| **РБНФ** | **Код для перевірки РБНФ** |
| --- | --- |
|  |  |
|  |  |
| program\_name = ident; | program\_name = SAME\_RULE(ident); |
| value\_type = "Integer"; | value\_type = SAME\_RULE(tokenInteger); |
| declaration\_element = ident , [ "[", unsigned\_value , "]" ]; | declaration\_element = ident >> -(tokenLEFTSQUAREBRACKETS >> unsigned\_value >> tokenRIGHTSQUAREBRACKETS); |
| other\_declaration\_ident = "," , declaration\_element; | other\_declaration\_ident = tokenCOMMA >> declaration\_element; |
| declaration = value\_type , declaration\_element , {other\_declaration\_ident}; | declaration = value\_type >> declaration\_element >> \*other\_declaration\_ident; |
| index\_action = "[" , expression , "]"; | index\_action = tokenLEFTSQUAREBRACKETS >> expression >> tokenRIGHTSQUAREBRACKETS; |
| unary\_operator = "!="; | unary\_operator = SAME\_RULE(tokenNOT); |
| unary\_operation = unary\_operator , expression; | unary\_operation = unary\_operator >> expression; |
| binary\_operator = "&&" | "||" | "==" | "!=" | "Le" | "Ge" | "+" | "-" | "Mul" | "Div" | "Mod"; | binary\_operator = tokenAND | tokenOR | tokenEQUAL | tokenNOTEQUAL | tokenLESSOREQUAL | tokenGREATEROREQUAL | tokenPLUS | tokenMINUS | tokenMUL | tokenDIV | tokenMOD; |
| binary\_action = binary\_operator , expression; | binary\_action = binary\_operator >> expression; |
| left\_expression = group\_expression | unary\_operation | ident , [index\_action] | value | cond\_block\_\_with\_optionally\_return\_value; | left\_expression = group\_expression | unary\_operation | ident >> -index\_action | value | cond\_block\_\_with\_optionally\_return\_value; |
| expression = left\_expression , {binary\_action}; | expression = left\_expression >> \*binary\_action; |
| group\_expression = "(" , expression , ")"; | group\_expression = tokenGROUPEXPRESSIONBEGIN >> expression >> tokenGROUPEXPRESSIONEND; |
| bind\_left\_to\_right = expression , "->" , ident , [index\_action]; | bind\_left\_to\_right = expression >> tokenLRBIND >> ident >> -index\_action; |
| if\_expression = expression; | if\_expression = SAME\_RULE(expression); |
| body\_for\_true\_\_with\_optionally\_return\_value = block\_statements\_\_with\_optionally\_return\_value; | body\_for\_true\_\_with\_optionally\_return\_value = SAME\_RULE(block\_statements\_\_with\_optionally\_return\_value); |
| false\_cond\_block\_without\_else\_\_with\_optionally\_return\_value = "Else" , "If" , if\_expression , body\_for\_true\_\_with\_optionally\_return\_value; | false\_cond\_block\_without\_else\_\_with\_optionally\_return\_value = tokenELSE >> tokenIF >> if\_expression >> body\_for\_true\_\_with\_optionally\_return\_value; |
| body\_for\_false\_\_with\_optionally\_return\_value = "Else" , block\_statements\_\_with\_optionally\_return\_value; | body\_for\_false\_\_with\_optionally\_return\_value = tokenELSE >> block\_statements\_\_with\_optionally\_return\_value; |
| cond\_block\_\_with\_optionally\_return\_value = "If" , if\_expression , body\_for\_true\_\_with\_optionally\_return\_value , {false\_cond\_block\_without\_else\_\_with\_optionally\_return\_value} , [body\_for\_false\_\_with\_optionally\_return\_value]; | cond\_block\_\_with\_optionally\_return\_value = tokenIF >> if\_expression >> body\_for\_true\_\_with\_optionally\_return\_value >> \*false\_cond\_block\_without\_else\_\_with\_optionally\_return\_value >> -body\_for\_false\_\_with\_optionally\_return\_value; |
| cond\_block\_\_with\_optionally\_return\_value\_and\_optionally\_bind = cond\_block\_\_with\_optionally\_return\_value , [tokenLRBIND , ident , [index\_action]]; | cond\_block\_\_with\_optionally\_return\_value\_and\_optionally\_bind = cond\_block\_\_with\_optionally\_return\_value >> -(tokenLRBIND >> ident >> -index\_action); |
