

<p style="text-align: center;"><b>РБНФ №1</b> <b>(опис синтаксису всіма допустимими засобами РБНФ)</b></p>	<p style="text-align: center;"><b>РБНФ №2</b> <b>(опис формальної граматики засобами РБНФ)</b></p>	<p style="text-align: center;">Формальна граматика</p>	<p style="text-align: center;">Формальна граматика з специфікацією lookahead у правилах для LL(2)-аналізатора</p>		<p style="text-align: center;">/* Перевірка РБНФ №2 за допомогою коду (помістити у файл "EBNF_N2.h") */</p>	<p style="text-align: center;">/* 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). (помістити у файл "LexicaByRegExAndSyntaxByLL2prototype.h") УВАГА: при копіюванні зважайте, щоб у кожному рядку після символу «\» не містилось жодних інших символів. */</p>
		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, 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 }</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, 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 }</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, \ 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, \ program_rule, \ 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, \ 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 }</pre>	
		<pre>T = { "INTEGER16", ",", "NOT", "AND", "OR", "==", "!=", "&lt;=", "&gt;=", "+", "-", "**", "DIV", "MOD", "(",</pre>	<pre>T = { "INTEGER16", ",", "NOT", "AND", "OR", "==", "!=", "&lt;=", "&gt;=", "+", "-", "**", "DIV", "MOD", "(",</pre>	<pre>#define TOKENS \ tokenINTEGER16, \ tokenCOMMA, \ tokenNOT, \ tokenAND, \ tokenOR, \ tokenEQUAL, \ tokenNOTEQUAL, \ tokenLESS, \ tokenGREATER, \ tokenPLUS, \ tokenMINUS, \ tokenMUL, \ tokenDIV, \ tokenMOD, \ tokenGROUPEXPRESSIONBEGIN, \</pre>		



				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 = "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 = "*" >> 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 ""
				tokenBREAK = "break" >> STRICT_BOUNDARIES;	#define T_EXIT WHILE_0 "break" #define T_EXIT WHILE_1 "" #define T_EXIT WHILE_2 "" #define T_EXIT WHILE_3 ""
				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;	#define T_INPUT_0 "read" #define T_INPUT_1 "" #define T_INPUT_2 "" #define T_INPUT_3 ""
				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;	#define T_NAME_0 "program" #define T_NAME_1 "" #define T_NAME_2 "" #define T_NAME_3 ""
				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;	#define T_DATA_0 "var" #define T_DATA_1 "" #define T_DATA_2 "" #define T_DATA_3 ""
				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;	#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);	{ LA_IS, {"ident_terminal"}, {"program_name"}, \(\{ LA_IS, {""}, 1, {"ident"}\}\)}
value_type = "int32";	value_type = "int32";	value_type → "int32"	value_type(1: "int32") → "int32"	value_type = SAME_RULE(tokenINT32);	{ 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 >> "]";	{ 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);	{ LA_IS, {"ident_terminal"}, {"declaration_element"}, \(\{ LA_IS, {""}, 2, {"ident", "array_specify_optional"}\}\)}
	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_IS, {""}, {"array_specify_optional"}, \(\{ LA_IS, {""}, 1, {"array_specify"}\}\)\} { LA_NOT, {"["}, {"array_specify_optional"}, \(\{ LA_IS, {""}, 0, {"["}\}\)}
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;	{ 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: "INTEGER16") → value_type declaration_element other_declaration_ident_iteration	declaration = value_type >> declaration_element >> *other_declaration_ident;	{ 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 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_iteration other_declaration_ident_iteration   "";	{ LA_IS, {T_COMA_0}, {"other_declaration_ident_iteration"}, \(\{ LA_IS, {""}, 2, {"other_declaration_ident", "other_declaration_ident_iteration"}\}\)\} { LA_NOT, {T_COMA_0}, {"other_declaration_ident_iteration"}, \(\{ LA_IS, {""}, 0, {""}\}\)}
