

<u>РБНФ</u>	<u>Код для перевірки РБНФ</u>
program_name = ident;	program_name = SAME_RULE(ident);
value_type = "int32_t";	value_type = SAME_RULE(tokenINTEGER16);
declaration_element = ident , [ "[" , unsigned_value , "]" ];	declaration_element = ident >> -(tokenLEFTSQUAREBRACKETS >> unsigned_value >> tokenRIGHTSQUAREBRACKETS);
other_declaration_ident = " , " , declaration_element;	other_declaration_ident = tokenCOMMA >> declaration_element;
declaration = value_type , declaration_element , {other_declaration_ident};	declaration = value_type >> declaration_element >> *other_declaration_ident;
index_action = "[" , expression , "]" ;	index_action = tokenLEFTSQUAREBRACKETS >> expression >> tokenRIGHTSQUAREBRACKETS;
unary_operator = "not";	unary_operator = SAME_RULE(tokenNOT);
unary_operation = unary_operator , expression;	unary_operation = unary_operator >> expression;
binary_operator = "and"   " "   "eg"   "ne"   "<<"   ">>"   "add"   "sub"   "*"   "/"   "%";	binary_operator = tokenAND   tokenOR   tokenEQUAL   tokenNOTEQUAL   tokenLESS   tokenGREATER   tokenPLUS   tokenMINUS   tokenMUL   tokenDIV   tokenMOD;
binary_action = binary_operator , expression;	binary_action = binary_operator >> expression;
left_expression = group_expression   unary_operation   ident , [index_action]   value   cond_block_with_optionally_return_value;	left_expression = group_expression   unary_operation   ident >> - index_action   value   cond_block_with_optionally_return_value;
expression = left_expression , {binary_action};	expression = left_expression >> *binary_action;
group_expression = "(" , expression , ")";	group_expression = tokenGROUPEXPRESSIONBEGIN >> expression >> tokenGROUPEXPRESSIONEND;
bind_left_to_right = expression , " :> " , ident , [index_action];	bind_left_to_right = expression >> tokenLRBIND >> ident >> -index_action;
if_expression = expression;	if_expression = SAME_RULE(expression);
body_for_true__with_optionally_return_value = block_statements__with_optionally_return_value;	body_for_true__with_optionally_return_value = SAME_RULE(block_statements__with_optionally_return_value);
false_cond_block_without_else__with_optionally_return_value = "else" , "if" , if_expression , body_for_true__with_optionally_return_value;	false_cond_block_without_else__with_optionally_return_value = tokenELSE >> tokenIF >> if_expression >> body_for_true__with_optionally_return_value;
body_for_false__with_optionally_return_value = "else" , block_statements__with_optionally_return_value;	body_for_false__with_optionally_return_value = tokenELSE >> block_statements__with_optionally_return_value;
cond_block__with_optionally_return_value = "if" , if_expression , body_for_true__with_optionally_return_value , {false_cond_block_without_else__with_optionally_return_value} , [body_for_false__with_optionally_return_value];	cond_block__with_optionally_return_value = tokenIF >> if_expression >> body_for_true__with_optionally_return_value >> *false_cond_block_without_else__with_optionally_return_value >> - body_for_false__with_optionally_return_value;

