

	<p>РБНФ №2 (опис формальної граматики засобами РБНФ)</p> <p>РБНФ №1 (опис синтаксису всіма допустимими засобами РБНФ)</p>	Формальна граматика	Формальна граматика з специфікацією lookahead у правилах для LL(2)-аналізатора	<pre>/* Перевірка РБНФ №2 за допомогою коду (помістити у файл "EBNF_N2.h") */</pre>	<pre>/* Перевірка РБНФ №1 за допомогою коду (помістити у файл "EBNF_N1.h") */</pre>	<pre>/* Перевірки прототипу LL(2)-синтаксичного аналізатора (спеціальна структура) та прототипу лексичного аналізатора (регулярні вирази) за допомогою коду. Лексеми для синтаксичного аналізатором, тому синтаксичний аналізатор не аналізує їх повторно (як показано в РБНФ). (помістити у файл "LexicaByRegExAndSyntaxByLL2prototype.h") УВАГА: при копіюванні зважайте, щоб у кожному рядку після символу \x не містилось жодних інших символів. */</pre>
		G = (N, T, P, S)	G = (N, T, P, S)			
		S → program_rule	S → program_rule			
		<pre>N = { program_name, value_type, array_specify, declaration_element, array_specify_optional, other_declaration_ident, declaration, other_declaration_ident_iteration, index_action, unary_operator, unary_operation, binary_operator, binary_action, left_expression, group_expression, index_action_optional, expression, binary_action_iteration, expression_or_cond_block_with_optional_assign, assign_to_right, assign_to_right_optional, if_expression, body_for_true, false_cond_block_without_else, body_for_false, cond_block, false_cond_block_without_else_iteration, body_for_false_optional, statement, repeat_until_cycle_cond, repeat_until_cycle, statements_or_block_statements, block_statements, input_rule, argument_for_input, output_rule, statement_iteration, expression_optional, program_rule, declaration_optional, non_zero_digit, digit_iteration, digit, unsigned_value, value, sign_optional, sign, ident, letter_in_upper_case, letter_in_lower_case, sign_plus, sign_minus }</pre>	<pre>N = { program_name, value_type, array_specify, declaration_element, array_specify_optional, other_declaration_ident, declaration, other_declaration_ident_iteration, index_action, unary_operator, unary_operation, binary_operator, binary_action, left_expression, group_expression, index_action_optional, expression, binary_action_iteration, expression_or_cond_block_with_optional_assign, assign_to_right, assign_to_right_optional, if_expression, body_for_true, false_cond_block_without_else, body_for_false, cond_block, false_cond_block_without_else_iteration, body_for_false_optional, statement, repeat_until_cycle_cond, repeat_until_cycle, statements_or_block_statements, block_statements, input_rule, argument_for_input, output_rule, statement_iteration, expression_optional, program_rule, declaration_optional, non_zero_digit, digit_iteration, digit, unsigned_value, value, sign_optional, sign, ident, letter_in_upper_case, letter_in_lower_case, sign_plus, sign_minus }</pre>	<pre>#define NONTERMINALS program_name, \ value_type, \ array_specify, \ declaration_element, \ array_specify_optional, \ other_declaration_ident, \ declaration, \ other_declaration_ident_iteration, \ index_action, \ unary_operator, \ unary_operation, \ binary_operator, \ binary_action, \ left_expression, \ group_expression, \ index_action_optional, \ expression, \ binary_action_iteration, \ expression_or_cond_block_with_optional_assign, \ assign_to_right, \ assign_to_right_optional, \ if_expression, \ body_for_true, \ false_cond_block_without_else, \ body_for_false, \ cond_block, \ false_cond_block_without_else_iteration, \ body_for_false_optional, \ statement, \ repeat_until_cycle_cond, \ repeat_until_cycle, \ statements_or_block_statements, \ block_statements, \ input_rule, \ argument_for_input, \ output_rule, \ statement_iteration, \ expression_optional, \ program_rule, \ declaration_optional, \ non_zero_digit, \ digit_iteration, \ digit, \ unsigned_value, \ value, \ sign_optional, \ sign, \ ident, \ letter_in_upper_case, \ letter_in_lower_case, \ sign_plus, \ sign_minus }</pre>	<pre>#define NONTERMINALS program_name, \ value_type, \ array_specify, \ declaration_element, \ array_specify_optional, \ other_declaration_ident, \ declaration, \ other_declaration_ident_iteration, \ index_action, \ unary_operator, \ unary_operation, \ binary_operator, \ binary_action, \ left_expression, \ group_expression, \ index_action_optional, \ expression, \ binary_action_iteration, \ expression_or_cond_block_with_optional_assign, \ assign_to_right, \ assign_to_right_optional, \ if_expression, \ body_for_true, \ false_cond_block_without_else, \ body_for_false, \ cond_block, \ false_cond_block_without_else_iteration, \ body_for_false_optional, \ statement, \ repeat_until_cycle_cond, \ repeat_until_cycle, \ statements_or_block_statements, \ block_statements, \ input_rule, \ argument_for_input, \ output_rule, \ statement_iteration, \ expression_optional, \ program_rule, \ declaration_optional, \ non_zero_digit, \ digit_iteration, \ digit, \ unsigned_value, \ value, \ sign_optional, \ sign, \ ident, \ letter_in_upper_case, \ letter_in_lower_case, \ sign_plus, \ sign_minus }</pre>	
		<pre>T = { "INT_4", " ", "NOT", "&", "OR", "EQ", "NE", "LT", "GT", "ADD", "SUB", "MUL", "DIV", "MOD", "(", ")", ">", "ELSE", "IF", "EXIT"}</pre>	<pre>T = { "INT_4", " ", "NOT", "&", "OR", "EQ", "NE", "LT", "GT", "ADD", "SUB", "MUL", "DIV", "MOD", "(", ")", ">", "ELSE", "IF", "EXIT"}</pre>	<pre>#define TOKENS \ tokenINTEGER16, \ tokenCOMMA, \ tokenNOT, \ tokenAND, \ tokenOR, \ tokenEQUAL, \ tokenNOTEQUAL, \ tokenLESS, \ tokenGREATER, \ tokenPLUS, \ tokenMINUS, \ tokenMUL, \ tokenDIV, \ tokenMOD, \ tokenGROUPEXPRESSIONBEGIN, \ tokenGROUPEXPRESSIONEND, \ tokenLRASSIGN, \ tokenELSE, \ tokenIF, \ tokenEXIT, \ </pre>	<pre>#define TOKENS \ tokenINTEGER16, \ tokenCOMMA, \ tokenNOT, \ tokenAND, \ tokenOR, \ tokenEQUAL, \ tokenNOTEQUAL, \ tokenLESS, \ tokenGREATER, \ tokenPLUS, \ tokenMINUS, \ tokenMUL, \ tokenDIV, \ tokenMOD, \ tokenGROUPEXPRESSIONBEGIN, \ tokenGROUPEXPRESSIONEND, \ tokenLRASSIGN, \ tokenELSE, \ tokenIF, \ tokenEXIT, \ </pre>	

