The Database Programming Language DBPL Rationale and Report

Joachim W. Schmidt
Florian Matthes

The database programming language DBPL is based on notion of bulk type and iteration abstraction, supports data persistence and transaction procedures, and has Modula-2 as its algorithmic kernel. This document describes the rationale behind DBPL and defines the elements of the language.

Status: Final, June 1992

Arbeitsbereich DBIS
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln Straße 30
D-2000 Hamburg 54
Federal Republic of Germany

This work was supported by the European Commission, Basic Research, Contract # 4092.

Contents

1 The Rationale behind DBPL **3**
- 1.1 On Data-Intensive Applications . . . 3
- 1.2 Conceptual Foundations of DBPL . . . 4
- 1.3 Language Design Considerations . . . 5
- 1.4 The DBPL System and its Use . . . 6

2 Syntax **8**

3 Vocabulary and representation **8**

4 Declarations and scope rules **10**

5 Constant declarations **11**

6 Type declarations **11**
- 6.1 Basic types . . . 12
- 6.2 Enumerations . . . 13
- 6.3 Subrange types . . . 13
- 6.4 Array types . . . 13
- 6.5 Record types . . . 14
- 6.6 Set types . . . 15
- 6.7 Pointer types . . . 15
- 6.8 Procedure types . . . 16
- 6.9 Relation types . . . 16
- 6.10 Selector types . . . 17
- 6.11 Constructor types . . . 18

7 Variable declarations **18**

8 Expressions **19**
- 8.1 Access expressions . . . 19
- 8.2 Selectors . . . 20
- 8.3 Constructors . . . 21
- 8.4 Operands . . . 21
  - 8.4.1 Designators . . . 22
  - 8.4.2 Aggregate expressions . . . 23
  - 8.4.3 Relation expressions . . . 23
- 8.5 Operators . . . 24
  - 8.5.1 Arithmetic operators . . . 25
  - 8.5.2 Logical operators . . . 25
  - 8.5.3 Set operators . . . 26
  - 8.5.4 Relational operators . . . 26

9 Statements **27**
9.1 Assignments . . . 28
9.2 Procedure calls . . . 29
9.3 Statement sequences . . . 30
9.4 If statements . . . 30
9.5 Case statements . . . 31
9.6 While statements . . . 31
9.7 Repeat statements . . . 32
9.8 For statements . . . 32
9.9 Loop statements . . . 33
9.10 With statements . . . 33
9.11 Return and exit statements . . . 34

10 Selector declarations **34**

11 Constructor declarations **35**

12 Procedure declarations **36**
12.1 Formal parameters . . . 37
12.2 Standard procedures . . . 39

13 Modules **40**

14 System-dependent facilities **43**

15 Processes **44**
15.1 Creating a process and transfer of control . . . 45
15.2 Device processes and interrupts . . . 45

16 Compilation units **46**

Bibliography **47**

1 The Rationale behind DBPL

The DBPL language orthogonally integrates sets and first-order predicates into the strongly and statically typed programming language Modula-2. The DBPL environment supports the language with full database functionality including persistence, query optimization and transaction management. The application of modern language technology to database concepts results in new insights into the relationship between types and schemas, expressions/iterators and queries, selectors and views, or functions and transactions. Furthermore, it allows the exploitation of type theory and formal semantics of programming languages and thus connects database application development with results from program specification and verification.

Section 1 reviews the rationale behind the design of the language and the implementation of the DBPL environment. The subsequent sections define the elements of the language. This definition is not intended as a programmer's tutorial (see [SM91a, MS92, MSS91]). It is intentionally kept concise, and in a sense, minimal. Its function is to serve as a reference for programmers, implementors, and manual writers, and as an arbiter, should they find disagreement.

The first DBPL language definition was published in [EEK+85]. A later revision [SEM88] removed several limitations on the orthogonality of language constructs. The current version of the report corrects minor ambiguities and errors of its predecessor and is based on the Modula-2 Report by N. Wirth as published in N. Wirth: Programming in Modula-2, Springer-Verlag, Berlin, Heidelberg, New York, 3rd Edition, 1985. All modifications to the Modula-2 Report are indicated at the margins ($\top$, $\bot$).

1.1 On Data-Intensive Applications

The development of data-intensive applications, such as information systems or design applications in mechanical or software engineering, involves requirements modeling, design specification, and, last but not least, the implementation and maintenance of large database application programs. Although the database programming language DBPL concentrates on the last issue and offers a uniform and consistent framework for the *efficient implementation* of entire database applications, DBPL was designed, extended and evaluated with a declarative style in mind.

Data-intensive applications may be characterized by their needs to model and manipulate *heavily constrained* data that are *long-lived* and *shared* by a user community. These requirements result directly from the fact that databases serve as (partial) representations of some organizational unit or physical structure that exist in their own constraining context and on their own time-scale independent of any computer system. Due to the size of the target system and the level of detail by which it is represented, such representational data may become extremely voluminous – in current data-intensive applications up to $O(10^9)$ or even higher. In strong contrast to the need for global management of large amounts of *representational data*, data-intensive applications also have a strong demand to process small amounts of local *computational data* that implement individual states or state transitions.

In essence, it is that broad spectrum of demands – the difference in purpose, size, lifetime, availability etc. of data – and the need to cope with all these demands within a single conceptual framework that has to guide the design of a database programming language.

DBPL is a successor of Pascal/R [Sch77] and addresses the above issues by incorporating a set- and rule-based view of extended relational database modelling into the well-understood system programming language Modula-2. DBPL extends Modula-2 into three dimensions:

- bulk data management through a data type *set (relation)*;
- abstraction from bulk iteration through *access expressions*;
- *persistent modules* and *transactions* for sharing, concurrency and recovery control.

1.2 Conceptual Foundations of DBPL

The leitmotiv behind the DBPL design was simplicity and power by orthogonality. DBPL aims at exploiting and *integrating* a solid, well-understood conceptual framework capable to consistently capture the wide range of data-intensive application requirements outline above — and not the desire to implement new theoretical concepts in isolation.

