Portable Object Oriented Language

Language Description and Reference

Revision: see Change Index

Büro für Datentechnik GmbH
35418 Buseck
Germany

1 Contents

2 Change Index

Date	Author	Rev.	Ref.	Type	Description
2005-09-19	Uwe Kühn	1.02.00	various	format content	Formatting of special characters. Updated examples.
2005-09-15	Uwe Kühn	1.01.00	all	editorial	Translation to English language based on “POOL 2 deutsch.doc” as of 2003-07-03.

3 Introduction

Conventional programming languages like Pascal, FORTRAN or C are designed so that over the way of a compiler an independent machine code executable program is created from the corresponding source instructions. The real compiler enters into no interaction with the user. Smallest changes cause a new compilation of the program and the testability of programs and modules is very restricted. Newer integrated programming environments take this circumstance into account and hardly leave wishes unfulfilled on programming and test by an event-driven and interactive user interface. These tools nevertheless are still intended to produce an independent executable program. This mode of operation, however, is rather unfavourable for the operation of a test bed or for the direct interaction with an attached electronic device. Besides the demand to be able to generate programs the urgent need insists here to be able to execute immediate actions by direct command line instructions. Once, BASIC interpreters from the initial years of microprocessor technology already went pretty much close to this claim without delivering a really efficient or really object-oriented tool with BASIC, though. This type of the line wise system control has got itself at the level of operating system shells obstinately why so-called shell languages also are here talking. The basic syntax of Shell languages is mostly

```
Command <Parameter(s)> <Option(s)>
```

in which the options represent a primitive opportunity to adjust environmental variables for this command. Since avoiding of unnecessary keyboard typing in the interactive mode always means a time-saver, such shell languages have all a more or less mysterious appearance (see DOS, Unix, even BSK-Diagnosis), what lets the make - and to an even higher extent the comprehension - from programs in such languages get easy for a frustrating experience.

On the other side conventional programming languages are unsuitably to interactive work because they need a certain part of purely formal code and - what weighs more - kind things, like leaving out parameters, the expansion of parameter lists or even the submission of options are not possible. Nevertheless, there are also attempts to use programming languages as shell languages.

POOL has tried to achieve - obliged to it’s name - this two aforementioned basic demands:

As in the case of every compiler data structures, procedures, functions etc. are translated and summarized to modules, there isn't, however, any main program. After all required modules which were made in advance are loaded, all objects declared as public are at the user's disposal and all his directives, still in POOL syntax, represent in fact the main function of a POOL program.

4 Language Description of POOL

POOL resembles all Pascal based languages. For a user who masters Pascal, it is therefore very easy to use POOL. POOL also is to be easily learned for beginners, since it does without partial features of Pascal. In opposition to Wirth's Pascal, however, achievements of the Borland's Pascal compilers were taken up into the object-oriented expansions. The possibilities of the parameter submission were also enhanced to procedures and functions. As a concession to the line interactive use of pool the definition of omitting parameters and the use of untyped lists were made possible. The essential syntactic difference compared with Pascal is the use of special end keywords at all structured instructions and the so caused introduction of new keywords such as elseif. This expansion shall emphasize the program’s structure more strongly and in addition saves many begins. Other essential differences compared with the common Pascal variants are indicated in italics as help for Pascal experts in the respective sections *(e.g. BP = Borland Pascal).*

The pool source files (<name>.pool) are pure text files with usual line hyphen(s) common to the used operating system. Since there is no line extension character and 'carriage return' (<cr>) is interpreted as white space, even DOS text files (containing <cr><lf>) can be processed directly under UNIX. As the character set an extended 1 byte ASCII set is presupposed. The individual lines may be at most 200 characters long. The different structure levels are marked by indentation, e.g. by two characters, where obligatory house rules (see chapter 6) improve the maintainability of program sources. Unless in comments and text constants merely <tab>s, <lf>s, <vt>s, <ff>s, <cr> and the ASCII characters $ 20 to $ 7 E ('~') are permitted, where all control characters are interpreted as delimiting characters. Comments and text constants apart from <nul>, <cr> and <lf> may contain arbitrary signs, what possibly makes the program no longer portable, however, and can't be processed with any arbitrary editor any more either. The translated text constants may contain all characters from <nul> ($00) up to $FF, the few not allowed characters have to be input as escape sequences (e.g. cChar := '\r'; csString := 'ABC\x0A';).

4.1 The Backus-Naur Notation

Traditionally for the definition of a programming syntax the Backus-Naur notation (BNF = Backus Naur Form) is used. The syntax of a language is described thus completely. The representation of the language syntax into syntax graphs is, however, a more open type, an equal one. Both notations serve to the direct realization of the syntax Parser. The BNF uses symbolic representations which are in this form not part of the described language.

Syntactic elements are identifiers in English language, are written with lower-case letters and included in the angle brackets < and >.

::= one would translate "may consist of" and serves the exact specification of the syntax elements.

Alternatives at the specification of the syntax elements are separated by the vertical line |.

In curly brackets { and } included sequences are optional and may be also used repeatedly (don't apply to the description of the comments and compiler switches).

<empty> points to the permissible particularly use of an empty element.

4.2 The POOL Vocabulary

The complete basic vocabulary of POOL consists of a number of symbols which are classified as follows:

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | _

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<bin digit> ::= 0 | 1

<oct digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<hex digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f

<special symbol> ::= + | – | * | / | ! | & | $ | | | @ | = | <> | != | < | > | <= | >= | (|) | [|] | { | } | (* | *) | := | . | .. | , | ; | : | ' | " | ^ | << | >> | <reserved word>

<reserved word> ::= absolute | and | array | begin | Boolean | breakfor | breakrepeat | breakwhile | Byte | ByteString | case | Char | CharString | const | constructor | contfor | continue | contrepeat | contwhile | destructor | div | do | downto | DWord | else | elseif | end | endcase | endfor | endif | endwhile | endwith | exit | external | false | for | forward | from | function | goto | if | import | in | inherited | Int8 | Int16 | Int32 | Int64 | label | loop | module | mod | nil | not | object | of | or | packed | Pointer | private | procedure | program | public | Real32 | Real64 | QWord | record | repeat | return | set | shl | shr | signed | static | String | then | to | true | type | UChar | UCharString | unsigned | until | var | variant | virtual | WChar | while | WideString | with | withopt | Word | WordString | xor

<space> ::= < > | <tab> | <cr> | <lf> | <ff>

Blank characters <space> is treated as a hyphen and is needed only where lack of hyphens can lead to misunderstandable interpretation. Since an unusual notation affects the readability and thus also considerably the maintainability of program sources the above notation (all words in lower case letters except from the type identifiers) became introduced obligatorily. Also the use of synonymous operators (e.g. not und !) should be regulated uniformly, compiler switches could be introduced if necessary in order to distinguish between respective conventions (e.g. not as logical, ! as arithmetical operator).

4.3 Comments

Comments can be inserted to any place between identifiers or special symbols. They are embraced by { and } or (∗ and ∗) and may be nested as deep as necessary, resp. the are led in by // and end automatically at the same line's end. Although nesting in most of the other languages is not usual, it was introduced intentionally in favour of practical considerations.

<comment> ::= <comment1> | <comment2> | <comment3>

<comment1> := { <comment1 string> } |

{ <comment1 string> <comment> <comment1 string> }

<comment2> ::= (∗ <comment2 string> ∗) |

(∗ <comment2 string> <comment> <comment2 string> ∗)

<comment1 string> ::= <any sequence of characters not starting with $ and

not containing {, } and (∗>

<comment2 string> ::= <any sequence of characters not containing {, (∗ and ∗)>

<comment3> ::= // <comment3 string> <end of line>

<comment3 string> ::= <any sequence of characters not containing <end of line>>

Example:

```
(∗ This Comment includes (∗ this Comment ∗) ∗)
```

4.4 Identifiers

Identifiers serve as references for types, variables, objects and functions. Within their validity range (scope) identifier must be unambiguous, where the first 63 characters are significant. Identifiers distinguish also between upper and lower case letters, mainly in order to force a unique writing. The use of identical identifiers with different meaning has to be avoided.

There is no differentiation between lower and upper case letters in Pascal.

<identifier> ::= <letter> { <letter> | <digit> }

The above syntax applies to all identifiers in POOL and therefore isn't repeated at the special identifiers appearing in the following, e.g. <type identifier> ::= <identifier>, without being indicated explicitly in the corresponding section again.

Examples (see Chapter 6):

```
csName, nenMonday, csABC, nenPik7, boDoorOpen
```

4.5 Numbers

Decimal numbers are represented in the usual notation as an integer or broken number. Integers are allowed in a hexadecimal, octal and binary representation additionally.

<number> ::= <integer number> | <real number>

<integer number> ::= <unsigned integer> | <sign><unsigned integer>

<sign> ::= + | –

<unsigned integer> ::= <digit> { <digit> } | <digit> { <hex digit> } <hex radix specifier> |
<oct digit> { <oct digit> } <octal radix specifier> | <bin digit> { bin digit> } <binary radix specifier> | $ <hex digit> { <hex digit> } | 0x <hex digit> { <hex digit> }

<hexadecimal radix specifier> ::= h | H

<octal radix specifier> ::= o | O | q | Q

<binary radix specifier> ::= b | B

<real number> ::= <unsigned real> | <sign><unsigned real>

<unsigned real> ::= <digit sequence>.<digit sequence> |

<digit sequence>.<digit sequence><exponent prefix><scale factor> |

<digit sequence><exponent prefix><scale factor>

<digit sequence> ::= <digit> { <digit> }

<exponent prefix> ::= e | E

<scale factor> ::= <digit sequence> | <sign> <digit sequence>

Remark:

From the view of the compiler the sign is not part of the number, but treated separately as a unary operator. Integers beginning with '0' will be, depending on the compiler switch $COct, treated either as octal numbers or refused as illegal numbers. As an expansion of the above syntax real numbers may also begin with the decimal point if a numerical character follows next.

Examples:

```
N = 123456789;
{$ifdef POOL} { POOL only, not BP }
  nABin  = 01010101b;
  nAOct1 = 1234567o;
  nAOct2 = 1234567q;
  nAHex  = 0ABCDEFh;
  {$COct+} { allow C syntax for octal numbers }
    nCOct  = 01234567;
```

```
  {$COct-} { forbid C syntax for octal numbers }
  nCHex  = 0xABCDEF;
{$endif}
  nPHex  = $ABCDEF;
  nReal1 = 12e1;
  nReal2 = 12.;
  nReal3 = 1.2;
  nReal4 = 1.2e-1;
```

4.6 Strings

Strings are data types which have some length information and a data area. The length information says how many elements the accompanying data area contains. In practice Strings are from their length as well as not restricted and can take a size depending on implementation which is nearby the address range of the target system. In an internal representation strings are actually structured types, they are implemented, however, to the advantage of the user like simple types. Due to their possible size strings are always managed dynamically, i.e. their memory requirements are adapted at **every** operation to the size required in the end. With their flexibility strings are one of the most important benefits and one of the greatest strengths at all the language POOL offers.