index_action = "[" , expression , "]";	index_action = "[" , expression , "]";	index_action → "[" expression "]"	index_action(1: "[" ) → "[" expression "]"	index_action = tokenLEFTSQUAREBRACKETS >> expression >> tokenRIGHTSQUAREBRACKETS;	{ LA_IS, {""}, {"index_action"}, \(\{ LA_IS, {""}, 3, {"expression", ""]"}\}\)}
unary_operator = "!";	unary_operator = "!";	unary_operator → "!"	unary_operator(1: "!"") → "!"	unary_operator = SAME_RULE(tokenNOT);	{ LA_IS, {T_NOT_0}, {"unary_operator"}, \(\{ LA_IS, {""}, 4, {T_NOT_0}\}\)}
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;	{ LA_IS, {T_NOT_0}, {"unary_operation"}, \(\{ LA_IS, {""}, 2, {"unary_operator", "expression"}\}\)}
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 → "&" binary_operator → "!" 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_IS, {T_AND_0}, {"binary_operator"}, \(\{ LA_IS, {""}, 1, {T_AND_0}\}\)\} { LA_IS, {T_OR_0}, {"binary_operator"}, \(\{ LA_IS, {""}, 1, {T_OR_0}\}\)\} { LA_IS, {T_EQUAL_0}, {"binary_operator"}, \(\{ LA_IS, {""}, 1, {T_EQUAL_0}\}\)\} { LA_IS, {T_NOTEQUAL_0}, {"binary_operator"}, \(\{ LA_IS, {""}, 1, {T_NOTEQUAL_0}\}\)\} { LA_IS, {T_LESS_0}, {"binary_operator"}, \(\{ LA_IS, {""}, 1, {T_LESS_0}\}\)\} { LA_IS, {T_GREATER_0}, {"binary_operator"}, \(\{ LA_IS, {""}, 1, {T_GREATER_0}\}\)\} { LA_IS, {T_ADD_0}, {"binary_operator"}, \(\{ LA_IS, {""}, 1, {T_ADD_0}\}\)\}

					{ LA_IS, { T_SUB_0 }, { "binary_operator", \\\n        { LA_IS, { "" }, 1, { T_SUB_0 } }\\\n    }},\n    { LA_IS, { T_MUL_0 }, { "binary_operator", \\\n        { LA_IS, { "" }, 1, { T_MUL_0 } }\\\n    }},\n    { LA_IS, { T_DIV_0 }, { "binary_operator", \\\n        { LA_IS, { "" }, 1, { T_DIV_0 } }\\\n    }},\n    { LA_IS, { T_MOD_0 }, { "binary_operator", \\\n        { LA_IS, { "" }, 1, { T_MOD_0 } }\\\n    }},\n}
binary_action = binary_operator , expression;	binary_action = binary_operator , expression;	binary_action → binary_operator expression	binary_action(1: "8", " ", "=" , "!=" , "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", \\\n        { LA_IS, { "" }, 2, { "binary_operator", "expression" } }\\\n    }},\n}
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;	{ LA_IS, { "(" }, { "left_expression", \\\n        { LA_IS, { "" }, 1, { "group_expression" } }\\\n    }},\n    { LA_IS, { T_NOT_0 }, { "left_expression", \\\n        { LA_IS, { "" }, 1, { "unary_operation" } }\\\n    }},\n    { LA_IS, { T_IF_0 }, { "left_expression", \\\n        { LA_IS, { "" }, 1, { "cond_block" } }\\\n    }},\n    { LA_IS, { "unsigned_value_terminal" }, { "left_expression", \\\n        { LA_IS, { "" }, 1, { "value" } }\\\n    }},\n    { LA_IS, { T_ADD_0 , T_SUB_0 }, { "left_expression", \\\n        { LA_IS, { "unsigned_value_terminal" }, 1, { "value" } }\\\n    }},\n    { LA_NOT, { "unsigned_value_terminal" }, 1, { "unary_operation" } }* /\n},\n{ LA_IS, { "ident_terminal" }, { "left_expression", \\\n        { LA_IS, { "" }, 2, { "ident", "index_action__optional" } }\\\n    }},\n}
expression = left_expression , {binary_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, { "(" }, { "index_action__optional", \\\n        { LA_IS, { "" }, 1, { "index_action" } }\\\n    }},\n    { LA_NOT, { "(" }, { "index_action__optional", \\\n        { LA_IS, { "" }, 0, { "" } }\\\n    }},\n}
group_expression = ("", expression , "");	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__iteration;	{ LA_IS, { "(" , T_NOT_0 , T_ADD_0 , T_SUB_0 , "ident_terminal", "unsigned_value_terminal", T_IF_0 }, { "expression", \\\n        { LA_IS, { "" }, 2, { "left_expression", "binary_action__iteration" } }\\\n    }},\n}