		"REPEAT", "UNTIL", "READ", "WRITE", "STARTPROGRAM", "STARTBLOCK", "DATA", "BEGIN", "ENDBLOCK", "{" , "}" , "[" , "]" , ";" , "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", " ", "−", "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")	"REPEAT", "UNTIL", "READ", "WRITE", "STARTPROGRAM", "STARTBLOCK", "DATA", "BEGIN", "ENDBLOCK", "{" , "}" , "[" , "]" , ";" , "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", " ", "−", "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")	tokenREPEAT, \ tokenUNTIL, \ tokenGET, \ tokenPUT, \ tokenNAME, \ tokenBODY, \ tokenDATA, \ tokenBEGIN, \ tokenEND, \ tokenBEGINBLOCK, \ tokenENDBLOCK, \ tokenLEFTSQUAREBRACKETS, \ tokenRIGHTSQUAREBRACKETS, \ tokenSEMICOLON, \ digit_0, \ digit_1, \ digit_2, \ digit_3, \ digit_4, \ digit_5, \ digit_6, \ digit_7, \ digit_8, \ digit_9, \ tokenUNDERSCORE, \ 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	tokenREPEAT, \ tokenUNTIL, \ tokenGET, \ tokenPUT, \ tokenNAME, \ tokenBODY, \ tokenDATA, \ tokenBEGIN, \ tokenEND, \ tokenBEGINBLOCK, \ tokenENDBLOCK, \ tokenLEFTSQUAREBRACKETS, \ tokenRIGHTSQUAREBRACKETS, \ tokenSEMICOLON, \ digit_0, \ digit_1, \ digit_2, \ digit_3, \ digit_4, \ digit_5, \ digit_6, \ digit_7, \ digit_8, \ digit_9, \ tokenUNDERSCORE, \ 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
			#define COMMENT_BEGIN_STR "##" #define COMMENT_END_STR "##"	#define COMMENT_BEGIN_STR "##" #define COMMENT_END_STR "##"	#define COMMENT_BEGIN_STR "##" #define COMMENT_END_STR "##"
					#define TOKENS_RE "> & _0-9a-zA-Z [^\\n\\r\\v\\n]"
					#define IDENTIFIERS_RE "[a-zA-Z][a-zA-Z0-9][0-9]"
					#define UNSIGNEDVALUES_RE "0 [1-9]0-9**"
			tokenGROUPEXPRESSIONBEGIN = "(" >> BOUNDARIES;	tokenGROUPEXPRESSIONBEGIN = "(" >> BOUNDARIES;	#define T_BEGIN_GROUPEXPRESSION_0 "(" #define T_BEGIN_GROUPEXPRESSION_1 "" #define T_BEGIN_GROUPEXPRESSION_2 "" #define T_BEGIN_GROUPEXPRESSION_3 ""
			tokenGROUPEXPRESSIONEND = ")" >> BOUNDARIES;	tokenGROUPEXPRESSIONEND = ")" >> BOUNDARIES;	#define T_END_GROUPEXPRESSION_0 ")" #define T_END_GROUPEXPRESSION_1 "" #define T_END_GROUPEXPRESSION_2 "" #define T_END_GROUPEXPRESSION_3 ""
			tokenLEFTSQUAREBRACKETS = "[" >> BOUNDARIES;	tokenLEFTSQUAREBRACKETS = "[" >> BOUNDARIES;	#define T_LEFT_SQUAREBRACKETS_0 "[" #define T_LEFT_SQUAREBRACKETS_1 "[" #define T_LEFT_SQUAREBRACKETS_2 "[" #define T_LEFT_SQUAREBRACKETS_3 "["
			tokenRIGHTSQUAREBRACKETS = "]" >> BOUNDARIES;	tokenRIGHTSQUAREBRACKETS = "]" >> BOUNDARIES;	#define T_RIGHT_SQUAREBRACKETS_0 "]" #define T_RIGHT_SQUAREBRACKETS_1 "]" #define T_RIGHT_SQUAREBRACKETS_2 "]" #define T_RIGHT_SQUAREBRACKETS_3 "]"
			tokenBEGINBLOCK = "{" >> BOUNDARIES;	tokenBEGINBLOCK = "{" >> BOUNDARIES;	#define T_BEGIN_BLOCK_0 "{" #define T_BEGIN_BLOCK_1 "{" #define T_BEGIN_BLOCK_2 "[" #define T_BEGIN_BLOCK_3 "["
			tokenENDBLOCK = "}" >> BOUNDARIES;	tokenENDBLOCK = "}" >> BOUNDARIES;	#define T_END_BLOCK_0 "}" #define T_END_BLOCK_1 "}" #define T_END_BLOCK_2 "}"

						#define T_END_BLOCK_3 ""
				tokenSEMICOLON = ";" >> BOUNDARIES;	tokenSEMICOLON = ";" >> BOUNDARIES;	#define T_SEMICOLON_0 ":"; #define T_SEMICOLON_1 "" #define T_SEMICOLON_2 "" #define T_SEMICOLON_3 ""
				tokenINTEGER16 = "INT_4" >> STRICT_BOUNDARIES;	tokenINTEGER16 = "INT_4" >> STRICT_BOUNDARIES;	#define T_DATA_TYPE_0 "INT_4" #define T_DATA_TYPE_1 "" #define T_DATA_TYPE_2 "" #define T_DATA_TYPE_3 ""
				tokenCOMMA = "," >> BOUNDARIES;	tokenCOMMA = "," >> BOUNDARIES;	#define T_COMA_0 "," #define T_COMA_1 "" #define T_COMA_2 "" #define T_COMA_3 ""
						#define T_BITWISE_NOT_0 "~" #define T_BITWISE_NOT_1 "" #define T_BITWISE_NOT_2 "" #define T_BITWISE_NOT_3 ""
				tokenNOT = "NOT" >> STRICT_BOUNDARIES;	tokenNOT = "NOT" >> STRICT_BOUNDARIES;	#define T_NOT_0 "NOT" #define T_NOT_1 "" #define T_NOT_2 "" #define T_NOT_3 ""
						#define T_BITWISE_AND_0 "&" #define T_BITWISE_AND_1 "" #define T_BITWISE_AND_2 "" #define T_BITWISE_AND_3 ""
				tokenAND = "&" >> STRICT_BOUNDARIES;	tokenAND = "&" >> STRICT_BOUNDARIES;	#define T_AND_0 "AND" #define T_AND_1 "" #define T_AND_2 "" #define T_AND_3 ""
						#define T_BITWISE_OR_0 " " #define T_BITWISE_OR_1 "" #define T_BITWISE_OR_2 "" #define T_BITWISE_OR_3 ""
				tokenOR = "OR" >> STRICT_BOUNDARIES;	tokenOR = "OR" >> STRICT_BOUNDARIES;	#define T_OR_0 "OR" #define T_OR_1 "" #define T_OR_2 "" #define T_OR_3 ""
				tokenEQUAL = "EQ" >> BOUNDARIES;	tokenEQUAL = "EQ" >> BOUNDARIES;	#define T_EQUAL_0 "EQ" #define T_EQUAL_1 "" #define T_EQUAL_2 "" #define T_EQUAL_3 ""
				tokenNOTEQUAL = "NE" >> BOUNDARIES;	tokenNOTEQUAL = "NE" >> BOUNDARIES;	#define T_NOTEQUAL_0 "NE" #define T_NOTEQUAL_1 "" #define T_NOTEQUAL_2 "" #define T_NOTEQUAL_3 ""
				tokenLESS = "LT" >> BOUNDARIES;	tokenLESS = "LT" >> BOUNDARIES;	#define T_LESS_0 "LT" #define T_LESS_1 "" #define T_LESS_2 "" #define T_LESS_3 ""
				tokenGREATER = "GT" >> BOUNDARIES;	tokenGREATER = "GT" >> BOUNDARIES;	#define T_GREATER_0 "GT" #define T_GREATER_1 "" #define T_GREATER_2 "" #define T_GREATER_3 ""
				tokenPLUS = "ADD" >> BOUNDARIES;	tokenPLUS = "ADD" >> BOUNDARIES;	#define T_ADD_0 "ADD" #define T_ADD_1 "" #define T_ADD_2 "" #define T_ADD_3 ""
				tokenMINUS = "SUB" >> BOUNDARIES;	tokenMINUS = "SUB" >> BOUNDARIES;	#define T_SUB_0 "SUB" #define T_SUB_1 "" #define T_SUB_2 "" #define T_SUB_3 ""
				tokenMUL = "MUL" >> BOUNDARIES;	tokenMUL = "MUL" >> BOUNDARIES;	#define T_MUL_0 "MUL" #define T_MUL_1 "" #define T_MUL_2 "" #define T_MUL_3 ""
				tokenDIV = "DIV" >> STRICT_BOUNDARIES;	tokenDIV = "DIV" >> STRICT_BOUNDARIES;	#define T_DIV_0 "DIV" #define T_DIV_1 "" #define T_DIV_2 "" #define T_DIV_3 ""
				tokenMOD = "MOD" >> STRICT_BOUNDARIES;	tokenMOD = "MOD" >> STRICT_BOUNDARIES;	#define T_MOD_0 "MOD" #define T_MOD_1 "" #define T_MOD_2 "" #define T_MOD_3 ""
				tokenLRASSIGN = ">" >> BOUNDARIES;	tokenLRASSIGN = ">" >> BOUNDARIES;	#define T_LRASSIGN_0 ">" #define T_LRASSIGN_1 "" #define T_LRASSIGN_2 "" #define T_LRASSIGN_3 ""
						#define T_THEN_BLOCK_0 "(" #define T_THEN_BLOCK_1 "" #define T_THEN_BLOCK_2 "" #define T_THEN_BLOCK_3 ""
				tokenELSE = "ELSE" >> STRICT_BOUNDARIES;	tokenELSE = "ELSE" >> STRICT_BOUNDARIES;	#define T_ELSE_BLOCK_0 "ELSE" #define T_ELSE_BLOCK_1 T_BEGIN_BLOCK_0 #define T_ELSE_BLOCK_2 "" #define T_ELSE_BLOCK_3 ""
				tokenIF = "IF" >> STRICT_BOUNDARIES;	tokenIF = "IF" >> STRICT_BOUNDARIES;	#define T_IF_0 "IF" #define T_IF_1 "" #define T_IF_2 "" #define T_IF_3 ""
