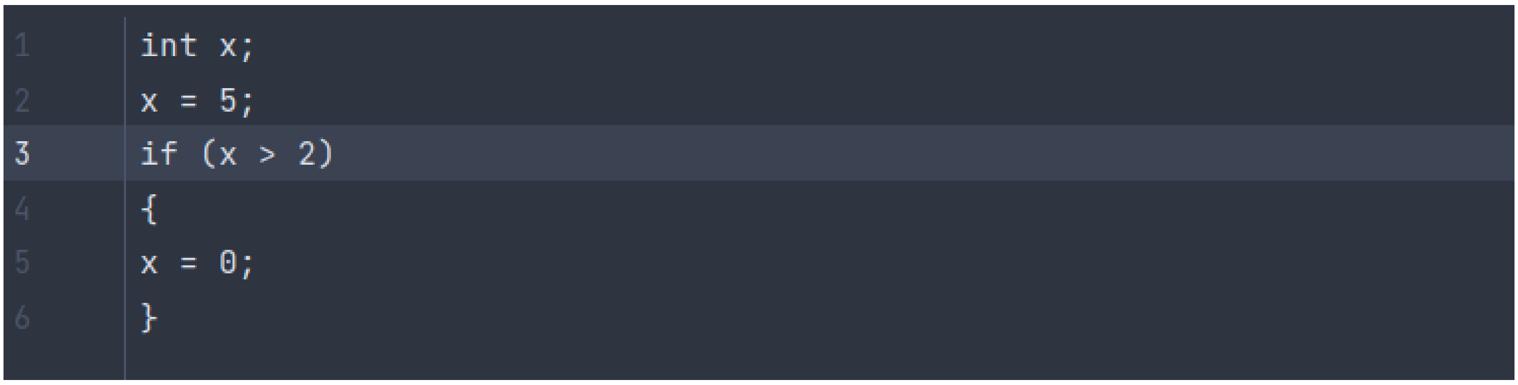
```
letter = a-z | A-Z
digit = 0 - 9
id: letter (letter|digit)*
digits = digit+
{boolean int float}
num: digit+ | digit+ . digits ( \L | E digits)
relop: \=\= | !\= | > | >\= | < | <\=
assign: =
{ if else while }
[; , \( \) { }]
addop: \+ | \-
mulop: \* | /
```

- The Input grammar file:



```
# METHOD_BODY = STATEMENT_LIST
# STATEMENT_LIST = STATEMENT | STATEMENT_LIST STATEMENT
# STATEMENT = DECLARATION

| IF
| WHILE
| ASSIGNMENT

# DECLARATION = PRIMITIVE_TYPE 'id' ';'
# PRIMITIVE_TYPE = 'int' | 'float'
# IF = 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
# WHILE = 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
# ASSIGNMENT = 'id' '=' EXPRESSION ';'
# EXPRESSION = SIMPLE_EXPRESSION

| SIMPLE_EXPRESSION 'relop' SIMPLE_EXPRESSION

# SIMPLE_EXPRESSION = TERM | SIGN TERM | SIMPLE_EXPRESSION 'addop' TERM
# TERM = FACTOR | TERM 'mulop' FACTOR
# FACTOR = 'id' | 'num' | '(' EXPRESSION ')'
# SIGN = '+' | '-'
```

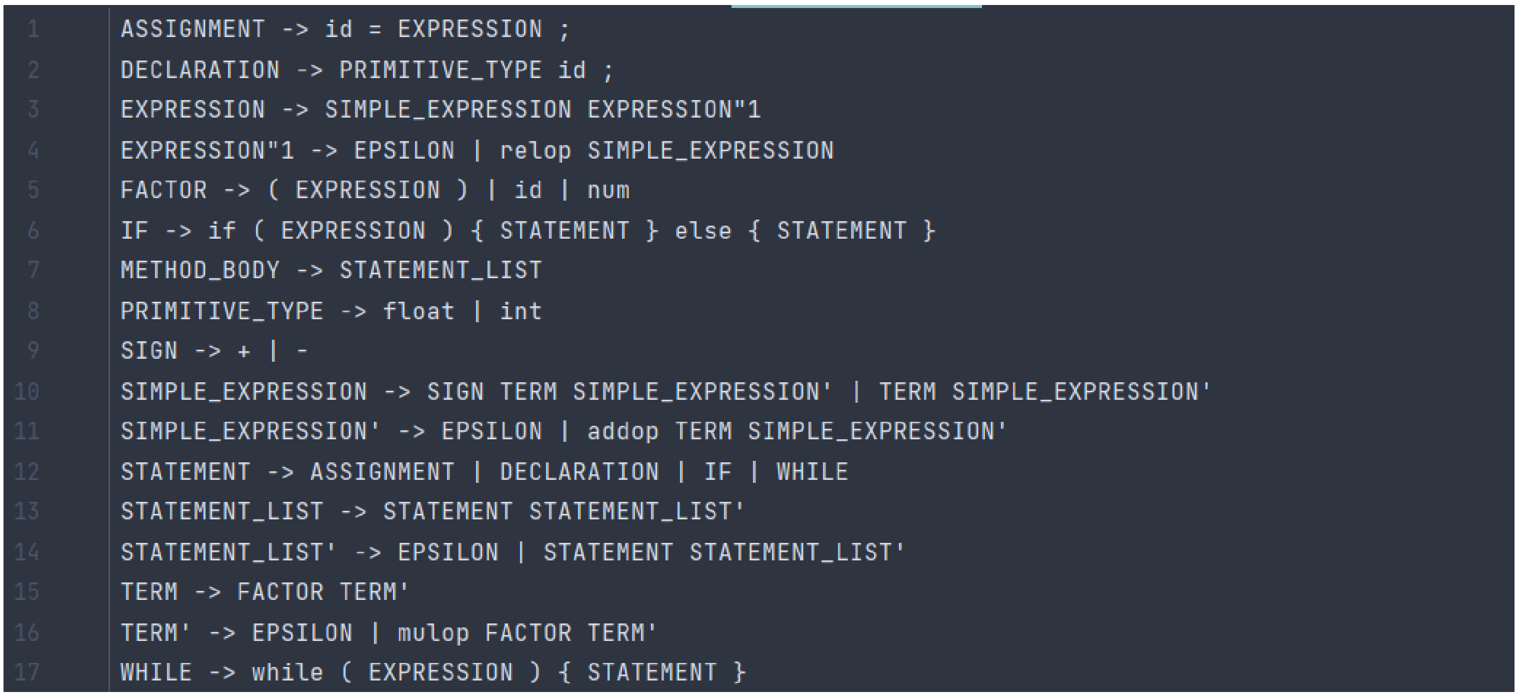
- The Input test example:



```
int x;
x = 5;
if (x > 2)
{
x = 0;
}
```

