Programming Languages Translation

Phase 2: Parser Generator

Shahenda Elsayed 3695

Nouran Mahmoud Bakry 3679

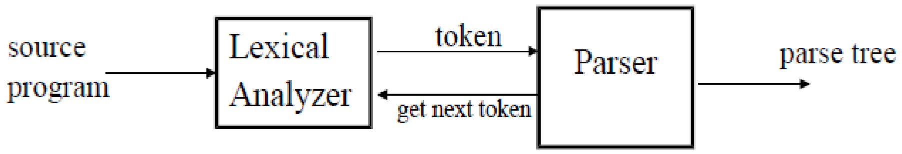
Khalid AlSafawany 3211

Maram Elsayed 3526

Introduction:

• Syntax Analyzer creates the syntactic structure of the given source program.
• The syntax of a programming is described by a context-free grammar (CFG).
• The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
– If it satisfies, the parser creates the parse tree of that program.
– Otherwise the parser gives the error messages.

The parser generator expects an LL (1) grammar as input so it eliminating grammar left recursion and performing left factoring then computes First and Follow sets and uses them to construct a predictive parsing table for the grammar.
The table is to be used to drive a predictive top-down parser.



Left Recursion and Left Factoring:

Class LL1 is responsible for left recursion and left factoring to produce an ambiguity free LL(1) grammar.

Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

Elimination of Left Recursion Algorithm:

> - Arrange non-terminals in some order: $A_1 \ldots A_n$
> - **for** i **from** 1 **to** n **do** {
>   - **for** j **from** 1 **to** i-1 **do** {
>   replace each production
> $A_i \rightarrow A_j \gamma$
>   by
> $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$
>   where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$
>   }
>   - eliminate immediate left-recursions among A_i productions
> }

Class LL1 has a function eliminate which returns an object of type CFG that contains the new productions after elimination of left recursion and after applying left factoring. A LinkedHashmap is used to for carrying all the newly formed productions and the array list of non terminals is also provided. If the grammar has no left recursion then the productions will not be modified and non terminals will remain the same.

After looping through productions and removing any left recursion found the functions checks if the current grammar requires left factoring.

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top down parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen to make the right choice.

Left Factoring Algorithm:

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

  $$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

  convert it into

  $$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
  $$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

The function then loops through the productions and upon finding a production that requires left factoring applies the rule and replaces the production in the LinkedHashmap.

The final output of this class will be CFG object containing ambiguity free grammar.

Assumptions: epsilon is represented by the symbol '~'

First and Follow :

In first_follow class ≫ we implement a recursion function for computing first that take all non terminals and return it is first depending on this cases :

- If X is a terminal symbol ➔ FIRST(X)={X}
- If X is a non-terminal symbol and $X \to \varepsilon$ is a production rule
  ➔ ε is in FIRST(X).
- If X is a non-terminal symbol and $X \to Y_1Y_2..Y_n$ is a production rule
  ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then **a** is in FIRST(X).
  ➔ if ε is in all FIRST(Y_j) for j=1,...,n then ε is in FIRST(X).

Data structure :

Arraylist of String LHS for all non terminals .

Arraylist of String for terminals.

Arraylist of String first for all first that return from function.

In a follow function implement this cases :

- If S is the start symbol ➔ \$ is in FOLLOW(S)
- if $A \to \alpha B\beta$ is a production rule
  ➔ everything in FIRST(β) is FOLLOW(B) except ε
- If ($A \to \alpha B$ is a production rule) or
  ($A \to \alpha B\beta$ is a production rule and ε is in FIRST(β))
  ➔ everything in FOLLOW(A) is in FOLLOW(B).

We apply these rules until nothing more can be added to any follow set.

Data structure :

Arraylist of String LHS for all non terminals .

Arraylist of String for terminals.

Arraylist of String for right to get the production for nonterminals .

Arraylist of String follow for each follow .

Map of string and Arraylist of String follow_set to indicate for each non terminal and it is follow .

LL(1) Parsing Tables :

We use first and follow to construct the table so in class parser table it takes CFG and first and follow .

In function createTable() : it creates row of terminals and column of non terminals .