Important notes for C programmers:

1. *Strings are real independent types in pool and must be also used in this way. CharString (synonymous with String) may be used only, where "readable" characters are stored. For common buffers the type ByteString only (with prefix "bs") has to be used.*
2. *Pool isn't C, therefore MemMove must never be used in a program for copying strings. Firstly, the construct will get round all type checking with that, secondly, it's illegibly and thirdly also wrong, moreover because the string data laid out dynamically cannot be copied. Furthermore MemMove isn't necessary, since CharStrings are assignment compatible to zero-terminated Char-Arrays (if $zstr+) as ByteStrings are to zero-terminated Byte Arrays. Special functions exist for cases in which this cannot be made use of like AB2BStr, HStr2Str, Move2Str and Str2HStr (see below).*
3. *The address references to the data contents of strings can change at **every** operation since strings are managed dynamically in POOL. The use of the address operator is therefore warned against urgently on string elements (although permitted syntactically). For example: The pointer resolved by the expression @csStr[3] will be usually pointing to an address outside the data area of csStr after **any** manipulation of the String. Such an access usually isn't necessary either.*

4.6.1 Character Strings

Character strings (type name CharString, synonymous String) are sequences of characters enclosed by quotes ' or double quotes ". If the characters ' and " should be

part of the strings themselves, they have to be headed by a \ (Backslash). Special signs are represented with escape Characters like usual in C. Constant Character strings with only one character are represented internally as Char constants which has no meaning, however, for the user, since single characters are in principle assignment compatible to strings.

<char string> ::= ' { <character1> } ' | " { <character2> } "

<character1> ::= <character except CR, LF, ' and \> | <escape character>

<character2> ::= <character except CR, LF, " and \> | <escape character>

<escape character> ::= \ <escape code>

<escape code> ::= <character escape code> | <octal escape code> | <hex escape code>

<character escape code> ::= a | b | f | n | r | t | v | \ | ' | "

<octal escape code> ::= <oct digit> | <oct digit> <oct digit> | <oct digit> <oct digit> <oct digit>

<hex escape code> ::= x <hex digit> { <hex digit> }

Character Escape Codes:

a	alert (z.B. bell)	t	horizontal tab
b	backspace	v	vertical tab
f	form feed	\	backslash
n	newline	'	single quote
r	carriage return	"	double quote

Examples:

```
'String, containing the " char'
"String, containing the ' char"
'String with '' and "', "\0", "\x00AB" { that's only one char! }
"\0111" { these are two chars! }
"\090"  { these are three chars! }
"\n"    { this is one char, but will be possibly expanded to two chars
          when written into a text file }
```

4.6.2 Byte Strings

Byte Strings (type name ByteString) represent a generalised form of strings, where their elements are not of the Character type, but of the Byte type. As they belong by definition to the simple types, this kind of strings are often liked to be used as dynamic buffers with an implicit length information und can therefore be used very simple as submitting parameters as well as function results. Most string operations are in principle also possible with byte strings so that manipulating these buffers appears to be extremely comfortable. Character strings and byte strings are identical on data base and can be led into each other by a simple type casting. The strict type checking of POOL, however, requires two different types.

Unlike character strings there is no possibility to set up Byte strings as constants, neither typed, nor untyped. On the other hand, however, the keyword bcstr can be applied, that permits to create functions and procedures which work similarly with Character or Byte strings without further distinction

4.7 Type Definitions

Type definitions allow the creation of new data types based on the existing set of standard types. Type definitions are defined in an independent definition block which is headed with the keyword type.

<type block> ::= type <type definition> { <type definition> }

<type definition> ::= <type identifier> = <type> ;

Examples:

```
type
  tbIndex = Byte;
  tbCounter  = tbIndex;
```

4.8 Data Types

Data types in the real meaning are data structure descriptions which are assigned to variables. The variables themselves then form the instances of these data structures.

<type> ::= <simple type> | <pointer type> | <structured type>

4.8.1 Simple Types

Simple Types generally are unstructured data types.

<simple type> ::= <standard type> | <type identifier>

<standard type> ::= <ordinal type> | <real type> | <string type>

<ordinal type> ::= <integer type> | <subrange type> | <enum type> | Boolean | Char

<integer type> ::= Int8 | Byte | Int16 | Word | Int32 | DWord

<real type> ::= Real32 | Real64

<enum type> ::= (<identifier> { , <identifier> })

<subrange type> ::= <constant> .. <constant>

<string type> ::= String | ByteString

Value ranges of Integer types:

Type	Range	Format
Int8	-128 .. 127	8 Bit signed
Int16	-32768 .. 32767	16 Bit signed
Int32	-2147483648 .. 2147483647	32 Bit signed
Byte	0 .. 255	8 Bit unsigned
Word	0 .. 65535	16 Bit unsigned
DWord	0 .. 4294967295	32 Bit unsigned

Range and precision of the Real types:

Type	Range		Accuracy	Format
Real32	$1.2*10^{-38}$ ($1.5*10^{-45}$)	to $3.4*10^{38}$	6-7 digits	32 Bit IEEE
Real64	$2.2*10^{-308}$ ($5.0*10^{-324}$)	to $1.7*10^{308}$	15-16 digits	64 Bit IEEE

(the value in brackets isn't normalised any more)

The standard types represent the basic data types for the construction of structured types and aren't re-definable. The string types in principle are structured, however, unstructured in their manifestation, thus being counted formally to the Simple types. The type classes <integer type> and <real type> define assignment compatible data types which are transferable into each other as long as the destination type can hold the content of the source type.

Examples:

```
Int16, (red, green, blue), 10..20, Char, Real64
```

4.8.2 Pointer Types

Pointer types describe variables which contain address references to other variables instead of data. Address references can be won both from static and dynamic (i.e. runtime created) variables. A pointer which keeps the value nil doesn't provide any reference to a variable.

<pointer type> ::= ^<type identifier> | Pointer

Untyped pointers of the pure type pointer can point to variables of any arbitrary type, they are assignment compatible to all other pointer types, but can't get de-referenced without a previous type cast, however.

Example:

```
tpstCicle = ^tstCircle;
```

4.8.3 Structured Types

Structured data types are indexed or not indexed orders of simple types.

<structured type> ::= <array type> | <record type> | <object type>

4.8.3.1 Indexed Types (Array Type)

Array types describe a finite number of identical components of another type.

<array type> ::= array [<index type> { , <index type> }] of <type>
<index type> ::= <ordinal type>

The individual components of Arrays are provided with references by the index. If the component type of an Arrays is also an Array, one can treat the result either as an Array of Array or as a single multidimensional Array. The individual elements of strings (e.g. Char) also are treated like an Array with index, but since no normal Array access takes place internally, however, the string index must be indicated one by one (Str[n] equals to Str.PText^[n]).

"Open" Arrays are also supported by being copied from "indecent" big Arrays. To be able to use such Arrays, the constant nMaxOASize exists in the POOL system, but dependent on the target system, on which such Arrays are based in units of bytes as the maximum size. Independent from that, in reality such "open" Arrays are dynamic data, of course, whose actual size has to be determined by HeapBlockSize while the number of allocated elements can be calculated for them over the size of every element. Merely in some special cases such Arrays appear also static, e.g. in type definitions as constants which the compiler allocates already. All functions which work with such data must know and take into account an end criterion as a consequence! The POOL compiler supports the use of "open" Arrays by a short notation at the index range specification (see examples).

Examples:

```
tacCA10   = array [0..9] of Char;
taaar64RA = array [Boolean] of array ['A'..'Z'] of array [0..9] of Real64;
             { the compiler is interpreting just the same as
               array [Boolean, 'A'..'Z', 0..9] of Real64}

{ "open" Array with most possible size in an actual implementation: }
```

```
tacOAC    = array [0..nMaxOASize div SizeOf(Char)-1] of Char;

{ the same "open" Array in an open short notation: }
tacOAC    = array [0..] of Char;
```

4.8.3.2 Record Types

A Record type describes a structure which consists of a finite number - of as a rule different - components. Each of these components, called field, gets an identifier of it's own and a type of it's own.

<record type> ::= record <field list> end | record <parent> <field list> end

<field list> ::= <fixed part> | <fixed part> ; <variant part>

<parent> ::= (<record type identifier>)

<fixed part> ::= <identifier list> : <type> { ; <identifier list> : <type> } | <empty>

<identifier list> ::= <field identifier> { , <field identifier> }

<variant part> ::= variant <variant> { ; <variant> }

<variant> ::= (<record section> { ; <record section> })

For public, global Records the view of other modules to the fields of records can be limited by the keywords private and public (these keywords work as switches, i.e. all field identifiers from private to public or until the end aren't visible for other modules.

Similar to object types Records can be derived from existing Record types at POOL (see below).

Variant parts of Records (*unions in C)* are introduced by the keyword variant and describe data structures which come to lay over the same memory area. Records in POOL are guaranteed to be always "packed", so that these parallel structures will never happen to lay over alignment caused gaps. Therefore variants are used in practice often as multi-structured buffers instead of genuine type casts. Since the defined variants not necessarily must have the same physical total size (this can already depend on the implementation alone), variant parts never can lie between non-variant parts of the Record definition but only to their end. Due to their special treatment in POOL dynamic strings are not permitted in variant parts of Records, neither directly nor indirectly.

Example (*much further practice near examples in POOL.PLI*):

```
stData1 = record (stData0)
            variant
              (i32L: Int32);
             (dwDW: DWord);
              (abB:  array [0..3] of Byte);
          end;
```

In Pascal there isn't any heredity at Records as well as no public or private. In contrast and in advantage to Pascal in POOL variants are not named and are not described in form of the there not even obvious Case construct either.

4.8.3.3 Object Types (Classes)

Objects are structured types similar to Records which, however, still define so-called methods besides data fields which carry out certain operations with the object. Such methods are quite normal procedures and functions which are part, however, of the object and are bound partly dynamically (virtual). Object types can be declared as global and as type identifiers only (therefore not var Obj: object ... end;).

<object type> ::= object <element list> end | object <parent> <element list> end

<parent> ::= (<object type identifier>)

<element list> ::= <field list> | <method list> | <field list> ; <method list>

<field list> ::= <identifier list> : <type> { ; <identifier list> : <type> } | <empty>

<identifier list> ::= <field identifier> { , <field identifier> }

<method list> ::= <method declaration>{; <method declaration>}

<method declaration> ::= <method prototype> | <method prototype> ; virtual | <method prototype> ; external <integer constant>

<method prototype> ::= <procedure declaration> | <function declaration> | <constructor declaration> | <destructor declaration>

At public objects, i.e. objects declared within the public segment, the view from other modules at the fields and methods of these objects can be restricted by use of the keywords private and public (these keywords work as switches, i.e. all identifiers from private to public or until end aren't visible for other modules).

Restrictive to the definition above constructors must not be virtual since the virtual method table (VMT) isn't initialized at its call yet. Since a just defined object type already may appear as a parameter in its methods and the compiler also must know it’s size, all data fields, as indicated above, must be declared **before** the method’s declaration. Objects can pass their data structures as well as their methods. The heir declares from which object he is deduced. Unlike the data fields methods can be declared with identical names as those of the ancestors, through what the corresponding method is overwritten. At virtual methods exactly the same prototype must be used, though, static methods can be redefined (however, they have to remain static). Except at the overwriting of methods all visible names must be clear and therefore being not allowed to be used as a parameter or local variable in methods either. Merely private names of objects of other modules can be reused freely.

In BP therefore the mixture of data fields and methods via private and PUBLIC is possible which has the consequence, though, that some faults are recognized and reported only at the implementation of the methods outside the object's type declaration. In addition, external is possible only at the implementation, there is the additional number at virtual instead of at external and static methods may be overwritten with virtual methods.

Examples:

```
toPoint = object
            i16X,i16Y: Int16;
            procedure vDraw; virtual;
          end;

toCircle = object (toPoint)
            i16Radius: Int16;
            constructor poInit (XX,YY,R: Int16);
            destructor vDone; virtual;
            procedure vDraw; virtual;
          end;
```

4.9 Constant Definitions

Constants permit the use of unchanging values by the use of their identifiers. In POOL, constants allocate just the same as variables, their use, however, is of advantage since expressions with constants are already evaluated by the compiler. Since there, however, aren't any special constant data types, constants are submitted, e.g. at parameter lists (variable number of parameters), like variables. To prevent from runtime errors at write accesses on the constant segment, a function which classifies a submitted address is provided (Code, Var, Static, Const, Stack, Heap, nil or invalid) which allows the user to take own measures if it comes to handle illegal types. Constant definitions are carried out in an independent block which is headed with the keyword const.