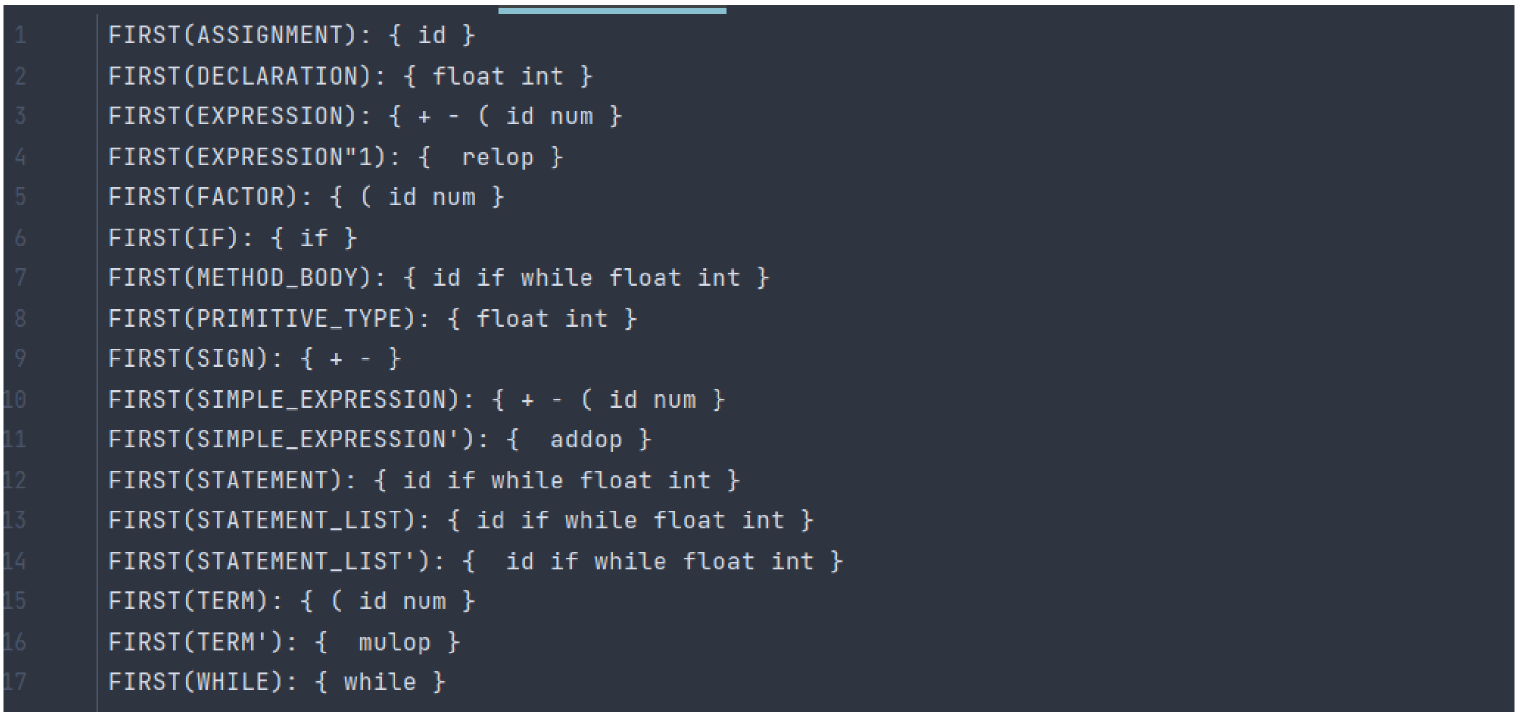
The output files we have:

- The grammar after eliminating left recursion and applying left factoring:



