Assignment 2

Program 1 -

```
%% This is the first program, where we just have to verify whether the input string adheres to
the grammar.
%% Here I have taken care of precedence and associativity rule, Moreover eliminated the left
recursion.
%% Code is written in such a way that it is self explanatory and dont require much of comments
I believe
%% Elimination of left recursion in Command and declaration is not done using alpha beta rule,
but by the method taught in class.
%% For expressions, they are first made left associative then eliminated left recursion using
alpha beta rule.

program --> block,[.].
block --> [begin], declaration, [;], command, [end].

declaration --> single_declaration, [;], declaration.
declaration --> single_declaration.
single_declaration --> [const], identifier, [=], number.
single_declaration --> [var], identifier.

command --> single_command, [;], command.
command --> single_command.
single_command --> identifier, [:=], expression.
single_command --> [if], boolean_exp, [then], command, [else], command, [endif].
single_command --> [while], boolean_exp, [do], command, [endwhile].
single_command --> block.

boolean_exp --> [true].
boolean_exp --> [false].
boolean_exp --> expression, [=], expression.
boolean_exp --> [not], boolean_exp.

expression --> term1, expression_not.
expression_not --> [+], term1, expression_not | [].
term1 --> term2, term1_not.
term1_not --> [-], term2, term1_not | [].
term2 --> term3, term2_not.
term2_not --> [*], term3, term2_not | [].
term3 --> term4, term3_not.
term3_not --> [/], term4, term3_not | [].
term4 --> identifier.
term4 --> number.
```

```
identifier --> [x] | [y] | [z] | [u] | [v].
number --> [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9].
```

```
?- program([begin, const, x, =, 8, ;, var, y, ;, var, z, ;, z, :=, 0, ;, if, x, =, y, +, 2,
then, z , := , 5, else, z, :=, 3, endif, ;, while, not, x, =, z, do, z, :=, z, +, 2, endwhile,
end, .],[]).
true ;
```

Program 2

```
%% This is the second program where parse tree is created for the input string that follows
the grammar


program(t_program(X,.)) --> block(X),[.].
block(t_block(begin,X,;,Y,end)) --> [begin], declaration(X), [;], command(Y), [end].

declaration(t_declaration(X,;,Y)) --> single_declaration(X), [;], declaration(Y).
declaration(t_declaration(X)) --> single_declaration(X).
single_declaration(t_single_declaration(const,X,=,Y)) --> [const], identifier(X), [=],
number(Y).
single_declaration(t_single_declaration(var,X)) --> [var], identifier(X).

command(t_command(X,;,Y)) --> single_command(X), [;], command(Y).
command(t_command(X)) --> single_command(X).
single_command(t_single_command(X,:=,Y)) --> identifier(X), [:=], expression(Y).
single_command(t_single_command(if,X,then,Y,else,Z,endif)) --> [if], boolean_exp(X), [then],
command(Y), [else], command(Z), [endif].
single_command(t_single_command(while,X,do,Y,endwhile)) --> [while], boolean_exp(X), [do],
command(Y), [endwhile].
single_command(t_single_command(X)) --> block(X).

boolean_exp(t_boolean_exp(true)) --> [true].
boolean_exp(t_boolean_exp(false)) --> [false].
boolean_exp(t_boolean_exp(X,=,Y)) --> expression(X), [=], expression(Y).
boolean_exp(t_boolean_exp(not,X)) --> [not], boolean_exp(X).

expression(t_expression(X,Y)) --> term1(X), expression_not(Y).
expression_not(t_plus(+,X,Y)) --> [+], term1(X), expression_not(Y).
expression_not(t_plus(e)) --> [].