<const block> ::= const <constant definition> { ; <constant definition> } ;

<constant definition> ::= <constant identifier> = <constant> |
<constant identifier> : <type> = <typed constant>

<constant> ::= <number> | <char> | <char string> | nil

<typed constant> ::= <constant> | <array constant> | <record or object constant>

<array constant> ::= (<typed constant> { , <typed constant> })

<record or object constant> ::= (<empty>) | (<field identifier> : <typed constant> { ;
<field identifier> : <typed constant> })

Definitions which contain a type information specify so-called typed constants. If not all elements or fields are indicated, the rest of the occupied memory area becomes byte wise filled with the value zero. 'array[..] of Char'-Types can be initialized by Character Strings alternatively. For a use as a C string it has to be taken into account, that the

string is filled up with zeros only if the string length is smaller than the array size. Therefore it is recommended to add a terminating zero character explicitly if necessary. Only one variant can always be initialized at variant Records. For untyped constants the smallest type which can take the given constant is created (in bytes).

Examples:

```
const
  nrFakt        = 1.234;
  ncsTextConst1 = 'Text' + '1';
  ncCharConst1  = 'C';
  ncsTextConst2 = ncsTextConst1 + ncCharConst1;
  ncsTextConst3 = ncsTextConst1 + ncsTextConst2;
  ncsTextConst4 = ncCharConst1 + ncCharConst1;
  ncsTextConst5 = '\1\2\3\4\5\6\7\10\100\200\377\a\b\f\n\r\t\v''\"\?\0\0';
  npTestNil     = nil;
  nboTestRelOp  = (1>2);
  nTestExprI    = 3 * (1+2);
  nTestExprR    = 1/3;
  nTestUMinus   = -10;
  nTestUPlus    = +10;
  nTestANot     = not 10;
  nboTestLNot   = not false;

type
  tstPunkt = record
    x,y: Int16;
    end;
  tstP1 = record (tstPunkt)
    r: Int16;
    stRec: record
      variant
        ( i,j,k: Int16 );
        ( b0:  Byte; b1: Byte; b2: Byte; b3: Byte );
        ( abB: array[0..3] of Byte );
        ( acC: array[0..3] of Char );
      end;
    end;

const
  ndwLongConstant: DWord = 5;
  nstP1_1: tstP1 = (x:0; y:0; r:0; stRec:(i:0; j:0; k:0));
  nstP1_2: tstP1 = (x:0; y:0; r:0; stRec:(b0:0; b1:0; b2:0; b3:0));
  nstP1_3: tstP1 = (x:0; y:0; r:0; stRec:(abB:(0,1,2,3)));
  nstP1_4: tstP1 = (x:0; y:0; r:0; stRec:(abB:(0,1)));
  nstP1_5: tstP1 = (x:0; y:0; r:0; stRec:());
  nstP1_6: tstP1 = (x:0; y:0; r:0);
  nstP1_7: tstP1 = (x:0; y:0; r:0; stRec:(acC:('a','b','c','\0')));
  nstP1_8: tstP1 = (x:0; y:0; r:0; stRec:(acC: 'abc\0'));
```

4.10 Variable Declaration

Variables are declared by assigning an individual data type to every variable identifier. If the identical type shall be assigned to several identifiers, simplifying a list of identifiers can be given. Variable declarations are carried out in independent blocks, these are initialized with the keywords var or static.

<var block> ::= <variable storage class> <variable declaration> { ; <variable declaration> } ;

<variable storage class> ::= var | static

<variable declaration> ::= <var identifier> { , <var identifier> } : <type> |
<var identifier> : <type> = <constant>

Variables to which a constant was assigned at the declaration are said initialized or static variables. The syntax is identical to that one with which typed constants are initialized. Unlike constants, however, initialized variables can be changed again while runtime. Such static variables are different from var variables by the fact that they are available only once, even then, when they were created as a variable with a local validity range (scope) (e.g. in procedures) and have the possibility of keeping their value from one call to the next one.

The type checking of POOL is strict, i.e. var variables *must not* and static variables *must* be initialized.

A variable or typed constant of an object type is described as the instance of the object. Only initialized object instances can be used without constructor calls (even assignments like Obj2 := Obj1 can **never** replace the constructor call for not initialised object instances!).

Examples:

```
var
  i,j,k: Int16;
  pi16Ptr: ^Int16 = nil;
static
  boListOutput: Boolean = false;
```

4.11 Use of Variables, Fields and Constants

Variables and constants are used as complete variable, component of a variable or as pointer referenced variables by their given identifier.

<variable> ::= <variable identifier> { <qualifier> } | <typecast> { <qualifier> } |
<function designator> { <qualifier> }

<variable identifier> ::= <identifier> | <module identifier> . <identifier> |
. <module identifier> . <identifier>

<typecast> ::= <type identifier> (<variable>)

<function designator> ::= <function identifier> |
<function identifier> (<actual parameter> { , <actual parameter> })

<qualifier> ::= <index> | <field designator> | ^

<index> ::= [<expression> { , <expression> }]

<field designator> ::= . <field identifier>

The dot preceding the module name serves to indicate the following identifier as an unequivocal module name and is usually only necessary in the (hopefully) rare cases where a (local) record variable exists with the same name. However, in principle, to distinguish global variables of other modules from record elements clearly, it can make sense to use the leading point at module names. Typecasts with different integer types are partly changed by the compiler into transfer functions; the result, of course, is not longer a variable and therefore no qualifier can follow either (e.g. Int16(WordVar) is a variable, while Int16(ByteVar) is the result of a transfer function). In principle, a real type casting is possible only if the size of the variables corresponds to the size of the type which the type identifier exactly stands for. Function results can not be used as variables, however, if a function returns a pointer this can be de-referenced by a succeeding ^ and this stands for the variable at which the pointer points. See section 'function calls' for the general definition of <function designator>.

Examples:

```
i, .MODULE1.i
Int16(wVar), toRec2(stRec1Var).i16X
astA[2], adwB[2*3+1,5], csStr[Length(csStr)-1]
oCircle.i16Radius, stData.bHi, stData.stR.pA2
pstDataPtr^.bLo, pst := pstPrtFunc(i16A,i16B,i16C)
```

4.12 Expressions

Expressions are links of operands by operators. Operands are variables, constants or functions. Operators contain certain processing regulations for the assigned operands. The result of an expression is always a value of a valid and existing data type. Operators obey a predefined hierarchy. Not all combinations of operands and operators make sense and all combinations therefore are not allowed either.

Remark:

Common to all high level languages the parts of composite arithmetical terms are calculated independently from the order of their appearance in the source text! Therefore it is not permitted to formulate expressions, where the terms are dependent of their order, e.g. several function calls with side effects. Also mathematically unnecessary brackets or type casts do not influence the order of the computing. If a certain calculation order has to be kept, the expressions has to be split.

<expression> ::= <simple expression> |

<simple expression> <relational operator> <simple expression>

<relational operator> := < | > | <= | >= | = | <> | !=

<simple expression> ::= <term> | <simple expression> <adding operator> <term>

<adding operator> :== + | – | or | | | xor

<term> ::= <factor> | <term> <multiplying operator> <factor>

<multiplying operator> ::= * | / | div | mod | and | & | shl | << | shr | >>

<factor> ::= <variable> | <unsigned constant> | (<expression>) | <not> <factor> | <sign> <factor> | @ <variable> | <function designator> | <typecast>

<unsigned constant> ::= <unsigned number> | <char string> | <char> | <constant identifier> | nil

<not> ::= not | !

Examples:

```
(a+b*c), pi>0, 3*(–5), a+1 > b, @stRec, Sin(x/2)
```

4.13 Operators

The operators are arranged hierarchically into four levels. They are graded descending into monadic (unary) operators, multiplying operators, operators adding up as well as comparative operators. Operators of a higher level "adhere" to the adjacent operands more tightly and are executed preferentially. The operators are usually evaluated in the expression within the hierarchy levels from left to right, but the compiler may carry out rearrangements for the optimization of the computing path, except for logical or and and. The logical operators or and and are evaluated strictly from left to right and also in the short circuit procedure. By brackets (and) hierarchy levels can be preferred, however, without forcing a certain evaluation order. (e.g. a+(b+c) can be evaluated the same way as a+b+c). Operators only have an effect on operands of cooperating types, normally. Only some operators have an effect on operands of different types or connect even operands of different types. The operators are subdivided into logical operators who only have an effect on Boolean operands and arithmetical operators who have an effect on all numerical operands. Within the arithmetical operators there are some who only have an effect on integral operands. As an extension the addition operator + can be applied also to strings and characters (Char).

The arithmetical operators, except /, shl, <<, shr and >>, cause at application to different integer types a type extension according to the table below. According to these rules the compiler investigates the result type and whether an expression is permitted. The result type again is used to check the permission of assignments. Particularly at terms with more than two operands the calculations are executed also with higher precision if necessary. The type of the destination variables at assignments influences the evaluation of terms under no conditions. Besides the standard types (I8=Int8, BY=Byte, I16=Int16, WD=Word, I32=Int32, DW=DWord) the compiler also uses the intermediate types 7Bit (0..$7F), 15Bit ($0100..$7FFF) and 31Bit ($00010000..$7FFFFFFF), particularly for being able to assign constants of the appropriate range both to signed and unsigned variables. Due to the strict type checking of POOL otherwise either

ByteVar := 0 or Int8Var := 0 would be rejected, depending of the constant 0 to be interpreted internally as Int8 or as Byte.

	7Bit	I8	BY	15Bit	I16	WD	31Bit	I32	DW
7Bit	7Bit	I8	BY	15Bit	I16	WD	31Bit	I32	DW
I8	I8	I8	I16	I16	I16	I32	I32	I32	Err
BY	BY	I16	BY	15Bit	I16	WD	31Bit	I32	DW
15Bit	15Bit	I16	15Bit	15Bit	I16	WD	31Bit	I32	DW
I16	I16	I16	I16	I16	I16	I32	I32	I32	Err
WD	WD	I32	WD	WD	I32	WD	31Bit	I32	DW
31Bit	31Bit	I32	31Bit	31Bit	I32	31Bit	31Bit	I32	DW
I32	I32	I32	I32	I32	I32	I32	I32	I32	Err
DW	DW	Err	DW	DW	Err	DW	DW	Err	DW

4.13.1 Monadic Operators

Monadic operators most strongly adhere to operands and are considered an inseparable unit with the operand. They only have an effect on the one following operand. One also can understand the monadic operators as a short notation for functions omitting the brackets, what at the addressing operator @ gets particularly clear. The two negation operators not and ! are synonymous and cause an inversion at Boolean operands, resp. the execution of the 1's complement at integral operands.

<monadic operator> ::= not | ! | @ | <sign>

Examples:

```
not true, @Data, !10110110b
```

4.13.2 Multiplying Operators

These operators are not really multiplying in the severe sense, or rather operate at the hierarchy level of the multiplying calculations and therefore are described so. They always stand between the operands which they connect.

<multiplying operator> ::= * | / | div | mod | and | & | shl | << | shr | >>

The operator couples and | &, shl | << and shr | << are synonymous. Mode of action of the operators, cooperating operand types and result types of the operators shows the table below. Other combinations are not permitted. In this context it is advantageous to use the type classes for the operands that have been introduced before.