Using Constructing LL(1) parsing Table Algorithm :

Constructing LL(1) Parsing Table -- Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
  - for each terminal a in FIRST(α)
    ➔ add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α)
    ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α) and \$ in FOLLOW(A)
    ➔ add $A \rightarrow \alpha$ to M[A,\$]

Data structure :

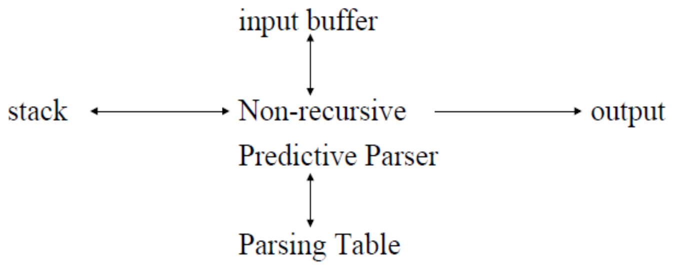
2D Array of String for table.

Map of string and Arraylist of String followMap to take the follow for each non terminal in first_follow class.

Array of String first to get the first .

Arraylist of String to full the productions in the table

LL(1) Parser :

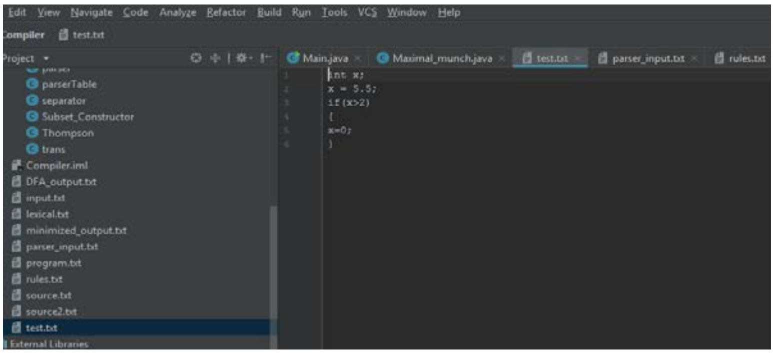


• Input: Parsing table and string of tokens.
• Output: Left derivation, stack, input and production.

We created parser class.

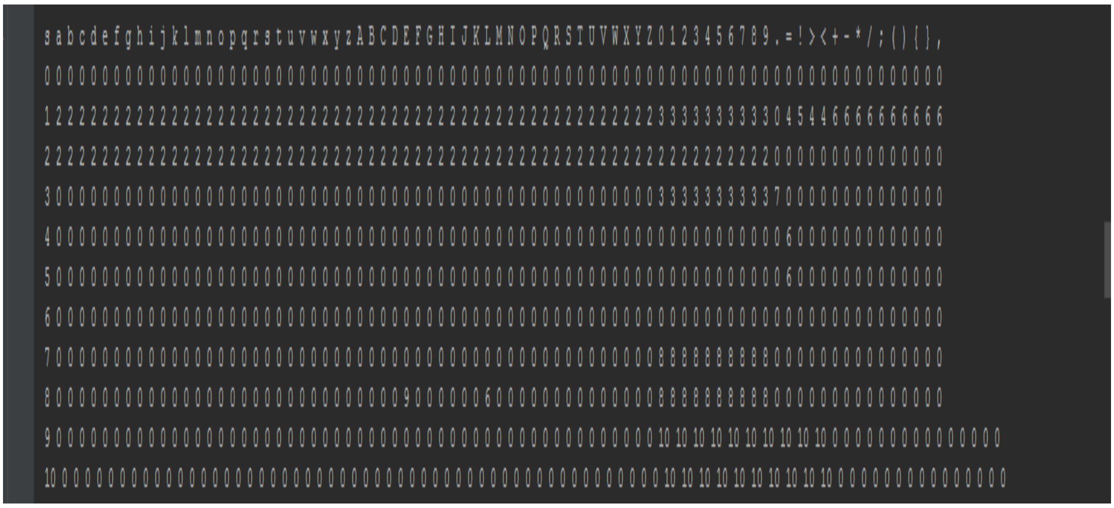
In parser class >> Frist we create a stack and push “$” and the start symbol.
A while loop is then created to iterate on the input string, the current input token is compared with the top of the stack if they are equal the input is consumed and we will move to the next token, else the parsing table is checked to get the production.
If there is a production in the table cell [non_terminal][input token] the production is reversed using function reversereverse(String str) and is then pushed in the stack.
If cell was null Panic-Mode Error Recovery is applied and the input token is discard, else if it was SYNC the top of the stack is discarded.
If all the input string is consumed and the stack was not empty the stack is emptied.
An arraylist is created which hold the left most derivation.
An index is used to point to the first non_terminal in the arraylist every time the a new production is used the non_terminal is removed and replaced by the new production.