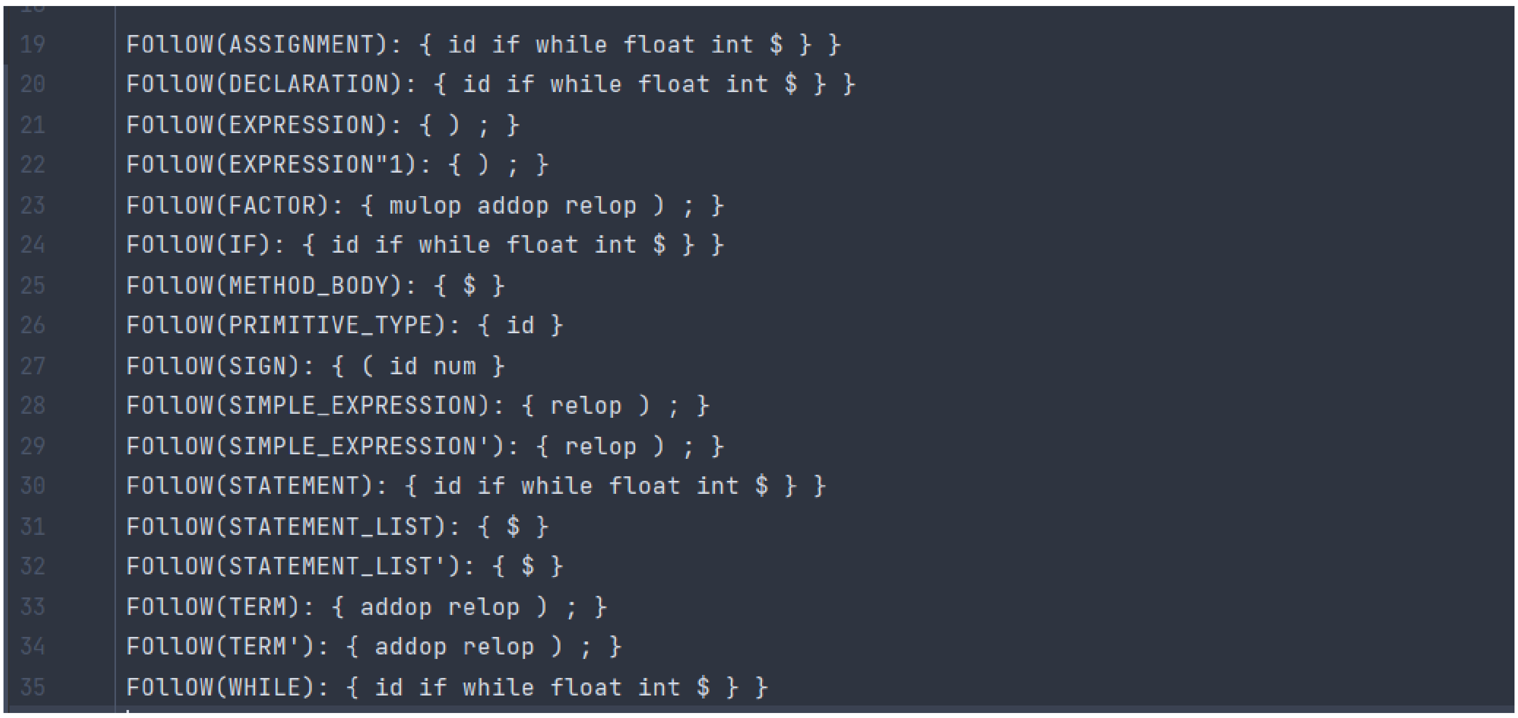
```
ASSIGNMENT -> id = EXPRESSION ;
DECLARATION -> PRIMITIVE_TYPE id ;
EXPRESSION -> SIMPLE_EXPRESSION EXPRESSION"1
EXPRESSION"1 -> EPSILON | relop SIMPLE_EXPRESSION
FACTOR -> ( EXPRESSION ) | id | num
IF -> if ( EXPRESSION ) { STATEMENT } else { STATEMENT }
METHOD_BODY -> STATEMENT_LIST
PRIMITIVE_TYPE -> float | int
SIGN -> + | -
SIMPLE_EXPRESSION -> SIGN TERM SIMPLE_EXPRESSION' | TERM SIMPLE_EXPRESSION'
SIMPLE_EXPRESSION' -> EPSILON | addop TERM SIMPLE_EXPRESSION'
STATEMENT -> ASSIGNMENT | DECLARATION | IF | WHILE
STATEMENT_LIST -> STATEMENT STATEMENT_LIST'
STATEMENT_LIST' -> EPSILON | STATEMENT STATEMENT_LIST'
TERM -> FACTOR TERM'
TERM' -> EPSILON | mulop FACTOR TERM'
WHILE -> while ( EXPRESSION ) { STATEMENT }
```

- The first of each non terminal we have after left factoring and eliminating left recursion:



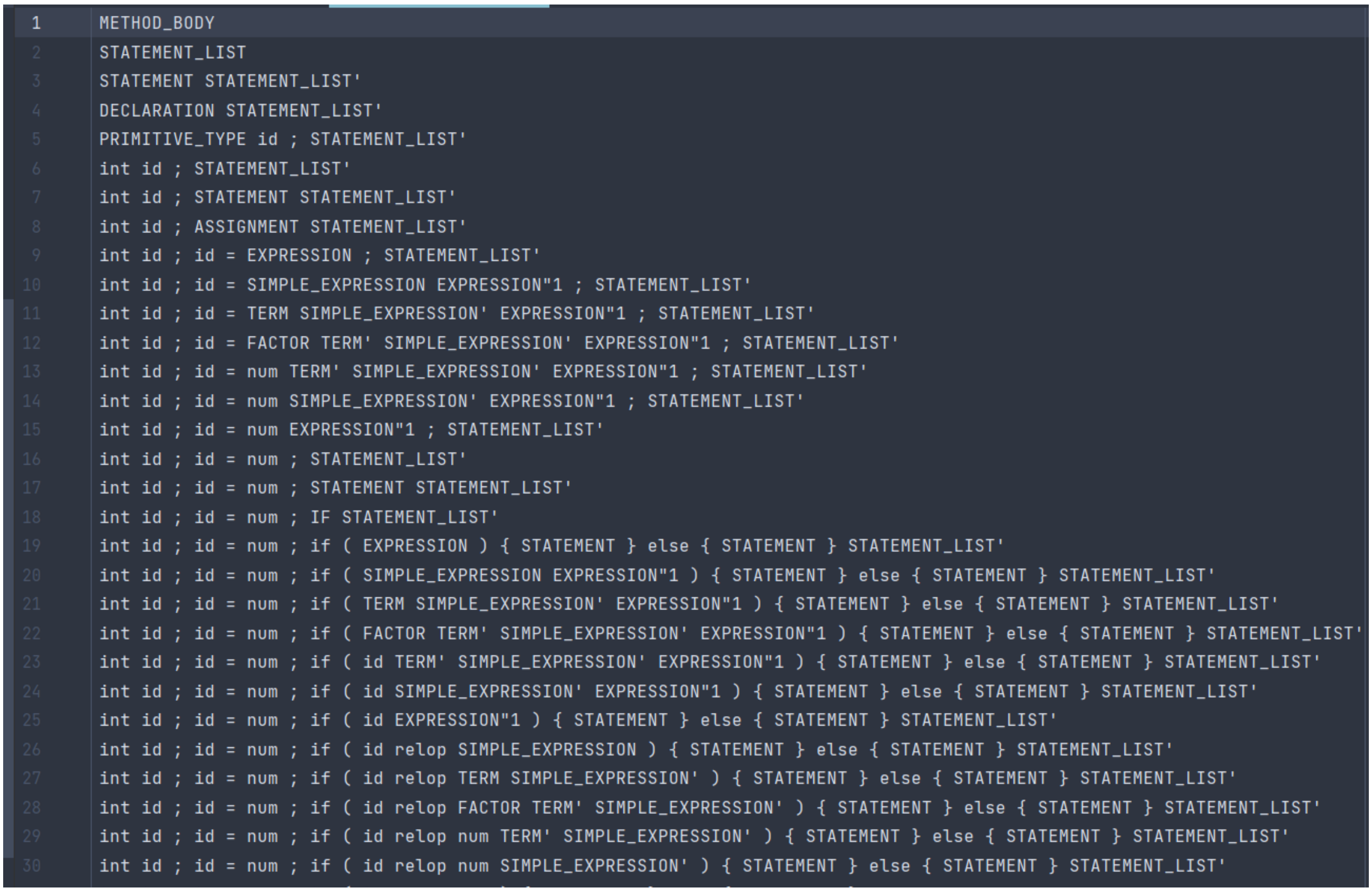
```
FIRST(ASSIGNMENT): { id }
FIRST(DECLARATION): { float int }
FIRST(EXPRESSION): { + - ( id num }
FIRST(EXPRESSION"1): {  relop }
FIRST(FACTOR): { ( id num }
FIRST(IF): { if }
FIRST(METHOD_BODY): { id if while float int }
FIRST(PRIMITIVE_TYPE): { float int }
FIRST(SIGN): { + - }
FIRST(SIMPLE_EXPRESSION): { + - ( id num }
FIRST(SIMPLE_EXPRESSION'): {  addop }
FIRST(STATEMENT): { id if while float int }
FIRST(STATEMENT_LIST): { id if while float int }
FIRST(STATEMENT_LIST'): {  id if while float int }
FIRST(TERM): { ( id num }
FIRST(TERM'): {  mulop }
FIRST(WHILE): { while }
```