cond_block__with_optionally_return_value_and_optionally_bind = cond_block__with_optionally_return_value , [tokenLRBIND , ident , [index_action]];	cond_block__with_optionally_return_value_and_optionally_bind = cond_block__with_optionally_return_value >> -(tokenLRBIND >> ident >> - index_action);
cycle_begin_expression = expression;	cycle_begin_expression = SAME_RULE(expression);
cycle_end_expression = expression;	cycle_end_expression = SAME_RULE(expression);
cycle_counter = ident;	cycle_counter = SAME_RULE(ident);
cycle_counter_lr_init = cycle_begin_expression , ">" , cycle_counter;	cycle_counter_lr_init = cycle_begin_expression >> tokenLRBIND >> cycle_counter;
cycle_counter_init = cycle_counter_lr_init;	cycle_counter_init = SAME_RULE(cycle_counter_lr_init);
cycle_counter_last_value = cycle_end_expression;	cycle_counter_last_value = SAME_RULE(cycle_end_expression);
cycle_body = "do" , ({statement}   block_statements);	cycle_body = tokenDO >> (statement   block_statements);
forto_cycle = "for" , cycle_counter_init , "downto" , cycle_counter_last_value , cycle_body;	forto_cycle = tokenFOR >> cycle_counter_init >> tokenDOWNTO >> cycle_counter_last_value >> cycle_body;
input = "scan" , ( ident , [index_action]   "(" , ident , [index_action] , ")" );	input = tokenGET >> (ident >> -index_action   tokenGROUPEXPRESSIONBEGIN >> ident >> -index_action >> tokenGROUPEXPRESSIONEND);
output = "print" , expression;	output = tokenPUT >> expression;
statement = bind_left_to_right   cond_block__with_optionally_return_value_and_optionally_bind   forto_cycle   input   output   ";" ;	statement = bind_left_to_right   cond_block__with_optionally_return_value_and_optionally_bind   forto_cycle   input   output   tokenSEMICOLON;
block_statements = "{" , {statement} , "}";	block_statements = tokenBEGINBLOCK >> *statement >> tokenENDBLOCK;
block_statements__with_optionally_return_value = "{" , {statement} , [expression] , "}";	block_statements__with_optionally_return_value = tokenBEGINBLOCK >> *statement >> -expression >> tokenENDBLOCK;
program = "mainprogram" , "DATA" , [declaration] , ";" , "start" , {statement} , "end";	program = BOUNDARIES >> tokenNAME >> tokenDATA >> (-declaration) >> tokenSEMICOLON >> tokenBODY >> *statement >> tokenEND;
digit = "0"   "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9";	digit = digit_0   digit_1   digit_2   digit_3   digit_4   digit_5   digit_6   digit_7   digit_8   digit_9;
non_zero_digit = "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9";	non_zero_digit = digit_1   digit_2   digit_3   digit_4   digit_5   digit_6   digit_7   digit_8   digit_9;
unsigned_value = (non_zero_digit , {digit})   "0";	unsigned_value = ((non_zero_digit >> *digit)   digit_0) >> BOUNDARIES;
value = [sign] , unsigned_value;	value = -sign >> unsigned_value >> BOUNDARIES;
letter_in_lower_case = "a"   "b"   "c"   "d"   "e"   "f"   "g"   "h"   "i"   "j"   "k"   "l"   "m"   "n"   "o"   "p"   "q"   "r"   "s"   "t"   "u"   "v"   "w"   "x"   "y"   "z";	letter_in_lower_case = a   b   c   d   e   f   g   h   i   j   k   l   m   n   o   p   q   r   s   t   u   v   w   x   y   z;

letter_in_upper_case = "A"   "B"   "C"   "D"   "E"   "F"   "G"   "H"   "I"   "J"   "K"   "L"   "M"   "N"   "O"   "P"   "Q"   "R"   "S"   "T"   "U"   "V"   "W"   "X"   "Y"   "Z";	letter_in_upper_case = A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P   Q   R   S   T   U   V   W   X   Y   Z;
ident = letter_in_lower_case, digit, digit	ident = letter_in_lower_case >> digit >> digit >> STRICT_BOUNDARIES;
sign = "+"   "-";	sign = sign_plus   sign_minus;
	sign_plus = SAME_RULE(tokenPLUS);
	sign_minus = SAME_RULE(tokenMINUS);
	digit_0 = '0';
	digit_1 = '1';
	digit_2 = '2';
	digit_3 = '3';
	digit_4 = '4';
	digit_5 = '5';
	digit_6 = '6';
	digit_7 = '7';
	digit_8 = '8';
	digit_9 = '9';
	tokenINTEGER16 = "int32_t" >> STRICT_BOUNDARIES;
	tokenCOMMA = "," >> BOUNDARIES;
	tokenNOT = "not" >> STRICT_BOUNDARIES;
	tokenAND = "and" >> STRICT_BOUNDARIES;
	tokenOR = " " >> BOUNDARIES;
	tokenEQUAL = "eg" >> STRICT_BOUNDARIES;
	tokenNOTEQUAL = "ne" >> STRICT_BOUNDARIES;
	tokenLESS = "<<" >> BOUNDARIES;
	tokenGREATER = ">>" >> BOUNDARIES;
	tokenPLUS = "add" >> STRICT_BOUNDARIES;
	tokenMINUS = "sub" >> STRICT_BOUNDARIES;
	tokenMUL = "*" >> BOUNDARIES;
	tokenDIV = "/" >> BOUNDARIES;
	tokenMOD = "%" >> BOUNDARIES;