expression_or_cond_block_with_optional_assign = expression , ["=", ident , [index_action]];	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 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", \\\n        { LA_IS, { "" }, 2, { "binary_action", "binary_action__iteration" } }\\\n    }},\n    { 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", \\\n        { LA_IS, { "" }, 0, { "" } }\\\n    }},\n}
assign_to_right = ">" , ident , index_action__optional;	group_expression = ("", expression , "");	group_expression → "(" expression ")"	group_expression(1: "") → "(" expression ")"	group_expression = tokenGROUPEXPRESSIONBEGIN >> expression >> tokenGROUPEXPRESSIONEND;	{ LA_IS, { "(" , { "group_expression", \\\n        { LA_IS, { "" }, 3, { "(" , "expression", ")" }\\\n    }},\n}
assign_to_right = ">" , ident , index_action__optional;	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);	{ 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", \\\n        { LA_IS, { "" }, 2, { "expression", "assign_to_right__optional" } }\\\n    }},\n}
assign_to_right__optional = assign_to_right   ε;	if_expression = expression;	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", \\\n        { LA_IS, { "" }, 3, { T_LRASSIGN_0 , "ident", "index_action__optional" } }\\\n    }},\n}
if_expression = expression;	body_for_true = block_statements_in_while_and_if_body;	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);	{ LA_IS, { "(" , T_NOT_0 , T_ADD_0 , T_SUB_0 , "ident_terminal", "unsigned_value_terminal", T_IF_0 }, { "if_expression", \\\n        { LA_IS, { "" }, 1, { "expression" } }\\\n    }},\n}
body_for_true = block_statements_in_while_and_if_body;	false_cond_block_without_else = "else" , "if" , if_expression , body_for_true;	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);	{ LA_IS, { T_BEGIN_BLOCK_0 }, { "body_for_true", \\\n        { LA_IS, { "" }, 1, { "block_statements_in_while_and_if_body" } }\\\n    }},\n
false_cond_block_without_else = "else" , "if" , if_expression , body_for_true;	body_for_false = "else" , block_statements_in_while_and_if_body;	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;	{ LA_IS, { T_ELSE_IF_0 }, { "false_cond_block_without_else", \\\n        { LA_IS, { "" }, 4, { T_ELSE_IF_0 , T_ELSE_IF_1 , "if_expression", "body_for_true" } }\\\n    }},\n}
body_for_false = "else" , block_statements_in_while_and_if_body;	cond_block = "if" , if_expression , body_for_true , false_cond_block_without_else_iteration , body_for_false__optional;	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;	{ LA_IS, { T_ELSE_BLOCK_0 }, { "body_for_false", \\\n        { LA_IS, { "" }, 2, { T_ELSE_BLOCK_0 , "block_statements" } }\\\n    }},\n}
cond_block = "if" , if_expression , body_for_true , {false_cond_block_without_else} , {body_for_false};	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_iteration >> body_for_false__optional;	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", \\\n        { LA_IS, { "" }, 5, { T_IF_0 , "if_expression", "body_for_true", "false_cond_block_without_else_iteration", "body_for_false__optional" } }\\\n    }},\n}



					{LA_IS, { T_INPUT_0 }, { "statement", \\\n(LA_IS, {"\"}, 1, {"input_rule"})\\\n}}\n{LA_IS, { T_OUTPUT_0 }, { "statement", \\\n(LA_IS, {"\"}, 1, {"output_rule"})\\\n}}\n{LA_IS, { T_SEMICOLON_0 }, { "statement", \\\n(LA_IS, {"\"}, 1, {"\"})\\\n}}\n
	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", "if") → statement statement__iteration statement__iteration(1: "_" _, !("!, !"!, !"0", !"1", !"2", !"3", !"4", !"5", !"6", !"7", !"8", !"9", !"add", !"sub", !"if", !"while", !"read", !"write", !"if") → ε		statement__iteration = statement >> statement__iteration   "";
