DELPHI - BNF

```
start= program  | unit | library | package .

  identifier_list= ID_NAME { ',' ID_NAME } .
  unit_qualified_identifier= ID_NAME { '.' ID_NAME } .

  type_name= TYPE_NAME | STRING | FILE .
  unit_qualified_type_name= type_name [ '.' type_name ] .

  function_result_type= type_name .

  constant_expression= F .
  string_expression= ( STRING_NAME | STRING_LITTERAL )
     { '+' ( STRING_NAME | STRING_LITTERAL ) } .

  variable_access= ( ACCESS_NAME | STRING ) { end_access_ } .
    end_access_= { array_access_ | record_access_ | '^' | function_parameters_ } .
      array_access_= '[' constant_expression { ',' constant_expression } ']' .
      record_access_= '.' variable_access .
      function_parameters_= '(' [ constant_expression { ',' constant_expression } ] ')' .

  set_factor= '[' [ set_element { ',' set_element } ] ']' .
    set_element= constant_expression [ '..' constant_expression ] .

  constant_expression= simple_expression__ [ ('=' | '<>' | '<' | '<=' | '>' | '>=' | IN  )
      simple_expression__ ] .
    simple_expression__= [ '+' | '-' ] term__ { ('+' | '-' | OR | XOR ) term__  } .
      term__= factor__ { ('*' | '/' | DIV | MOD | AND | SHR | SHL ) factor__ } .
        factor__= NUMBER | STRING_LITTERAL | NIL
            | variable_access
            | NOT factor__ | '@' factor__ | set_factor
            | '^' NAME
            | '(' constant_expression ')'.

  typed_constant= simple_expression_ [ ('=' | '<>' | '<' | '<=' | '>' | '>=' | IN  )
      simple_expression_ ] .
    simple_expression_= [ '+' | '-' ] term_ { ('+' | '-' | OR | XOR ) term_ } .
      term_= factor_ { ('*' | '/' | DIV | MOD | AND | SHR | SHL ) factor_ } .
        factor_= NUMBER | STRING_LITTERAL | NIL
            // -- id or field "(f1: v1; f2: v2)"
            | variable_access [ ':' typed_constant
                { ';' variable_access ':' typed_constant } ]
            | NOT factor_ | '@' factor_
```

```
          | '^' NAME
          | '(' [ typed_constant_]  ')'
          | set_factor .
        // -- array "(1, 2, 3)" or "fn(p1, p2")
        typed_constant_= typed_constant { ',' typed_constant } .
formal_parameters= '(' formal_parameter { ';' formal_parameter } ')' .
  formal_parameter= [ parameter | var_parameter
      | const_parameter | out_parameter | in_parameter ] .
    parameter_name_list= PARAMETER_NAME { ',' PARAMETER_NAME } .
    array_or_name_type= ARRAY OF ( CONST | unit_qualified_type_name )
        | unit_qualified_type_name .
    parameter= parameter_name_list ':' array_or_name_type
        ['=' constant_expression ]  .
    var_parameter= VAR parameter_name_list [ ':' array_or_name_type ] .
    const_parameter= CONST parameter_name_list
        [ ':' array_or_name_type ['=' constant_expression ] ] .
    out_parameter= OUT parameter_name_list [ ':' array_or_name_type ] .
    in_parameter= IN parameter .

dos_directives= NEAR | FAR | EXPORT | ASSEMBLER .
calling_directives= CDECL | PASCAL | REGISTER | SAFECALL | STDCALL .
overload_directive= OVERLOAD .
method_directives= ABSTRACT | VIRTUAL | DYNAMIC
      | OVERRIDE | REINTRODUCE | MESSAGE constant_expression .

const_type_var_declarations= constant_definitions | resource_defintions
    | type_definitions | variable_declarations .

  type= keyed_types | type_0 .
    // -- called by i_type
    enumeration_type= '(' identifier_list ')' .
    expression_t= simple_expression_t
        [ ( ('=' | '<>' | '<' | '<=' | '>' | '>=' | IN  ) simple_expression_t
          | '..' end_range_type ) ] .
      simple_expression_t= [ '+' | '-' ] term_t { ('+' | '-' | OR | XOR ) term_t } .
        term_t= factor_t { ('*' | '/' | DIV | MOD | AND | SHR | SHL ) factor_t } .
          factor_t= NUMBER | STRING_LITTERAL | NIL
              | variable_access
              | NOT factor_t | '@' factor_t
              | '^' NAME
              | '(' expression_t ')'
              | set_factor .
      end_range_type= simple_expression_t .
    type_0= ( NUMBER | STRING_LITTERAL | NIL | NOT | '+' | '-' | '@' | '(' | '[' | NAME )
     $i_type .
  keyed_types= string_type | structured_type | pointer_type | procedural_type .