The test file



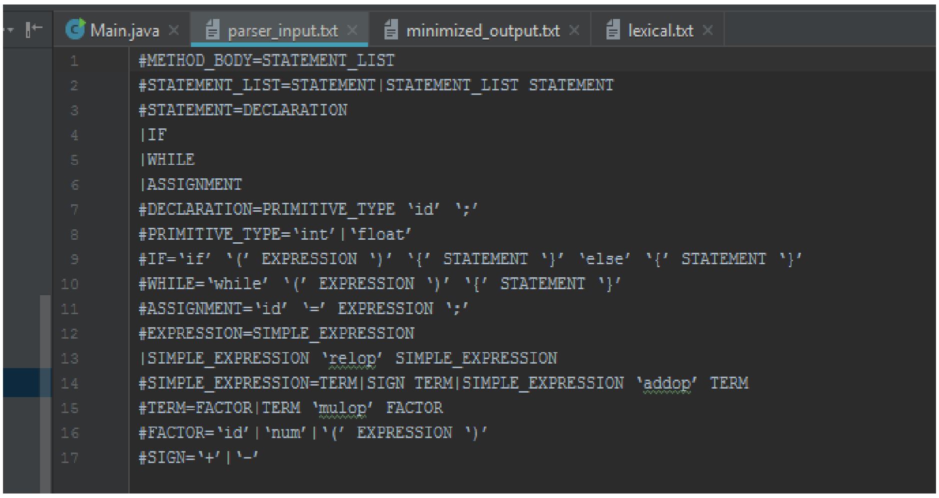

```
int x;
x = 5.5;
if(x>2)
{
x=0;
}
```

The transition table from lexical



```
s a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 . = ! > < + - * / ; ( ) { } ,
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 0 4 5 4 4 6 6 6 6 6 6 6 6 6 6
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 3 3 3 3 3 3 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0
6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 8 8 8 8 8 8 8 8 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 8 8 8 8 8 8 8 8 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 10 10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10 10 10 10 10 10 10 10 10 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The input of parser




```
#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='+'|'-'
```

LL(1) Parser

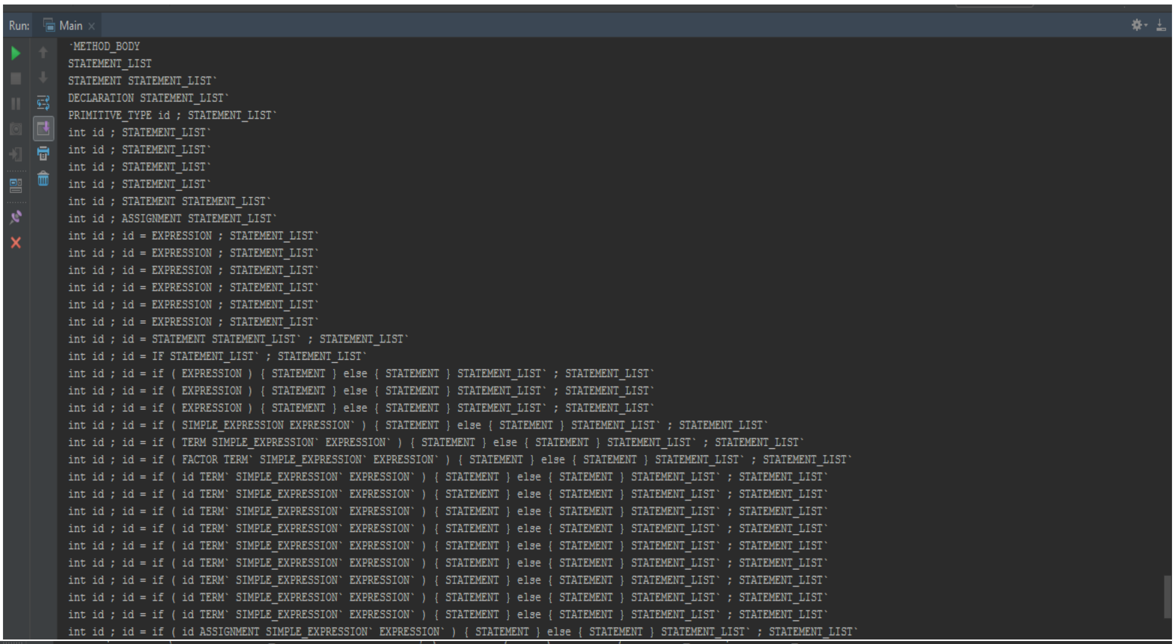
Predictive parsing table

