

РБНФ №1 (опис синтаксису всіма допустимими засобами РБНФ)	РБНФ №2 (опис формальної граматики засобами РБНФ)	Формальна граматика	Формальна граматика з специфікацією lookahead у правилах для LL(2)-аналізатора	Перевірка РБНФ №1 за допомогою коду <i>(помістити у файл "EBNF_N1.h")</i> */	Перевірка РБНФ №2 за допомогою коду <i>(помістити у файл "EBNF_N2.h")</i> */	Перевірки прототипу LL(2)-синтаксичного аналізатора (спеціальна структура) та прототипу лексичного аналізатора (регулярні вирази) за допомогою коду. Лексеми для синтаксичного аналізатора обробляються лексичним аналізатором, тому синтаксичний аналізатор не аналізує їх повторно (як показано в РБНФ). <i>(помістити у файл "LexicaByRegExAndSyntaxByLL2protototype.h")</i> УВАГА: при копіюванні вражайте, щоб у кожному рядку після символу «\» не містилось жодних інших символів. */
		G = {N, T, P, S}	G = {N, T, P, S}			
		S → program_rule	S → program_rule			
		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, continue_while, break_while, statement_in_while_and_if_body, statement, block_statements_in_while_and_if_body, statement_in_while_and_if_body_iteration, while_cycle_head_expression, while_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 }	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, continue_while, break_while, statement_in_while_and_if_body, statement, block_statements_in_while_and_if_body, statement_in_while_and_if_body_iteration, while_cycle_head_expression, while_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 }	#define NONTERMINALS program_name, \ value_type, \ array_specify, \ declaration_element, \ \ other_declaration_ident, \ declaration, \ \ index_action, \ unary_operator, \ unary_operation, \ binary_operator, \ binary_action, \ left_expression, \ group_expression, \ \ expression, \ \ expression_or_cond_block_with_optional_assign, \ assign_to_right, \ \ 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, \ continue_while, \ break_while, \ statement_in_while_and_if_body, \ statement, \ block_statements_in_while_and_if_body, \ statement_in_while_and_if_body_iteration, \ while_cycle_head_expression, \ while_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	#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, \ continue_while, \ break_while, \ statement_in_while_and_if_body, \ statement, \ block_statements_in_while_and_if_body, \ statement_in_while_and_if_body_iteration, \ while_cycle_head_expression, \ while_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	
		T = { "INTEGER16", "n", "NOT", "AND", "OR", "==" , "!=", "<", ">", "+", "-", "*", "/", "DIV", "MOD", "(", 	"INTEGER16", "n", "NOT", "AND", "OR", "==" , "!=", "<", ">", "+", "-", "*", "/", "DIV", "MOD", "(", 	#define TOKENS \ tokenINTEGER16, \ tokenCOMMA, \ tokenNOT, \ tokenAND, \ tokenOR, \ tokenEQUAL, \ tokenNOTEQUAL, \ tokenLESS, \ tokenGREATER, \ tokenPLUS, \ tokenMINUS, \ tokenMUL, \ tokenDIV, \ tokenMOD, \ tokenGROUPEXPRESSIONBEGIN, \ 	#define TOKENS \ tokenINTEGER16, \ tokenCOMMA, \ tokenNOT, \ tokenAND, \ tokenOR, \ tokenEQUAL, \ tokenNOTEQUAL, \ tokenLESS, \ tokenGREATER, \ tokenPLUS, \ tokenMINUS, \ tokenMUL, \ tokenDIV, \ tokenMOD, \ tokenGROUPEXPRESSIONBEGIN, \ 	

				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 = "int32" >> STRICT_BOUNDARIES;	tokenINTEGER16 = "int32" >> STRICT_BOUNDARIES;	#define T_DATA_TYPE_0 "int32" #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 = "!" >> STRICT_BOUNDARIES;	tokenNOT = "!" >> STRICT_BOUNDARIES;	#define T_NOT_0 "!" #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 "&" #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 = " " >> STRICT_BOUNDARIES;	tokenOR = " " >> STRICT_BOUNDARIES;	#define T_OR_0 " " #define T_OR_1 "" #define T_OR_2 "" #define T_OR_3 ""