						#define T_ELSE_IF_0 T_ELSE_BLOCK_0 #define T_ELSE_IF_1 T_IF_0 #define T_ELSE_IF_2 "" #define T_ELSE_IF_3 ""
				tokenEXIT = "EXIT" >> STRICT_BOUNDARIES;	tokenEXIT = "EXIT" >> STRICT_BOUNDARIES;	#define T_EXIT_0 "EXIT" #define T_EXIT_1 "" #define T_EXIT_2 "" #define T_EXIT_3 ""
				tokenREPEAT = "REPEAT" >> STRICT_BOUNDARIES;	tokenREPEAT = "REPEAT" >> STRICT_BOUNDARIES;	#define T_REPEAT_0 "REPEAT" #define T_REPEAT_1 "" #define T_REPEAT_2 "" #define T_REPEAT_3 ""
				tokenUNTIL = "UNTIL" >> STRICT_BOUNDARIES;	tokenUNTIL = "UNTIL" >> STRICT_BOUNDARIES;	#define T_UNTIL_0 "UNTIL" #define T_UNTIL_1 "" #define T_UNTIL_2 "" #define T_UNTIL_3 ""
				tokenGET = "READ" >> STRICT_BOUNDARIES;	tokenGET = "READ" >> STRICT_BOUNDARIES;	#define T_INPUT_0 "READ"

						#define T_INPUT_1 "" #define T_INPUT_2 "" #define T_INPUT_3 ""
				tokenPUT = "WRITE" >> STRICT_BOUNDARIES;	tokenPUT = "WRITE" >> STRICT_BOUNDARIES;	#define T_OUTPUT_0 "WRITE" #define T_OUTPUT_1 "" #define T_OUTPUT_2 "" #define T_OUTPUT_3 ""
				tokenNAME = "STARTPROGRAM" >> STRICT_BOUNDARIES;	tokenNAME = "STARTPROGRAM" >> STRICT_BOUNDARIES;	#define T_NAME_0 "STARTPROGRAM" #define T_NAME_1 "" #define T_NAME_2 "" #define T_NAME_3 ""
				tokenBODY = "STARTBLOCK" >> STRICT_BOUNDARIES;	tokenBODY = "STARTBLOCK" >> STRICT_BOUNDARIES;	#define T_BODY_0 "STARTBLOCK" #define T_BODY_1 "" #define T_BODY_2 "" #define T_BODY_3 ""
				tokenDATA = "DATA" >> STRICT_BOUNDARIES;	tokenDATA = "DATA" >> STRICT_BOUNDARIES;	#define T_DATA_0 "DATA" #define T_DATA_1 "" #define T_DATA_2 "" #define T_DATA_3 ""
				tokenBEGIN = "BEGIN" >> STRICT_BOUNDARIES;	tokenBEGIN = "BEGIN" >> STRICT_BOUNDARIES;	#define T_BEGIN_0 "BEGIN" #define T_BEGIN_1 "" #define T_BEGIN_2 "" #define T_BEGIN_3 ""
				tokenEND = "ENDBLOCK" >> STRICT_BOUNDARIES;	tokenEND = "ENDBLOCK" >> STRICT_BOUNDARIES;	#define T_END_0 "ENDBLOCK" #define T_END_1 "" #define T_END_2 "" #define T_END_3 ""
						#define T_NULL_STATEMENT_0 "NULL" #define T_NULL_STATEMENT_1 "STATEMENT" #define T_NULL_STATEMENT_2 "" #define T_NULL_STATEMENT_3 ""
						#define GRAMMAR_LL2_2025 (\
program_name = ident;	program_name = ident;	program_name → ident	program_name(1: "ident_terminal") → ident	program_name = SAME_RULE(ident);	program_name = SAME_RULE(ident);	{ LA_IS, ("ident_terminal"), { "program_name", \(\{ LA_IS, (""), 1, {"ident"}\}\) }\}
value_type = "INT_4";	value_type = "INT_4";	value_type → "INT_4"	value_type(1: "INT_4") → "INT_4"	value_type = SAME_RULE(tokenINTEGER16);	value_type = SAME_RULE(tokenINTEGER16);	{ LA_IS, {T_DATA_TYPE_0}, { "value_type", \(\{ LA_IS, (""), 1, {T_DATA_TYPE_0}\}\) }\}
array_specify = "[" , unsigned_value , "]";	array_specify = "[" , unsigned_value , "]";	array_specify → "[" unsigned_value "]"	array_specify(1: "[") → "[" unsigned_value "]"	array_specify = "[" >> unsigned_value >> "]";	array_specify = "[" >> unsigned_value >> "]";	{ LA_IS, ("[") , { "array_specify", \(\{ LA_IS, (""), 3, ("[", "unsigned_value", "]")\}\) }\}
declaration_element = ident, ["[", unsigned_value , "]"];	declaration_element = ident, ["[", unsigned_value , "]"];	declaration_element → ident array_specify_optional;	declaration_element(1: "ident_terminal") → ident array_specify_optional	declaration_element = ident >> -(tokenLEFTSQUAREBRACKETS >> unsigned_value >> tokenRIGHTSQUAREBRACKETS);	declaration_element = ident >> array_specify_optional;	{ LA_IS, ("ident_terminal"), { "declaration_element", \(\{ LA_IS, (""), 2, {"ident", "array_specify_optional"}\)\) }\}
array_specify_optional = array_specify ε;	array_specify_optional = array_specify ε;	array_specify_optional → array_specify array_specify_optional → ε	array_specify_optional(1: "[") → array_specify array_specify_optional(1: ")") → ε	array_specify_optional = array_specify "";	array_specify_optional = array_specify "";	{ LA_IS, ("array_specify_optional"), { \(\{ LA_IS, (""), 1, {"array_specify"}\)\) }\}\}
other_declaration_ident = "", declaration_element;	other_declaration_ident = "", declaration_element;	other_declaration_ident → "", declaration_element	other_declaration_ident(1: ":") → "", declaration_element	other_declaration_ident = tokenCOMMA >> declaration_element;	other_declaration_ident = tokenCOMMA >> declaration_element;	{ LA_IS, {T_COMA_0}, { "other_declaration_ident", \(\{ LA_IS, (""), 2, {T_COMA_0, "declaration_element"}\)\) }\}
declaration = value_type, declaration_element, {other_declaration_ident};	declaration = value_type, declaration_element, {other_declaration_ident};	declaration → value_type declaration_element other_declaration_ident_iteration;	declaration(1: "INT_4") → value_type declaration_element other_declaration_ident_iteration	declaration = value_type >> declaration_element >> *other_declaration_ident;	declaration = value_type >> declaration_element >> other_declaration_ident_iteration;	{ LA_IS, {T_DATA_TYPE_0}, { "declaration", \(\{ LA_IS, (""), 3, {"value_type", "declaration_element", "other_declaration_ident_iteration"}\)\) }\}
other_declaration_ident_iteration = other_declaration_ident, other_declaration_ident_iteration ε;	other_declaration_ident_iteration = other_declaration_ident, other_declaration_ident_iteration ε;	other_declaration_ident_iteration → other_declaration_ident other_declaration_ident_iteration false_cond_block_without_else_iteration → ε	other_declaration_ident_iteration(1: ":") → other_declaration_ident other_declaration_ident_iteration false_cond_block_without_else_iteration(1: ")") → ε		other_declaration_ident_iteration = other_declaration_ident >> other_declaration_ident_iteration "";	{ LA_IS, {T_COMA_0}, { "other_declaration_ident_iteration", \(\{ LA_IS, (""), 2, {"other_declaration_ident", "other_declaration_ident_iteration"}\)\) }\}\}
index_action = "[" , expression , "]";	index_action = "[" , expression , "]";	index_action → "[" expression "]"	index_action(1: "[") → "[" expression "]"	index_action = tokenLEFTSQUAREBRACKETS >> expression >> tokenRIGHTSQUAREBRACKETS;	index_action = tokenLEFTSQUAREBRACKETS >> expression >> tokenRIGHTSQUAREBRACKETS;	{ LA_IS, ("[") , { "index_action", \(\{ LA_IS, (""), 3, ("[", "expression", "]")\)\) }\}\}