Classical *type systems* and the *relational data model* constitute the cornerstones of the DBPL language design. Both contributions provide an enlightening foundation from a technological as well as from a theoretical point of view. However, the exciting new experiences are made at the borderline where types (and expressions, selectors, functions ...) and data models (and queries, views, transactions ...) meet and have to be understood, one in the light of the other.

During DBPL development, several, mostly unnecessary and ad-hoc restrictions on the programming language as well as on the data model became obvious. Some limitations, as, for example, the lack of data *type orthogonality* (i.e. the restriction to first normal form relations) had already been realized, others, like *orthogonal persistence* (providing persistence for all types, not just for relations) were new for the database community (but already discovered by the persistent programming language people). Other insights as, for example, the relationship between abstract access expressions and queries, iterators and views, or between recursive access expressions and deductive databases, came as surprises, at least to us.

Globally speaking, the DBPL language and system both heavily rely on conceptual and theoretical input from following areas:

Programming Language Foundations: typing, scoping, binding; [MS89, SM90b]

Compilation Technology: type checking, local data flow analysis, intermediate languages, separate compilation; [SM91b]

Extended Relational Modeling: data constructors, bulk operations, query languages and their expressive power;

Query Optimization: transformations at compile- and run-time, cost models, search strategies; [JK83]

Recursive Query Evaluation: fixed point semantics, stratification, recursive rule definition; [ERMS91]

Concurrency Control: multi-level concurrency control and recovery for complex objects; [SM91b]

Distribution Models: transaction procedures, compensating transactions [JGL+88, JLRS88].

In summary, one can say that the DBPL project stayed more or less inside the boundaries drawn by conventional language technology (e.g. by *mathematical* typing schemes) and concentrated on improving the linguistic support for state-of-the-art database technology. Currently, however, we are heavily intrigued by novel programming language concepts (e.g. by *taxonomical* typing schemes [Car88]) and are quite positive that they provide the basis for substantial extensions of database technology [MS91].

1.3 Language Design Considerations

The DBPL language design is also influenced by requirements of the DAIDA environment for database application development [SWBM89, BJM+89, JMW+90] that asks for a target language that makes extensive use of sets and first-order predicates, and provides a complete separation of typing and persistence.

The main guideline for the design of DBPL can be characterized by the slogan "power through orthogonality". Instead of designing a new language from scratch (with its own naming, binding, scoping and typing rules), DBPL extends an existing language and puts particular emphasis on the interoperability of the new "database" concepts with those already present in the programming language. In particular, DBPL aims to overcome the traditional competence and impedance mismatch between programming languages and database management systems by providing

- a uniform treatment of volatile and persistent data (DBPL supports, for example, relational variables local to DBPL procedures or as function parameters);
- a uniform treatment of large quantities of objects with a simple structure and small quantities of objects with a complex structure, as well as
- a uniform (static) compatibility check between the declaration and the use of each value.

Implementation details (e.g., storage layout of records, clustering of data, existence of secondary index structures, query evaluation strategies, concurrency and recovery mechanisms) are deliberately hidden from the DBPL programmer. A key idea in the design and implementation of DBPL is to let the runtime system choose appropriate implementation strategies based on "high-level" information extracted from the application programs. As it turns out, the widespread use of *access expressions* (i.e. first-order logic abstractions of bulk data access) in typical DBPL programs facilitates such an approach.

DBPL also follows Modula-2 by occasionally sacrificing "language orthogonality" out of engineering and efficiency considerations. For example, DBPL does not support persistence of pointers and procedure variables. This was not only motivated by the predominance of associative identification mechanisms in the classical Relational Data Model, but also by the far-reaching consequences of this identification mechanism on the concurrency control and distribution support [BJS86, JLRS88].

Modula-2 was chosen as the basis for DBPL because of its software engineering qualities. It provides a clear module concept and a strict type system and is further excelled by its balance between simplicity and expressive power. Finally, our group had a long tradition in Modula-2 compiler construction.

A design decision with far-reaching consequences is the compatibility between DBPL and Modula-2. DBPL is designed to be fully upward compatible with Modula-2, i.e. every correct Modula-2 program has to be correct DBPL program. This decision not only limits the freedom in language extensions (e.g. the keyword *SET* is already used for bit sets in Modula-2 and is not available for "true" sets in DBPL), but also forces adherence to language mechanisms (e.g. variant records, string handling, local modules) for which nowadays "better" solutions are available. The main advantage of our compatibility decision is to lower the conceptual and technological burden for DBPL users since they do not have to learn yet another language. Furthermore, this decision allows to smoothly integrate DBPL into existing, fully-fledged software development environments (e.g. debuggers, profilers or version managers), to benefit from software libraries and from the interfaces to a variety of other languages and systems.

In contrast to some persistent programming languages, the evolution of database schemata and of application programs is left outside the scope of DBPL. This was based on the insight that this is a complex issue in itself that should be delegated to a specific environment (see DAIDA environment, e.g. [JMW+90]).

To summarize, the most prominent feature of the DBPL language is the type-complete integration of sets and first-order predicates into a strongly typed, monomorphic language with persistence as an orthogonal property of individual compilation units. The DBPL system is further characterized by covering a wide range of operational database demands, such as query optimization, transaction and distribution management and computer-aided support for database application development. Readers interested in specific language and system aspects are referred to the following publications [JLS85, JGL+88, MS89, SGLJ89, SM89, SM90a, SM90b, SM91b]

1.4 The DBPL System and its Use

The DBPL project always had a strong commitment to implementability. A multi-user DBPL system under VAX/VMS has served many times since 1985 for lab courses on database programming at the Universities of Frankfurt and Hamburg. There exist several DBPL system extensions that experiment with alternatives for concurrency (optimistic, pessimistic and mixed strategies) [BJS86] and integrity control [Böt90], storage structures for complex objects, recursive queries [JLS85, SL85] and distribution [JLRS88, JGL+88]. The construction of a distributed DBPL system is based on ISO/OSI communication standards and involves, for example, a re-implementation of the DBPL compiler to generate native code for IBM-PC/AT clients in cooperation with VAX/VMS servers.