				tokenEQUAL = "==" >> BOUNDARIES;	tokenEQUAL = "==" >> BOUNDARIES;	#define T_EQUAL_0 "==" #define T_EQUAL_1 "" #define T_EQUAL_2 "" #define T_EQUAL_3 ""
				tokenNOTEQUAL = "!=" >> BOUNDARIES;	tokenNOTEQUAL = "!=" >> BOUNDARIES;	#define T_NOT_EQUAL_0 "!=" #define T_NOT_EQUAL_1 "" #define T_NOT_EQUAL_2 "" #define T_NOT_EQUAL_3 ""
				tokenLESS = "<" >> BOUNDARIES;	tokenLESS = "<" >> BOUNDARIES;	#define T_LESS_0 "<" #define T_LESS_1 "" #define T_LESS_2 "" #define T_LESS_3 ""
				tokenGREATER = ">" >> BOUNDARIES;	tokenGREATER = ">" >> BOUNDARIES;	#define T_GREATER_0 ">" #define T_GREATER_1 "" #define T_GREATER_2 "" #define T_GREATER_3 ""
				tokenPLUS = "+" >> BOUNDARIES;	tokenPLUS = "+" >> BOUNDARIES;	#define T_ADD_0 "+" #define T_ADD_1 "" #define T_ADD_2 "" #define T_ADD_3 ""
				tokenMINUS = "-" >> BOUNDARIES;	tokenMINUS = "-" >> BOUNDARIES;	#define T_SUB_0 "-" #define T_SUB_1 "" #define T_SUB_2 "" #define T_SUB_3 ""
				tokenMUL = "*" >> BOUNDARIES;	tokenMUL = "*" >> BOUNDARIES;	#define T_MUL_0 "" #define T_MUL_1 "" #define T_MUL_2 "" #define T_MUL_3 ""
				tokenDIV = "/" >> STRICT_BOUNDARIES;	tokenDIV = "/" >> STRICT_BOUNDARIES;	#define T_DIV_0 "/" #define T_DIV_1 "" #define T_DIV_2 "" #define T_DIV_3 ""
				tokenMOD = "%" >> STRICT_BOUNDARIES;	tokenMOD = "%" >> STRICT_BOUNDARIES;	#define T_MOD_0 "" #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 "else" #define T_ELSE_IF_1 T_IF_0 #define T_ELSE_IF_2 "" #define T_ELSE_IF_3 ""
				tokenWHILE = "while" >> STRICT_BOUNDARIES;	tokenWHILE = "while" >> STRICT_BOUNDARIES;	#define T_WHILE_0 "while" #define T_WHILE_1 "" #define T_WHILE_2 "" #define T_WHILE_3 ""
				tokenCONTINUE = "continue" >> STRICT_BOUNDARIES;	tokenCONTINUE = "continue" >> STRICT_BOUNDARIES;	#define T_CONTINUE_WHILE_0 "continue" #define T_CONTINUE_WHILE_1 "" #define T_CONTINUE_WHILE_2 ""

						#define T_CONTINUE_WHILE_3 ""
						#define T_EXIT_WHILE_0 "break" #define T_EXIT_WHILE_1 "" #define T_EXIT_WHILE_2 "" #define T_EXIT_WHILE_3 ""
				tokenBREAK = "break" >> STRICT_BOUNDARIES;	tokenBREAK = "break" >> STRICT_BOUNDARIES;	
				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 ""
				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 = "program" >> STRICT_BOUNDARIES;	tokenNAME = "program" >> STRICT_BOUNDARIES;	#define T_NAME_0 "program" #define T_NAME_1 "" #define T_NAME_2 "" #define T_NAME_3 ""
				tokenBODY = "begin" >> STRICT_BOUNDARIES;	tokenBODY = "begin" >> STRICT_BOUNDARIES;	#define T_BODY_0 "begin" #define T_BODY_1 "" #define T_BODY_2 "" #define T_BODY_3 ""
				tokenDATA = "var" >> STRICT_BOUNDARIES;	tokenDATA = "var" >> STRICT_BOUNDARIES;	#define T_DATA_0 "var" #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 = "end" >> STRICT_BOUNDARIES;	tokenEND = "end" >> STRICT_BOUNDARIES;	#define T_END_0 "end" #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_JS, ("ident_terminal"), { "program_name", {\
value_type = "int32";	value_type = "int32";	value_type → "int32"	value_type(1: "int32") → "int32"	value_type = SAME_RULE(tokenINT32);	value_type = SAME_RULE(tokenINT32);	{ LA_JS, (T_DATA_TYPE_0), { "value_type", {\
	array_specify = "[", unsigned_value, "]";	array_specify → "[" unsigned_value "]"	array_specify(1: "["") → "[" unsigned_value "]"		array_specify = "[" >> unsigned_value >> "]"	{ LA_JS, ("["), { "array_specify", {\
declaration_element = ident, array_specify__optional;	declaration_element = ident, array_specify__optional;	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_JS, ("ident_terminal"), { "declaration_element", {\