- The follow of each non terminal we have after left factoring and eliminating left recursion:

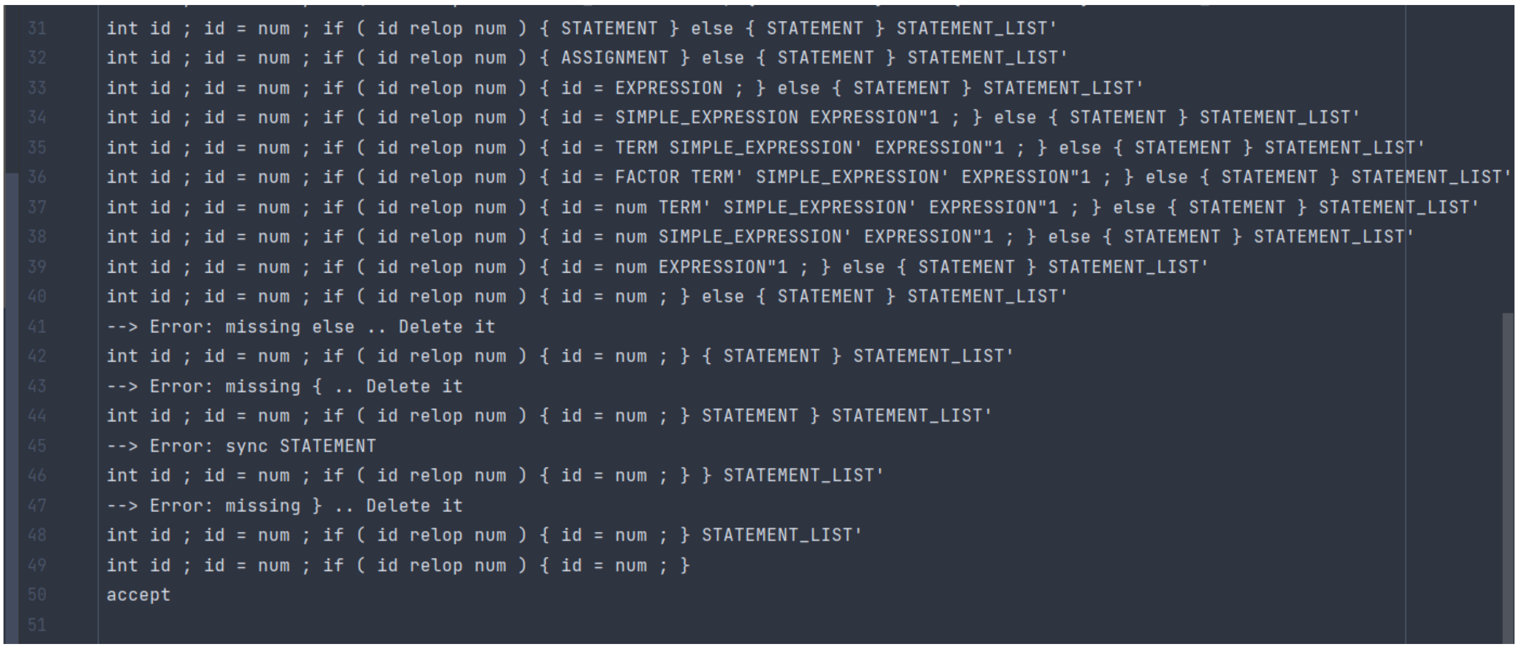


```
FOllOW(ASSIGNMENT): { id if while float int $ } }
FOllOW(DECLARATION): { id if while float int $ } }
FOllOW(EXPRESSION): { ) ; }
FOllOW(EXPRESSION"1): { ) ; }
FOllOW(FACTOR): { mulop addop relop ) ; }
FOllOW(IF): { id if while float int $ } }
FOllOW(METHOD_BODY): { $ }
FOllOW(PRIMITIVE_TYPE): { id }
FOllOW(SIGN): { ( id num }
FOllOW(SIMPLE_EXPRESSION): { relop ) ; }
FOllOW(SIMPLE_EXPRESSION'): { relop ) ; }
FOllOW(STATEMENT): { id if while float int $ } }
FOllOW(STATEMENT_LIST): { $ }
FOllOW(STATEMENT_LIST'): { $ }
FOllOW(TERM): { addop relop ) ; }
FOllOW(TERM'): { addop relop ) ; }
FOllOW(WHILE): { id if while float int $ } }
```