In 1991, a substantial effort was made to integrate the experience gained with these prototypes into a new, portable implementation of the DBPL runtime system on various platforms (VAX/VMS, Sun-3, Sun-4/Unix, IBM RISC). By utilizing Sun's optimizing compiler backend, the DBPL compiler achieves "production-quality" performance and interoperability. Implementation aspects of the various layers of the DBPL database system are discussed in [SM91b] and [MS92].

Finally, since 1992 there exists an optimizing gatway between the DBPL runtime system and commercial SQL database engines (Ingres, Oracle) that allows DBPL programs to transparently access existing external database relations in addition to the internal type-complete DBPL objects.

We expect the DBPL language and system to be used in research and devleopment mainly for the following three tasks:

Concept validation: As outlined above, we strongly believe in the necessity of experimental evaluation of proposed system solutions (e.g. of new concurrency control protocols or a new workstation-server architectures). In many cases, the interaction with several system components (e.g. the recovery management or the query optimizer), or the lack of universality severly impairs the utility of a seemingly advantageous paper-and-pencil solution.

Database education: Our experience in using DBPL intensively in lab classes convinces us that it is an appropriate tool for teaching the essential problems and solutions in database application development. Without being distracted by idiosyncratic surface syntax and deficiencies of traditional preprocessor database interfaces, it is much easier to isolate and communicate the repeating patterns in database applications and to concentrate on an *abstract and complete* picture of database application programming.

Application prototyping: From Modula-2 the DBPL language has inherited software engineering qualities that can not be found in commercial database environments and which qualify DBPL as an appropriate tool for designs and implementations. This use of DBPL is further supported by the quality of the commercial platforms (DEC/VAX/VMS and SUN/RISC/UNIX) on which the DBPL system is realized and the depth of its integration into professional environmets for software development and maintenance.

Acknowledgements

We are grateful for the many contributions by our colleagues, in particular Stefan Böttcher, Henning Eckhardt, Jürgen Edelmann, Matthias Jarke, Jürgen Koch, Manuel Mall and Andreas Rudloff.

2 Syntax

A language is an infinite set of sentences, namely the sentences well formed according to its syntax. In DBPL, these sentences are called *compilation units*. Each unit is a finite sequence of symbols from a finite *vocabulary*. The vocabulary of DBPL consists of identifiers, numbers, strings, operators, and delimiters. They are called lexical *symbols* and are composed of sequences of characters. (Note the distinction between symbols and characters.)

To describe the syntax, an extended Backus-Naur Formalism called EBNF is used. Angular brackets [ ] denote optionality of the enclosed sentential form, and curly brackets { } denote its repetition (possibly 0 times). Syntactic entities (non-terminal symbols) are denoted by English words expressing their intuitive meaning. Symbols of the language vocabulary (terminal symbols) are strings enclosed in quote marks or words written in capital letters, so-called *reserved* words. Syntactic rules (productions) are designated by a $ sign at the left margin of the line.

3 Vocabulary and representation

The representation of symbols in terms of characters depends on the underlying character set. The ASCII set is used in this paper, and the following lexical rules must be observed. Blanks must not occur within symbols (except in strings). Blanks and line breaks are ignored unless they are essential to separate two consecutive symbols.

1. *Identifiers* are sequences of letters and digits. The first character must be a letter.

   $ ident = letter {letter | digit}.

   Examples:

   ```
   x scan Modula ETH GetSymbol firstLetter
   ```

2. *Numbers* are (unsigned) integers or real numbers. Integers are sequences of digits. If the number is followed by the letter B, it is taken as an octal number; if it is followed by the letter H, it is taken as a hexadecimal number; if it is followed by the letter C, it denotes the character with the given (octal) ordinal number (and is of type CHAR, see 6.1).

   An integer i in the range 0 <= i <= MaxInt can be considered as either of type INTEGER or CARDINAL; if it is in the range MaxInt < i <= MaxCard, it is of type CARDINAL. For 16-bit computers: MaxInt = 32767, MaxCard = 65535.

   A real number always contains a decimal point. Optionally it may also contain a decimal scale factor. The letter E is pronounced as "ten to the power of". A real number is of type REAL.

   $ number = integer | real.
   $ integer = digit {digit} | octalDigit {octalDigit} ("B" | "C") |

```
$         digit {hexDigit} "H".
$ real = digit {digit} "." {digit} [ScaleFactor].
$ ScaleFactor = "E" ["+" | "-"] digit {digit}.
$ hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
$ digit = octalDigit | "8" | "9".
$ octalDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7".
```

Examples:

```
1980  3764B  7BCH  33C  12.3  45.67E-8
```

3. *Strings* are sequences of characters enclosed in quote marks. Both double quotes and single quotes (apostrophes) may be used as quote marks. However, the opening and closing marks must be the same character, and this character cannot occur within the string. A string must not extend over the end of a line.

   $ string = "'" {character} "'" | """ {character} """.

   A string consisting of n characters is of type (see 6.4).

   ```
   ARRAY [0..n-1] OF CHAR
   ```

   Examples:

   ```
   "DBPL"  "Don't worry!"  'codeword "Barbarossa"'
   ```

4. *Operators and delimiters* are the special characters, character pairs, or reserved words listed below. These reserved words consist exclusively of capital letters and *must not* be used in the role of identifiers. The symbols # and <> are synonyms, and so are &, AND, and ~, NOT. ⊤

+	<	ALL	EXPORT	QUALIFIED
-	>	AND	FOR	RECORD
*	<>	ARRAY	FROM	RELATION
/	<=	BEGIN	IF	REPEAT
:=	>=	BY	IMPLEMENTATION	RETURN
&	..	CASE	IMPORT	SELECTOR
.	:	CONST	IN	SET
,	)	CONSTRUCTOR	LOOP	SOME
;	]	DATABASE	MOD	THEN
(	}	DEFINITION	MODULE	TO
[	|	DIV	NOT	TRANSACTION
{	:+	DO	OF	TYPE
↑	:-	EACH	ON	UNTIL
=	:&	ELSE	OR	USING
#		ELSIF	POINTER	VAR
~		END	PROCEDURE	WHILE
		EXIT		WITH

5. *Comments* may be inserted between any two symbols in a program. They are arbitrary character sequences opened by the bracket (∗ and closed by ∗). Comments may be nested, and they do not affect the meaning of a program.