	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 "";	{ LA_JS, ("["), { "array_specify__optional", {\
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_JS, (T_COMA_0), { "other_declaration_ident", {\
declaration = value_type, declaration_element, other_declaration_ident__iteration;	declaration = value_type, declaration_element, other_declaration_ident__iteration;	declaration → value_type declaration_element other_declaration_ident__iteration	declaration(1: "INTEGER16") → 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_JS, (T_DATA_TYPE_0), { "declaration", {\
	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_JS, (T_COMA_0), { "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_JS, ("["), { "index_action", {\
unary_operator = "!";	unary_operator = "!";	unary_operator → "!"	unary_operator(1: "!") → "!"	unary_operator = SAME_RULE(tokenNOT);	unary_operator = SAME_RULE(tokenNOT);	{ LA_JS, (T_NOT_0), { "unary_operator", {\
unary_operation = unary_operator , expression;	unary_operation = unary_operator , expression;	unary_operation → unary_operator expression	unary_operation(1: "!") → unary_operator expression	unary_operation = unary_operator >> expression;	unary_operation = unary_operator >> expression;	{ LA_JS, (T_NOT_0), { "unary_operation", {\
binary_operator = "&" " " "==" "!=" "lt" "gt" "add" "sub" "*" "/" "%";	binary_operator = "&" " " "==" "!=" "lt" "gt" "add" "sub" "*" "/" "%";	binary_operator → "&" binary_operator → " " binary_operator → "==" binary_operator → "!=" binary_operator → "lt" binary_operator → "gt" binary_operator → "add" binary_operator → "sub" binary_operator → "*" binary_operator → "/" binary_operator → "%"	binary_operator(1: "&") → "&" binary_operator(1: " ") → " " binary_operator(1: "==") → "==" binary_operator(1: "!=") → "!=" binary_operator(1: "lt") → "lt" binary_operator(1: "gt") → "gt" binary_operator(1: "add") → "add" binary_operator(1: "sub") → "sub" binary_operator(1: "**") → "**" binary_operator(1: "/") → "/" binary_operator(1: "%") → "%"	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_JS, (T_AND_0), { "binary_operator", {\
binary_operator = "&" " " "==" "!=" "lt" "gt" "add" "sub" "*" "/" "%";	binary_operator = "&" " " "==" "!=" "lt" "gt" "add" "sub" "*" "/" "%";	binary_operator → "&" binary_operator → " " binary_operator → "==" binary_operator → "!=" binary_operator → "lt" binary_operator → "gt" binary_operator → "add" binary_operator → "sub" binary_operator → "*" binary_operator → "/" binary_operator → "%"	binary_operator(1: "&") → "&" binary_operator(1: " ") → " " binary_operator(1: "==") → "==" binary_operator(1: "!=") → "!=" binary_operator(1: "lt") → "lt" binary_operator(1: "gt") → "gt" binary_operator(1: "add") → "add" binary_operator(1: "sub") → "sub" binary_operator(1: "**") → "**" binary_operator(1: "/") → "/" binary_operator(1: "%") → "%"	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_JS, (T_AND_0), { "binary_operator", {\

						{ LA_IS, { T_SUB_0 }, { "binary_operator", {\ { LA_IS, { "" }, 1, { T_SUB_0 }} }\ }};\ { LA_IS, { T_MUL_0 }, { "binary_operator", {\ { LA_IS, { "" }, 1, { T_MUL_0 }} }\ }};\ { LA_IS, { T_DIV_0 }, { "binary_operator", {\ { LA_IS, { "" }, 1, { T_DIV_0 }} }\ }};\ { 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: "&", " ", "=", "!=", "lt", "gt", "add", "sub", "**", "/", "%") → binary_operator expression		binary_action = binary_operator >> expression;	{ LA_IS, { T_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" }} }\ }};\