unary_operator = "NOT";	unary_operator = "NOT";	unary_operator → "NOT"	unary_operator(1: "NOT") → "NOT"	unary_operator = SAME_RULE(tokenNOT);	unary_operator = SAME_RULE(tokenNOT);	{ LA_IS, {T_NOT_0}, { "unary_operator", \(\{ LA_IS, (""), 1, {T_NOT_0}\}\) }\}
unary_operation = unary_operator, expression;	unary_operation = unary_operator, expression;	unary_operation → unary_operator expression	unary_operation(1: "NOT") → unary_operator expression	unary_operation = unary_operator >> expression;	unary_operation = unary_operator >> expression;	{ LA_IS, {T_NOT_0}, { "unary_operation", \(\{ LA_IS, (""), 2, {"unary_operator", "expression"}\)\) }\}
binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator → "&" binary_operator → "OR" binary_operator → "EQ" binary_operator → "NE" binary_operator → "LT" binary_operator → "GT" binary_operator → "ADD" binary_operator → "SUB" binary_operator → "MUL" binary_operator → "DIV" binary_operator → "MOD"	binary_operator(1: "AND") → "&" binary_operator(1: "OR") → "OR" binary_operator(1: "=") → "EQ" binary_operator(1: "!=") → "NE" binary_operator(1: "<") → "LT" binary_operator(1: ">") → "GT" binary_operator(1: "+") → "ADD" binary_operator(1: "-") → "SUB" binary_operator(1: "*") → "MUL" binary_operator(1: "/") → "DIV" binary_operator(1: "%") → "MOD"		binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	{ LA_IS, {T_BITWISE_AND_0}, { "binary_operator", \(\{ LA_IS, (""), 1, {T_BITWISE_AND_0}\}\) }\}\}
binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";			binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	{ LA_IS, {T_OR_0}, { "binary_operator", \(\{ LA_IS, (""), 1, {T_OR_0}\}\) }\}\}
binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";			binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	{ LA_IS, {T_EQUAL_0}, { "binary_operator", \(\{ LA_IS, (""), 1, {T_EQUAL_0}\}\) }\}\}
binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";			binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	{ LA_IS, {T_NOTEQUAL_0}, { "binary_operator", \(\{ LA_IS, (""), 1, {T_NOTEQUAL_0}\}\) }\}\}
binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";			binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	{ LA_IS, {T_LESS_0}, { "binary_operator", \(\{ LA_IS, (""), 1, {T_LESS_0}\}\) }\}\}
binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";			binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	{ LA_IS, {T_GREATER_0}, { "binary_operator", \(\{ LA_IS, (""), 1, {T_GREATER_0}\}\) }\}\}
binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";			binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	{ LA_IS, {T_ADD_0}, { "binary_operator", \(\{ LA_IS, (""), 1, {T_ADD_0}\}\) }\}\}
binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";			binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	{ LA_IS, {T_SUB_0}, { "binary_operator", \(\{ LA_IS, (""), 1, {T_SUB_0}\}\) }\}\}
binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";			binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	{ LA_IS, {T_MUL_0}, { "binary_operator", \(\{ LA_IS, (""), 1, {T_MUL_0}\}\) }\}\}
binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";			binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	{ LA_IS, {T_DIV_0}, { "binary_operator", \(\{ LA_IS, (""), 1, {T_DIV_0}\}\) }\}\}
binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";	binary_operator = "&" "OR" "EQ" "NE" "LT" "GT" "ADD" "SUB" "MUL" "DIV" "MOD";			binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	binary_operator = tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD;	{ LA_IS, {T_MOD_0}, { "binary_operator", \(\{ LA_IS, (""), 1, {T_MOD_0}\}\) }\}\}

						}}\}
binary_action = binary_operator , expression;	binary_action = binary_operator , expression;	binary_action → binary_operator expression	binary_action(1: "&", "OR", "EQ", "NE", "LT", "GT", "ADD", "SUB", "MUL", "DIV", "MOD") → binary_operator expression		binary_action = binary_operator >> expression;	{ LA_IS, { T_BITWISE_AND_0, T_OR_0, T_EQUAL_0, T_NOT_EQUAL_0, T_LESS_0, T_GREATER_0, T_ADD_0, T_SUB_0, T_MUL_0, T_DIV_0, T_MOD_0 }, { "binary_action", { { LA_IS, { "", 2, { "binary_operator", "expression" } } } } } }
binary_action = binary_operator , expression;	left_expression = group_expression unary_operation cond_block value ident , index_action__optional;	left_expression → group_expression left_expression → unary_operation left_expression → cond_block left_expression → value left_expression → ident , index_action__optional	left_expression(1: "(") → group_expression left_expression(1: "NOT") → unary_operation left_expression(1: "IF") → cond_block left_expression(1: "0", "1", "2", "3", "4", "5", "6", "7", "8", "9") → value left_expression(1: "ADD", "SUB", 2: "0", "1", "2", "3", "4", "5", "6", "7", "8", "9") → value left_expression(1: "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 , index_action__optional		left_expression = group_expression unary_operation cond_block value ident >> -index_action;	{ LA_IS, { "", 1, { "group_expression" } } } { LA_IS, { T_NOT_0 }, { "left_expression", { { LA_IS, { "", 1, { "unary_operation" } } } } } } { LA_IS, { T_IF_0 }, { "left_expression", { { LA_IS, { "", 1, { "cond_block" } } } } } } { LA_IS, { "unsigned_value_terminal" }, { "left_expression", { { LA_IS, { "", 1, { "left_expression" } } } } } } { LA_IS, { "left_expression", { { LA_IS, { "", 1, { "value" } } } } } } { LA_IS, { T_ADD_0, T_SUB_0 }, { "left_expression", { { LA_IS, { "unsigned_value_terminal" }, 1, { "value" } } } } } /{ LA_NOT, { "unsigned_value_terminal" }, 1, { "unary_operation" } } * /{ LA_IS, { "ident_terminal" }, { "left_expression", { { LA_IS, { "", 2, { "ident", "index_action_optional" } } } } } } }
left_expression = group_expression unary_operation cond_block value ident , [index_action];	index_action__optional = index_action ε;	index_action__optional → index_action index_action__optional → ε	index_action__optional(1: "[") → index_action index_action__optional(1: ")" → ε		index_action__optional = index_action "";	{ LA_IS, { "", 1, { "index_action" } } } { LA_NOT, { "" }, { "index_action_optional", { { LA_IS, { "", 0, { "" } } } } } }