block_statements = "(" , statement__iteration , ")" ;	block_statements = "(" , statement__iteration , ")" ;	block_statements → "(" statement__iteration ")"	block_statements(1: "(") → "(" statement__iteration ")"	block_statements = tokenBEGINBLOCK >> statement__iteration >> tokenENDBLOCK;	{LA_IS, { T_BEGIN_BLOCK_0 }, { "block_statements", \\\n(LA_IS, {"\"}, 3, { T_BEGIN_BLOCK_0, "statement__iteration", T_END_BLOCK_0 })\\\n}}\n
program_rule = "program" , program_name , ";" , "begin" , "var" , declaration__optional , ";" , statement__iteration , "end" ;	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", "!", "!", "!", "!", "2", "!", "3", "!", "4", "!", "5", "!", "6", "!", "7", "!", "8", "!", "9", "if") → ε		expression__optional = expression   ""; (LA_IS, { "!", T_NOT_0, T_ADD_0, T_SUB_0, T_IF_0, "ident_terminal", "unsigned_value_terminal", \\\n(LA_IS, {"\"}, 1, { "expression" })\\\n})\n{LA_NOT, { "!", T_NOT_0, T_ADD_0, T_SUB_0, T_IF_0, "ident_terminal", "unsigned_value_terminal", T_ADD_0, \\\n(LA_IS, {"\"}, 0, { "!" })\\\n})\n
sign = sign_plus   sign_minus;	declaration__optional = declaration   "";	declaration__optional → declaration declaration__optional → ε	declaration__optional(1: "int32") → declaration declaration__optional(1: !"int32") → ε		{LA_IS, { T_NAME_0 }, { "program_rule", \\\n(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 })\\\n}}\n
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__optional >> unsigned_value >> BOUNDARIES;	{LA_IS, { "unsigned_value_terminal", T_ADD_0, T_SUB_0, { "value" }, \\\n(LA_IS, {"\"}, 2, { "sign_optional", "unsigned_value" })\\\n})\n
sign_plus = "add";	sign__optional = sign   ε;	sign__optional → sign sign__optional → ε	sign__optional(1: "add", "sub") → sign sign__optional(1: !"add", !"sub") → ε		{LA_IS, { T_ADD_0, T_SUB_0, { "sign_optional", \\\n(LA_IS, {"\"}, 1, { "sign" })\\\n}}\n{LA_NOT, { T_ADD_0, T_SUB_0, { "sign_optional", \\\n(LA_IS, {"\"}, 0, { "!" })\\\n}}\n
sign_minus = "sub";	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;	{LA_IS, { T_ADD_0 }, { "sign", \\\n(LA_IS, {"\"}, 1, { "sign_plus" })\\\n})\n{LA_IS, { T_SUB_0 }, { "sign", \\\n(LA_IS, {"\"}, 1, { "sign_minus" })\\\n}}\n
unsigned_value = non_zero_digit , {digit}   "0";	sign_plus = "add";	sign_plus → "add"	sign_plus(1: "add") → "add"	sign_plus = SAME_RULE(tokenPLUS);	{LA_IS, { T_ADD_0 }, { "sign_plus", \\\n(LA_IS, {"\"}, 1, { T_ADD_0 })\\\n})\n
digit = "0"   non_zero_digit;	sign_minus = "sub";	sign_minus → "sub"	sign_minus(1: "sub") → "sub"	sign_minus = SAME_RULE(tokenMINUS);	{LA_IS, { T_SUB_0 }, { "sign_minus", \\\n(LA_IS, {"\"}, 1, { T_SUB_0 })\\\n})\n
non_zero_digit = "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9";	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__iteration   digit_0) >> BOUNDARIES;	/* unsigned_value_token represents unsigned_value in lexical analyzer */\n{LA_IS, { "unsigned_value_terminal" }, { "unsigned_value" }, \\\n(LA_IS, {"\"}, 1, { "unsigned_value_terminal" })\\\n})\n
non_zero_digit = "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9";	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   "";	\
ident = letter_in_upper_case , letter_in_upper_case , letter_in_upper_case;	digit → "0"	digit → "0"	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;	\
ident = letter_in_upper_case , 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 = {\n(tokenINTEGER16   tokenCOMMA   tokenNOT   tokenAND   tokenOR   tokenEQUAL   tokenNOTEQUAL   tokenLESS   tokenGREATER	/* ident_token represents ident in lexical analyzer */\n{LA_IS, { "ident_terminal" }, { "ident", \\\n(LA_IS, {"\"}, 1, { "ident_terminal" })\\\n}}\n