left_expression = group_expression unary_operation cond_block value ident , [index_action];	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: " ") → 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: ".") → ident , index_action__optional	left_expression = group_expression unary_operation cond_block value ident >> -index_action;	left_expression = group_expression unary_operation cond_block value ident >> index_action__optional;	{LA_IS, { "(" }, { "left_expression", {\ {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", {\ "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" }} }\ }};\
	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, { "(" }, { "index_action__optional", {\ {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" , "+" , "-" , " " , "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;	{LA_IS, { "(" , T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal", T_IF_0 }, {\ "expression", {\ {LA_IS, { "" }, 2, { "left_expression", "binary_action__iteration" }} }\ }};\
	binary_action__iteration = binary_action, binary_action__iteration ε;	binary_action__iteration → binary_action binary_action__iteration binary_action__iteration → ε	binary_action__iteration(1: "AND", "OR", "=", "!=", "<", ">", "+", "-", "**", "DIV", "MOD") → binary_action binary_action__iteration(1: "! "AND", "! "OR", "! "=", "! "!=", "! "<", "! ">, "! "+", "! "-", "! "**, "! "DIV", "! "MOD") → ε		binary_action__iteration = binary_action >> binary_action__iteration "";	{LA_IS, { T_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__iteration", {\ {LA_IS, { "" }, 2, { "binary_action", "binary_action__iteration" }} }\ }};\ {LA_NOT, { T_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__iteration", {\ {LA_IS, { "" }, 0, { "" }} }\ }};\
group_expression = "(" , expression , ")";	group_expression = "(" , expression , ")";	group_expression → "(" expression ")"	group_expression(1: "(") → "(" expression ")"	group_expression = tokenGROUPEXPRESSSIONBEGIN >> expression >> tokenGROUPEXPRESSSIONEND;	group_expression = tokenGROUPEXPRESSSIONBEGIN >> expression >> tokenGROUPEXPRESSSIONEND;	{LA_IS, { "(" }, { "group_expression", {\ {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: "(" , "!", "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;	{LA_IS, { "(" , T_NOT_0, T_ADD_0, T_SUB_0, "ident_terminal", "unsigned_value_terminal", T_IF_0 }, {\ "expression_or_cond_block_with_optional_assign", {\ {LA_IS, { "" }, 2, { "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 = 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 = 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: "(" , "!", "add", "sub", " " , "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "if") → expression	if_expression = SAME_RULE(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_in_while_and_if_body;	body_for_true = block_statements_in_while_and_if_body;	body_for_true → block_statements_in_while_and_if_body	body_for_true(1: "(") → block_statements_in_while_and_if_body	body_for_true = SAME_RULE(block_statements_in_while_and_if_body);	body_for_true = SAME_RULE(block_statements_in_while_and_if_body);	{LA_IS, { T_BEGIN_BLOCK_0 }, {\ "body_for_true", {\ {LA_IS, { "" }, 1, {\ "block_statements_in_while_and_if_body" }} }\ }};\
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_in_while_and_if_body;	body_for_false = "else" , block_statements_in_while_and_if_body;	body_for_false → "else" block_statements_in_while_and_if_body	body_for_false(1: "else") → "else" block_statements_in_while_and_if_body	body_for_false = tokenELSE >> block_statements_in_while_and_if_body;	body_for_false = tokenELSE >> block_statements_in_while_and_if_body;	{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: !"if") → ε false_cond_block_without_else_iteration(1: !"else") → ε		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, {T_ELSE_IF_1}, 2, { "false_cond_block_without_else", "false_cond_block_without_else__iteration" }}\ LA_NOT, { T_ELSE_IF_1 }, 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 = body_for_false "";	{LA_IS, { T_ELSE_BLOCK_0 }, { "body_for_false__optional", {\ LA_IS, {""}, 1, { "body_for_false" }}\ }}\