expression = left_expression , {binary_action};	expression = left_expression , binary_action__iteration;	expression → left_expression binary_action__iteration	expression(1: "(", "NOT", "ADD", "SUB", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → left_expression binary_action__iteration		expression = left_expression >> *binary_action;	expression = left_expression >> binary_action__iteration;
group_expression = (" , expression , ")";	group_expression = "(" , expression , ")";	group_expression → "(" expression ")"	group_expression(1: "(") → "(" expression ")"	group_expression = tokenGROUPEXPRESSIONBEGIN >> expression >> tokenGROUPEXPRESSIONEND;	group_expression = tokenGROUPEXPRESSIONBEGIN >> expression >> tokenGROUPEXPRESSIONEND;	{ LA_IS, { "", 3, { "(", "expression", ")" } } }
expression_or_cond_block__with_optional_assign = expression , [">"] , ident , [index_action];	expression_or_cond_block__with_optional_assign = expression , assign_to_right__optional;	expression_or_cond_block__with_optional_assign → expression assign_to_right__optional	expression_or_cond_block__with_optional_assign(1: "(", "NOT", "ADD", "SUB", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → expression assign_to_right__optional		expression_or_cond_block__with_optional_assign = expression >> (tokenLRASSIGN >> ident >> -index_action);	expression_or_cond_block__with_optional_assign = expression >> assign_to_right__optional;
assign_to_right = ">" , ident , index_action__optional;	assign_to_right → ">" ident index_action__optional	assign_to_right(1: ">") → ">" ident index_action__optional	assign_to_right(1: ">") → ">" ident index_action__optional		assign_to_right = tokenLRASSIGN >> ident >> index_action__optional;	{ LA_IS, { T_LRASSIGN_0 }, { "assign_to_right", { { LA_IS, { "", 3, { "T_LRASSIGN_0", "ident", "index_action_optional" } } } } } }
assign_to_right__optional = assign_to_right ε;	assign_to_right__optional → assign_to_right assign_to_right__optional → ε;	assign_to_right__optional(1: ">") → assign_to_right assign_to_right__optional(1: "!" → ε)	assign_to_right__optional(1: ">") → assign_to_right assign_to_right__optional(1: "!" → ε)		assign_to_right__optional = assign_to_right "";	{ LA_IS, { T_LRASSIGN_0 }, { "assign_to_right_optional", { { LA_IS, { "", 1, { "assign_to_right" } } } } } } { LA_NOT, { T_LRASSIGN_0 }, { "assign_to_right_optional", { { LA_IS, { "", 0, { "" } } } } } }
if_expression = expression;	if_expression = expression;	if_expression → expression	if_expression(1: "(", "NOT", "ADD", "SUB", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → expression		if_expression = SAME_RULE(expression);	{ LA_IS, { T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal", T_IF_0 }, { "if_expression", { { LA_IS, { "", 1, { "expression" } } } } } }
body_for_true = block_statements;	body_for_true = block_statements;	body_for_true → block_statements	body_for_true(1: "(") → block_statements	body_for_true = SAME_RULE(block_statements);	body_for_true = SAME_RULE(block_statements);	{ LA_IS, { T_BEGIN_BLOCK_0 }, { "body_for_true", { { LA_IS, { "", 1, { "block_statements" } } } } } }
false_cond_block_without_else = "ELSE" , "IF" , if_expression , body_for_true;	false_cond_block_without_else = "ELSE" , "IF" , if_expression , body_for_true;	false_cond_block_without_else → "ELSE" "IF" if_expression body_for_true	false_cond_block_without_else(1: "ELSE") → "ELSE" "IF" if_expression body_for_true	false_cond_block_without_else = tokenELSE >> tokenIF >> if_expression >> body_for_true;	false_cond_block_without_else = tokenELSE >> tokenIF >> if_expression >> body_for_true;	{ LA_IS, { T_ELSE_IF_0 }, { "false_cond_block_without_else", { { LA_IS, { "", 4, { T_ELSE_IF_0, T_ELSE_IF_1, "if_expression", "body_for_true" } } } } } }
body_for_false = "ELSE" , block_statements;	body_for_false = "ELSE" , block_statements;	body_for_false → "ELSE" block_statements	body_for_false(1: "ELSE") → "ELSE" block_statements	body_for_false = tokenELSE >> block_statements;	body_for_false = tokenELSE >> block_statements;	{ LA_IS, { T_ELSE_BLOCK_0 }, { "body_for_false", { { LA_IS, { "", 2, { T_ELSE_BLOCK_0, "block_statements" } } } } } }
cond_block = "IF" , if_expression , body_for_true , false_cond_block_without_else_iteration , body_for_false__optional;	cond_block = "IF" , if_expression , body_for_true , false_cond_block_without_else_iteration , body_for_false__optional;	cond_block → "IF" if_expression body_for_true false_cond_block_without_else_iteration body_for_false__optional	cond_block(1: "IF") → "IF" if_expression body_for_true false_cond_block_without_else_iteration body_for_false__optional	cond_block = tokenIF >> if_expression >> body_for_true >> *false_cond_block_without_else->-body_for_false;	cond_block = tokenIF >> if_expression >> body_for_true >> false_cond_block_without_else_iteration >> body_for_false__optional;	{ LA_IS, { T_IF_0 }, { "cond_block", { { LA_IS, { "", 5, { T_IF_0, "if_expression", "body_for_true", "false_cond_block_without_else_iteration", "body_for_false_optional" } } } } } }
false_cond_block_without_else_iteration = false_cond_block_without_else , false_cond_block_without_else_iteration ε;	false_cond_block_without_else_iteration → false_cond_block_without_else false_cond_block_without_else_iteration false_cond_block_without_else_iteration → ε	false_cond_block_without_else_iteration(1: "ELSE", 2: "IF") → false_cond_block_without_else false_cond_block_without_else_iteration false_cond_block_without_else_iteration(1: "ELSE", 2: "!" → ε)	false_cond_block_without_else_iteration(1: "ELSE", 2: "IF") → false_cond_block_without_else false_cond_block_without_else_iteration false_cond_block_without_else_iteration(1: "ELSE", 2: "!" → ε)	false_cond_block_without_else_iteration = false_cond_block_without_else >> false_cond_block_without_else_iteration "";	false_cond_block_without_else_iteration = false_cond_block_without_else >> false_cond_block_without_else_iteration "";	{ LA_IS, { T_ELSE_IF_0 }, { "false_cond_block_without_else_iteration", { { LA_IS, { "", 2, { T_ELSE_IF_1, "false_cond_block_without_else", "false_cond_block_without_else_iteration" } } } } } } { LA_NOT, { T_ELSE_IF_0 }, { "false_cond_block_without_else_iteration", { { LA_IS, { "", 0, { "" } } } } } }