4 Declarations and scope rules

Every identifier occurring in a program must be introduced by a declaration, unless it is a standard identifier. The latter are considered to be predeclared, and they are valid in all parts of a program. For this reason they are called *pervasive*. Declarations also serve to specify certain permanent properties of an object, such as whether it is a constant, a type, a variable, a procedure, or a module.

The identifier is then used to refer to the associated object. This is possible in those parts of a program only which are within the so-called *scope* of the declaration. In general, the scope extends over the entire block (procedure or module declaration) to which the declaration belongs and to which the object is local. The scope rule is augmented by the following cases:

1. If an identifier x defined by a declaration D1 is used in another declaration (not statement) D2, then D1 must textually precede D2.

2. A type T1 can be used in a declaration of a pointer type T (see 6.7) which textually precedes the declaration of T1, if both T and T1 are declared in the same block. This is a relaxation of rule 1.

3. If an identifier defined in a module M1 is exported, the scope expands over the block which contains M1. If M1 is a compilation unit (see Ch. 16), it extends to all those units which import M1.

4. Field identifiers of a record declaration (see 6.5) are valid only in field designators and in with statements referring to a variable of that record type.

An identifier may be *qualified*. In this case it is prefixed by another identifier which designates the module (see Ch. 13) in which the qualified identifier is defined. The prefix and the identifier are separated by a period. Standard identifiers appear below.

$ qualident = ident {"." ident}.

ABS	(12.2)	INTEGER	(6.1)
BITSET	(6.6)	LONGINT	(6.1)
BOOLEAN	(6.1)	LONGREAL	(6.1)
CAP	(12.2)	LOWEST	(12.2)
CARD	(12.2)	MAX	(12.2)
CARDINAL	(6.1)	MIN	(12.2)
CHAR	(6.1)	NEXT	(12.2)
CHR	(12.2)	NIL	(6.7)
DEC	(12.2)	ODD	(12.2)
EOR	(12.2)	ORD	(12.2)
EXCL	(12.2)	PRIOR	(12.2)
FALSE	(6.1)	PROC	(6.8)
FLOAT	(12.2)	REAL	(6.1)
HALT	(12.2)	THIS	(12.2)
HIGH	(12.2)	SIZE	(12.2)
HIGHEST	(12.2)	TRUE	(6.1)
INC	(12.2)	TRUNC	(12.2)
INCL	(12.2)	VAL	(12.2)

⊥

5 Constant declarations

A constant declaration associates an identifier with a constant value.

$ ConstantDeclaration = ident "=" ConstExpression.
$ ConstExpression = expression.

A constant expression is an expression, which can be evaluated by a mere textual scan without actually executing the program. Its operands are constants. (see Ch. 8).

Examples of constant declarations are

```
N     =  100
limit =  2*N-1
all   = {0..WordSize-1}
bound = MAX(INTEGER) - N
```

6 Type declarations

A data type determines a set of values which variables of that type may assume, and it associates an identifier with the type. In the case of structured types, it also defines the structure of variables of this type. There are four different structures, namely arrays, records, sets and relations. ⊤ ⊥ ⊤

```
$ TypeDeclaration = ident "=" type.
$ type = SimpleType | ArrayType | RecordType | SetType |
$        RelationType | PointerType | ProcedureType | SelectorType |
$        ConstructorType.
$ SimpleType = qualident | enumeration | SubrangeType.
```

⊥

Examples:

```
Color = (red,green,blue)
Index = [0..80]
Card = ARRAY Index OF CHAR
Node = RECORD key: CARDINAL;
         left,right: TreePtr
       END
Tint = SET OF Color
TreePtr = POINTER TO Node
Function = PROCEDURE(CARDINAL):CARDINAL
```

⊤

```
String = ARRAY Index OF CHAR
Item = RECORD code: Color;
         itemname: String;
         price: INTEGER;
         connectedTo: RELATION OF String;
       END
Items = RELATION itemname OF Item
Connections = RELATION OF RECORD from,to: String END
ItemsInRange = SELECTOR ON (Items) WITH (CARDINAL,CARDINAL)
```

⊥

6.1 Basic types

The following basic types are predeclared and denoted by standard identifiers:

1. INTEGER comprises the integers between MIN(INTEGER) and MAX(INTEGER).
2. CARDINAL comprises the integers between 0 and MAX(CARDINAL).
3. BOOLEAN comprises the truth values TRUE or FALSE.
4. CHAR denotes the character set provided by the used computer system.
5. REAL (and LONGREAL) denote finite sets of real numbers.
6. LONGINT comprises the integers between MIN(LONGINT) and MAX(LONGINT).

6.2 Enumerations

An enumeration is a list of identifiers that denote the values which constitute a data type. These identifiers are used as constants in the program. They, and no other values, belong to this type. The values are ordered, and the ordering relation is defined by their sequence in the enumeration. The ordinal number of the first value is 0.

$ enumeration = "(" IdentList ")".
$ IdentList = ident {"," ident}.

Examples of enumerations:

```
(red,green,blue)
(club,diamond,heart,spade)
(Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday)
```

6.3 Subrange types

A type T may be defined as a subrange of another, basic or enumeration type T1 (except REAL) by specification of the least and the highest value in the subrange.

$ SubrangeType = [ident] "[" ConstExpression ".." ConstExpression "]".

The first constant specifies the lower bound, and must not be greater than the upper bound. The type T1 of the bounds is called the *base type* of T, and all operators applicable to operands of type T1 are also applicable to operands of type T. However, a value to be assigned to a variable of a subrange type must lie within the specified interval. The base type can be specified by an identifier preceding the bounds. If it is omitted, and the lower bound is a non-negative integer, the base type of the subrange is taken to be CARDINAL; if it is a negative integer, it is INTEGER.