Operand type A	Operator	Operand type B	Result type
<integer type>	*	<integer type>	<integer type>
<real type>	*	<integer type> | <real type>	Real64
<integer type> | <real type>	*	<real type>	Real64
<integer type> | <real type>	/	<integer type> | <real type>	Real64
<integer type>	div | mod | and | & | shl | << | shr | >>	<integer type>	<integer type>
Boolean	and | &	Boolean	Boolean

Examples:

```
3*8, 122 div 6, 10h shl 5, b & 00001100b
```

4.13.3 Adding Operators

The operators adding up are used like the multiplying operators, they work, however, at the hierarchy level of the dash type calculations and therefore get called that way.

<adding operator> ::= + | – | or | | | xor

The operators or and **|** are synonymous. Mode of operation, cooperating operand types and result types of the operators are shown in the table below.

Operand type A	Operator	Operand type B	Result type
<integer type>	+ | – | or | | | xor	<integer type>	<integer type>
<real type>	+ | –	<integer type> | <real type>	Real64
<integer type> | <real type>	+ | –	<real type>	Real64
Boolean	or | | | xor	Boolean	Boolean
String | Char	+	String | Char	String

Examples:

```
a+3, 55 xor 0FFh, c or d
```

4.13.4 Comparative Operators

The comparative operators take a special position both regarding their hierarchy level (undermost) and their result type which always is of the type boolean. Furthermore comparative operators exclusively connect operands of equal type or at least equal type classes (so that not apples get comparable with pears). Character strings are compared in accordance with their character code in which the leftmost characters take the most significant positions. A string is also regarded as greater if it is longer than the other one and is in addition identical with it up to the length of the shorter.

<relational operator> := < | > | <= | >= | = | <> | !=

The Operators <> and != are synonymous. Comparisons basically are allowed only between simple, unstructured, compatible types. Supplementary the following combinations are legal: <char string> | <char> with <char string> | <char> and <real type> with <integer type>. For pointers only = and <> (resp. !=) are allowed and this only then, if the pointers point at the same type or one at least is an untyped pointer (nil or of the type pointer).

Examples:

```
a > b, (f < 1) <> true, csStr = 'N', pstPtr <> nil
```

4.14 Statements

Statements are the algorithmic execution steps of a program.

<statement> ::= <simple statement> | <structured statement>

4.14.1 Simple Statements

Such are simple statements which do not contain any further statements. The empty statement which causes no action is also part of them.

<simple statement> ::= <assignment statement> | <procedure statement> | <control statement>

4.14.2 Assignments

Assignments allow to transfer new data contents into variables. Assignments are allowed only between assignment compatible data types. Data types are assignment compatible if they belong at least to the same type class, or if one is a real and the other one an integer type and the destination variable is big enough that it can take the assigned data contents or else if the assignment destination is a subset of the assignment expression. E.g. for objects oFather := oSon is allowed, but oSon := oFather is not, since not all fields of oSon get a value assigned. Only within functions or constructors the function name can occur as a destination of an assignment like a variable through which the function result is inserted.

<assignment statement> ::= <variable> := <expression> | <function identifier> := <expression>

Examples:

```
cCharacter := 'A', i := 0, i32GgT := a
```

4.14.3 Procedure Calls

Procedure calls consist of the identifier of the invoked procedure as well as an optional list of parameters submitted to, which are included into the round brackets (and). The parameters must correspond to the details of the formal parameters of the procedure's head in type and order. There are four types of parameters to be submitted:

1. The submission by value.
2. The submission by reference (var parameter), where it has to be distinguished between typed and untyped parameters. For typed parameters internally a pointer is submitted which will be de-referenced automatically at every access within the procedure, while untyped var parameters will submit a tstUArg structure as value (with type and pointer).

3. The submission of an unspecified list.
4. The submission of the empty parameter. (The empty parameter is permitted only for untyped parameters, e.g. within the list of a variable number of parameters, see also section 'Procedure Declarations'.)

<procedure statement> ::= <procedure identifier> <actual parameter list> |
<function identifier> <actual parameter list> |
procedure (<function identifier> <actual parameter list>) |
<object variable> . <method identifier> <actual parameter list> |
inherited <method identifier> <actual parameter list> |
<method identifier> <actual parameter list> |
<object type identifier> . <method identifier> <actual parameter list>

<actual parameter list> ::= <empty> | (<actual parameter> { , <actual parameter> })

<actual parameter> ::= <expression> | <variable> | <empty>

<method identifier> ::= <procedure identifier> | <function identifier> | <constructor identifier> |
<destructor identifier>

Limiting to the definition above functions may be used as procedures only if this syntax extension has been switched on by the compiler switch $IFR (ignore function result). In addition, method calls without a preceding object variable are possible only within the methods of the object type in question or one of the descendants.

Examples:

```
vPrintXY (1,10,"Hello World"),
toFather.Draw, procedure(i32IntFunc(a,b,c)), inherited poInit(x)
```

4.14.4 Function Calls

The call of functions is carried out analogously to that one of the procedures. Unlike the procedures the functions return, however, a function value which can be used as a data source in assignments and terms. The equal restrictions as above apply to method calls. Thereby it has to be taken into account, that no procedures and destructors are allowed (destructors are always procedures) and constructors always, i.e. independently from the compiler switch $IFR, can be used either as procedures or as functions of the type Pointer.

<function designator> ::= <function identifier> <actual parameter list> |
<object variable> . <method identifier> <actual parameter list> |
inherited <method identifier> <actual parameter list> |
<method identifier> <actual parameter list> |
<object type identifier> . <method identifier> <actual parameter list>

Examples:

```
a := i32Maximum (a, b);
if inherited poInit(x) <> nil then ...
```

4.14.5 Type Transformations

In principle, there are two classes of type transformations: ***Implicit*** and ***explicit*** type transformations. The explicit type transformations disintegrate into the two ways of the ***direct*** and the ***indirect*** type transformation.

Implicit type transformations take place (see above) at assignments by the compiler automatically. The destination variable determines the type of the result. The transformation is only possible between assignment compatible types. It is supervised by the compiler, whether the assignment is allowed. The compiler also decides itself, whether a real internal transfer function has to be used or whether the transformation by a direct data take-over is possible. As a rule, this kind of type transformations is uncritical and straightforward.

If the result type is not known, e.g. at the submission of a value into an untyped list (such as at the Write function) one has to produce consciously the desired type information divergently from the original type, for example the real output of an ordinal value by multiplication by 1.0 or addition with 0.0.

Work with ***explicit*** type transformations (type casts) can be very advisable, it is, however, also dangerous, since the examining possibilities of the compiler are limited hereby or even pried out.

<typecast> ::= <type identifier> (<variable>) | <type identifier> (<expression>)

The ***direct*** type transformations are temporary, i.e., in the instant of its application, they only have an effect on the argument within the simple statement in which they are. There are two types of direct type transformations. The ***first type*** allows the transformation of one variable of a certain type into another type. At this way of transformation the two types must have exactly the same size (in bytes). Structured and unstructured types may be combined similarly, the result can be used in all respects like a variable of the destination type, i.e. qualifiers also may follow.

Caution: At this kind of type transformation merely the memory contents of the original variables are interpreted in the way of the type cast over. In the following first example, no real correspondence of an integer value arises from that. Therefore the result of the operation is generally senseless in this example. The example is listed only here to clarify how mercilessly type casts are executed and that caution is advised with their use.

Examples:

```
r := Real32 (l);        { r: Real32; l: Int32; }
c := Char (b);          { c: Char; b: Byte;    }
i := Int16 (w);         { i: Int16; w: Word;   }
```

By the ***second way*** of direct type transformations values which also may be the results of terms and variables of different sizes are converted. For this transformation a suitable transfer function is used by the compiler, therefore this function is possible only with the predefined ordinal types and the result is always a value. This transformation can bring about a shortening or also an extension of the original value if the quantity of the specified type is unequal to that one of the expression. With extension of values the sign is kept, thus also the value for signed destination types, with shortenings necessarily not.

Examples:

```
i := Int16 (65535);         { results in –1 }
i := Int8 (-129);           { results in 127 }
i := Int16 (b);             { b (Byte) will be extended to 16 Bits }
i := Int16 (l);             { l (Int32) will be cut to 16 Bits }
```

Remark: Type casts applied to the results of functions are allowed only when the destination type is the type of a permissible function result.

The form of the ***indirect*** type transformation finally suspends all control possibilities of the compiler. It only has an effect on variables and is carried out over the detour of the type transformation of a temporary pointer. It is not language elements and therefore is listed here only to complete the picture.

Example:

```
type tpWord = ^Word;
...
w := tpWord(@axData[n])^;
```

In this example a complete data word is read from the field named Data at the position of the n-th component and this one is assigned to the variable w. Therefore the address of Data[n] is investigated first, this then is understood as a word pointer and in conclusion the so referenced data word is read from the field.

4.14.6 Control Statements

The control statements influence the process of structured statements. While break terminates loop executions of the nearest for, repeat or while command, return leads to the exit of the current subroutine level, namely the executing function or procedure. continue skips the rest of the loop kernel and starts the next loop execution if applicable. Beyond that there are some special break and continue commands corresponding to every kind of loop in order to be able to leave loops in most cases even from deeper levels without executing goto commands (POOL has no goto, only SetJump and LongJump are supported).

<control statement> ::= break | breakfor | breakrep | breakwhile | continue | contfor | contrep | contwhile | return

4.14.7 Structured Statements

Structured statements are constructions arranged from other statements which are executed sequentially, either conditionally or repeating.

<structured statement> ::= <compound statement> | <conditional statement> | <repetitive statement> | <with statement>

4.14.8 Sequential Statements

Sequential statements are exactly executed in the order in which they were written. A group of sequential statements is embraced by the keywords begin and end.

<compound statement> ::= begin <statement list> end

<statement list> ::= <empty> | <statement> { ; <statement> }

4.14.9 Conditional Statements

Conditional statements result in the branch of the program depending on the result of the conditional expression.

<conditional statement> ::= <if statement> | <case statement>

4.14.9.1 if Statement

The if statement allows the execution of a statement if the test condition produced the Boolean value "true". Provided that an else construct exists, these statements are executed in the "false" case.

<if statement> ::= if <expression> then <statement list> endif |
if <expression> then <statement list> <else construct> endif |
if <expression> then begin <statement list> end |
if <expression> then begin <statement list> end else <statement>

<else construct> ::= else <statement list> | elseif <expression> then <statement list> |
elseif <expression> then <statement list> <else construct>

For the concatenation of equivalent conditional branches the use of the keyword elseif is recommended. It equals to the separated notation else if known from Pascal, it avoids, however, the collection of endifs with which every single if statement must be completed. The syntax without endif is thought only to the simpler porting of Pascal programs and as a concession to Pascal developers too, however, if <expression> then <statement> is illegal under all circumstances. If no compatibility is necessary for Pascal, elseif and endif should be used generally, since the program structure gets even clearer so.

Example:

```
if a > 0 then
  Write ('+');
elseif a < 0 then
  Write ('-');
else
  Write (' ');
endif;
```

4.14.9.2 case Statement

The case statement consists of an expression as a selector and a following list of statements which are indicated by a list of constants of the selector type each. The statement is executed respectively whose constant is identical to the contents of the selector.

<case statement> ::= case <expression> of <case list element> { ; <case list element> }
<endcase> | case <expression> of <case list element> { ; <case list element> }
else <statement list> <endcase>

<case list element> ::= <case label list> : <statement>

<case label list> ::= <case label> { , <case label> }

<case label> ::= <ordinal constant>

<endcase> ::= end | endcase

The same remarks are for the syntax without endcase as mentioned for if.