						});\}
	body_for_false_optional = body_for_false ε;	body_for_false_optional → body_for_false body_for_false_optional → ε	body_for_false_optional(1: "FALSE") → body_for_false body_for_false_optional(1: !"FALSE") → ε		body_for_false_optional = body_for_false "";	{LA_IS, {T_ELSE_BLOCK_0}, {\ "body_for_false_optional", \\\ (LA_IS, {""}), 1, {"body_for_false"} }\} }{LA_NOT, {T_ELSE_BLOCK_0}, {\ "body_for_false_optional", \\\ (LA_IS, {""}), 0, {""} }\} });\}
repeat_until_cycle_cond = expression;	repeat_until_cycle_cond = expression;	repeat_until_cycle_cond → expression	repeat_until_cycle_cond(1: "(" "NOT" "ADD" "SUB", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → expression		repeat_until_cycle_cond = SAME_RULE(expression);	{LA_IS, {"", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal", T_IF_0}, {\ "repeat_until_cycle_cond", \\\ (LA_IS, {""}), 1, {"expression"} }\} });\}
repeat_until_cycle = "REPEAT", ((statement block_statements), "UNTIL", repeat_until_cycle_cond);	repeat_until_cycle = "REPEAT", statements_or_block_statements, "UNTIL", repeat_until_cycle_cond;	repeat_until_cycle → "REPEAT" statements_or_block_statements "UNTIL" repeat_until_cycle_cond	repeat_until_cycle(1: "REPEAT") → "REPEAT" statements_or_block_statements "UNTIL" repeat_until_cycle_cond	repeat_until_cycle = tokenREPEAT >> (*statement block_statements) >> tokenUNTIL >> repeat_until_cycle_cond;	repeat_until_cycle = tokenREPEAT >> statements_or_block_statements >> tokenUNTIL >> repeat_until_cycle_cond;	{LA_IS, {T_REPEAT_0}, {"repeat_until_cycle"}, \\\ (LA_IS, {""}), 4, {T_REPEAT_0, "statements_or_block_statements", T_UNTIL_0, "repeat_until_cycle_cond"} }\} });\}
input_rule = "READ", argument_for_input;	input_rule → "READ" argument_for_input	input_rule(1: "READ") → "READ" argument_for_input	input_rule(1: "READ") → "READ" argument_for_input	input_rule = tokenGET >> (ident >> -index_action tokenGROUPEXPRESSIONBEGIN >> ident >> -index_action >> tokenGROUPEXPRESSIONEND);	input_rule = tokenGET >> argument_for_input;	{LA_IS, {T_INPUT_0}, {"input_rule"}, \\\ (LA_IS, {""}), 2, {T_INPUT_0, "argument_for_input"} }\} });\}
input_rule = "READ", (ident, [index_action] ("", ident, [index_action]), "");	argument_for_input = ident, index_action_optional; argument_for_input = ("", ident, "index_action_optional", "");	argument_for_input → ident index_action_optional argument_for_input → ("", ident "index_action_optional", "")	argument_for_input(1: "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 index_action_optional argument_for_input(1: "") → ("", ident "index_action_optional", "")		argument_for_input = ident >> index_action_optional index_action_optional >> tokenGROUPEXPRESSIONEND;	{LA_IS, {"ident_terminal"}, {"argument_for_input"}, \\\ (LA_IS, {""}), 2, {"ident", "index_action_optional"} }\} {LA_IS, {"", T_BEGIN_BLOCK_0}, {"statements_or_block_statements"}, \\\ (LA_IS, {""}), 1, {"block_statements"} }\} });\}
output_rule = "WRITE", expression;	output_rule → "WRITE" expression	output(1: "WRITE") → "WRITE" expression	output(1: "WRITE") → "WRITE" expression		output_rule = tokenPUT >> expression;	{LA_IS, {T_OUTPUT_0}, {"output_rule"}, \\\ (LA_IS, {""}), 2, {T_OUTPUT_0, "expression"} }\} });\}
statement = expression_or_cond_block_with_optional_assign repeat_until_cycle input_rule output_rule ";"	statement → expression_or_cond_block_with_optional_assign statement → repeat_until_cycle statement → input_rule statement → output_rule statement → ";"	statement(1: "(", "NOT", "ADD", "SUB", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → expression_or_cond_block_with_optional_assign statement(1: "REPEAT") → repeat_until_cycle statement(1: "READ") → input_rule statement(1: "WRITE") → output_rule statement(1: "") → ;		statement = expression_or_cond_block_with_optional_assign repeat_until_cycle input_rule output_rule tokenSEMICOLON;	statement = expression_or_cond_block_with_optional_assign repeat_until_cycle input_rule output_rule tokenSEMICOLON;	{LA_IS, {"", T_NOT_0, "ident_terminal", "unsigned_value_terminal", T_ADD_0, T_SUB_0, T_IF_0}, {\ "statement", \\\ (LA_IS, {""}), 1, {"expression_or_cond_block_with_optional_assign"} }\} {LA_IS, {T_REPEAT_0}, {"statement"}, \\\ (LA_IS, {""}), 1, {"repeat_until_cycle"} }\} {LA_IS, {T_INPUT_0}, {"statement"}, \\\ (LA_IS, {""}), 1, {"input_rule"} }\} {LA_IS, {T_OUTPUT_0}, {"statement"}, \\\ (LA_IS, {""}), 1, {"output_rule"} }\} {LA_IS, {T_SEMICOLON_0}, {"statement"}, \\\ (LA_IS, {""}), 1, {""} }\} });\}
block_statements = ("", statement_iteration, ");"	statement_iteration = statement, statement_iteration ε;	statement_iteration → statement statement_iteration statement_iteration → ε	statement_iteration(1: "(", "NOT", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "ADD", "SUB", "IF", "REPEAT", "READ", "WRITE", ";") → statement statement_iteration statement_iteration(1: !"(", !"NOT", !"0", !"1", !"2", !"3", !"4", !"5", !"6", !"7", !"8", !"9", !"ADD", !"SUB", !"IF", !"REPEAT", !"READ", !"WRITE", !";") → ε		statement_iteration = statement >> statement_iteration "";	{LA_IS, {"ident_terminal", "(", T_NOT_0, "unsigned_value_terminal", T_ADD_0, T_SUB_0, T_IF_0, T_REPEAT_0, T_INPUT_0, T_OUTPUT_0, T_SEMICOLON_0}, {"statement", "statement_iteration"}, \\\ (LA_IS, {""}), 2, {"statement", "statement_iteration"} }\} {LA_NOT, {"ident_terminal", "(", T_NOT_0, "unsigned_value_terminal", T_ADD_0, T_SUB_0, T_IF_0, T_REPEAT_0, T_INPUT_0, T_OUTPUT_0, T_SEMICOLON_0}, {"statement", "statement_iteration"}, \\\ (LA_IS, {""}), 0, {""} }\} });\}
block_statements = ("", [statement], ");"	block_statements = ("", statement_iteration, ");"	block_statements → "(" statement_iteration ")"	block_statements(1: "(") → "(" statement_iteration ")"	block_statements = tokenBEGINBLOCK >> statement_iteration >> tokenENDBLOCK;	block_statements = tokenBEGINBLOCK >> statement_iteration >> tokenENDBLOCK;	{LA_IS, {T_BEGIN_BLOCK_0}, {"block_statements"}, \\\ (LA_IS, {""}), 3, {T_BEGIN_BLOCK_0, "statement_iteration", T_END_BLOCK_0 } }\} });\}