A type T1 is said to be *compatible* with a type T0, if it is declared either as T1 = T0 or as a subrange of T0, or if T0 is a subrange of T1, or if T0 and T1 are both subranges of the same (base) type.

Examples of subrange types:

```
[0..N-1]
["A".."Z"]
[Monday..Friday]
```

6.4 Array types

An array is a structure consisting of a fixed number of components which are all of the same type, called the *component type*. The elements of the array are designated by indices, values belonging to the *index type*. The latter must be an enumeration, a subrange type, or one of the basic types BOOLEAN or CHAR.

$ ArrayType = ARRAY SimpleType {"," SimpleType} OF type.

A declaration of the form

```
ARRAY T1, T2, ..., Tn OF T
```

with n index types T1 ... Tn must be understood as an abbreviation for the declaration

```
ARRAY T1 OF
  ARRAY T2 OF
    ...
     ARRAY Tn OF T
```

Examples of array types:

```
ARRAY [0..N-1] OF CARDINAL
ARRAY [1..10],[1..20] OF [0..99]
ARRAY [-10..+10] OF BOOLEAN
ARRAY WeekDay OF Color
ARRAY Color OF WeekDay
```

6.5 Record types

A record type is a structure consisting of a fixed number of components of possibly different types. The record type declaration specifies for each component, called *field*, its type and an identifier which denotes the field. The scope of these field identifiers is the record definition itself, and they are also accessible within field designators (see 8.4) refering to components of record variables, and within with statements.

A record type may have several variant sections, in which case the first field of the section is called the *tag field*. Its value indicates which variant is assumed by the section. Individual variant structures are identified by *case labels*. These labels are constants of the type indicated by the tag field.

$ RecordType = RECORD FieldListSequence END.
$ FieldListSequence = FieldList {";" FieldList}.
$ FieldList = [IdentList ":" type |
$ CASE [ident] ":" qualident OF variant {"|" variant}
$ [ELSE FieldListSequence] END].
$ variant = [CaseLabelList ":" FieldListSequence].
$ CaseLabelList = CaseLabels {"," CaseLabels}.
$ CaseLabels = ConstExpression [".." ConstExpression].

Examples of record types:

```
RECORD day: [1..31];
  month: [1..12];
  year:  [0..2000]
END

RECORD
  name,firstname: ARRAY [0..9] OF CHAR;
  age: [0..99];
  salary: REAL
END

RECORD x,y: T0;
  CASE tag0: Color OF
    red: a: Tr1; b: Tr2|
    green: c: Tg1; d: Tg2|
    blue: e: Tb1; f: Tb2
  END;
  z: T0;
  CASE tag1: BOOLEAN OF
    TRUE: u,v: INTEGER|
    FALSE: r,s: CARDINAL
  END
END
```

The example above contains two variant sections. The variant of the first section is indicated by the value of the tag field tag0, the one of the second section by the tag field tag1.

6.6 Set types

A set type defined as SET OF T comprises all sets of values of its base type T. This must be a subrange of the integers between 0 and N–1, or a (subrange of an) enumeration type with at most N values, where N is a small constant determined by the implementation, usually the computer's wordsize or a small multiple thereof.

$ SetType = SET OF SimpleType.

The standard type BITSET is defined as follows, where W is a constant defined by the implementation, usually the word size of the computer.

```
BITSET = SET OF [0..W-1]
```

6.7 Pointer types

Variables of a pointer type P assume as values pointers to variables of another type T. The pointer type P is said to be *bound* to T. A pointer value is generated by a call to an allocation procedure in a storage management module.

$ PointerType = POINTER TO type.

Besides such pointer values, a pointer variable may assume the value NIL, which can be thought as pointing to no variable at all.

6.8 Procedure types

Variables of a procedure type T may assume as their value a procedure P. The (types of the) formal parameters of P must be the same as those indicated in the formal type list of T. The same holds for the result type in the case of a function procedure.

Restriction: P must not be declared local to another procedure, and neither can it be a standard procedure.

$ ProcedureType = (PROCEDURE | TRANSACTION) [FormalTypeList].
$ FormalTypeList = "(" [[VAR] FormalType
$ {"," [VAR] FormalType}] ")" [":" qualident].

The standard type PROC denotes a parameterless procedure:

```
PROC = PROCEDURE
```

6.9 Relation types

T

A relation type declaration specifies a structure consisting of elements of the same type, called the relation element type. The number of elements, called the cardinality of the relation, is not fixed. A relation with zero elements is called empty. The relation type declaration specifies the element type as well as the relation key.

The relation key defines a list of components of the relation element type which uniquely determines a relation element. An empty key component list is a synonym for an exhaustive element component list; in this case, a relation is just a set of relation elements. A relation key can only be specified if the relation element type is an array or record structure. A key component is either specified by a qualified identifier designating a record field, or by a constant expression designating an array component. Key components must not be part of a variant in a variant record structure. The type of a key component must be a simple type or a string.

$ RelationType = RELATION [RelationKey] OF type.
$ RelationKey = KeyComponent {"," KeyComponent}.
$ KeyComponent = KeyDesignator {KeySubDesignator}.
$ KeyDesignator = ident | "[" ConstExpList "]".
$ KeySubDesignator = "." ident | "[" ConstExpList "]".
$ ConstExpList = ConstExpression {"," ConstExpression}.

Examples of relation types:

```
Address =    RECORD
               street: String;
               city: String
             END
AddrClass =  (office, stock)
Company =    RECORD
               companyname: String;
               address: ARRAY AddrClass OF Address;
               phonenumber: CARDINAL
             END
Companies =  RELATION companyname, address[office].city OF
               Company
Deliveries = RELATION itemname OF
               RECORD
                 itemname: String;
                 name, city: String;
                 quantity: CARDINAL
               END
```

⊥

6.10 Selector types

⊤

Variables of a selector type T may assume as their values a selector S. The types of the formal parameters and the result type of S must be the same as those indicated in the formal type list of T.