Example:

```
case cCommand of
  'A': vAbbort;
  'R': vRetry;
else
  vContinue;
endcase;
```

4.14.10 Repetitive Statements

By repetitive statements other statements become executed in a loop as long as the condition is fulfilled with regard to their execution. The two loop types repeat and while are different essentially in this, whether the loop condition shall be checked respectively before the execution of the loop body or after this. for makes the execution of a simple counted loop possible.

<repetitive statement> ::= <repeat statement> | <while statement> | <for statement>

4.14.10.1 repeat Statement

The repeat statement executes the loop body once in every fall before it checks the condition, whether the loop shall be continued. Unlike the while loop construction the until condition does not formulate a continuation but a break criterion.

<repeat statement> ::= repeat <statement list> until <expression>

Example:

```
repeat
  a := a * 1.2 + 1;
until a > 2000;
```

4.14.10.2 while Statement

At the while statement the loop body becomes executed as long as the condition expression produces the Boolean value "true". If the condition is not satisfied already at the first time, then the loop body is not executed at all.

<while statement> ::= while <expression> do <statement list> endwhile |
while <expression> do begin <statement list> end

For the syntax without endwhile the same remarks apply as mentioned for if.

Example:

```
while not boEndOfInput do
  vReadCharacter;
endwhile;
```

4.14.10.3 for Statement

The for statement allows the repetition of statements if the number of loops is known in advance.

<for statement> ::= for <control statement> do <statement list> endfor | for <control statement> do begin <statement list> end

<control statement> ::= <control variable> := <for list>

<for list> ::= <initial value> to <final value> | <initial value> downto <final value>

<control variable> ::= <variable identifier>

<for count> ::= <expression>

<initial value> ::= <expression>

<final value> ::= <expression>

The number of loop executions is calculated once at the beginning as an Int32 type out of the difference between <initial value> and <final value>, thus the expressions have to result in ordinal types, except DWord, (and must be assignment compatible to the control variable, of course). The control variable will never be affected by the loop, but merely incremented (to) or decremented (downto) at every loop's execution behind the list of statements. Therefore assignments to the control variable do not influence the number of loops, but can cause the control variable running out of the given limits. For that reason and because the internal treatment of the for loop can change in other compiler implementations, those assignments are acceptable only just in case.

For the syntax without endfor the same remarks apply as mentioned for if.

The statement

for i := Expr1 to Expr2 do StatementList endfor;

thus leads to the following construction:

```
Temp := Ord(Expr2)-Ord(Expr1)+1;
while Temp>0 do
  Dec(Temp);
  StatementList;
  Inc(i);
endwhile;
```

Example:

```
for i := i-1 to i+10 do
  Write(' ',i);
endfor;
```

4.14.11 with Statement

The last one of the structured statements, the with statement, sets the validity range (scope) of identifiers. The effect consists that the fields and methods of the structured variables listed within the with statement can be referenced by the mere name of their identifiers. The identifiers in the statements section are then searched in the named spaces of all given variables, starting from the innermost. Field or method names therefore hide all fields, methods, variables etc. of identical names in the enclosing name spaces, to which the access is merely possible with their full names like <module name>.<variable identifier>. The declaration of a list of variables is equivalent to a corresponding number of interlocked with directives, from which follows that the name space of any variable is valid also for all other variables in the list. If the addressing of a record or object variable requires the indexing of an array or the calculation of a pointer address, the regarding action takes place before the statement section, e.g. the assignment to a pointer variable RecPtr in the statement section doesn't change the reference established by with RecPtr^ do.

<with statement> ::= with <record or object variable list> do <statement list> endwith |
with <record or object variable list> do begin <statement list> end

<record or object variable list> ::= <record or object variable> { , <record or object variable> }

<record or object variable> ::= <record variable> | <object variable>

For the syntax without endwith the same remarks apply as mentioned for if.

Example:

```
with Data do
  stR.bA3 := 5;             { Data.R.A3 }
  abB[2] := 0AAh;           { Data.B }
endwith;
pstRecPtr := @stRec1;
with pstRec^ do
  bA := 0;                  { Rec1.A }
```

```
  pstRec := @stRec2;
  bA := 0;                  { still Rec1.A, not RecPtr^.A !!! }
endwith;
```

4.15 Procedure Declarations

With procedure declarations whole program parts are separated from the calling program and given an identifier by which they can be invoked any time.

<procedure declaration> ::= <procedure heading> { <procedure modifier> ; } <block> |
<procedure heading> forward ; | <procedure heading> external <unsigned integer> ;

<procedure modifier> ::= WITHOPT

<block> ::= { <definition and declaration part> } <statement part>

<definition and declaration part> ::= <constant definition part> | <type definition part> |
<variable declaration part> | <procedure and function declaration part>

The procedure header consists of the identifier with which the procedure is called and - provided that available - the definition of the formal parameters which takes over the role of local variables. The declaration with forward allows to declare the procedure block with an abbreviated or complete header in later passages of the module which is necessary particularly at procedures and functions that invoke each other. All procedures, functions and methods in the public section of modules (public) and in the type definition of objects are implicitly declared forward unless they are explicitly declared external. A procedure block therefore can be placed exclusively also in the non-public part of modules (private). The declaration with external <unsigned integer> indicates the compiler that the regarding procedure, function or method doesn't exist in POOL code, but has to be invoked as a C function with the given index. Due to different invoking mechanisms no addresses can be gained from external procedures etc., they can not be used as virtual methods and after forward (explicit or implicit) external is no longer allowed.

The listed procedure modifiers are some kind of compiler switch with a restricted domain on the procedure in question.

<procedure heading> ::= procedure <identifier> ; |
procedure <identifier> (<formal parameters>) ;

<formal parameters> ::= <formal parameter declaration> { ; <formal parameter declaration> } | {
<formal parameter declaration> ; } ..

<formal parameter declaration> ::= <parameter list> : <type identifier> | var <parameter list> :
<type identifier> | var <parameter list>

<parameter list> ::= <identifier> { , <identifier> }

A parameter group without a preceding specification contains only members who are submitted by their value alone (val parameter). Parameters with the specification var are only referenced by their address and also may be untyped. As the last or the only one parameter **..** can represent an untyped parameter list. Only untyped parameters can remain empty at the call. Untyped var parameters are submitted as val parameters <parameter identifier>: tstUArg, for parameter lists in addition to that one val parameter pstVAStart: tpstVArg.

Hint for C programmers

Different from C, where parameters can be submitted only "by value", what requires the use of pointers in many cases, POOL provides var parameters. E.g. in such cases where pointers have to be used in C there is no necessity in POOL or the pointer has to be de-referenced.

The definition and declaration part of procedures contain information which have - like the formal parameters - an only local validity range within the procedure.

Local variables initialise always to zero, i.e. strings are empty, Booleans are „false", pointers are „nil", integers and reals are 0. Although this initialisation costs processing power, the decision was made at the conception of the POOL system to avoid harmful mistakes from un-initialised data structures that often happen in practise. One of the major reasons for this decision was primarily the introduction of dynamic strings at which the access would lead to fatal aborts due to un-initialised internal pointers.

<constant definition part> ::= <empty> | const <constant definition>{; <constant definition>};

<type definition part> ::= <empty> | type <type definition>{; <type definition>};

<variable declaration part> ::= <empty> | var <variable declaration>{; <variable declaration>};

The statements section finally contains the executable subroutine of the procedure. The call of the procedure within its own statements section leads to recursion and is expressly allowed.

Examples:

```
procedure vCountIntParams(..);
var
  n: Int16;
  pstVT: tpstType;
begin
  while pstVAStart<>nil do
    pstVT := pstVAStart^.pstType;
    if (pstVT<>nil) and (pstVT^.enTypeClass=nenTCInteger) then
      Inc(n);
    endif;
    pstVAStart := PVANext(pstVAStart);
  endwhile;
  { Rem.: This version does not support embedded lists or untyped }
  {       var parameters, so normally VAFirst and VANext is used! }
  Writeln ('vCountIntParams: ',n,' integer parameters found');
end; { vCountIntParams }
```

```
procedure vUpCaseStr (var cs: CharString);
var i: tSize;
begin
  for i := 0 to Length(cs)-1 do
    cs[i] := ChUpper(cs[i]);
  endfor;
end; { vUpCaseStr }
```

4.16 Function Declarations

Functions represent - exactly like procedures - subroutine parts which, however, give back a calculated value, unlike the latter. Function values therefore can be elements of terms or can be assigned to variables of equal type. Assignments to a function identifier are allowed only within the function themselves, this respectively last assigned value before the end of the statements section is then the result of the function. So at least one assignment should occur in every possible branch of the statements section, yet this is not checked by the compiler at present. The function declaration corresponds exactly to the declaration of the procedures, with the exception that an additional result type must be specified.

<function declaration> ::= <function heading> { <function modifier> } <block> |
<function heading> forward ; | <function heading> external <unsigned integer> ;

<function modifier> ::= BCSTR | IFR | WITHOPT

<function heading> ::= function <identifier> : <result type> ; |
function <identifier> (<formal parameters>) : <result type> ;

<result type> ::= <type identifier>

The result type of a function is restricted to unstructured types and the string type. The use of a function within its own statements section leads to recursion and is expressly allowed.

The given function modifiers are a kind of compiler switch with a restricted domain on the function in question.

The modifier BCSTR indicates functions with parameters which are defined by the type CharString which, however, shall work also with ByteStrings (e.g. for Length, Copy, Delete etc.). A function result defined as CharString is then assignment compatible to ByteStrings, too. In doubt the result of such a function remains a CharString, though.

The modifier IFR indicates functions, their use as procedures shall be allowed, that is without the use of the function result (IFR = **I**gnore **F**unction **R**esult).

Examples:

```
function i16Max (a,b: Int16): Int16;
begin
  if a>b then i16Max := a
  else i16Max := b; endif;
```

```
end; { i16Max }
function  Copy (bcs: String; xoIndex, xoCount: tSize): String;
  external 1; bcstr;
```

4.17 Method Declaration

The declaration of a method within an object type (class) corresponds, except as external, to a forward declaration, the just defined object type is yet allowed as a parameter.

After the declaration of the object type any method has to be implemented by it's qualified method identifier (<object type identifier>.<method identifier>) within the same scope, but outside of the object type definition in the private section of the module. The implementation follows the rules of procedure or function declarations, where the keyword procedure has to be replaced by constructor or destructor for constructors and destructors and virtual methods have to be marked with the keyword virtual.

<method declaration> ::= <method heading> <block> | <method heading> virtual ; <block>

<method heading> ::= function <qualified method identifier> ; | function
<qualified method identifier> : <result type> ; |
function <qualified method identifier> (<formal parameters>) : <result type> ; |
procedure <qualified method identifier> ; | procedure <qualified method identifier>
(<formal parameters>) ; | constructor <qualified method identifier> ; | constructor
<qualified method identifier> (<formal parameters>) ; | destructor
<qualified method identifier> ; | destructor <qualified method identifier>
(<formal parameters>) ; |

<qualified method identifier> ::= <object type identifier> . <method identifier>

All methods are automatically (but hidden) declared with the parameter oSelf before all given parameters. At every call one object of the current or the derived type is submitted like a var parameter (var oSelf:<object type identifier>). Within the entire method block all identifiers first are searched in the name space of oSelf, as if the method block resided in a with directive of the form with oSelf do ... endwith. For that reason oSelf ist not allowed as an identifier for components or parameters and leads to the error message „Duplicate identifier: oSelf".

Static methods can be defined differing by their descendents, but have to remain static. If otherwise the object is submitted as one of it's ancestors, e.g. in inherited methods, in this case the methods of the ancestor will be used!