	expression_optional = expression "";	expression_optional → expression expression_optional → ε	expression_optional(1: "(", "NOT", "ADD", "SUB", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "IF") → expression expression_optional(1: !"(", !"NOT", !"ADD", !"SUB", !"0", !"1", !"2", !"3", !"4", !"5", !"6", !"7", !"8", !"9", !"IF") → ε		expression_optional = expression "";	{LA_IS, {"T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal", T_IF_0}, {\ "expression_optional", \\\ (LA_IS, {""}), 1, {"expression"} }\} {LA_NOT, {"", T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal", T_IF_0}, {\ "expression_optional", \\\ (LA_IS, {""}), 0, {""} }\} });\}
program_rule = "STARTPROGRAM", "DATA", declaration_optional, "", "STARTBLOCK", statement_iteration, "ENDBLOCK"; program_rule = "STARTPROGRAM", "DATA", [declaration], ";", "STARTBLOCK", [statement], "ENDBLOCK";	program_rule → "STARTPROGRAM" "DATA" declaration_optional "STARTBLOCK" statement_iteration, "ENDBLOCK";	program_rule(1: "STARTPROGRAM") → "STARTPROGRAM" "DATA" declaration_optional "STARTBLOCK" statement_iteration "ENDBLOCK"	program_rule(1: "STARTPROGRAM") → "STARTPROGRAM" "DATA" declaration_optional "STARTBLOCK" statement_iteration "ENDBLOCK"	program_rule = BOUNDARIES >> tokenNAME >> tokenDATA >> declaration_optional >> tokenSEMICOLON >> tokenBODY >> (-declaration) >> tokenEND; tokenEND;	program_rule = BOUNDARIES >> tokenNAME >> tokenDATA >> declaration_optional >> tokenSEMICOLON >> tokenBODY >> statement_iteration >> tokenEND;	{LA_IS, {T_NAME_0, "program_rule"}, {\ LA_IS, {""}, 9, {T_NAME_0, T_DATA_0, "declaration_optional", T_SEMICOLON_0, T_BODY_0, "statement_iteration", T_END_0} }\} });\}
	declaration_optional = declaration "";	declaration_optional → declaration declaration_optional → ε	declaration_optional(1: "INT_4") → declaration declaration_optional(1: !"INT_4") → ε		declaration_optional = declaration "";	{LA_IS, {T_DATA_TYPE_0}, {"declaration_optional"}, \\\ (LA_IS, {""}), 1, {"declaration"} }\} {LA_NOT, {T_DATA_TYPE_0}, {"declaration_optional"}, \\\ (LA_IS, {""}), 0, {""} }\} });\}
value = sign_optional, unsigned_value;	value → sign_optional unsigned_value	value(1: "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "ADD", "SUB") → sign_optional unsigned_value	value(1: "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "ADD", "SUB") → sign_optional unsigned_value	value = sign_optional >> unsigned_value >> BOUNDARIES;	value = sign_optional >> unsigned_value >> BOUNDARIES;	{LA_IS, {"UNSIGNED_VALUE_TERMINAL", T_ADD_0, T_SUB_0}, {"value"}, \\\ (LA_IS, {""}), 2, {"sign_optional", "unsigned_value"} }\} });\}
value = [sign], unsigned_value;	sign_optional = sign ε;	sign_optional → sign sign_optional → ε	sign_optional(1: "ADD", "SUB") → sign sign_optional(1: !"ADD", !"SUB") → ε		sign_optional = sign "";	{LA_IS, {T_ADD_0, T_SUB_0}, {"sign_optional"}, \\\ (LA_IS, {""}), 1, {"sign"} }\} {LA_NOT, {T_ADD_0, T_SUB_0}, {"sign_optional"}, \\\ (LA_IS, {""}), 0, {""} }\} });\}

)\}
sign = sign_plus sign_minus;	sign = sign_plus sign_minus; sign → sign_plus sign → sign_minus	sign → sign_plus sign → sign_minus	sign(1: "ADD") → sign_plus sign(1: "SUB") → sign_minus	sign = sign_plus sign_minus;	{LA_IS, { T_ADD_0 }, { "sign", { \(\LA_IS, {""}, 1, { "sign_plus" })\} }}, {LA_IS, { T_SUB_0 }, { "sign", { \(\LA_IS, {""}, 1, { "sign_minus" })\} }},\}
sign_plus = "ADD";	sign_plus = "ADD";	sign_plus → "ADD"	sign_plus(1: "ADD") → "ADD"	sign_plus = SAME_RULE(tokenPLUS);	{LA_IS, { T_ADD_0 }, { "sign_plus", { \(\LA_IS, {""}, 1, { T_ADD_0 }\) }\} }}, {LA_IS, { T_SUB_0 }, { "sign_minus", { \(\LA_IS, {""}, 1, { T_SUB_0 }\) }\} }},\}
sign_minus = "SUB";	sign_minus = "SUB";	sign_minus → "SUB"	sign_minus(1: ".") → "SUB"	sign_minus = SAME_RULE(tokenMINUS);	{LA_IS, { T_SUB_0 }, { "sign_minus", { \(\LA_IS, {""}, 1, { T_SUB_0 }\) }\} }},\}
unsigned_value = non_zero_digit , digit_iteration "0";	unsigned_value = non_zero_digit , digit_iteration "0";	unsigned_value → non_zero_digit digit_iteration "0";	unsigned_value(1: "1", "2", "3", "4", "5", "6", "7", "8", "9") → non_zero_digit digit_iteration unsigned_value(1: "0") → "0"	unsigned_value(1: "1", "2", "3", "4", "5", "6", "7", "8", "9") → non_zero_digit digit_iteration unsigned_value(1: "0") → "0"	/* unsigned_value token represents unsigned_value in lexical analyzer */ {LA_IS, { "unsigned_value_terminal" }, { "unsigned_value", { \(\LA_IS, {""}, 1, { "unsigned_value_terminal" }\) }\} }},\}
	digit_iteration = digit , digit_iteration ε;	digit_iteration → digit digit_iteration digit_iteration → ε	digit_iteration(1: "0", "1", "2", "3", "4", "5", "6", "7", "8", "9") → digit digit_iteration digit_iteration(1: "0", "1", "2", "3", "4", "5", "6", "7", "8", "9") → ε		digit_iteration = digit >> digit_iteration "";
digit = "0" non_zero_digit;	digit = "0" non_zero_digit;	digit → "0" digit → non_zero_digit	digit(1: "0") → "0" digit(1: "1", "2", "3", "4", "5", "6", "7", "8", "9") → non_zero_digit	digit_0 = '0'; digit = digit_0 non_zero_digit;	\
non_zero_digit = "1" "2" "3" "4" "5" "6" "7" "8" "9";	non_zero_digit = "1" "2" "3" "4" "5" "6" "7" "8" "9";	non_zero_digit → "1" non_zero_digit → "2" non_zero_digit → "3" non_zero_digit → "4" non_zero_digit → "5" non_zero_digit → "6" non_zero_digit → "7" non_zero_digit → "8" non_zero_digit → "9"	non_zero_digit(1: "1") → "1" non_zero_digit(1: "2") → "2" non_zero_digit(1: "3") → "3" non_zero_digit(1: "4") → "4" non_zero_digit(1: "5") → "5" non_zero_digit(1: "6") → "6" non_zero_digit(1: "7") → "7" non_zero_digit(1: "8") → "8" non_zero_digit(1: "9") → "9"	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'; non_zero_digit = digit_1 digit_2 digit_3 digit_4 digit_5 digit_6 digit_7 digit_8 digit_9;	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'; non_zero_digit = digit_1 digit_2 digit_3 digit_4 digit_5 digit_6 digit_7 digit_8 digit_9;