				<pre> tokenPLUS    tokenMINUS    tokenMUL    tokenDIV    tokenMOD    tokenGROUPEXPRESSIONBEGIN    tokenGROUPEXPRESSIONEND    tokenLASSIGN    tokenELSE    tokenIF    tokenWHILE    tokenCONTINUE    tokenBREAK    tokenEXIT    tokenGET    tokenPUT    tokenNAME    tokenBODY    tokenDATA    tokenBEGIN    tokenEND    tokenBEGINBLOCK    tokenENDBLOCK    tokenLEFTSQUAREBRACKETS    tokenRIGHTSQUAREBRACKETS    tokenSEMICOLON  ) &gt;&gt;  letter_in_upper_case &gt;&gt; letter_in_upper_case &gt;&gt;  letter_in_upper_case &gt;&gt; STRICT_BOUNDARIES; </pre>	<pre> tokenPLUS    tokenMINUS    tokenMUL    tokenDIV    tokenMOD    tokenGROUPEXPRESSIONBEGIN    tokenGROUPEXPRESSIONEND    tokenLASSIGN    tokenELSE    tokenIF    tokenWHILE    tokenCONTINUE    tokenBREAK    tokenEXIT    tokenGET    tokenPUT    tokenNAME    tokenBODY    tokenDATA    tokenBEGIN    tokenEND    tokenBEGINBLOCK    tokenENDBLOCK    tokenLEFTSQUAREBRACKETS    tokenRIGHTSQUAREBRACKETS    tokenSEMICOLON  ) &gt;&gt;  letter_in_upper_case &gt;&gt; letter_in_upper_case &gt;&gt;  letter_in_upper_case &gt;&gt; STRICT_BOUNDARIES; </pre>			
				<pre> 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"; </pre>	<pre> 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" </pre>	<pre> 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" </pre>	<pre> 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"; </pre>	<pre> letter_in_lower_case = 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 = 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; </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> 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" </pre>	<pre> 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" </pre>	<pre> 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"; </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> 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> STRICK_BOUNDARIES = (BOUNDARY &gt;&gt; *(BOUNDARY))    (!qi::alpha   qi::char("_")); </pre>	<pre> STRICK_BOUNDARIES = (BOUNDARY &gt;&gt; *(BOUNDARY))    (!qi::alpha   qi::char("_")); </pre>	
					<pre> BOUNDARIES = (BOUNDARY &gt;&gt; *(BOUNDARY))    NO_BOUNDARY; </pre>	<pre> BOUNDARIES = (BOUNDARY &gt;&gt; *(BOUNDARY))    NO_BOUNDARY; </pre>		
					<pre> BOUNDARY = BOUNDARY__SPACE   BOUNDARY__TAB    BOUNDARY__VERTICAL_TAB   BOUNDARY__FORM_FEED    BOUNDARY__CARRIAGE_RETURN   BOUNDARY__LINE_FEED    BOUNDARY__NULL; </pre>	<pre> BOUNDARY = BOUNDARY__SPACE   BOUNDARY__TAB    BOUNDARY__VERTICAL_TAB   BOUNDARY__FORM_FEED    BOUNDARY__CARRIAGE_RETURN    BOUNDARY__LINE_FEED   BOUNDARY__NULL; </pre>		
					<pre> BOUNDARY__SPACE = " "; </pre>	<pre> BOUNDARY__SPACE = " "; </pre>		
					<pre> BOUNDARY__TAB = "\t"; </pre>	<pre> BOUNDARY__TAB = "\t"; </pre>		
					<pre> BOUNDARY__VERTICAL_TAB = "\v"; </pre>	<pre> BOUNDARY__VERTICAL_TAB = "\v"; </pre>		
					<pre> BOUNDARY__FORM_FEED = "\f"; </pre>	<pre> BOUNDARY__FORM_FEED = "\f"; </pre>		
					<pre> BOUNDARY__CARRIAGE_RETURN = "\r"; </pre>	<pre> BOUNDARY__CARRIAGE_RETURN = "\r"; </pre>		
					<pre> BOUNDARY__LINE_FEED = "\n"; </pre>	<pre> BOUNDARY__LINE_FEED = "\n"; </pre>		
					<pre> BOUNDARY__NULL = "\0"; </pre>	<pre> BOUNDARY__NULL = "\0"; </pre>		
					<pre> NO_BOUNDARY = ""; </pre>	<pre> NO_BOUNDARY = ""; </pre>		
					<pre> #define WHITESPACES \ </pre>	<pre> #define WHITESPACES \ </pre>		
					<pre> STRICK_BOUNDARIES, \ </pre>	<pre> STRICK_BOUNDARIES, \ </pre>		
					<pre> BOUNDARIES, \ </pre>	<pre> BOUNDARIES, \ </pre>		
					<pre> BOUNDARY, \ </pre>	<pre> BOUNDARY, \ </pre>		
					<pre> BOUNDARY__SPACE, \ </pre>	<pre> BOUNDARY__SPACE, \ </pre>		
					<pre> BOUNDARY__TAB, \ </pre>	<pre> BOUNDARY__TAB, \ </pre>		

				<pre>BOUNDARY_VERTICAL_TAB, \ BOUNDARY_FORM_FEED, \ BOUNDARY_CARRIAGE_RETURN, \ BOUNDARY_LINE_FEED, \ BOUNDARY_NULL, \ NO_BOUNDARY</pre>	<pre>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, "program_name", \     T_SEMICOLON_0, T_BODY_0, T_DATA_0, \     "declaration_optional", T_SEMICOLON_0 } } \ } } } \ \ \ } \ \ "program_rule"</pre>	