	tokenGROUPEXPRESSIONBEGIN = "(" >> BOUNDARIES;
	tokenGROUPEXPRESSIONEND = ")" >> BOUNDARIES;
	tokenLRBIND = ":" >> BOUNDARIES;
	tokenELSE = "else" >> STRICT_BOUNDARIES;
	tokenIF = "if" >> STRICT_BOUNDARIES;
	tokenDO = "do" >> STRICT_BOUNDARIES;
	tokenFOR = "for" >> STRICT_BOUNDARIES;
	tokenDOWNT0 = "downto" >> STRICT_BOUNDARIES;
	tokenGET = "scan" >> STRICT_BOUNDARIES;
	tokenPUT = "print" >> STRICT_BOUNDARIES;
	tokenNAME = "mainprogram" >> STRICT_BOUNDARIES;
	tokenBODY = "start" >> STRICT_BOUNDARIES;
	tokenDATA = "var" >> STRICT_BOUNDARIES;
	tokenEND = "end" >> STRICT_BOUNDARIES;
	tokenBEGINBLOCK = "{" >> BOUNDARIES;
	tokenENDBLOCK = "}" >> BOUNDARIES;
	tokenLEFTSQUAREBRACKETS = "[" >> BOUNDARIES;
	tokenRIGHTSQUAREBRACKETS = "]" >> BOUNDARIES;
	tokenSEMICOLON = ";" >> BOUNDARIES;
	STRICT_BOUNDARIES = (BOUNDARY >> *(BOUNDARY))   (!{qi::alpha   qi::char_("_")});
	BOUNDARIES = (BOUNDARY >> *(BOUNDARY)   NO_BOUNDARY);
	BOUNDARY = BOUNDARY_SPACE   BOUNDARY_TAB   BOUNDARY_CARRIAGE_RETURN   BOUNDARY_LINE_FEED   BOUNDARY_NULL;
	BOUNDARY_SPACE = " ";
	BOUNDARY_TAB = "\t";
	BOUNDARY_CARRIAGE_RETURN = "\r";
	BOUNDARY_LINE_FEED = "\n";
	BOUNDARY_NULL = "\0";
	NO_BOUNDARY = "";

	tokenUNDERSCORE = "_";
	A = "A";
	B = "B";
	C = "C";
	D = "D";
	E = "E";
	F = "F";
	G = "G";
	H = "H";
	I = "I";
	J = "J";
	K = "K";
	L = "L";
	M = "M";
	N = "N";
	O = "O";
	P = "P";
	Q = "Q";
	R = "R";
	S = "S";
	T = "T";
	U = "U";
	V = "V";
	W = "W";
	X = "X";
	Y = "Y";
	Z = "Z";
	a = "a";
	b = "b";
	c = "c";
	d = "d";

	e = "e";
	f = "f";
	g = "g";
	h = "h";
	i = "i";
	j = "j";
	k = "k";
	l = "l";
	m = "m";
	n = "n";
	o = "o";
	p = "p";
	q = "q";
	r = "r";
	s = "s";
	t = "t";
	u = "u";
	v = "v";
	w = "w";
	x = "x";
	y = "y";
	z = "z";