- Left most derivation:

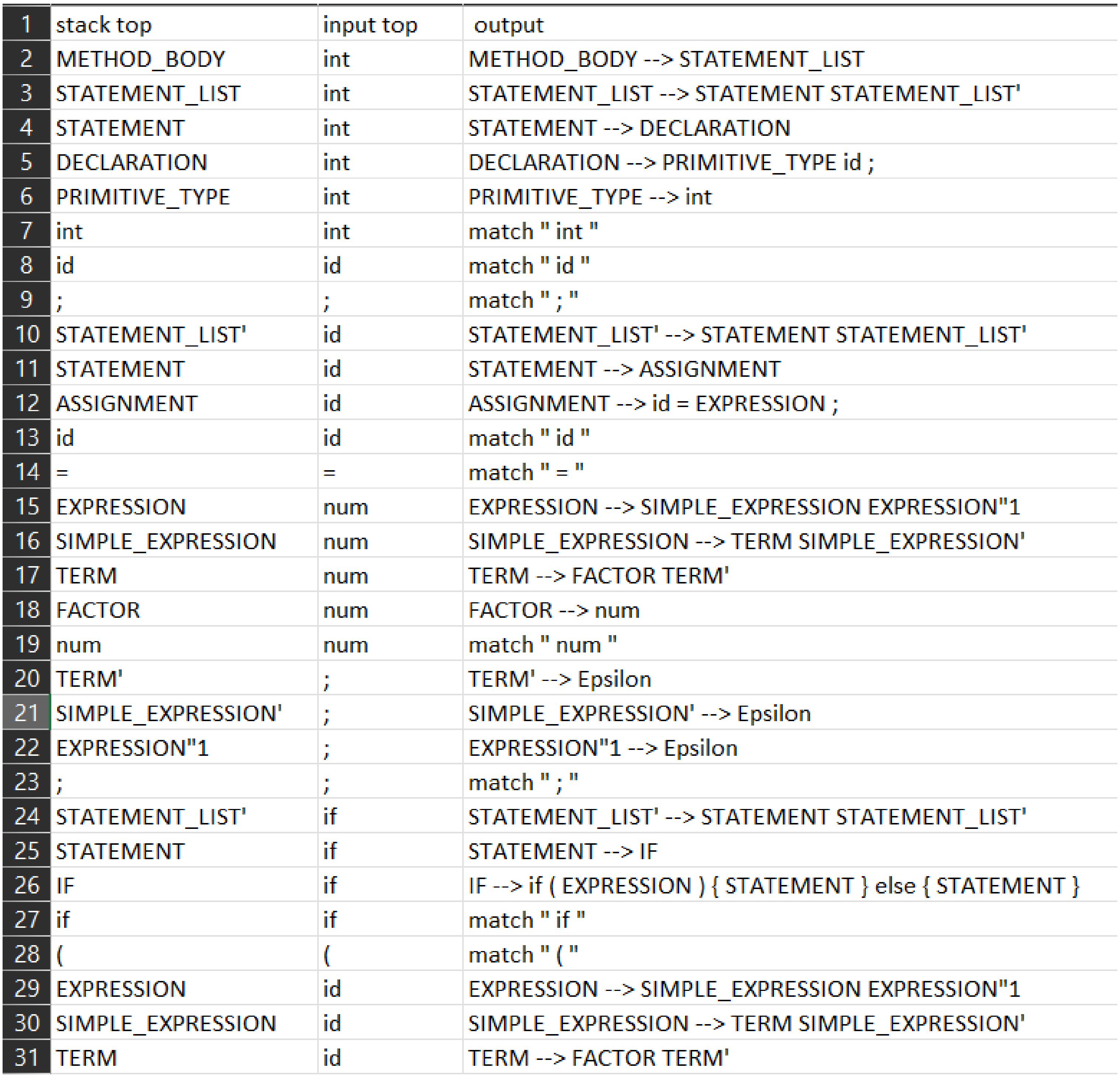


```
METHOD_BODY
STATEMENT_LIST
STATEMENT STATEMENT_LIST'
DECLARATION STATEMENT_LIST'
PRIMITIVE_TYPE id ; STATEMENT_LIST'
int id ; STATEMENT_LIST'
int id ; STATEMENT STATEMENT_LIST'
int id ; ASSIGNMENT STATEMENT_LIST'
int id ; id = EXPRESSION ; STATEMENT_LIST'
int id ; id = SIMPLE_EXPRESSION EXPRESSION"1 ; STATEMENT_LIST'
int id ; id = TERM SIMPLE_EXPRESSION' EXPRESSION"1 ; STATEMENT_LIST'
int id ; id = FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION"1 ; STATEMENT_LIST'
int id ; id = num TERM' SIMPLE_EXPRESSION' EXPRESSION"1 ; STATEMENT_LIST'
int id ; id = num SIMPLE_EXPRESSION' EXPRESSION"1 ; STATEMENT_LIST'
int id ; id = num EXPRESSION"1 ; STATEMENT_LIST'
int id ; id = num ; STATEMENT_LIST'
int id ; id = num ; STATEMENT STATEMENT_LIST'
int id ; id = num ; IF STATEMENT_LIST'
int id ; id = num ; if ( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( SIMPLE_EXPRESSION EXPRESSION"1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( TERM SIMPLE_EXPRESSION' EXPRESSION"1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION"1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id TERM' SIMPLE_EXPRESSION' EXPRESSION"1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id SIMPLE_EXPRESSION' EXPRESSION"1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id EXPRESSION"1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop SIMPLE_EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop TERM SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop FACTOR TERM' SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop num TERM' SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop num SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
```



```
int id ; id = num ; if ( id relop num ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop num ) { ASSIGNMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop num ) { id = EXPRESSION ; } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop num ) { id = SIMPLE_EXPRESSION EXPRESSION"1 ; } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop num ) { id = TERM SIMPLE_EXPRESSION' EXPRESSION"1 ; } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop num ) { id = FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION"1 ; } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop num ) { id = num TERM' SIMPLE_EXPRESSION' EXPRESSION"1 ; } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop num ) { id = num SIMPLE_EXPRESSION' EXPRESSION"1 ; } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop num ) { id = num EXPRESSION"1 ; } else { STATEMENT } STATEMENT_LIST'
int id ; id = num ; if ( id relop num ) { id = num ; } else { STATEMENT } STATEMENT_LIST'
--> Error: missing else .. Delete it
int id ; id = num ; if ( id relop num ) { id = num ; } { STATEMENT } STATEMENT_LIST'
--> Error: missing { .. Delete it
int id ; id = num ; if ( id relop num ) { id = num ; } STATEMENT } STATEMENT_LIST'
--> Error: sync STATEMENT
int id ; id = num ; if ( id relop num ) { id = num ; } } STATEMENT_LIST'
--> Error: missing } .. Delete it
int id ; id = num ; if ( id relop num ) { id = num ; } STATEMENT_LIST'
int id ; id = num ; if ( id relop num ) { id = num ; }
accept
```