```
predictive parsing table
[null, id, ;, int, float, if, (, ), {, }, else, while, =, relop, addop, mulop, num, +, -, $]
[`METHOD_BODY, STATEMENT_LIST, null, STATEMENT_LIST, STATEMENT_LIST, STATEMENT_LIST, null, null, null, null, null, STATEMENT_LIST, null, null, null, null, null, null, null, SYNC]
[STATEMENT_LIST, STATEMENT STATEMENT_LIST`, null, STATEMENT STATEMENT_LIST`, STATEMENT STATEMENT_LIST`, STATEMENT STATEMENT_LIST`, null, null, null, null, null, STATEMENT STATEMENT_LIS
[STATEMENT_LIST`, STATEMENT STATEMENT_LIST`, null, STATEMENT STATEMENT_LIST`, STATEMENT STATEMENT_LIST`, STATEMENT STATEMENT_LIST`, null, null, null, null, null, STATEMENT STATEMENT_LI
[STATEMENT, ASSIGNMENT, null, DECLARATION, DECLARATION, IF, null, null, null, SYNC, null, WHILE, null, null, null, null, null, null, null, SYNC]
[DECLARATION, SYNC, null, PRIMITIVE_TYPE id ;, PRIMITIVE_TYPE id ;, SYNC, null, null, null, SYNC, null, SYNC, null, null, null, null, null, null, null, SYNC]
[PRIMITIVE_TYPE, SYNC, null, int, float, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null]
[IF, SYNC, null, SYNC, SYNC, if ( EXPRESSION ) { STATEMENT } else { STATEMENT } , null, null, null, SYNC, null, SYNC, null, null, null, null, null, null, null, SYNC]
[WHILE, SYNC, null, SYNC, SYNC, SYNC, null, null, null, SYNC, null, while ( EXPRESSION ) { STATEMENT } , null, null, null, null, null, null, null, SYNC]
[ASSIGNMENT, id = EXPRESSION ; , null, SYNC, SYNC, SYNC, null, null, null, SYNC, null, SYNC, null, null, null, null, null, null, null, SYNC]
[EXPRESSION, SIMPLE_EXPRESSION EXPRESSION`, SYNC, null, null, null, SIMPLE_EXPRESSION EXPRESSION`, SYNC, null, null, null, null, null, null, null, null, SIMPLE_EXPRESSION EXPRESSION`, 
[EXPRESSION`, null, ~, null, null, null, null, ~, null, null, null, null, null, relop SIMPLE_EXPRESSION , null, null, null, null, null, null]
[SIMPLE_EXPRESSION, TERM SIMPLE_EXPRESSION`, SYNC, null, null, null, TERM SIMPLE_EXPRESSION`, SYNC, null, null, null, null, null, null, null, null, TERM SIMPLE_EXPRESSION`, SIGN TERM S
[SIMPLE_EXPRESSION`, null, ~, null, null, null, null, ~, null, null, null, null, null, null, addop TERM SIMPLE_EXPRESSION` , null, null, null, null, null]
[TERM, FACTOR TERM`, SYNC, null, null, null, FACTOR TERM`, SYNC, null, null, null, null, null, null, SYNC, null, FACTOR TERM`, null, null, null]
[TERM`, null, ~, null, null, null, null, ~, null, null, null, null, null, ~, mulop FACTOR TERM` , null, null, null, null]
[FACTOR, id, SYNC, null, null, null, ( EXPRESSION ) , SYNC, null, null, null, null, null, null, SYNC, SYNC, num, null, null, null]
[SIGN, SYNC, SYNC, null, null, null, SYNC, SYNC, null, null, null, null, null, null, null, null, SYNC, +, -, null]
```

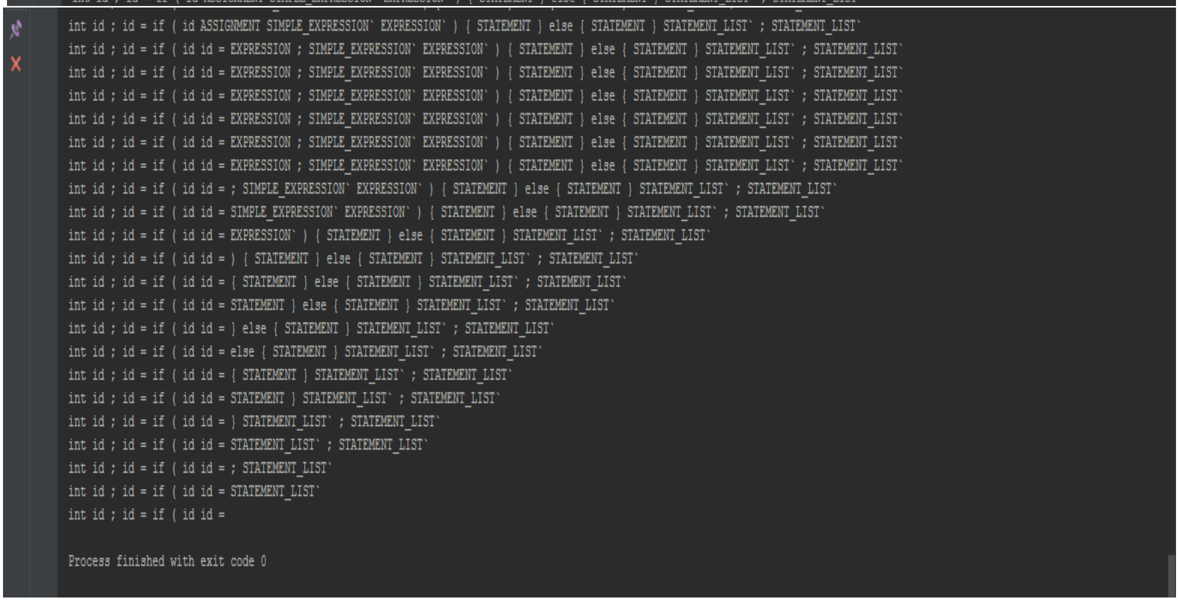
Stack input output