Two selector types are compatible, if their formal parameter types and result types agree, and if their access restrictions are compatible.

Access restrictions are used in selector declarations and in the intention list of a transaction (see Ch. 12). They define the set of operations applicable to selected relation variables and persistent variables.

The access symbol = restricts the use of these variables to expressions, to range relations of FOR EACH statements and as parameters in the standard relation handling procedures (see Ch. 12.2). The access symbols :=, :+, :–, :& enable the use of a selected variable as the left side of the corresponding relation update operations. Defining no access restriction is synonymous to the full set of access symbols. If a selected relation variable is to be passed as a variable parameter, all access rights have to be specified.

Access restriction A is called a *restriction* of access restriction B, if A is a subset of B. Two access restrictions A and B are compatible, if A and B are equal.

```
$ SelectorType = SELECTOR [ON "("qualident")"] [WITH TypeList]
$         AccessRestriction [":" qualident].
$ TypeList = "(" FormalType {"," FormalType} ")".
$ AccessRestriction = [FOR "(" AccessRight {"," AccessRight ")"}].
$ AccessRight = ":=" | ":+" | ":–" | ":&" | "=".
```

Example of a selector type:

```
ReadOnlyItems = SELECTOR FOR (=): Items
```

6.11 Constructor types

Variables of a constructor type T may assume as their values a constructor C. The types of the formal parameters and the result type of C must be the same as those indicated in the formal type list of T.

Two constructor types are compatible, if their formal pararameter types and result types agree.

$ ConstructorType = CONSTRUCTOR [ON TypeList] [WITH TypeList]":" qualident.

Example of a constructor type:

```
ConnectionRule = CONSTRUCTOR ON (Items): Connections
```

7 Variable declarations

Variable declarations serve to introduce variables and associate them with a unique identifier and a fixed data type and structure. Variables whose identifiers appear in the same list all obtain the same type.

$ VariableDeclaration = IdentList ":" type.

The data type determines the set of values that a variable may assume and the operators that are applicable; it also defines the structure of the variable.

Examples of variable declarations (see examples in Ch. 6):

```
i,j: CARDINAL
k: INTEGER
p,q: BOOLEAN
s: BITSET
F: Function
a: ARRAY Index OF CARDINAL
w: ARRAY [0..7] OF
     RECORD ch: CHAR;
       count: CARDINAL
     END
t: TreePtr
```

```
thispart: Item
parts: Items
oldparts: Items
readOnlyItems :ReadOnlyItems
suppliers: Companies
orders: Deliveries
InRange: ItemsInRange
adjacent: ConnectionRule
```

8 Expressions

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators. Expressions consist of operands and operators. Parentheses may be used to express specific associations of operators and operands.

8.1 Access expressions

Access expressions denote rules for accessing relation variables. Access expressions can be named and parameterized. A parameter substitution is used to build new access expressions which can be stored in variables of type selector and constructor.

```
$ AccessExpression = SelectiveAccessExpression | ConstructiveAccessExpression.
$ SelectiveAccessExpression = ElementDenotation ":"expression.
$ ConstructiveAccessExpression = expression |
$        expression OF ElementDenotation {"," ElementDenotation} ":" expression.
$ ElementDenotation = EACH ident IN range.
$ range = expression | designator.
```

There are selective and constructive access expressions.

A selective access expression has the form

```
EACH v IN rg: se
```

It denotes the selection rule that refers to exactly those elements v of the range rg, that make the selection expression se of type BOOLEAN true. The (lexical) scope of an element variable is the selection expression se.

A constructive access expression has the form

```
e OF EACH v1 IN r1, EACH v2 IN r2, ..., EACH vn IN rn: se
```

It denotes the construction rule that evaluates the expression e for exactly those combinations of the elements v1, v2,..., vn of the ranges r1, r2,..., rn that fulfill the selection expression se. The (lexical) scope of the element variables are the expressions e and se.

8.2 Selectors

Selectors denote selective access expressions where the range relation is a designator. They are introduced by selector declarations and have a type (see 6.10 and 10).

Selectors are used in relations (see 8.4.3), in for statements (see 9.8), in the intention list of transactions (see Ch. 12), and as designators (see 8.4.1).

$ selector = designator ["(" [designator] ")" [ActualParameters]].

Let S be a selector declared as:

```
SELECTOR S ON (RPar: RType) WITH (ParList) FOR (AccList): RType;
BEGIN EACH r IN RPar: p(r,ParList) END S
```

Such a parameterized selector can be used to define new selectors by a (partial) parameter substitution:

- The substitution of the ON-parameter of S by a designator R, S(R), denotes a non-ON-parameterized selector SR, that is bound to R as its range relation. That selector is equivalent to the declaration

  ```
  SELECTOR SR WITH (ParList) FOR (AccList): RType;
  BEGIN EACH r IN R: p(r,ParList) END SR
  ```

- The relational ON-parameter of a selector can also be substituted by a selected relation variable denoted by any type-compatible (non-parameterized) selector [SRP]. The resulting selector, S([SRP]), has as its access restriction the intersection of the access restrictions of S and SRP.

- Substitution of the WITH-parameter list of S by actual parameters, S()(ActList), denotes a selector equivalent to the declaration

  ```
  SELECTOR SP ON (RPar: RType) FOR (AccList): RType;
  BEGIN EACH r IN RPar: pp(r) END SP
  ```

  where pp is the predicate p after substitution of each occurence of the formal parameters by the current values of the actual parameters.

- ON- and WITH-parameters can be substituted simultaneously: S(R)(ActList) is an unparameterized selector equivalent to

  ```
  SELECTOR SRP FOR (AccList): RType;
  BEGIN EACH r IN R: pp(r) END SRP
  ```

A selector with an ON-parameter of type RType but without a WITH-parameter can be applied to a relation variable of type RType: R[SP] denotes a *selected relation variable*. For the special case of a non-parameterized selector the application is denoted by [SRP]. The designators R[SP], [SRP], [SP(R)] and [SR(P)] are synonymous. They denote a subvariable of R, which is defined by the access expression

```
EACH r IN R: pp(r)
```

A selected relation variable can only be used in a context defined by the access restrictions (AccList) of the selector type.