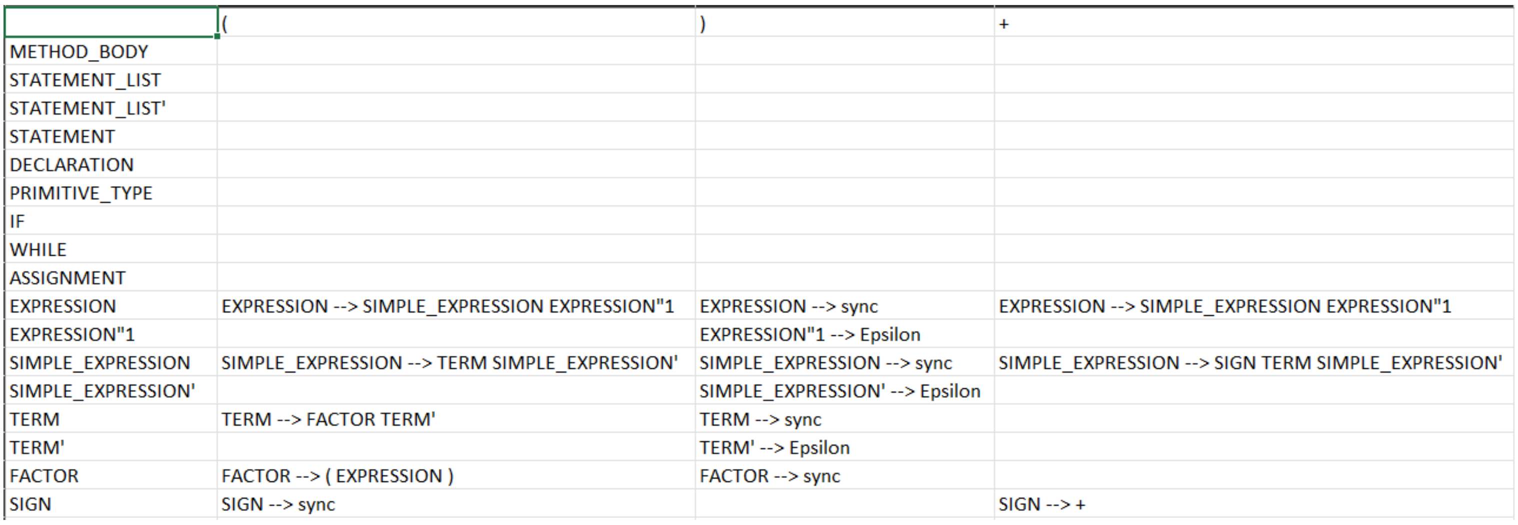
- The file stack.csv:



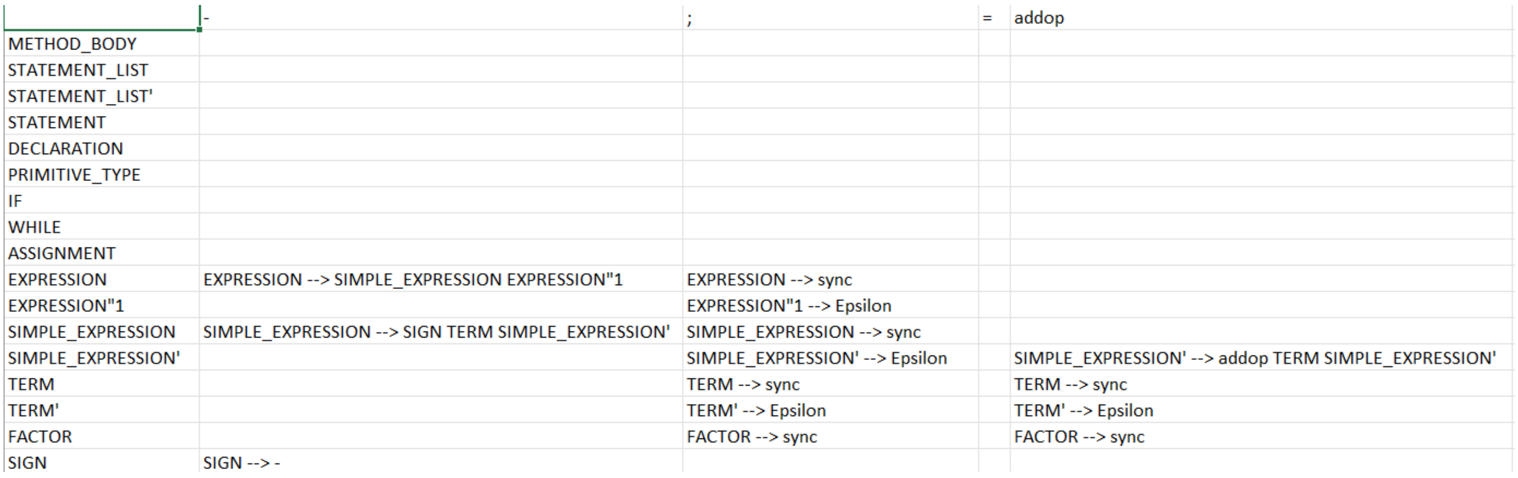
1	stack top	input top	output
2	METHOD_BODY	int	METHOD_BODY --> STATEMENT_LIST
3	STATEMENT_LIST	int	STATEMENT_LIST --> STATEMENT STATEMENT_LIST'
4	STATEMENT	int	STATEMENT --> DECLARATION
5	DECLARATION	int	DECLARATION --> PRIMITIVE_TYPE id ;
6	PRIMITIVE_TYPE	int	PRIMITIVE_TYPE --> int
7	int	int	match " int "
8	id	id	match " id "
9	;	;	match " ; "
10	STATEMENT_LIST'	id	STATEMENT_LIST' --> STATEMENT STATEMENT_LIST'
11	STATEMENT	id	STATEMENT --> ASSIGNMENT
12	ASSIGNMENT	id	ASSIGNMENT --> id = EXPRESSION ;
13	id	id	match " id "
14	=	=	match " = "
15	EXPRESSION	num	EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION"1
16	SIMPLE_EXPRESSION	num	SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION'
17	TERM	num	TERM --> FACTOR TERM'
18	FACTOR	num	FACTOR --> num
19	num	num	match " num "
20	TERM'	;	TERM' --> Epsilon
21	SIMPLE_EXPRESSION'	;	SIMPLE_EXPRESSION' --> Epsilon
22	EXPRESSION"1	;	EXPRESSION"1 --> Epsilon
23	;	;	match " ; "
24	STATEMENT_LIST'	if	STATEMENT_LIST' --> STATEMENT STATEMENT_LIST'
25	STATEMENT	if	STATEMENT --> IF
26	IF	if	IF --> if (EXPRESSION) { STATEMENT } else { STATEMENT }
27	if	if	match " if "
28	(	(	match " ("
29	EXPRESSION	id	EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION"1
30	SIMPLE_EXPRESSION	id	SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION'
31	TERM	id	TERM --> FACTOR TERM'