   // -- handle STRING as array[index_type]
   string_type= STRING [ '[' constant_expression ']' ] .
   structured_type= [ PACKED ] ( array_type | record_type | set_type | file_type  ) .
     array_type= ARRAY [ '[' index_type { ',' index_type } ']' ] OF type .
       index_type= constant_expression [ '..' constant_expression ] .
     record_type= RECORD field_list END .
       field_list= { common_field ';' } [ variant_fields ] .
         common_field= identifier_list ':' type .
         variant_fields= CASE tag OF cases { cases } .
           tag= VARIANT_TAG_NAME [ ':' unit_qualified_type_name ] .
           cases= constant_expression { ',' constant_expression }
               ':' one_case .
             one_case= '(' [ common_field { ';' [ ( common_field | variant_fields ) ] }
                           | variant_fields ]
                        ')' [ ';' ] .
     set_type= SET OF type .
     file_type= FILE [ OF type ] .
   pointer_type= '^' POINTED_NAME .
   procedural_type= ( PROCEDURE [ formal_parameters ]
           | FUNCTION [ formal_parameters ] ':' function_result_type )
       $<dir( [ OF OBJECT ] | i_procedural_type_directives ) $>dir  .
     procedural_type_directives= calling_directives  .
     i_procedural_type_directives=  ( ';'
         | CDECL | PASCAL | REGISTER | SAFECALL | STDCALL ) $i_directives .

constant_definitions= CONST constant_definition { constant_definition  }  .
  constant_definition= CONST_NAME [ ':' type ] '=' typed_constant ';' .

resource_defintions= RESOURCESTRING resource_definition { resource_definition } .
  resource_definition= RESOURCE_NAME '=' string_expression ';' .

type_definitions= TYPE type_definition  { type_definition  } .
  type_definition= TYPE_NAME '=' [ TYPE ] ( class_type | interface_type | type ) ';' .

   // -- used in INTERFACE also
   property= PROPERTY $>priv PROPERTY_NAME [ property_type ] property_specifiers .
     property_type= [ property_indexes ] ':' unit_qualified_type_name .
       property_indexes= '[' property_index { ';' property_index } ']' .
         property_index= [ CONST ] INDEX_NAME { ',' INDEX_NAME }
             ':' unit_qualified_type_name .
     property_specifiers= $<prop [ INDEX constant_expression ] $>prop
         // -- "READ FTabSize.Y"
```

```
      $<prop [ READ variable_access | READONLY ]
        [ WRITE WRITE_NAME | WRITEONLY ] $>prop
      // -- some params called "dispid"
      $<prop [ DISPID constant_expression ] [ ';' ] $>prop
      $<prop { storage_specifier [';' ] } $>prop
      [ IMPLEMENTS unit_qualified_identifier { ',' unit_qualified_identifier } ';' ] .
    storage_specifier= storage_stored | storage_default  | storage_no_default  .
     storage_stored= STORED [ constant_expression ]  .
     storage_default= DEFAULT [ constant_expression ] .
     storage_no_default= NODEFAULT .

// -- the ; is in the type_definitions
class_type= CLASS [ class_reference | class_definition ] .
  class_reference= OF unit_qualified_type_name .
  // -- class_definition : can be foward with inheritance
  class_definition= [ inheritance ] [ class_body ] .
    inheritance= '(' unit_qualified_type_name { ',' unit_qualified_type_name } ')' .
    class_body= fields_and_procs_section { fields_and_procs_section } END .
      fields_and_procs_section= $<priv protection fields_and_procs $>priv .
        protection= [ PRIVATE  | PROTECTED | PUBLIC | PUBLISHED ]  .
        fields_and_procs= { class_field } { class_methods | property $<priv } .
          class_field= identifier_list $>priv ':' type ';' $<priv .
          class_methods= constructor | destructor |
            [ CLASS ] ( class_procedure | class_function ) .
          method_directives_= $<dir
              { (method_directives | overload_directive | calling_directives)
              [ ';'] } $>dir .
          // -- if interfaces : "FUNCTION i_xxx.yyy = zzz;"
          rename_method= '.' NAME '=' NAME ';' .
          constructor= CONSTRUCTOR $>priv PR_NAME [ formal_parameters ] ';'
            method_directives_ $<priv .
          destructor= DESTRUCTOR $>priv PR_NAME [ formal_parameters ] ';'
            method_directives_ $<priv .
          class_procedure= PROCEDURE $>priv PR_NAME
            ( rename_method | [ formal_parameters ] ';'
              method_directives_ ) $<priv .
          class_function= FUNCTION $>priv FN_NAME
            ( rename_method | [ formal_parameters ] ':' function_result_type ';'
              method_directives_ ) $<priv .