The interpretation of a selected relation variable depends on its actual context:

- If [SRP] is used in an expression it is synonymous to the relation expression

```
RType{EACH r IN R: pp(r)}
```

- The semantics of assignments to selected relation variables are defined in 9.1.
- The substitution of a variable parameter in a procedure call by a selected relation variable implies repeated evaluation of the access expression in the procedure or transaction.

Examples of selectors and selected relation variables (see examples in Ch. 10 together with their types in Ch. 6 and 7):

```
InRange()(0,N)                  SELECTOR ON (Items): Items
InRange(parts)                  SELECTOR WITH
                                  (CARDINAL, CARDINAL): Items
InRange([RedItems(oldparts)])   SELECTOR WITH
                                  (CARDINAL, CARDINAL): Items
[RedItems(parts)]               Items
orders[OfParts()("Printer")]    Deliveries
[InRange(oldparts)(0,k)]        Items
```

8.3 Constructors

Constructors denote access expression lists. They are introduced by constructor declarations and have a type that is determined by the constructor heading (see 6.11 and 11).

The use of constructors is restricted to relations and the intention list of transactions.

$ constructor = designator [ActualParameters [ActualParameters]].

Constructors can be generated by substituting formal parameters of existing constructors. The rules for subtitution are the same as those for selectors with the exception, that a constructor may have more than one ON-parameter.

8.4 Operands

Operands are denoted by literal constants, i.e. numbers, strings and sets (see Ch. 5), designators, aggregates or relations.

8.4.1 Designators

A designator consists of an identifier referring to the constant, variable, procedure, transaction, selector or constructor to be designated, or a selector enclosed in square brackets denoting a selected relation variable (see 8.2).

```
$ designator = (qualident | "[" selector "]") { "." ident | "[" ExpList "]" | "↑"}.
$ ExpList = expression {"," expression}.
```

This identifier may possibly be qualified by module identifiers (see Ch. 4 and 13), and it may be followed by component selectors, if the designated object is an element of a structure. If the structure is an array A, then the designator A[E] denotes that component of A whose index is the current value of the expression E. The index type of A must be *assignment compatible* with the type of E (see 9.1). A designator of the form

A[E1,E2,...,En] stands for A[E1][E2]...[En].

If the structure is a relation R declared by

R: RELATION k1, k2,..., kn OF ElementType

then the designator R[E1,...,En] denotes that element of R whose key component values are the current values of the selection expressions E1 to En. A variable designated in this way is called a selected element variable. The types of the selection expressions must be assignment compatible with the types of the key components k1,...,kn identified by the relation type. The type of a selected element variable is defined by the relation element type with the additional constraint that the values of the key components are restricted to the values of the selection expressions. This implies that the values of the key components of a selected element variable cannot be altered. Note also, that R[E1,...,En] denotes an object of type ElementType only if an element with key value E1,E2,...,En exists in R. Thus selected element variables may cause conflicts with type definitions and other constraints (see 9.1).

If the structure is a record R, then the designator R.f denotes the field f of R. The designator P↑denotes the variable which is referenced by the pointer P.

If the designated object is a variable, then the designator refers to the variable's current value. If the object is a function procedure, a designator without parameter list refers to that procedure. If it is followed by a (possibly empty) parameter list, the designator implies an activation of the procedure and stands for the value resulting from its execution, i.e. for the "returned" value. The (types of these) actual parameters must correspond to the formal parameters as specified in the procedure's declaration (see Ch. 12).

Examples of designators (see examples in Ch. 7):

```
k                                   (INTEGER)
a[i]                                (CARDINAL)
w[3].ch                             (CHAR)
t .key                              (CARDINAL)
t .left .right                      (TreePtr)
```

```
parts["Printer"]                                (Item)
suppliers["Brown & Co","Zurich"].phonenumber (CARDINAL)
```

8.4.2 Aggregate expressions

An aggregate must be either of type RECORD or type ARRAY. The identifier preceding the left bracket of an aggregate specifies the type of the aggregate. If an element of a relation or a component of an aggregate is an aggregate the type identifier preceding the aggregate can be omitted.

If the aggregate is of type ARRAY, all components must be assignment compatible with the array element type and the number of components must be the same as the number of array elements: if A=ARRAY [i..j] OF T and A{E0,...,En} is an aggregate then n=j–i and the component Ek corresponds to the array element with the index [i+k].

If the aggregate is of type RECORD the components and the record fields are associated in the sequence of their declaration. The component expressions must be assignment compatible with their corresponding record fields. Components associated with tag fields must be constant expressions (see Ch. 5). Records containing variants without tag fields cannot be built with an aggregate.

$ aggregate = [qualident] "{" ExpList "}".

Examples of aggregates:

```
Item{green,"Printer",999, {"Computer"}}      (Item)
Node{25,t .left,NIL}                         (Node)
```

8.4.3 Relation expressions

RType{} denotes the empty relation of type RType.

Relation expressions denote sets of relation elements; relation elements can be given by

- selective or constructive access expressions as defined in 8.1,
- unparameterized selectors denoting elements defined by the selective access expressions of their bodies,
- unparameterized constructors denoting the access expression list of their bodies.

Note, that constructive access expressions include the case that a relation element is given by an expression that evaluates a single value of the relation's element type.