		continue_while = "continue";	continue_while → "continue"	continue_while(1: "continue") → "continue"	continue_while = SAME_RULE(tokenCONTINUE);	{LA_IS, { T_CONTINUE_WHILE_0 }, { "continue_while", {\ LA_IS, {""}, 1, { T_CONTINUE_WHILE_0 }}\ }}\
				continue_while = SAME_RULE(tokenCONTINUE);		
	break_while = "break";	break_while → "break"	break_while(1: "break") → "break"	break_while = SAME_RULE(tokenBREAK);	break_while = SAME_RULE(tokenBREAK);	{LA_IS, { T_EXIT_WHILE_0 }, { "break_while", {\ LA_IS, {""}, 1, { T_EXIT_WHILE_0 }}\ }}\
statement_in_while_and_if_body = statement "continue" "break";	statement_in_while_and_if_body = statement continue_while break_while;	statement_in_while_and_if_body → statement statement_in_while_and_if_body → continue_while statement_in_while_and_if_body → break_while	statement_in_while_and_if_body(1: " " , "(", "!", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "add", "sub", "if", "while", "read", "write", ",") → statement statement_in_while_and_if_body(1: "continue") → continue_while statement_in_while_and_if_body(1: "break") → break_while	statement_in_while_and_if_body = statement continue_while break_while;	statement_in_while_and_if_body = statement continue_while break_while;	{LA_IS, { "ident_terminal", "(", T_NOT_0, "unsigned_value_terminal", T_ADD_0, T_SUB_0, T_IF_0, T_WHILE_0, T_INPUT_0, T_OUTPUT_0, T_SEMICOLON_0 }, { "statement_in_while_and_if_body", {\ LA_IS, {""}, 1, { "statement" }}\ }}\
block_statements_in_while_and_if_body = "(" , {statement_in_while_and_if_body}, ")";	block_statements_in_while_and_if_body = "(" , statement_in_while_and_if_body__iteration , ")";	block_statements_in_while_and_if_body → "(" statement_in_while_and_if_body__iteration ")"	block_statements_in_while_and_if_body(1: "(") → "(" statement_in_while_and_if_body__iteration ")"	block_statements_in_while_and_if_body = tokenBEGINBLOCK >> *statement_in_while_and_if_body >> tokenENDBLOCK;	block_statements_in_while_and_if_body = tokenBEGINBLOCK >> statement_in_while_and_if_body__iteration >> tokenENDBLOCK;	{LA_IS, { T_BEGIN_BLOCK_0 }, { "block_statements_in_while_and_if_body", {\ LA_IS, {""}, 3, { T_BEGIN_BLOCK_0, "statement_in_while_and_if_body__iteration", T_END_BLOCK_0 }}\ }}\
	statement_in_while_and_if_body_iteration = statement_in_while_and_if_body , statement_in_while_and_if_body__iteration ε;	statement_in_while_and_if_body_iteration → statement_in_while_and_if_body statement_in_while_and_if_body__iteration statement_in_while_and_if_body__iteration → ε	statement_in_while_and_if_body_iteration(1: " " , "(", "!", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "add", "sub", "if", "while", "read", "write", ",," , "continue", ",") → statement_in_while_and_if_body statement_in_while_and_if_body__iteration statement_in_while_and_if_body__iteration(1: " " , "(", "!", "!", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "!", "9", "!", "+", "!", "-", "!", "!", "!", "while", "!", "read", "!", "write", "!", ":", "!", "continue", "!", "break") → ε		statement_in_while_and_if_body_iteration = statement_in_while_and_if_body >> statement_in_while_and_if_body__iteration "";	{LA_IS, { "ident_terminal", "(", T_NOT_0, "unsigned_value_terminal", T_ADD_0, T_SUB_0, T_IF_0, T_WHILE_0, T_INPUT_0, T_OUTPUT_0, T_SEMICOLON_0, T_CONTINUE_WHILE_0, T_EXIT_WHILE_0 }, { "statement_in_while_and_if_body__iteration", {\ LA_IS, {""}, 2, { "statement_in_while_and_if_body", "statement_in_while_and_if_body__iteration" }}\ }}\