| cycle\_begin\_expression = expression; | cycle\_begin\_expression = SAME\_RULE(expression); |
| cycle\_end\_expression = expression; | cycle\_end\_expression = SAME\_RULE(expression); |
| cycle\_counter = ident; | cycle\_counter = SAME\_RULE(ident); |
| cycle\_counter\_lr\_init = cycle\_begin\_expression , "=:" , cycle\_counter; | cycle\_counter\_lr\_init = cycle\_begin\_expression >> tokenLRBIND >> cycle\_counter; |
| cycle\_counter\_init = cycle\_counter\_lr\_init; | cycle\_counter\_init = SAME\_RULE(cycle\_counter\_lr\_init); |
| cycle\_counter\_last\_value = cycle\_end\_expression; | cycle\_counter\_last\_value = SAME\_RULE(cycle\_end\_expression); |
|  | cycle\_body = tokenDO >> (statement | block\_statements); |
|  | forto\_cycle = tokenFOR >> cycle\_counter\_init >> (tokenTO | tokenDOWNTO) >> cycle\_counter\_last\_value >> cycle\_body; |
|  | continue\_while = SAME\_RULE(tokenCONTINUE); |
|  | break\_while = SAME\_RULE(tokenBREAK); |
| statement\_in\_while\_and\_if\_body = statement | "Continue" | "Break"; | statement\_in\_while\_and\_if\_body = statement | continue\_while | break\_while; |
| block\_statements\_in\_while\_and\_if\_body = "{" , {statement\_in\_while\_and\_if\_body} , "}"; | block\_statements\_in\_while\_and\_if\_body = tokenBEGINBLOCK >> \*statement\_in\_while\_and\_if\_body >> tokenENDBLOCK; |
| while\_cycle\_head\_expression = expression; | while\_cycle\_head\_expression = SAME\_RULE(expression); |
| while\_cycle = "While" , while\_cycle\_head\_expression , block\_statements\_in\_while\_and\_if\_body; | while\_cycle = tokenWHILE >> while\_cycle\_head\_expression >> block\_statements\_in\_while\_and\_if\_body; |
|  |  |
|  |  |
| input = "Input" , ( ident , [index\_action] | "(" , ident , [index\_action] , ")" ); | input = tokenInput >> (ident >> -index\_action | tokenGROUPEXPRESSIONBEGIN >> ident >> -index\_action >> tokenGROUPEXPRESSIONEND); |
| output = "Output" , expression; | output = tokenOutput >> expression; |
| statement = bind\_left\_to\_right | cond\_block\_\_with\_optionally\_return\_value\_and\_optionally\_bind | forto\_cycle | while\_cycle | repeat\_until\_cycle | labeled\_point | goto\_label | input | output | ";"; | statement = bind\_left\_to\_right | cond\_block\_\_with\_optionally\_return\_value\_and\_optionally\_bind | forto\_cycle | while\_cycle | repeat\_until\_cycle | labeled\_point | goto\_label | input | output | tokenSEMICOLON; |
| block\_statements = "{" , {statement} , "}"; | block\_statements = tokenBEGINBLOCK >> \*statement >> tokenENDBLOCK; |
| block\_statements\_\_with\_optionally\_return\_value = "{" , {statement\_in\_while\_and\_if\_body} , [expression] , "}"; | block\_statements\_\_with\_optionally\_return\_value = tokenBEGINBLOCK >> \*statement\_in\_while\_and\_if\_body >> -expression >> tokenENDBLOCK; |
| program = "Program" , program\_name , ";" , "Var" , [declaration] , ";" ,  “Start”, {statement} , "Finish"; | program = BOUNDARIES >> tokenNAME >> program\_name >> tokenSEMICOLON >> tokenDATA >> (-declaration) >> tokenSEMICOLON >> tokenStart >> \*statement >> tokenEnd; |
| digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"; | digit = digit\_0 | digit\_1 | digit\_2 | digit\_3 | digit\_4 | digit\_5 | digit\_6 | digit\_7 | digit\_8 | digit\_9; |
| non\_zero\_digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"; | non\_zero\_digit = digit\_1 | digit\_2 | digit\_3 | digit\_4 | digit\_5 | digit\_6 | digit\_7 | digit\_8 | digit\_9; |
| unsigned\_value = (non\_zero\_digit , {digit}) | "0"; | unsigned\_value = ((non\_zero\_digit >> \*digit) | digit\_0) >> BOUNDARIES; |
| value = [sign] , unsigned\_value; | value = -sign >> unsigned\_value >> BOUNDARIES; |
| letter\_in\_lower\_case = "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"; | letter\_in\_lower\_case = 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; |