interface_type= ( INTERFACE | DISPINTERFACE ) [ interface_definition ] .
  interface_definition= [ interface_heritage] [interface_g_u_i_d ]
      interface_member_list END .
    interface_heritage= '(' identifier_list ')' .
```

```
          interface_g_u_i_d= '[' string_expression ']' .
          interface_member_list= { class_procedure_ | class_function_ | property } .
            interface_directives_= $<dir
                { (method_directives | overload_directive | calling_directives | dispid )
                [ ';'] } $>dir .
              dispid= DISPID constant_expression .
            // -- redefinition "PROCEDURE x.y= z;" (axctrls)
            class_procedure_= ( PROCEDURE | CONSTRUCTOR | DESTRUCTOR )
               PR_NAME [ formal_parameters ] ';' interface_directives_ .
            class_function_= FUNCTION FN_NAME  [ formal_parameters ] ':' function_result_type
';'
               interface_directives_ .

      variable_declarations= (THREADVAR | VAR) variable_declaration { variable_declaration } .
        // -- has separated in 2 because of initialization
        // -- absolute can be after a list, but not with initialization
        variable_declaration= ID_NAME
           ( ':' type  [ '=' typed_constant | absolute ] ';'
            | { ',' ID_NAME }  ':' type  [ absolute ] ';' ) .
          absolute= ABSOLUTE OTHER_VAR_NAME .

  // -- code

  expression= simple_expression [ ('=' | '<>' | '<' | '<=' | '>' | '>=' | IN | IS )
      simple_expression ] .
    simple_expression= [ '+' | '-' ] term { ('+' | '-' | OR | XOR ) term } .
      term= factor { ('*' | '/' | DIV | MOD | AND | SHR | SHL ) factor } .
        // -- called by $i_access_or_expression
        // -- can be empty if fn call "fn()"
        parenthized_expression= '(' [ expression { ',' expression } ] ')' .
        factor= NUMBER | STRING_LITTERAL | NIL
            | NOT factor | '@' factor | INHERITED [ factor ]
            | '^' NAME
            | set_factor
            // -- x= (Sender AS tButton).Caption
            // -- the AS is only for the level 0
            | ( NAME | STRING ) { parenthized_expression | end_access }
            | parenthized_expression { end_access } .
          end_access= { array_access | record_access | '^' | as_access  } .
            array_access= '[' expression { ',' expression } ']' .
            record_access= '.' expression .
            as_access= AS NAME .

  // -- instructions
```

```
asm= ASM { asm_statement } END .
  // -- for pasting in i_asm
  asm_statement_= { NAME | NUMBER | STRING_LITTERAL
      | '[' | ']' | '.'  | ','
      | ':'
      | '+' | '-' | '*' | '/'
      | NOT  | AND | OR | XOR | SHR | SHL | DIV } .
  label_=  '@' [ '@'] ( ALL_NAME | NUMBER ) .

  asm_statement= ( NAME | NUMBER | STRING_LITTERAL
      | '[' | ']' | '.'  | ','
      | '@'
      | ':'
      | '+' | '-' | '*' | '/'
      | NOT  | AND | OR | XOR | SHR | SHL | DIV ) $i_asm .

composed_instruction= F .

// -- allows empty ";" instruction
instruction_list= [ instruction ] { ';' [ instruction ] } .
  instruction= { assignment_or_call | structured_instruction } .

    // -- this covers "x[3].z:= u;" or "my_proc(3+ zz)";
    // -- acces or (pchar+ 1)^ := ...
    assignment_or_call= expression [ end_assignment ] .
      // -- "(Sender As tButton).Caption:= xxx"
      end_assignment= ':=' expression .

    structured_instruction= composed_instruction | test | repetition | with
        | try | inherited_call | raise_statement | asm .
      test= if | case .
        if= IF expression THEN instruction [ ELSE instruction ] .
        // -- D5: ';' after last instr or before ELSE optional !
        case= CASE expression OF case_element
            { ';' [ ELSE $NOREAD | END $NOREAD | case_element ] }
              [ ELSE instruction_list ] END .
          case_element= case_label ':' instruction .
            // -- a general constant constant_expression, but no set [],
            // --   unless in a function call
            case_label= constant_expression
              { ( ',' constant_expression | '..' constant_expression ) } .
      repetition= while | repeat | for .
        while= WHILE expression DO instruction .
        repeat= REPEAT instruction_list UNTIL expression .
        for= FOR unit_qualified_identifier ':=' expression [ TO | DOWNTO ]
          expression DO instruction .