while_cycle_head_expression = expression;	while_cycle_head_expression = expression;	while_cycle_head_expression → expression	while_cycle_head_expression(1: "(" , "!", "add", "sub", " " , "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "!", "if") → expression	while_cycle_head_expression = SAME_RULE(expression);	while_cycle_head_expression = SAME_RULE(expression);	{LA_NOT, { "ident_terminal", "(", T_NOT_0, "unsigned_value_terminal", T_ADD_0, T_SUB_0, T_IF_0, T_WHILE_0, T_INPUT_0, T_OUTPUT_0, T_SEMICOLON_0, T_CONTINUE_WHILE_0, T_EXIT_WHILE_0 }, { "statement_in_while_and_if_body__iteration", {\ LA_IS, {""}, 0, { "" }}\ }}\
while_cycle = "while" , while_cycle_head_expression , block_statements_in_while_and_if_body;	while_cycle = "while" , while_cycle_head_expression , block_statements_in_while_and_if_body;	while_cycle → "while" while_cycle_head_expression block_statements_in_while_and_if_body	while_cycle(1: "while") → "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;	while_cycle = tokenWHILE >> while_cycle_head_expression >> while_cycle_head_expression >> block_statements_in_while_and_if_body;	{LA_IS, { T_WHILE_0 }, { "while_cycle", {\ LA_IS, {""}, 3, { T_WHILE_0, "while_cycle_head_expression", "block_statements_in_while_and_if_body" }}\ }}\
	statements_or_block_statements = statement__iteration block_statements;	statements_or_block_statements → statement__iteration statements_or_block_statements → block_statements	statements_or_block_statements(1: " " , "(", "!", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "add", "sub", "if", "while", "read", "write", ",") → statement__iteration statements_or_block_statements(1: "(") → block_statements		statements_or_block_statements = statement__iteration block_statements;	{LA_IS, { "ident_terminal", "(", T_NOT_0, "unsigned_value_terminal", T_ADD_0, T_SUB_0, T_IF_0, T_WHILE_0, T_INPUT_0, T_OUTPUT_0, T_SEMICOLON_0 }, { "statements_or_block_statements", {\ LA_IS, {""}, 1, { "statement__iteration" }}\ }}\
input_rule = "read" , (ident , [index_action] "(" , ident , [index_action] , ")");	input_rule = "read" , argument_for_input;	input_rule → "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" }}\ }}\
	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: " ") → ident index_action__optional argument_for_input(1: "(") → "(" "ident" "index_action__optional" ")"		argument_for_input = ident >> index_action__optional tokenGROUPEXPRESSIONBEGIN >> ident >> index_action__optional >> tokenGROUPEXPRESSIONEND;	{LA_IS, { " ident_terminal " }, { "argument_for_input", {\ LA_IS, {""}, 2, { "ident", "index_action__optional" }}\ }}\
output_rule = "write" , expression;	output_rule = "write", expression;	output_rule → "write" expression	output(1: "write") → "write" expression	output_rule = tokenPUT >> 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 while_cycle input_rule output_rule ";"	statement = expression_or_cond_block__with_optional_assign while_cycle input_rule output_rule ";"	statement → expression_or_cond_block__with_optional_assign statement → while_cycle statement → input_rule statement → output_rule statement → ";"	statement(1: " " , 2: "!" :") → expression_or_cond_block__with_optional_assign statement(1: "(" , "!", "add", "sub", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "!", "if") → expression_or_cond_block__with_optional_assign statement(1: "while") → while_cycle statement(1: "read") → input_rule statement(1: "write") → output_rule statement(1: ";") → ";"	statement = expression_or_cond_block__with_optional_assign while_cycle input_rule output_rule tokenSEMICOLON;	statement = expression_or_cond_block__with_optional_assign while_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" }}\ }}\