ident = letter_in_lower_case , letter_in_lower_case , letter_in_lower_case , digit , digit;	ident = letter_in_lower_case , letter_in_lower_case , letter_in_lower_case , digit , digit;	ident → letter_in_lower_case letter_in_lower_case letter_in_lower_case digit digit	ident(1: "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z") → letter_in_lower_case letter_in_lower_case letter_in_lower_case digit digit	tokenUNDERSCORE = "_"; ident = tokenINTEGER16 tokenCOMMA tokenNOT tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD tokenGROUPEXPRESSIONBEGIN tokenGROUPEXPRESSIONEND tokenLRASSIGN tokenELSE tokenIF tokenEXIT tokenREPEAT tokenUNTIL tokenGET tokenPUT tokenNAME tokenBODY tokenDATA tokenBEGIN tokenEND tokenBEGINBLOCK tokenENDBLOCK tokenLEFTSQUAREBRACKETS tokenRIGHTSQUAREBRACKETS tokenSEMICOLON)>> letter_in_lower_case >> letter_in_lower_case >> letter_in_lower_case >> digit >> digit >> letter_in_lower_case >> digit >> STRICT_BOUNDARIES;	/* ident token represents ident in lexical analyzer */ {LA_IS, { "ident_terminal" }, { "ident", { \(\LA_IS, {""}, 1, { "ident_terminal" }\) }\} }},\}
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_lower_case → "a" letter_in_lower_case → "b" letter_in_lower_case → "c" letter_in_lower_case → "d" letter_in_lower_case → "e" letter_in_lower_case → "f" letter_in_lower_case → "g" letter_in_lower_case → "h" letter_in_lower_case → "i" letter_in_lower_case → "j" letter_in_lower_case → "k" letter_in_lower_case → "l" letter_in_lower_case → "m" letter_in_lower_case → "n" letter_in_lower_case → "o" letter_in_lower_case → "p" letter_in_lower_case → "q" letter_in_lower_case → "r" letter_in_lower_case → "s" letter_in_lower_case → "t" letter_in_lower_case → "u" letter_in_lower_case → "v" letter_in_lower_case → "w" letter_in_lower_case → "x" letter_in_lower_case → "y" letter_in_lower_case → "z"	letter_in_lower_case(1: "a") → "a" letter_in_lower_case(1: "b") → "b" letter_in_lower_case(1: "c") → "c" letter_in_lower_case(1: "d") → "d" letter_in_lower_case(1: "e") → "e" letter_in_lower_case(1: "f") → "f" letter_in_lower_case(1: "g") → "g" letter_in_lower_case(1: "h") → "h" letter_in_lower_case(1: "i") → "i" letter_in_lower_case(1: "j") → "j" letter_in_lower_case(1: "k") → "k" letter_in_lower_case(1: "l") → "l" letter_in_lower_case(1: "m") → "m" letter_in_lower_case(1: "n") → "n" letter_in_lower_case(1: "o") → "o" letter_in_lower_case(1: "p") → "p" letter_in_lower_case(1: "q") → "q" letter_in_lower_case(1: "r") → "r" letter_in_lower_case(1: "s") → "s" letter_in_lower_case(1: "t") → "t" letter_in_lower_case(1: "u") → "u" letter_in_lower_case(1: "v") → "v" letter_in_lower_case(1: "w") → "w" letter_in_lower_case(1: "x") → "x" letter_in_lower_case(1: "y") → "y" letter_in_lower_case(1: "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"; 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;	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"; 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"	letter_in_upper_case = "A" "B" "C" "D" "E"	letter_in_upper_case → "A"	letter_in_upper_case(1: "A") → "A"	a = "a";	\

"O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z";	<pre> "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" " W" "X" "Y" "Z";</pre>	<pre>letter_in_upper_case → "B" letter_in_upper_case → "C" letter_in_upper_case → "D" letter_in_upper_case → "E" letter_in_upper_case → "F" letter_in_upper_case → "G" letter_in_upper_case → "H" letter_in_upper_case → "I" letter_in_upper_case → "J" letter_in_upper_case → "K" letter_in_upper_case → "L" letter_in_upper_case → "M" letter_in_upper_case → "N" letter_in_upper_case → "O" letter_in_upper_case → "P" letter_in_upper_case → "Q" letter_in_upper_case → "R" letter_in_upper_case → "S" letter_in_upper_case → "T" letter_in_upper_case → "U" letter_in_upper_case → "V" letter_in_upper_case → "W" letter_in_upper_case → "X" letter_in_upper_case → "Y" letter_in_upper_case → "Z"</pre>	<pre>letter_in_upper_case(1: "B") → "B" letter_in_upper_case(1: "C") → "C" letter_in_upper_case(1: "D") → "D" letter_in_upper_case(1: "E") → "E" letter_in_upper_case(1: "F") → "F" letter_in_upper_case(1: "G") → "G" letter_in_upper_case(1: "H") → "H" letter_in_upper_case(1: "I") → "I" letter_in_upper_case(1: "J") → "J" letter_in_upper_case(1: "K") → "K" letter_in_upper_case(1: "L") → "L" letter_in_upper_case(1: "M") → "M" letter_in_upper_case(1: "N") → "N" letter_in_upper_case(1: "O") → "O" letter_in_upper_case(1: "P") → "P" letter_in_upper_case(1: "Q") → "Q" letter_in_upper_case(1: "R") → "R" letter_in_upper_case(1: "S") → "S" letter_in_upper_case(1: "T") → "T" letter_in_upper_case(1: "U") → "U" letter_in_upper_case(1: "V") → "V" letter_in_upper_case(1: "W") → "W" letter_in_upper_case(1: "X") → "X" letter_in_upper_case(1: "Y") → "Y" letter_in_upper_case(1: "Z") → "Z"</pre>	<pre>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";</pre>	<pre>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";</pre>	<pre>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;</pre>
				<pre>STRICT_BOUNDARIES = (BOUNDARY >> *(BOUNDARY)) ((q:alpha q:i:char("_"))); BOUNDARIES = (BOUNDARY >> *(BOUNDARY) NO_BOUNDARY); BOUNDARY = BOUNDARY__SPACE BOUNDARY__TAB BOUNDARY__VERTICAL_TAB BOUNDARY__FORM_FEED BOUNDARY__CARRIAGE_RETURN BOUNDARY__LINE_FEED BOUNDARY__NULL; BOUNDARY__SPACE = " "; BOUNDARY__TAB = "\t"; BOUNDARY__VERTICAL_TAB = "\v"; BOUNDARY__FORM_FEED = "\f"; BOUNDARY__CARRIAGE_RETURN = "\r"; BOUNDARY__LINE_FEED = "\n"; BOUNDARY__NULL = "\0"; NO_BOUNDARY = ""; #define WHITESPACES \ STRICT_BOUNDARIES, \ BOUNDARIES, \ BOUNDARY, \ BOUNDARY__SPACE, \ BOUNDARY__TAB, \ BOUNDARY__VERTICAL_TAB, \ BOUNDARY__FORM_FEED, \ BOUNDARY__CARRIAGE_RETURN, \ BOUNDARY__LINE_FEED, \ BOUNDARY__NULL, \ NO_BOUNDARY</pre>	<pre>STRICT_BOUNDARIES = (BOUNDARY >> *(BOUNDARY)) ((q:alpha q:i:char("_"))); BOUNDARIES = (BOUNDARY >> *(BOUNDARY) NO_BOUNDARY); BOUNDARY = BOUNDARY__SPACE BOUNDARY__TAB BOUNDARY__VERTICAL_TAB BOUNDARY__FORM_FEED BOUNDARY__CARRIAGE_RETURN BOUNDARY__LINE_FEED BOUNDARY__NULL; BOUNDARY__SPACE = " "; BOUNDARY__TAB = "\t"; BOUNDARY__VERTICAL_TAB = "\v"; BOUNDARY__FORM_FEED = "\f"; BOUNDARY__CARRIAGE_RETURN = "\r"; BOUNDARY__LINE_FEED = "\n"; BOUNDARY__NULL = "\0"; NO_BOUNDARY = ""; #define WHITESPACES \ STRICT_BOUNDARIES, \ BOUNDARIES, \ BOUNDARY, \ BOUNDARY__SPACE, \ BOUNDARY__TAB, \ BOUNDARY__VERTICAL_TAB, \ BOUNDARY__FORM_FEED, \ BOUNDARY__CARRIAGE_RETURN, \ BOUNDARY__LINE_FEED, \ BOUNDARY__NULL, \ NO_BOUNDARY</pre>	<pre>\ \ \ \ \ \ { LA_IS, { T_NAME_0 }, { "program__part1", \ { LA_IS, {""}, 7, { T_NAME_0, T_DATA_0, \ "declaration_optional", T_SEMICOLON_0, T_BODY_0 \ }}}, \ }}, \ }, \ "program_rule"</pre>