Virtual methods are different from static methods in a way, that they are usually invoked indirectly over a pointer in the object itself (via the virtual methods table VMT). This mechanism causes, that also out of methods of an ancestor the virtual method of the descendants are invoked, if the current object variable (oSelf) is such a descendant. Virtual methods with a common denominator must therefore be declared also in the complete family tree with an identical head.

4.17.1 Method Calls

Call of dynamic methods:

<instance>.<method_name>: <instance>.VOT^.VMT[method_number] (<instance>)

Call of static methods:

<instance>.<method_name>: <type_of_instance>.method_name (<instance>)

(<type_of_instance> ist the declared type of the instance, regardless from the current value, who can be a descendant!)

Static call of methods within methods:

inherited <method_name>: <type_of_ancestor>.<method_name> (oSelf)

(looks for the latest ancestor that implements the method)

<object_type>.<method_name>: <object_type>.<method_name> (oSelf)

(<object_type> is the current type or type of an ancestor!

A preceded module name is allowed with <object_type> only!)

Call of dynamic methods within methods:

<method_name>: oSelf.VOT^.VMT [method_nummer] (oSelf)

oSelf.<method_name>: oSelf.VOT^.VMT[method_nummer] (oSelf)

Call of static methods within methods:

<method_name>: <current_object_type >.<method_name> (oSelf)

oSelf.<method_name>: <current_object_type >.<method_name> (oSelf)

4.17.2 Constructors and Destructors

Strictly speaking all advantages of object oriented languages are utilizable also without the semantic support of object types. Anyway all structures connected with objects are dissolved internally by the compiler into usual data types as well as pointers to data, functions and procedures. The advantage of the object orientation lies, however, in the more favourable notation and readability of the programs as well as in the submission of standard tasks already by the compiler which must be written down otherwise with difficulty. Nevertheless it must not be forget that objects are data structures themselves, these will completely or partly created and removed again dynamically. Exactly like pointers which must be initialized and accompanying memory areas must be requested, this work must be carried out for objects also. The POOL compiler supports the user by special types of methods with the names constructor and destructor.

Constructors and destructors are special procedures, since for the extended syntax of New and Dispose by implication additional parameters are submitted, and can be used as a function of the type Pointer. In the following description of the calling mechanism poSelf represents the pointer, that will be submitted for the var parameter oSelf.

4.17.2.1 Constructors

As all objects are derived from toRoot, there is at minimum the basic constructor poInit of this object.

The constructor is ***always static***, since at calling time yet no VMT exists (the initialisation of the VOT/VMT in particular is one of the reasons why special constructors must have been introduced)!

Pre-created objects (const or static) are initialised already and can be used immediately for that reason. However, only the fields (including of the hidden VOT field) become pre-created but no constructors are invoked! So pre-created objects are only sensible in simple falls. E.g. mutexes and events are not permissible as pre-created constants which the compiler, however, cannot check!

Not pre-created objects (local ones also) must always become initialized with a constructor call before use and, after which, de-initialised with a destructor call again! Since in POOL all data are erased and the destructor also erases the object, erased data are presumed for the constructor. If the data are not erased, e.g. also at multiple constructor calls without previous destructor call, nil (and EINVAL) is given back as an error and the object does not change (at least if all constructors are built up correctly and fields change only after a successful call of an inherited constructor). In case the data area of an object is still used otherwise, e.g. if the object stands in the variants part of a structure, the application must erase the "superposed" fields explicitly before it calls the constructor!

Any constructor has two invisible parameters in addition to oSelf, namely ?boHeapObj and ?pstVOT, as well as an invisible prologue, that, if applicable, requests memory for the object, and one invisible epilogue, by which the memory for the object is released again.

New (poObj, poInit(...)) compiles to poObj := <current_object_type>.poInit (nil, true, <current_object_tpye >, ...). I.e. poInit will be called by poSelf=nil and ?boHeapObj=true. After that poInit must request the needed memory in the invisible prologue (it's size is ?pstVOT^.xoSize) and must return a pointer to the new released memory area (or nil e.g. at lack of memory). Before return the resulting variable has to be checked as well, and, in case of an error (result = nil), the memory has to be released again. The invisible VOT field and the VMT also initialises in toRoot.poInit (by oSelf.pstVOT := ?pstVOT).

poObj^.poInit (...) compiles to <current_object_type.object_type>.poInit (poObj, false, <current_object_type>, ...), the constructor therefore does not execute additional functions.

The constructor is always a function which gives back a pointer to the object or nil at error. For most of the objects the only possible errors are lack of memory as a result of the call via New (poObj, poInit) or multiple constructor calls without destructors in among them. The result of the constructor at objects not created dynamically(!) can be ignored for objects at which there really are no other errors (therefore the constructor is also marked with "ifr" internally). In methods of the object, however, the result may be never ignored because the methods must work also for objects created dynamically, of course. In any case broader error possibilities should always be documented explicitly at the constructor!

Each constructor has to call explicitly one of the inherited constructors (usually with inherited) before the object can be used (because of toRoot one exists in any case). Since parameters still must be checked in case before and since an object can have several constructors, this cannot be made automatically. At the end of the constructor, however, is checked, whether the object was really initialized, and, if necessary, will be aborted with error message.

Unlike normal pointer functions, where the result is initialised automatically to zero, the result := nil has to be set explicitly, because the invisible constructor prologue leaved the result already set.

Constructor calls for oSelf (e.g. inherited poInit or toRoot.poInit) are allowed in the constructor itself only. Constructor calls with type casts, however, (e.g. tpoRoot(@oSelf)^.poInit) are allowed, but sensible only in extraordinary cases, as here not the type of oSelf, but the casted type is applied in the object.

Constructor calls for oSelf with object type given (e.g. toRoot.poInit) are useful only in case constructors of ancestors shall be skipped. At changes of the object hierarchy those calls possibly have to be adapted by hand and are therefore not recommended!

Local object variables (except static) may be used only within the thread they belong to, in particular the initialisation (constructor call) is allowed in this thread exclusively (e.g. it is forbidden to write the address in a global variable and to call a constructor by this variable out of another thread).

The simplest constructor contains only the call of the inherited constructor, e.g. poInit := inherited poInit (a, b);. Such an (actually empty) constructor is then superfluous, though, too.

After an error from the inherited constructor (e.g. inherited poInit = nil) no destructor must be called, since this would cause an avalanche of unreasonable destructor calls. In advance of the call of the inherited constructor usually no dynamic data become assigned, but if so, they still have to be released.

```
constructor toObj.poInit (i32DTms: Int32);
begin
  { Init of the ancestors }
  if inherited poInit=nil then
    poInit := nil;
    return;
  endif;
  { Own initialisation }
  oSelf.i32DT := i32DTms;
  { Remark: oSelf. can be omitted, but with oSelf the context is much more obvious }
```

```
  { ... (other code) }
end;
```

A constructor in which further (own) errors can occur must tidy up its legacy and in any case also invoke a destructor. Normally the clean up should be done by the destructor call, if not, however, the own destructor should be adequate extended or an equivalent own destructor has to be introduced, respectively!

```
constructor toObj.poInit(i32DTms: Int32);
begin
  { Init of the ancestors }
  if inherited poInit=nil then
    poInit := nil;
    return;
  endif;
  { Own initialisation }
  repeat { (for break at error only) }
    oSelf.i32DT := i32DTms;
    New(oSelf.pabArrayPtr1);
    if oSelf.pabArrayPtr1=nil then
      break;  { Error }
    endif;
    New(oSelf.pabArrayPtr2);
    if oSelf.pabArrayPtr2=nil then
      break;  { Error }
    endif;
    { ... (other code) }
    return; { Init successful }
  until true;
  { Clean up }
  vDone;
  { Error return }
  poInit := nil;
end;
```

4.17.2.2 Destructor

Since all objects are derived from toRoot, there is always at least the base destructor "vDone" from this object. This destructor is virtual and releases all contained dynamic strings if necessary and erases then the memory area of the object variables. However, the destructor of the object which also has created these data must always take care of possible dependent dynamic data structures.

The standard destructor vDone is always the first virtual method of all objects and can be invoked also at not initialised (but cleared) objects, either directly or by Dispose (in which case nothing will be executed). This special treatment primarily makes the clearing works easier, e.g. in vDeinit.

The destructor can be both a ***static*** *or* ***dynamic*** (virtual) method. Like ordinary methods those of same names must have always the same properties. Normally it is not even meaningful to define a static destructor, because it can be guaranteed for virtual methods only that actually the correct method is invoked instead of an ancestor’s method!

In addition to oSelf each destructor has one invisible parameter, ?boHeapObj, as well as an invisible epilogue in which by case the occupied memory for the object is released again.

Dispose (poObj, vDone) is assembled to poObj^.vDone (poObj, true); poObj := nil. poObj^.vDone is therefore called with poSelf = poObj and ?boHeapObj=true. As a result vDone has to release the memory before return.

poObj^.vDone is converted to poObj^.vDone (poObj, false), thus the destructor doesn't execute additional functions.

Any object has to be de-initialised after it's use by a destructor call as well as static an local objects!

The simplest destructor contains only the call of the inherited destructor, e.g. "inherited vDone;". Such a destructor, however, is superfluous in most cases, because the compiler invokes the method of an ancestor automatically, anyway.

Every destructor has to call one of it's inherited destructors (usually by inherited) after finishing it's own clean up work (due to toRoot there is certainly one). Since there can be multiple destructors, this cannot be automated. At the end of the destructor, however, it will be checked, if the object has been de-initialised, otherwise an error abortion occurs.

```
destructor toObj.vDone;
begin
  { Clean up }
  Dispose (oSelf.pabArrayPtr1);
  Dispose (oSelf.pabArrayPtr2);
  { ... (other code) }
  { Done of the ancestors }
  inherited vDone;
end;
```

4.18 Program Modules

Data, procedures and functions are summarized to so-called modules. Modules have the task of implicitly extending all identifiers declared in them by the module name. Through this there is a distinction possibility between identical named identifiers. Such extended identifiers are called qualified identifiers in general.

<module> ::= <module heading> <block> end.

<module heading> ::= module <module identifier> ;

The referencing of qualified identifiers from other modules happens similar to the referencing of record fields, see chapter 4.11 'Use of variables, fields and constants'.

Example:

```
module TEST_BENCH;              {Declaration}

import
  IBUS, DIAG;                   {In this module needed modules}

procedure vStart;               {Exported prototype}

private                         {Start of the private section}

procedure vStart;               {Statements section of the several functions}
begin
  ....
end;

begin                           {Initialisation part of the module}
end. {TEST_BENCH}

-----

TEST_BENCH.vStart;              {Referencing of a function/procedure}
```

Modules are not tied to files in principle. The current implementation of the compiler expects, however, only one module per file. Module and file name should agree fore the sake of simplicity here. The statements section of the module itself serves to initialization.

4.18.1 Validity Scopes

The validity scopes of identifiers within modules can be restricted.

<scope> ::= <public scope> | <private scope>

<public scope> ::= public <block>

<private scope> ::= private <block>

This allows the declaration of identifiers, which either can be referenced only within a module (private) or also from outside of the module (public). Without a special declaration the identifiers of a module are treated as public automatically. As long as a procedure, function or method is not declared as external, the public part contains only its header. The corresponding block, with a complete or abbreviated header, resides in the private part of the module.

4.18.2 Import of References from other Modules

In order to make identifiers of other modules referencable, the foreign module must be made accessible to the invoking module.