| letter\_in\_upper\_case = "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"; | letter\_in\_upper\_case = 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; |
| ident = letter\_in\_lower\_case , letter\_in\_lower\_case , letter\_in\_lower\_case , letter\_in\_lower\_case | ident = letter\_in\_lower\_case >> letter\_in\_lower\_case >> letter\_in\_lower\_case >> letter\_in\_lower\_case >> STRICT\_BOUNDARIES; |
| sign = "+" | "-"; | sign = sign\_plus | sign\_minus; |
|  | sign\_plus = SAME\_RULE(tokenPLUS); |
|  | sign\_minus = SAME\_RULE(tokenMINUS); |
|  | digit\_0 = '0'; |
|  | digit\_1 = '1'; |
|  | digit\_2 = '2'; |
|  | digit\_3 = '3'; |
|  | digit\_4 = '4'; |
|  | digit\_5 = '5'; |
|  | digit\_6 = '6'; |
|  | digit\_7 = '7'; |
|  | digit\_8 = '8'; |
|  | digit\_9 = '9'; |
|  |  |
|  |  |
|  | tokenInteger = "Integer" >> STRICT\_BOUNDARIES; |
|  | tokenCOMMA = "," >> BOUNDARIES; |
|  | tokenNOT = "!!" >> STRICT\_BOUNDARIES; |
|  | tokenAND = "&&" >> STRICT\_BOUNDARIES; |
|  | tokenOR = "||" >> STRICT\_BOUNDARIES; |
|  | tokenEQUAL = "==" >> BOUNDARIES; |
|  | tokenNOTEQUAL = "!=" >> BOUNDARIES; |
|  | tokenLESSOREQUAL = "Le" >> BOUNDARIES; |
|  | tokenGREATEROREQUAL = "Ge" >> BOUNDARIES; |
|  |  |
|  |  |
|  | tokenPLUS = "+" >> BOUNDARIES; |
|  | tokenMINUS = "-" >> BOUNDARIES; |
|  | tokenMUL = "Mul" >> BOUNDARIES; |
|  | tokenDIV = "Div" >> STRICT\_BOUNDARIES; |
|  | tokenMOD = "Mod" >> STRICT\_BOUNDARIES; |
|  | tokenGROUPEXPRESSIONBEGIN = "(" >> BOUNDARIES; |
|  | tokenGROUPEXPRESSIONEND = ")" >> BOUNDARIES; |
|  | tokenLRBIND = "->" >> BOUNDARIES; |
|  | tokenElse = "Else" >> STRICT\_BOUNDARIES; |
|  | tokenIf = "If" >> STRICT\_BOUNDARIES; |
|  |  |
|  |  |
|  |  |
|  |  |
|  | tokenWHILE = "While" >> STRICT\_BOUNDARIES; |
|  | tokenCONTINUE = "Continue" >> STRICT\_BOUNDARIES; |
|  | tokenBREAK = "Break" >> STRICT\_BOUNDARIES; |
|  |  |
|  |  |
|  |  |
|  | tokenInput = "Input" >> STRICT\_BOUNDARIES; |
|  | tokenOutput = "Output" >> STRICT\_BOUNDARIES; |
|  | tokenName = "Program" >> STRICT\_BOUNDARIES; |
|  | tokenStart = "Start" >> STRICT\_BOUNDARIES; |
|  | tokenDATA = "Var" >> STRICT\_BOUNDARIES; |
|  |  |
|  | tokenFinish = "Finish" >> STRICT\_BOUNDARIES; |
|  | tokenBEGINBLOCK = "{" >> BOUNDARIES; |
|  | tokenENDBLOCK = "}" >> BOUNDARIES; |
|  | tokenLEFTSQUAREBRACKETS = "[" >> BOUNDARIES; |
|  | tokenRIGHTSQUAREBRACKETS = "]" >> BOUNDARIES; |
|  | tokenSEMICOLON = ";" >> BOUNDARIES; |
|  | STRICT\_BOUNDARIES = (BOUNDARY >> \*(BOUNDARY)) | (!(qi::alpha | qi::char\_("\_"))); |
|  | BOUNDARIES = (BOUNDARY >> \*(BOUNDARY) | NO\_BOUNDARY); |
|  | BOUNDARY = BOUNDARY\_SPACE | BOUNDARY\_TAB | BOUNDARY\_CARRIAGE\_RETURN | BOUNDARY\_LINE\_FEED | BOUNDARY\_NULL; |
|  | BOUNDARY\_SPACE = " "; |
|  | BOUNDARY\_TAB = "\t"; |
|  | BOUNDARY\_CARRIAGE\_RETURN = "\r"; |
|  | BOUNDARY\_LINE\_FEED = "\n"; |
|  | BOUNDARY\_NULL = "\0"; |
|  | NO\_BOUNDARY = ""; |
|  | tokenUNDERSCORE = "\_"; |
|  | A = "A"; |
|  | B = "B"; |
|  | C = "C"; |
|  | D = "D"; |
|  | E = "E"; |
|  | F = "F"; |
|  | G = "G"; |
|  | H = "H"; |
|  | I = "I"; |
|  | J = "J"; |
|  | K = "K"; |
|  | L = "L"; |
|  | M = "M"; |
|  | N = "N"; |
|  | O = "O"; |
|  | P = "P"; |
|  | Q = "Q"; |
|  | R = "R"; |
|  | S = "S"; |
|  | T = "T"; |
|  | U = "U"; |
|  | V = "V"; |
|  | W = "W"; |
|  | X = "X"; |
|  | Y = "Y"; |
|  | Z = "Z"; |
|  | a = "a"; |
|  | b = "b"; |
|  | c = "c"; |
|  | d = "d"; |
|  | e = "e"; |
|  | f = "f"; |
|  | g = "g"; |
|  | h = "h"; |
|  | i = "i"; |
|  | j = "j"; |
|  | k = "k"; |
|  | l = "l"; |
|  | m = "m"; |
|  | n = "n"; |
|  | o = "o"; |
|  | p = "p"; |
|  | q = "q"; |
|  | r = "r"; |
|  | s = "s"; |
|  | t = "t"; |
|  | u = "u"; |
|  | v = "v"; |
|  | w = "w"; |
|  | x = "x"; |
|  | y = "y"; |
|  | z = "z"; |