				statement = expression_or_cond_block__with_optional_assign while_cycle input_rule output_rule tokenSEMICOLON;		{LA_IS, { T_WHILE_0 }, { "statement", {\ LA_IS, {""}, 1, {"while_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, {" ";" }}\ }}}\
	statement__iteration = statement, statement__iteration ε;	statement__iteration → statement statement__iteration statement__iteration → ε	statement__iteration(1: "_" , {"(", "!", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "add", "sub", "if", "while", "write", ";") → statement statement__iteration statement__iteration(1: !"_" , !"(", !"!", !"0", !"1", !"2", !"3", !"4", !"5", !"6", !"7", !"8", !"9", !"add", !"sub", !"if", !"while", !"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_WHILE_0, T_INPUT_0, T_OUTPUT_0, T_SEMICOLON_0 }, { "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_WHILE_0, T_INPUT_0, T_OUTPUT_0, T_SEMICOLON_0 }, { "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 >> 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: "(" , !"(", "add", "sub", "_", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "if") → expression expression__optional(1: !"(", !"!", !"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 = "program" , program_name , ";" , "begin" , "var" , declaration__optional , ";" , statement__iteration , "end";	program_rule = "program" , program_name , ";" , "begin" , "var" , declaration__optional , ";" , statement__iteration , "end";	program_rule → "program" program_name ";" "begin" "var" declaration__optional ";" statement__iteration "end"	program_rule(1: "program") → "program" program_name ";" "begin" "var" declaration__optional ";" statement__iteration "end"	program_rule = BOUNDARIES >> tokenNAME >> program_name >> tokenSEMICOLON >> tokenBODY >> tokenDATA >> (- declaration) >> tokenSEMICOLON >> *statement >> tokenEND;	program_rule = BOUNDARIES >> tokenNAME >> program_name >> tokenSEMICOLON >> tokenBODY >> tokenDATA >> declaration__optional >> tokenSEMICOLON >> statement__iteration >> tokenEND;	{ LA_IS, { T_NAME_0 }, { "program_rule", {\ { LA_IS, {""}, 9, { T_NAME_0 , "program_name", T_SEMICOLON_0, T_BODY_0, T_DATA_0, "declaration__optional", T_SEMICOLON_0, "statement__iteration", T_END_0 }}\ }}}\
	declaration__optional = declaration "";	declaration__optional → declaration declaration__optional → ε	declaration__optional(1: "int32") → declaration declaration__optional(1: "int32") → 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] , unsigned_value;	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 = -sign >> 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" }}\ }}}\
	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(1: "add") → sign_plus sign(1: "sub") → sign_minus	sign = sign_plus 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);	sign_plus = SAME_RULE(tokenPLUS);	{LA_IS, { T_ADD_0 }, { "sign_plus", {\ {LA_IS, {""}, 1, {T_ADD_0}}\ }}}\
sign_minus = "sub";	sign_minus = "sub";	sign_minus → "sub"	sign_minus(1: "sub") → "sub"	sign_minus = SAME_RULE(tokenMINUS);	sign_minus = SAME_RULE(tokenMINUS);	{LA_IS, { T_SUB_0 }, { "sign_minus", {\ {LA_IS, {""}, 1, {T_SUB_0}}\ }}}\