<import directive> ::= import <module identifier> { , <module identifier> } from <file name>

<file name> ::= <char string>

The path given in <file name> is a constant string and may contain relative references only. Absolute references (containing drive or host names, e.g. c:/myproject/test.pi or //server/project/modules/test.pi) are possible only by the use of environment variables (see below), so that POOL projects go on working, even if they have been copied to other computers, drives or directories. If there is no file name given, the module name in lower case letters will be inserted and extended by ".pi".

For compatibility reasons with other file systems the given path must not contain any \ (backslash), no : (colon) nor other special characters, but has to use / (slash) as a directory separator.

In the path declaration environment variables can be inserted by using the for the AIDA system usual notation $(<env_var_name>) or $(<env_var_name>/). The second form inserts a directory separator only if necessary, because at e.g. $(<env_var_name>)/test.pi the file would be searched in the root directory if the environment variable was empty. The environment variables, however, may be declared in the system conform notation, i.e. in this case they may contain also the there decisive special characters.

Examples:

```
import TEST from “test.pi”
import TEST from “../modules/test.pi”
import TEST from “$(ProjectPath/)test.pi”
  { where here ProjectPath is the name of an environment variable,
    which e.g. could refer to “C:\myproject\modules\” }
```

5 Compiler Instructions

For the control of the compiler itself instructions beyond the POOL instruction set are used. The compiler instructions can be divided up into three classes: Switches, parameters and conditions. Syntactical a compiler instruction is a special comment, in which immediately after the opening comment bracket ({) the dollar sign ($) follows. Next to that the name of the instruction follows, if applicable combined with switches or parameters. For simple commands may be further instructions given, separated by commas, before the closing comment bracket (}) terminates the entire command. All instructions must be written in lower case letters.

Some few of the compiler instructions can be utilised as function/procedure modifiers. Syntactically their usage follows the rules of the other modifiers virtual, external and forward, respectively applied to the function/procedure header. These compiler instructions then are effective only locally referred to the accompanying function or procedure.

5.1 Switches

$cbe <+|–>

The switch determines Boolean expressions to be evaluated completely. In contrast to that stands the so called short cut evaluation which terminates as soon as the result of the expression is not longer affected by further evaluation. The basic setting for this parameter is {$cbe-}.

$coct <+|–>

This switch determines the compiler to interpret integer numbers with leading zeros as octal values {$coct+} or to reject them as an error {$coct-}. Pre-setting is {$coct-}.

Remark: The use of coct is basically not recommended. POOL supports an unambiguous signing of octal numbers using the suffixes o or O and q or Q (Q is supported alternatively because it looks somewhat similar to O but cannot be mixed up as easy with 0 (Zero)).

$zstr <+|–>

This switch determines if the compiler allows {$zstr+} assignments of strings to variables of the type array[0..n] of Char or if they are rejected incorrect {$zstr-}. Default setting is {$zstr-}.

$ifr <+|–>

This switch decides if the compiles allows functions to be used as procdures {$ifr+} or rejects them incorrect {$ifr-}. Default setting is {$ifr-}. The name stands for 'Ignore Function Result'.

In addition to that this switch is usable also as a function modifier.

Remark: In order to avoid the erroneous usage of functions, e.g. like ignoring returned error conditions, this compiler switch better should not be used. Reasons:

1. Different from C there are not many functions in POOL where it seems sensible to ignore the function result.
2. Those few functions whose results are reasonably ignored (e.g. WriteStr etc.) are already marked ifr.
3. In particular cases any function can be used as a procedure using the type cast „procedure(..)“, which moreover documents that the function result was ignored deliberately.

5.2 Parameters

$i <file_name>

This instruction forces the compiler to include the file named by <file_name>. All content of this file will be inserted beginning from this instruction into the compiling source code. No further compiler instructions are allowed here within the same brackets. If <file_name> has no file extension, <file_name>.pool is assumed automatically. For the file names the same rules apply as for <file_name> in the import instruction.

There are three limitations in the use of include files, mainly to avoid erroneous or incomplete include files from causing error messages in other files.

1. private and public can be used as usual, but have no influence to the including file.
2. The reading file must contain a complete program block (e.g. complete function or procedure).
3. Every with {$if...} opened section has to be closed within the same file, i.e. every include file must contain to each {$if...} the corresponding {$endif}.

$stklen <len>

Using this instruction a developer can decide how much of stack space the current module may need. If at runtime, e.g. caused by recursion, more stack memory is needed than given, the program will abort with a runtime stack error.

Remark: As the user can not estimate the needed stack size anyway this adjustment should be reserved to the runtime system, that is this parameter should not be used. The existence of this parameters is justified only for the use in „small“ (embedded) Systems.

5.3 Conditional Compilation

$define <symbol_name>

$else

$endif

$ifdef <symbol_name>

$ifndef <symbol_name>

$undef <symbol_name>

The instructions for conditional compilation allow parts of the source code to be included or excluded depending on the existence of the given symbols. The syntax resembles the if instruction of POOL, where the symbols represent variables of the type Boolean which become assigned the values true or false by {$define ...} or {$undef ...}, respectively. Corresponding to that {$ifdef ...} or {$ifndef ...} are interpreted as if ...=true resp. if ...=false.

Example:

```
{$ifdef Debug}
  Writeln('i=',i);
{$endif}
{$ifdef LittleEndian}
  type
    tstWordRec = record bLo,bHi: Byte end;
{$endif}
```

5.3.1 Standard Symbols for Conditional Compilation

BigEndian

Defined if the compiler runs on a “high byte first” byte order machine (e.g. Motorola 68000 CPU).

DOS

Defined if the compiler runs on a DOS machine.

LittleEndian

Defined if the compiler runs on a “low byte first” byte order machine (e.g. Intel 80x86 CPU).

MiddleEndian

Defined if the compiler runs on machine with this special byte order (e.g. DEC Alpha CPU).

POOL

Always defined by the POOL compiler.

UNIX

Defined if the compiler runs on a UNIX machine.

6 The POOL System Module

The system module holds the basic data types, procedures and functions, which are not part of the language itself but are unrenounceable and available to every module. It resides in lib/pool.pi and will be imported to every other module automatically. All other libraries must be included manually. The public definitions (data types, procedures, functions etc.) of all provided libraries can be retrieved from the online documentation.

They can also be found in the corresponding POOL include files (lib/pli/*.pli).

Note: These files are provided only as a reference, but will **never** be included to other modules (unlike C header files)!

7 Internal Data Representation

7.1 Boolean

The data type Boolean will be stored in a single byte and can hold the both values "false" (=0) and "true" (=1).

7.2 Char

Values of the type Char claim exactly one byte data space like the type Byte (thus unsigned). Value range Chr(0) .. Chr(255).

7.3 Int8

The type Int8 is a signed byte between -128 until +127.

7.4 Byte

Byte types are unsigned values between 0 and up to 255.

7.5 Int16

The type Int16 covers two bytes and represents signed values between -32768 and 32767.

7.6 Word

Values of the type Word are the unsigned correspondence to Int16 in the range of 0 .. 65535 and covers two bytes, as well.

7.7 Int32

Int32 types cover four bytes, they are signed values and provide a range of -2147483648 .. 2147483647.

7.8 DWord

DWords cover 4 bytes like Int32 values and represent the range of 0 .. 4294967295.

7.9 Real32

The data type Real32 covers 4 bytes and consists of a normalised mantissa (23 Bits), a signed exponent (8 Bits) and a sign (1 Bit).

Value range about $1{,}5 \cdot 10^{-45}$ up to $3{,}4 \cdot 10^{38}$, resolution 7 to 8 digits, roughly.

7.10 Real64

Values of the data type Real64 claim 8 bytes. They consist also of a normalised mantissa (52 Bits), a signed exponent (11 Bits) and a sign (1 Bit).

Value range about $5{,}0 \cdot 10^{-324}$ to $1{,}7 \cdot 10^{308}$, resolution 15 to 16 digits, roughly.

7.11 Strings

Strings in general store chains of elements of the same type (e.g. Char, Byte ...). They also hold a length information. As POOL deals with dynamic strings, the string data is not stored in the string variable itself, string variables moreover are represented by a fixed data structure, that contains certain flags, the length information and also a pointer to the data contents. Direct accesses to this structure from POOL are not legal. The single elements of the strings are accessed as usual like an array [0..Length(Str)-1] of <element type>, the direct addressing over the pointer will be calculated by the compiler automatically. With e.g. @Str[i] it is made possible to retrieve the address of a string element and the subsequent use as a pointer. Hereby it has to be considered that each string operation may move the data content in the memory, by which the pointer becomes invalid and further accesses might cause severe errors or program abortions.

7.12 Arrays

Arrays cover as many data bytes in memory as the component type covers multiplied with the number of components. Those components with the lowest index reside in memory at lowest addresses.

7.13 Records

Data structures of the record type need as many data bytes as the sum of the memory consumption of the individual field types. Fields declared first reside in memory at lowest addresses.

7.14 Objects

Objects claim memory for their data in the same manner like records. Additionally objects maintain a pointer to their type descriptor (VOT) by which also the corresponding table of methods can be accessed, which again consists of pointers to the object's virtual methods.

7.15 Pointers

Pointer types carry the addresses of the variables they point to. They cover thus 4 bytes.

8 Formatting and Nomenclature

POOL follows just Pascal as an example in order to produce an easy to understand and an also well readable source code. The especially by “real programmers“ beloved „compact“ writing style, often coming across in C, was absolutely not intended for the development of POOL. Besides the semantic rules POOL gives no regulations how to write source code. Thus it’s possible to write rather awful code in POOL either. A unique nomenclature at the naming of the identifiers as well as a consequent formatting of the source code is a valuable contribution to the readability and clearness of the programs. For that reasons it is more than strongly recommended to hold on to the following rules.

8.1 Source Code Formatting

8.1.1 Headlines

The use of headlines enables the author to include additional information to the subsequent source code, for which there is nor the suitable room neither the suitable context within normal comments.

8.1.1.1 Module Header

The module header contains information for the entire module. Indispensable parts are title, module description and an up-to-date change list. The change list comprises of version, date of change, author and a textual description of the changes. The change list has to be ordered in a way that the latest changes lie on top of the list. This avoids at “mature” modules the unpleasant necessity of passing the whole history before finding the more interesting latest descriptions of changes.

8.1.1.2 Functions / Procedure / Method Headers

Any not trivial subfunction has to be described. At more complex subfunctions it can be even advisable to maintain the latest change description or a complete private change list. Especially public procedure and functions must have an abstract description to show them in the browser list.

8.1.2 Comments

The general rule reads as follows:

„Comment plentiful, but not senseless.“

That is, each not trivial source line should have an explaining comment. As a rule the comment is placed at the end of the source code line. Sometimes it happens to be sensible to comment a couple of line at once. In this case it is recommended to place the comment as a separate line on top of the succeeding source code lines.

8.1.3 Blank Lines

For the optical structure of the source text it is useful to insert blank lines at suitable positions. The only objective is a good readability, thus here are still some recommendations given:

Two blank lines should be inserted only on global, resp. top level hierarchy, e.g. before the declaration part, consisting of const, type and var blocks, before private and before subfunctions of highest level.

One blank line should be inserted between the const, type and var blocks of the declaration part and to separate local subfunctions.

No blank lines should be inserted between the several declarations and instructions, no between local declaration and instruction parts, not necessarily between prototypes, unless they contrast thematically.

8.1.4 Indentation

Indentations have to be done in POOL in general by **two blanks**. Although POOL hereby gives no regulations this has been proven a good compromise between distinctive separation and good readability on the other hand.

8.1.4.1 Modules

The module keyword module is always placed to the first column of the source code. The same applies to includes and the section designators privat and public.