```
Run: Main
$`METHOD_BODY   int id ; id = float ; if ( id relop int ) { id = int ; }    `METHOD_BODY->STATEMENT_LIST
$STATEMENT_LIST int id ; id = float ; if ( id relop int ) { id = int ; }    STATEMENT_LIST->STATEMENT STATEMENT_LIST`
$STATEMENT_LIST`STATEMENT   int id ; id = float ; if ( id relop int ) { id = int ; }    STATEMENT->DECLARATION
$STATEMENT_LIST`DECLARATION int id ; id = float ; if ( id relop int ) { id = int ; }    DECLARATION->PRIMITIVE_TYPE id ;
$STATEMENT_LIST`;idPRIMITIVE_TYPE   int id ; id = float ; if ( id relop int ) { id = int ; }    PRIMITIVE_TYPE->int
$STATEMENT_LIST`;idint  int id ; id = float ; if ( id relop int ) { id = int ; }
$STATEMENT_LIST`;id id ; id = float ; if ( id relop int ) { id = int ; }
$STATEMENT_LIST`;   ; id = float ; if ( id relop int ) { id = int ; }
$STATEMENT_LIST`    id = float ; if ( id relop int ) { id = int ; }     STATEMENT_LIST`->STATEMENT STATEMENT_LIST`
$STATEMENT_LIST`STATEMENT   id = float ; if ( id relop int ) { id = int ; }     STATEMENT->ASSIGNMENT
$STATEMENT_LIST`ASSIGNMENT  id = float ; if ( id relop int ) { id = int ; }     ASSIGNMENT->id = EXPRESSION ;
$STATEMENT_LIST`;EXPRESSION=id  id = float ; if ( id relop int ) { id = int ; }
$STATEMENT_LIST`;EXPRESSION=    = float ; if ( id relop int ) { id = int ; }
$STATEMENT_LIST`;EXPRESSION float ; if ( id relop int ) { id = int ; }  Error:(illegal EXPRESSION) - discard float
$STATEMENT_LIST`;EXPRESSION ; if ( id relop int ) { id = int ; }
$STATEMENT_LIST`;   ; if ( id relop int ) { id = int ; }
$STATEMENT_LIST`    if ( id relop int ) { id = int ; }  STATEMENT_LIST`->STATEMENT STATEMENT_LIST`
$STATEMENT_LIST`STATEMENT   if ( id relop int ) { id = int ; }  STATEMENT->IF
$STATEMENT_LIST`IF  if ( id relop int ) { id = int ; }  IF->if ( EXPRESSION ) { STATEMENT } else { STATEMENT }
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION(if    if ( id relop int ) { id = int ; }
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION(  ( id relop int ) { id = int ; }
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION   id relop int ) { id = int ; }   EXPRESSION->SIMPLE_EXPRESSION EXPRESSION`
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION`SIMPLE_EXPRESSION id relop int ) { id = int ; }   SIMPLE_EXPRESSION->TERM SIMPLE_EXPRESSION`
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION`SIMPLE_EXPRESSION`TERM    id relop int ) { id = int ; }   TERM->FACTOR TERM`
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION`SIMPLE_EXPRESSION`TERM`FACTOR id relop int ) { id = int ; }   FACTOR->id
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION`SIMPLE_EXPRESSION`TERM`id id relop int ) { id = int ; }
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION`SIMPLE_EXPRESSION`TERM`   relop int ) { id = int ; }  Error:(illegal TERM`) - discard relop
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION`SIMPLE_EXPRESSION`TERM`   int ) { id = int ; }    Error:(illegal TERM`) - discard int