```

```
term1(t_term1(X,Y)) --> term2(X), term1_not(Y).
term1_not(t_minus(-,X,Y)) --> [-], term2(X), term1_not(Y).
term1_not(t_minus(e)) --> [].
term2(t_term2(X,Y)) --> term3(X), term2_not(Y).
term2_not(t_multiply(*,X,Y)) --> [*], term3(X), term2_not(Y).
term2_not(t_multiply(e)) --> [].
term3(t_term3(X,Y)) --> term4(X), term3_not(Y).
term3_not(t_divide(/,X,Y)) --> [/], term4(X), term3_not(Y).
term3_not(t_divide(e)) --> [].
term4(t_id(X)) --> identifier(X).
term4(t_num(X)) --> number(X).

identifier(t_identifier(x)) --> [x].
identifier(t_identifier(y)) --> [y].
identifier(t_identifier(z)) --> [z].
identifier(t_identifier(u)) --> [u].
identifier(t_identifier(v)) --> [v].

number(t_number(0)) --> [0].
number(t_number(1)) --> [1].
number(t_number(2)) --> [2].
number(t_number(3)) --> [3].
number(t_number(4)) --> [4].
number(t_number(5)) --> [5].
number(t_number(6)) --> [6].
number(t_number(7)) --> [7].
number(t_number(8)) --> [8].
number(t_number(9)) --> [9].
```

```
L =  [begin, const, x, =, 8, ;, var, y, ;, var, z, ;, z, :=, 0, ;, if, x, =, y, +, 2, then, z
, := , 5, else, z, :=, 3, endif, ;, while, not, x, =, z, do, z, :=, z, +, 2, endwhile, end,
.],program(P, L, []).

L = [begin, const, x, =, 8, ;, var, y, ;, var, z, ;, z, :=, 0, ;, if, x, =, y, +, 2, then, z,
:=, 5, else, z, :=, 3, endif, ;, while, not, x, =, z, do, z, :=, z, +, 2, endwhile, end, '.'],

P = t_program(t_block(begin, t_declaration(t_single_declaration(const, t_identifier(x), =,
t_number(8)), ;, t_declaration(t_single_declaration(var, t_identifier(y)), ;,
t_declaration(t_single_declaration(var, t_identifier(z))))), ;,
t_command(t_single_command(t_identifier(z), :=,
t_expression(t_term1(t_term2(t_term3(t_num(t_number(0)), t_divide(e)), t_multiply(e)),
t_minus(e)), t_plus(e))), ;, t_command(t_single_command(if,
t_boolean_exp(t_expression(t_term1(t_term2(t_term3(t_id(t_identifier(x)), t_divide(e)),
t_multiply(e)), t_minus(e)), t_plus(e)), =,
t_expression(t_term1(t_term2(t_term3(t_id(t_identifier(y)), t_divide(e)), t_multiply(e)),
t_minus(e)), t_plus(+, t_term1(t_term2(t_term3(t_num(t_number(2)), t_divide(e)),
t_multiply(e)), t_minus(e)), t_plus(e)))), then, t_command(t_single_command(t_identifier(z),
```

```
:=, t_expression(t_term1(t_term2(t_term3(t_num(t_number(5)), t_divide(e)), t_multiply(e)),
t_minus(e)), t_plus(e)))), else, t_command(t_single_command(t_identifier(z), :=,
t_expression(t_term1(t_term2(t_term3(t_num(t_number(3)), t_divide(e)), t_multiply(e)),
t_minus(e)), t_plus(e)))), endif), ;, t_command(t_single_command(while, t_boolean_exp(not,
t_boolean_exp(t_expression(t_term1(t_term2(t_term3(t_id(t_identifier(x)), t_divide(e)),
t_multiply(e)), t_minus(e)), t_plus(e)), =,
t_expression(t_term1(t_term2(t_term3(t_id(t_identifier(z)), t_divide(e)), t_multiply(e)),
t_minus(e)), t_plus(e)))), do, t_command(t_single_command(t_identifier(z), :=,
t_expression(t_term1(t_term2(t_term3(t_id(t_identifier(z)), t_divide(e)), t_multiply(e)),
t_minus(e)), t_plus(+, t_term1(t_term2(t_term3(t_num(t_number(2)), t_divide(e)),
t_multiply(e)), t_minus(e)), t_plus(e))))), endwhile)))), end), '.') .
```