8.1.4.2 Declaration Area (const, type, var)

The declaration area designators const, var and type are placed on the same indentation level as the corresponding context, i.e. the designators which belong to the module context or to subfunctions of the highest level start in the first column. In contrast to that the actual declarations will be indented by two blanks.

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

```
var
  i:   Int16;
  r:   Real64;
  i32: Int32;
```

The : (colon) of the variable declaration are considered to be more related to the symbolic name of the variable than to the given type (this is in assimilation to the label writing style in assembler language – particularly this is being expressed in some case constructs). For this reason there is no space between the variable name and the colon, but at minimum one blank between the colon and the given type. POOL itself make here no regulations and accepts of course also the sometimes the popular style where the colons - as it were a vertical line - always stand one below the other.

8.1.4.3 Structures

Structures indent hierarchical corresponding to their construction, i.e. each substructure indents by two blanks to the right. The for records imperative end places in the same column as the corresponding identifier. The keyword variant places to the same column as variables of the same level, but the variants themselves indent in relation to the keyword by two columns. Since the variants themselves must be inserted enclosed in brackets and since lines should never begin in even columns, one blank each should be inserted after the opening or before the closing bracket.

Example:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890

var
  stRec1: record
    i32Alpha: Int32;
    variant
      ( a,b,c: Int16 );
      ( d,e,f: Word );
      ( stRec2: record
          i,j,k: Int16;
          r: Real64;
        end )
  end;
```

8.1.4.4 Objects

Objects, as also being structured data, get the same construction like records. The methods that belong to the object start from the highest level column of the object's data structures, i.e. they are indented by two columns in relation to the identifiers of the object.

8.1.4.5 Procedures, Functions, Methods

The formal declaration of the procedure and function head lines starts from the first column of the source code, unless it is about a pure local subfunction. In the last case the entire construct will be indented by two columns in relation to the higher context.

The unavoidable begins and ends of any subfunction place to the same column as the head line. In relation to that the lines of the instruction block are indented by two columns.

For a better readability blank characters are inserted before any opening bracket, after each ; (semicolon) at the declaration of formal parameters and after each : (colon). No blank characters should be inserted before a : (colon) and at the enumeration of identifiers to the same type.

At the call of subfunctions blank characters are inserted before the opening bracket as well as after every single parameter, except before the closing bracket. For functions that represent as it were numerical values within arithmetical expressions the blank character is omitted, e.g. like „Length(String)“.

Example:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890

function i32Work (a,b: Int32; cOp: Char): Int32;

  function i32Add (a,b: Int32): Int32;
  begin
    i32Add := a + b;
  end; { i32Add }

  function i32Sub (a,b: Int32): Int32;
  begin
    i32Sub := a - b;
  end; { i32Sub }

begin { i32Work }
  case cOp of
    'a': i32Work := i32Add (a, b);
    's': i32Work := i32Sub (a, b);
  else
    Writeln ('?i32Work: Illegal Operator!');
  endcase;
end; { i32Work }
```

8.1.4.6 begin / end

Corresponding begins and ends start always in the same column. From that follows informally that all end designators of blocks who possess implicit begins start in the same column as the block instruction (*see* while, for *etc.*).

8.1.4.7 if Constructs

The three possible elements of the if construction (if – else – endif) start each at the same column. In relation to that the enclosed instruction blocks of the if branch and the else branch are indented by two characters. By the way, it's recognized a good style to repeat parts of the decisional expression as a comment behind the closing endif. Particularly at heavy nested if constructs this improves the readability of the source code immensely. The same applies to the else line at complex constructions. In the trivial example below in practice one would better do without for both.

Remark: Please note that in contrast to Pascal (or C) the unpleasant, mostly three lines occupying „end/else/begin" constructs, in particular due to the in POOL imperative endifs, are not applicable and thus the source code becomes much easier readable.

Example:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890

function i32Work (a,b: Int32; cOp: Char): Int32;
begin
  if cOp = 'a' then
    i32Work := i32Add (a, b);
  else { op <> 'a' }
    i32Work := i32Sub (a, b);
  endif;  { op <> 'a' }
end; { i32Work }
```

8.1.4.8 while, for, repeat

For all loop constructs the instruction block is indented in relation to the loop instruction by two columns. The coupled loop instructions (for – endfor, while – endwhile, repeat – until) each place to the same column.

Example:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890

function i32Multiply (a,b: Int32): Int32;
var i32Res: Int32;
begin
  if b<0 then
    b := -b;
    b := -a;
  endif;
  while b > 1 do
    i32Res := i32Res + a;
    b := b - 1;
  endwhile;
  i32Multiply := i32Res;
end; { i32Multiply }
```

8.1.4.9 case

On highest level the case construct is established like the if construct, where the case branch accommodates all fulfillable conditions and the else branch all other not covered cases. From that follows an indentation by two columns analogous to the if construct for both instruction blocks. For the case constructs different cases have to be considered:

- There is a short case label and the case has one instruction only, then everything is written to one line.
- The case label is short, but the case consists of more lines, the begin is written in the label's line, the following instruction block is indented by ***four*** columns and the terminating end indents by two columns in relation to the case label.
- There is a long case label, e.g. it consists of several components, then the corresponding instruction block begins in the following line and indents by two columns. If there is a begin-end clause necessary the enclosed instructions become indented another two columns.

Aim of all the given formatting rules is to make both the case construct clearly visible and also to emphasize the single case labels by un-indentation in relation to the instruction parts. The below described case ‚a' one better would write in practice in only one line.

Example:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890

function i32Calculate (a,b: Int32; cOp: Char): Int32;
begin
  case cOp of
    'a', 'A', '+':
      begin
        i32Calculate := i32Add (a, b);
      end;
    's', 'S', '-': i32Calculate := i32Sub (a, b);
  else
    Writeln ('?i32Calculate: Illegal Operator!');
  endcase;
end; { i32Calculate }
end;
```

8.2 Nomenclature

8.2.1 Case Sensitivity

In contrast to Pascal POOL makes extensively use of case sensitive spelling. In the treatment of identifiers and reserved words, however, there are some distinctions.

8.2.1.1 Reserved Words

All reserved words are written like described in the POOL vocabulary, i.e. the designators of standard types in mixed spelling, all others general in lower case letters.

8.2.1.2 Identifier

Identifiers are written in mixed spelling, as possible without inserted _ (underbars), unless the underbar shall act as a distinctive separation or the identifier shall obtain a module name extension or one of the agreed suffixes shall be separated. Instead of the dividing underbar the single name parts have to be headed with upper case letters. Very popular, but nevertheless unreasonable is the practice to separate identifiers in upper case letters headed name parts, although the notion is of one word only in common language. Such naming has to be avoided (wrong: ~~stMailBox~~, correct: stMailbox). For all identifiers which represent data contents the below listed prefixes have to be used without any exception. Since prefixes always spell in lower case letters the heading characters of the actual name of the subsequent identifier must spell at least with one capital letter.

Excepted from the duty to use prefixes and to keep case sensitivity are trivial names only, e.g. local temporary, counting or indexing variables, that are popular to be named by i, j, k; a, b, c; l, m, n or x, y, z. Those identifiers must never act as global names in the source code!

Example:

```
bTerminal15
nFlag_Msk                   { _ due to the trailing suffix }
AWS_pstErrorMessage         { _ for the leading module name }
```

8.2.2 Prefixes

For the type characterisation of identifiers unified prefixes have to be used. Prefixes in general are spelled in lower case letters and lead the actual name without any additional separators. At identifiers of complex contexts prefixes are combined where the prefix of the most abstract designation places to the leftmost position. Used in this way prefixes allow the knowledge of every variable’s type by simply knowing it’s name. Since the use of prefixes is a very common stylistic element in the development of Unix software, some few of the prefixes show distinct influences of C data types. BSK applies prefixes consequently in the POOL library and in all the example programs. Their use is very well-tried and is thus strongly recommended to every user. In particular the use of prefixes helps the user by the assignment of identifiers for his data structures, because the actual name part can be maintained, e.g. at the naming of a structure and the pointer to it.

In case the actual name of the identifier stands for a POOL base type, the prefix for that type is omitted.

Example: ‚tpaByte' (not ~~tpabByte~~) as type of a pointer to a common array of bytes;

Counterexample: ‚tpabCount' must keep the ‚b' at the end of the prefixes in order to show that any single ‚Count' is of type byte.

The use of prefixes for identifiers of constants in general is left up to the user. E.g. the circular constant ‚PI' in correct writing had to be written ‚nrPI', but this would never improve readability nor adds something important to the naming. However, here also the use of suffixes is basically recommended, where constant values should lead in with the prefix‚n'.

Constant values which could be also variables by their meaning an usage, do not obtain this ‚n' in order to be re-definable if needed without the necessity to be sought for the use of the ‚n' prefix.

Pure ordinal constant values are designated by ‚n' only.

Local variables with trivial names like the popular ‚i', ‚j', ‚k' as indexing variables may be continued written in lower case letters and do not obtain a prefix, of course.

The following prefixes are agreed:

a..	array of ..
b	Byte
bo	Boolean
bs	ByteString
c	Char
cs	CharString
dw	DWord
en	(enumerating value)
fb	File of Byte (binary file)
fc	File of Char (text file)
fi	File
h	tHandle
hs	S(h)ort String (zero terminated array of short char)
i8	Int8
i16	Int16
i32	Int32
n..	(constant value)
ls	(reserved) Long String (zero terminated array of long/wide char)
o	object
p..	Pointer (to ..)
pv	untyped Pointer (*void*)
qw	QuadWord
r	(arbitrary real)
r32	Real32
r64	Real64

st	record (*structure*)
t..	type (of ..)
uc	(reserved) UChar (*unicode* char)
un	variant (*union*)
us	(reserved) UCharString, WideString (*unicode* string)
w	Word
ws	WordString
x	(arbitrary type)
xo	(arbitrary ordinal)

Examples (see Constant Definitions):

```
tpastScope:   a Type of a Pointer to an Array of Records defining a Scope
tenNPRes:     an Enumeration Type (Result of function NormPath)
nenNPOverrun: an Enumeration Element named NPOverrun
enNPRes:      an Enumeration Vaiable
tpapaChar:    a Type of a Pointer to an Array of Pointers to Arrays of Characters
```

9 Literature

[1] **Wirth, Niklaus**
Compilerbau : eine Einführung - Stuttgart : Teubner, 1977.
(Leitfäden der angewandten Mathematik und Mechanik ; Bd. 36)
(Teubner Studienbücher : Informatik)
ISBN 3-519-02338-5

[2] **Jensen, Kathleen - Wirth, Niklaus**
PASCAL : user manual and report.
"Springer study edition."
ISBN 0-387-90144-2

[3] **Borland GmbH**
Turbo Pascal 6.0 Manuals

[4] **Dick Pountain**
Schweizer Mittsommernachtstraum
Niklaus Wirths Programmiersprache Oberon
from c't 1991, issue 11, page 164
by permission of Byte 2/91, McGraw-Hill, Inc.

[5] **Albrecht, Harald**
Debug-Informationen von Turbo-C und Turbo-Pascal
from c't 1990, issue 3, page 361

[6] **Jobst, Fritz**
Compilerbau : Von der Quelle zum professionellen Assemblertext
(Hanser Programmtexte)
ISBN 3-446-16095-7

[7] **Kernighan, Brian W. - Ritchie, Dennis M.**
The C programming language.
Prentice-Hall International, Inc.
ISBN 0-13-110163-3

[8] **Borland GmbH**
Turbo C++ 3.0 Manuals

[9] **Arno Keppke**
Just in Time
Was Echtzeitbetriebssysteme von gewöhnlichen unterscheidet
from c't 1992, issue 8, page 52

[..] other

10 Index

L

M

N

O

P

R

S

T

U

V

W