      // -- "with xxx AS"
      with= WITH expression { ',' expression } DO instruction .
      try= TRY instruction_list
        ( EXCEPT except_block | FINALLY instruction_list ) END .
        except_block= on [ ELSE instruction_list ] | instruction_list .
        // -- can have "ON ezero DO ELSE xxx ;" or "ON xxx DO ;"
        on= handle_instruction { ';' [ handle_instruction ] } .
          exception_identifier= unit_qualified_identifier [ ':' unit_qualified_identifier ] .
          handle_instruction= ON exception_identifier DO [ instruction ';' ] .

      // -- "Inherited Items[Index]:= "
      inherited_call= INHERITED [ instruction ] .
      // inline_statement= INLINE '(' INTEGERCONST {'/' INTEGERCONST } ')' .
      raise_statement= $<at RAISE [ variable_access ] [ AT constant_expression ] $>at .

  composed_instruction= BEGIN instruction_list END .

  // -- bloc
  // -- VIRTUAL etc only in CLASS

  routine_header= class_methods_header | constructor_header | destructor_header
      | procedure_header | function_header .
    // -- methods have no directives in implementation
    class_methods_header= CLASS (class_procedure_method | class_function_method ) .
      class_procedure_method= PROCEDURE CLASS_NAME '.' PR_NAME [
formal_parameters ] ';' .
      // -- repeating the result is optional
      class_function_method= FUNCTION CLASS_NAME [ '.' FN_NAME ]
        [ formal_parameters ] [ ':' function_result_type ] ';' .
    constructor_header= CONSTRUCTOR CLASS_NAME '.' PR_NAME [ formal_parameters ] ';'
.
    destructor_header= DESTRUCTOR CLASS_NAME '.' PR_NAME [ formal_parameters ] ';' .
    // -- always ; before directives (for procedural cdecl is without ? )
    code_procedure_directives= $<dir { (dos_directives
      | calling_directives | overload_directive)
      [ ';'] } $>dir .
    procedure_header= PROCEDURE
      CLASS_OR_PR_NAME [ '.' PR_NAME ] [ formal_parameters ] ';'
        code_procedure_directives .
    // -- for the functions, STDCALL does not require ; "fn xxx: yyy STDCALL;"
    function_header= FUNCTION CLASS_OR_FN_NAME [ '.' FN_NAME ]
```

```
      [ formal_parameters ] [ ':' function_result_type ]
      [ ';' ] code_procedure_directives [ ';' ] .

bloc= F .
main_declarations= const_type_var_declarations | procedure_declarations_and_body .
  procedure_declarations_and_body= { procedure_declaration } .
    procedure_declaration= routine_header
        $<dir ( FORWARD  $>dir | EXTERNAL $>dir end_external | $>dir bloc ) ';' .
      // "procedure xxx; external;"
      // "procedure xxx; external 'xxx';"
      // "procedure xxx; external xxx;"
      // "procedure xxx; external xxx NAME 'MessageBoxA';"
      // "procedure xxx; external xxx 'MessageBoxA' INDEX 31;"
      end_external= [ constant_expression $<index [ index ] $>index ] '.' .
        index= INDEX constant_expression .
bloc= { main_declarations } ( composed_instruction | asm ) .

main_uses= USES uses_in { ',' uses_in } ';' .
  uses_in= UNIT_NAME [ IN constant_expression ] .

// -- program / units / library / packages

program= PROGRAM NAME ';' [ main_uses ] bloc '.' .

unit= UNIT UNIT_NAME ';' unit_interface unit_implementation unit_end '.' .
  uses= USES identifier_list ';' .
  unit_interface= INTERFACE [ uses ] { const_type_var_declarations | routine_header } .
  unit_implementation= IMPLEMENTATION [ uses ] { main_declarations } .
  unit_end= ( BEGIN instruction_list | initialization ) END .
    initialization= [ INITIALIZATION instruction_list [ FINALIZATION instruction_list ]] .

library= LIBRARY LIBRARY_NAME main_uses bloc '.' .

package= PACKAGE PACKAGE_NAME ';'
    $<pack [ requires_clause ] [ contains_clause ] $>pack END '.' .
  requires_clause= REQUIRES REQUIRES_NAME {',' REQUIRES_NAME } ';' .
  contains_clause= CONTAINS contains_statement {',' contains_statement } ';' .
    contains_statement= CONTAINS_NAME [ IN constant_expression ] .
.
```