unsigned_value = non_zero_digit , {digit} "0";	unsigned_value = non_zero_digit , digit__iteration "0";	unsigned_value → non_zero_digit digit__iteration unsigned_value → "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 = (non_zero_digit >> *digit digit_0) >> BOUNDARIES;	unsigned_value = (non_zero_digit >> digit__iteration digit_0) >> BOUNDARIES;	/* 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;	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_upper_case , letter_in_upper_case , letter_in_upper_case ;	ident = letter_in_upper_case , letter_in_upper_case , letter_in_upper_case ;	ident → letter_in_upper_case letter_in_upper_case letter_in_upper_case	Ident(1: "_") → letter_in_upper_case letter_in_upper_case letter_in_upper_case	ident = !(tokenINTEGER16 tokenCOMMA tokenNOT tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER	ident = !(tokenINTEGER16 tokenCOMMA tokenNOT tokenAND tokenOR tokenEQUAL tokenNOTEQUAL tokenLESS tokenGREATER	/* ident token represents ident in lexical analyzer */\ {LA_IS, { "ident_terminal" }, { "ident", {\ {LA_IS, {""}, 1, {"ident_terminal" }}\ }}}\

				<div>tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD tokenGROUPEXPRESSSIONBEGIN tokenGROUPEXPRESSSIONEND tokenLRASSIGN tokenELSE tokenIF tokenWHILE tokenCONTINUE tokenBREAK tokenEXIT tokenGET tokenPUT tokenNAME tokenBODY tokenDATA tokenBEGIN tokenEND tokenBEGINBLOCK tokenENDBLOCK tokenLEFTSQUAREBRACKETS tokenRIGHTSQUAREBRACKETS tokenSEMICOLON) >> letter_in_upper_case >> letter_in_upper_case >> letter_in_upper_case >> STRICT_BOUNDARIES;</div>	<div>tokenPLUS tokenMINUS tokenMUL tokenDIV tokenMOD tokenGROUPEXPRESSSIONBEGIN tokenGROUPEXPRESSSIONEND tokenLRASSIGN tokenELSE tokenIF tokenWHILE tokenCONTINUE tokenBREAK tokenEXIT tokenGET tokenPUT tokenNAME tokenBODY tokenDATA tokenBEGIN tokenEND tokenBEGINBLOCK tokenENDBLOCK tokenLEFTSQUAREBRACKETS tokenRIGHTSQUAREBRACKETS tokenSEMICOLON) >> letter_in_upper_case >> letter_in_upper_case >> letter_in_upper_case >> STRICT_BOUNDARIES;</div>	
<div>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";</div>	<div>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";</div>	<div>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"</div>	<div>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"</div>	<div>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;</div>	<div>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;</div>	<div>\</div>
<div>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";</div>	<div>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";</div>	<div>letter_in_upper_case → "A" 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"</div>	<div>letter_in_upper_case(1: "A") → "A" 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"</div>	<div>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_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;</div>	<div>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_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;</div>	<div>\</div>
				<div>STRICT_BOUNDARIES = (BOUNDARY >> *(BOUNDARY)) (!{qi::alpha qi::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, \ </div>	<div>STRICT_BOUNDARIES = (BOUNDARY >> *(BOUNDARY)) (!{qi::alpha qi::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, \ </div>	<div>\</div>

				BOUNDARY__VERTICAL_TAB, \ BOUNDARY__FORM_FEED, \ BOUNDARY__CARRIAGE_RETURN, \ BOUNDARY__LINE_FEED, \ BOUNDARY__NULL, \ NO_BOUNDARY	BOUNDARY__TAB, \ BOUNDARY__VERTICAL_TAB, \ BOUNDARY__FORM_FEED, \ BOUNDARY__CARRIAGE_RETURN, \ BOUNDARY__LINE_FEED, \ BOUNDARY__NULL, \ NO_BOUNDARY	
						 \ \ \ \ { LA_IS, { T_NAME_0 }, { "program____part1",{\br/> { LA_IS, {""}, 7, { T_NAME_0, "program_name", T_SEMICOLON_0, T_BODY_0, T_DATA_0, "declaration_optional", T_SEMICOLON_0 }}\ }};\br/> \ },\ "program_rule"