32	FACTOR	id	FACTOR --> id
33	id	id	match " id "
34	TERM'	relop	TERM' --> Epsilon
35	SIMPLE_EXPRESSION'	relop	SIMPLE_EXPRESSION' --> Epsilon
36	EXPRESSION"1	relop	EXPRESSION"1 --> relop SIMPLE_EXPRESSION
37	relop	relop	match " relop "
38	SIMPLE_EXPRESSION	num	SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION'
39	TERM	num	TERM --> FACTOR TERM'
40	FACTOR	num	FACTOR --> num
41	num	num	match " num "
42	TERM'	)	TERM' --> Epsilon
43	SIMPLE_EXPRESSION'	)	SIMPLE_EXPRESSION' --> Epsilon
44	)	)	match ") "
45	{	{	match " { "
46	STATEMENT	id	STATEMENT --> ASSIGNMENT
47	ASSIGNMENT	id	ASSIGNMENT --> id = EXPRESSION ;
48	id	id	match " id "
49	=	=	match " = "
50	EXPRESSION	num	EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION"1
51	SIMPLE_EXPRESSION	num	SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION'
52	TERM	num	TERM --> FACTOR TERM'
53	FACTOR	num	FACTOR --> num
54	num	num	match " num "
55	TERM'	;	TERM' --> Epsilon
56	SIMPLE_EXPRESSION'	;	SIMPLE_EXPRESSION' --> Epsilon
57	EXPRESSION"1	;	EXPRESSION"1 --> Epsilon
58	;	;	match " ; "
59	}	}	match " } "
60	else	$	Error: missing else .. pop from stack
61	{	$	Error: missing { .. pop from stack
62	STATEMENT	$	Error: sync STATEMENT
63	}	$	Error: missing } .. pop from stack
64	STATEMENT_LIST'	$	STATEMENT_LIST' --> Epsilon
65	$	$	accept

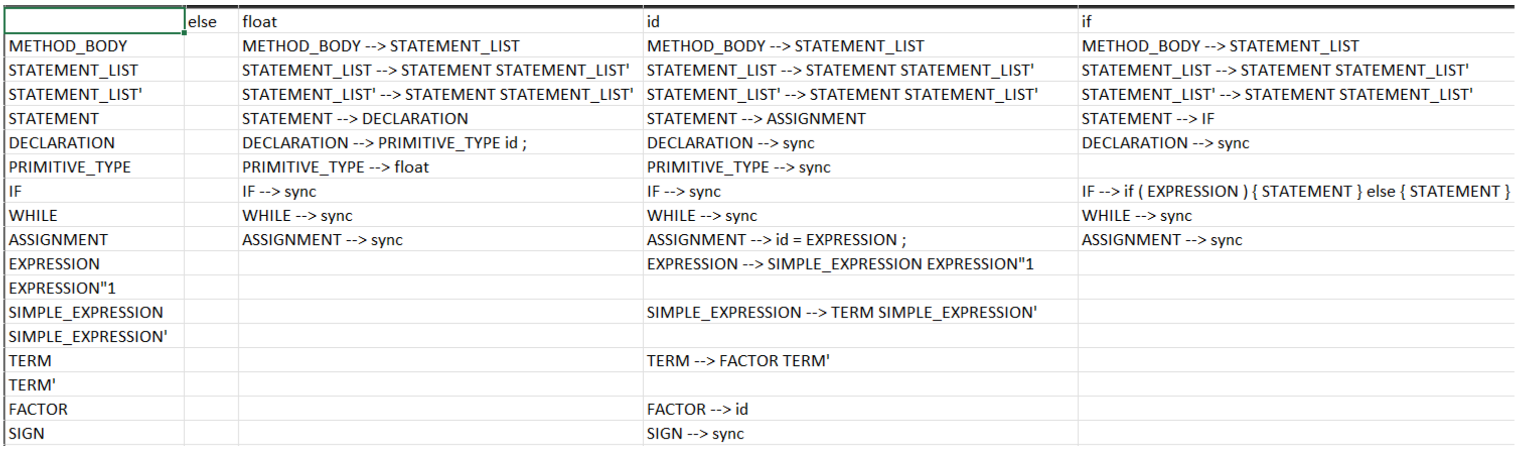
- The file table.csv:



	(	)	+
METHOD_BODY			
STATEMENT_LIST			
STATEMENT_LIST'			
STATEMENT			
DECLARATION			
PRIMITIVE_TYPE			
IF			
WHILE			
ASSIGNMENT			
EXPRESSION	EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION"1	EXPRESSION --> sync	EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION"1
EXPRESSION"1		EXPRESSION"1 --> Epsilon	
SIMPLE_EXPRESSION	SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION'	SIMPLE_EXPRESSION --> sync	SIMPLE_EXPRESSION --> SIGN TERM SIMPLE_EXPRESSION'
SIMPLE_EXPRESSION'		SIMPLE_EXPRESSION' --> Epsilon	
TERM	TERM --> FACTOR TERM'	TERM --> sync	
TERM'		TERM' --> Epsilon	
FACTOR	FACTOR --> (EXPRESSION)	FACTOR --> sync	
SIGN	SIGN --> sync		SIGN --> +



	-	;	=	addop
METHOD_BODY				
STATEMENT_LIST				
STATEMENT_LIST'				
STATEMENT				
DECLARATION				
PRIMITIVE_TYPE				
IF				
WHILE				
ASSIGNMENT				
EXPRESSION	EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION"1	EXPRESSION --> sync		
EXPRESSION"1		EXPRESSION"1 --> Epsilon		
SIMPLE_EXPRESSION	SIMPLE_EXPRESSION --> SIGN TERM SIMPLE_EXPRESSION'	SIMPLE_EXPRESSION --> sync		
SIMPLE_EXPRESSION'		SIMPLE_EXPRESSION' --> Epsilon		SIMPLE_EXPRESSION' --> addop TERM SIMPLE_EXPRESSION'
TERM		TERM --> sync		TERM --> sync
TERM'		TERM' --> Epsilon		TERM' --> Epsilon
FACTOR		FACTOR --> sync		FACTOR --> sync
SIGN	SIGN --> -			



	else	float	id	if
METHOD_BODY		METHOD_BODY --> STATEMENT_LIST	METHOD_BODY --> STATEMENT_LIST	METHOD_BODY --> STATEMENT_LIST
STATEMENT_LIST		STATEMENT_LIST --> STATEMENT STATEMENT_LIST'	STATEMENT_LIST --> STATEMENT STATEMENT_LIST'	STATEMENT_LIST --> STATEMENT STATEMENT_LIST'
STATEMENT_LIST'		STATEMENT_LIST' --> STATEMENT STATEMENT_LIST'	STATEMENT_LIST' --> STATEMENT STATEMENT_LIST'	STATEMENT_LIST' --> STATEMENT STATEMENT_LIST'
STATEMENT		STATEMENT --> DECLARATION	STATEMENT --> ASSIGNMENT	STATEMENT --> IF
DECLARATION		DECLARATION --> PRIMITIVE_TYPE id ;	DECLARATION --> sync	DECLARATION --> sync
PRIMITIVE_TYPE		PRIMITIVE_TYPE --> float	PRIMITIVE_TYPE --> sync	
IF		IF --> sync	IF --> sync	IF --> if (EXPRESSION) { STATEMENT } else { STATEMENT }
WHILE		WHILE --> sync	WHILE --> sync	WHILE --> sync
ASSIGNMENT		ASSIGNMENT --> sync	ASSIGNMENT --> id = EXPRESSION ;	ASSIGNMENT --> sync
EXPRESSION			EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION"1	
EXPRESSION"1				
SIMPLE_EXPRESSION			SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION'	
SIMPLE_EXPRESSION'				
TERM			TERM --> FACTOR TERM'	
TERM'				
FACTOR			FACTOR --> id	
SIGN			SIGN --> sync	

	int	mulop	num
METHOD_BODY	METHOD_BODY --> STATEMENT_LIST		
STATEMENT_LIST	STATEMENT_LIST --> STATEMENT STATEMENT_LIST'		
STATEMENT_LIST'	STATEMENT_LIST' --> STATEMENT STATEMENT_LIST'		
STATEMENT	STATEMENT --> DECLARATION		
DECLARATION	DECLARATION --> PRIMITIVE_TYPE id ;		
PRIMITIVE_TYPE	PRIMITIVE_TYPE --> int		
IF	IF --> sync		
WHILE	WHILE --> sync		
ASSIGNMENT	ASSIGNMENT --> sync		
EXPRESSION			EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION"1
EXPRESSION"1			
SIMPLE_EXPRESSION			SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION'
SIMPLE_EXPRESSION'			
TERM			TERM --> FACTOR TERM'
TERM'		TERM' --> mulop FACTOR TERM'	
FACTOR		FACTOR --> sync	FACTOR --> num
SIGN			SIGN --> sync

	relop	while	{	}	$
METHOD_BODY		METHOD_BODY --> STATEMENT_LIST			METHOD_BODY --> sync
STATEMENT_LIST		STATEMENT_LIST --> STATEMENT STATEMENT_LIST'			STATEMENT_LIST --> sync
STATEMENT_LIST'		STATEMENT_LIST' --> STATEMENT STATEMENT_LIST'			STATEMENT_LIST' --> Epsilon
STATEMENT		STATEMENT --> WHILE		STATEMENT --> sync	STATEMENT --> sync
DECLARATION		DECLARATION --> sync		DECLARATION --> sync	DECLARATION --> sync
PRIMITIVE_TYPE					
IF		IF --> sync		IF --> sync	IF --> sync
WHILE		WHILE --> while (EXPRESSION) { STATEMENT }		WHILE --> sync	WHILE --> sync
ASSIGNMENT		ASSIGNMENT --> sync		ASSIGNMENT --> sync	ASSIGNMENT --> sync
EXPRESSION					
EXPRESSION"1	EXPRESSION"1 --> relop SIMPLE_EXPRESSION				
SIMPLE_EXPRESSION	SIMPLE_EXPRESSION --> sync				
SIMPLE_EXPRESSION'	SIMPLE_EXPRESSION' --> Epsilon				
TERM	TERM --> sync				
TERM'	TERM' --> Epsilon				
FACTOR	FACTOR --> sync				
SIGN					