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION`SIMPLE_EXPRESSION`TERM`   ) { id = int ; }    TERM`->~
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION`SIMPLE_EXPRESSION`    ) { id = int ; }    SIMPLE_EXPRESSION`->~
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{)EXPRESSION`  ) { id = int ; }    EXPRESSION`->~
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{) ) { id = int ; }
$STATEMENT_LIST`}STATEMENT{else}STATEMENT{  { id = int ; }
$STATEMENT_LIST`}STATEMENT{else}STATEMENT   id = int ; }    STATEMENT->ASSIGNMENT
$STATEMENT_LIST`}STATEMENT{else}ASSIGNMENT  id = int ; }    ASSIGNMENT->id = EXPRESSION ;
$STATEMENT_LIST`}STATEMENT{else};EXPRESSION=id  id = int ; }
$STATEMENT_LIST`}STATEMENT{else};EXPRESSION=    = int ; }
$STATEMENT_LIST`}STATEMENT{else};EXPRESSION int ; }     Error:(illegal EXPRESSION) - discard int
$STATEMENT_LIST`}STATEMENT{else};EXPRESSION ; }
$STATEMENT_LIST`}STATEMENT{else};   ; }
$STATEMENT_LIST`}STATEMENT{else}    }
$STATEMENT_LIST`}STATEMENT{else
$STATEMENT_LIST`}STATEMENT{
$STATEMENT_LIST`}STATEMENT
$STATEMENT_LIST`}
$STATEMENT_LIST`
$
accept, successful completion
```

Leftmost derivation



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



```
int id ; id = if ( id ASSIGNMENT SIMPLE_EXPRESSION` EXPRESSION` ) { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = EXPRESSION ; SIMPLE_EXPRESSION` EXPRESSION` ) { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = EXPRESSION ; SIMPLE_EXPRESSION` EXPRESSION` ) { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = EXPRESSION ; SIMPLE_EXPRESSION` EXPRESSION` ) { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = EXPRESSION ; SIMPLE_EXPRESSION` EXPRESSION` ) { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = EXPRESSION ; SIMPLE_EXPRESSION` EXPRESSION` ) { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = EXPRESSION ; SIMPLE_EXPRESSION` EXPRESSION` ) { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = ; SIMPLE_EXPRESSION` EXPRESSION` ) { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = SIMPLE_EXPRESSION` EXPRESSION` ) { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = EXPRESSION` ) { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = ) { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = { STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = STATEMENT } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = } else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = else { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = { STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = STATEMENT } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = } STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = STATEMENT_LIST` ; STATEMENT_LIST`
int id ; id = if ( id id = ; STATEMENT_LIST`
int id ; id = if ( id id = STATEMENT_LIST`
int id ; id = if ( id id =

Process finished with exit code 0
```