If several relation elements are specified, they all have to be of the same type, which is the relation element type.

```
$ relation = qualident "{" AccessExpressionList "}".
$ AccessExpressionList = [AccessExpression { "," AccessExpression }].
```

Examples of relations:

```
Items{thispart}                                  (Items)
Items{EACH p IN parts:p.code = red}              (Items)
Items{InRange(parts)(0,7)}                       (Items)
Deliveries{EACH o IN orders: SOME p IN parts
             (o.itemname = p.itemname)}          (Deliveries)
Items{EACH p IN parts: p.code = thispart.code,
      thispart}                                  (Items)
Items{{p.code,p.itemname,1000,p.connectedTo} OF
      EACH p IN oldparts: p.price<1000}          (Items)
Connections{TransConn(parts)}                    (Connections)
```

8.5 Operators

The syntax of expressions specifies operator precedences according to four classes of operators. The operators NOT, SOME and ALL have the highest precedence, followed by the so-called multiplying operators, then the so-called adding operators, and finally, with the lowest precedence, the relational operators. Sequences of operators of the same precedence are executed from left to right.

```
$ expression = SimpleExpression [RelOperator SimpleExpression] |
$         constructor | selector.
$ RelOperator = "=" | "#" | "<>" | "<" | "<=" | ">" | ">=" | IN.
$ SimpleExpression = ["+" | "-"] term {AddOperator term}.
$ AddOperator = "+" | "-" | OR.
$ term = factor {MulOperator factor}.
$ MulOperator = "*" | "/" | DIV | MOD | AND | "&".
$ factor = number | string | set | designator [ActualParameters] |
$         relation | aggregate | QuantifiedExpression |
$         "(" expression ")" | NOT factor.
$ ActualParameters = "(" [ExpList] ")".
$ set = [qualident] "{" [element {"," element}] "}".
$ element = expression [".." expression].
$ QuantifiedExpression = (SOME | ALL) ident IN expression predicate.
$ predicate = "(" expression ")" | QuantifiedExpression.
```

The available operators are listed in the following tables. In some instances, several different operations are designated by the same operator symbol. In these cases, the actual operation is identified by the types of the operands.

8.5.1 Arithmetic operators

symbol	operation
+	addition
−	subtraction
*	multiplication
/	real division
DIV	integer division
MOD	modulus

These operators (except /) apply to operands of type INTEGER, CARDINAL, or subranges thereof. Both operands must be either of type CARDINAL or a subrange with base type CARDINAL, in which case the result is of type CARDINAL, or they must both be of type INTEGER or a subrange with base type INTEGER, in which case the result is of type INTEGER.

The operators +, −, and * also apply to operands of type REAL. In this case, both operands must be of type REAL, and the result is then also of type REAL. The division operator / applies to REAL operands only. When used as operators with a single operand only, − denotes sign inversion and + denotes the identity operation. Sign inversion applies to operands of type INTEGER or REAL. The operations DIV and MOD are defined by the following rules:

x DIV y is equal to the truncated quotient of x/y
x MOD y is equal to the remainder of the division x DIV y (for y > 0)
x = (x DIV y)*y+ (x MOD y)

8.5.2 Logical operators

symbol	operation
OR	logical disjunction
AND	logical conjunction
NOT	negation

These operators apply to BOOLEAN operands and yield a BOOLEAN result.

p OR q means "if p then TRUE, otherwise q"
p AND q means "if p then q, otherwise FALSE"

⊤

Quantifiers apply to operands of type RELATION and BOOLEAN and yield a BOOLEAN result.

symbol	operation
SOME	existential quantification
ALL	universal quantification

The expression in a quantified expression must be of type RELATION. The expression in a predicate must be of type BOOLEAN. Element variables in quantified expressions are called bound element variables. The scope of a bound element variable is the subsequent predicate, its type is the element type of the subsequent relation expression.

SOME r IN R (exp) is true, if some element r in the relation R makes the expression exp true. Analogously ALL r IN R (exp) is true, if all elements r in R fulfill the selection expression
⊥ exp.

8.5.3 Set operators

symbol	operation
+	set union
−	set difference
*	set intersection
/	symmetric set difference

These operations apply to operands of any set type and yield a result of the same type.

```
x IN (s1 + s2)    iff     (x IN s1) OR (x IN s2)
x IN (s1 - s2)    iff     (x IN s1) AND NOT (x IN s2)
x IN (s1 * s2)    iff     (x IN s1) AND (x IN s2)
x IN (s1 / s2)    iff     (x IN s1) # (x IN s2)
```

8.5.4 Relational operators

Relational operators yield a BOOLEAN result. They apply to the basic types INTEGER, CARDINAL, BOOLEAN, CHAR, REAL, to enumerations, and to subrange types.

symbol	relation
=	equal
#	unequal
<	less
<=	less or equal (set inclusion)
>	greater
>=	greater or equal (set inclusion)
IN	contained in (membership)

The relational operators = and # also apply to sets and pointers. If applied to sets, <= and
⊤ >= denote (improper) inclusion. The relational operators =, #, <, <=, >, >= may also be used to compare arrays of type string (see Ch. 3), and then denote alphabetical ordering according to the underlying character set. A string of length n1 can be compared with a string variable of length n2 > n1. In this case the string value is extended with a null character (0C) which will be the last character to be compared.

The relational operators =, #, <, <=, >, >= may also be used to express a subclass of relation comparisons which are key-based. The value of the expression

```
r1 <= r2
```

where r1, r2 are relation expressions is equal to the value of the quantified expression

```
ALL v1 IN r1 SOME v2 IN r2 (v1.key = v2.key).
```

where v1.key=v2.key stands for a conjunction of comparisons of the key attributes.

The relational operator IN denotes set or relation membership. In an expression of the form x IN e, the expression e must be of type SET OF T, where T is (compatible with) the type of x, or the expression e must be of relation type and x of its element type. ⊥

Examples of expressions (refer to examples in Ch. 7):

```
1980                                                    (CARDINAL)
k DIV 3                                                 (INTEGER)
NOT p OR q                                              (BOOLEAN)
(i+j) * (i-j)                                           (CARDINAL)
s - {8,9,13}                                            (BITSET)
a[i] + a[j]                                             (CARDINAL)
a[i+j] * a[i-j]                                         (CARDINAL)
(0<= k) & (k<100)                                       (BOOLEAN)
t .key = 0                                              (BOOLEAN)
{13..15} <= s                                           (BOOLEAN)
i IN {0,5..8, 15}                                       (BOOLEAN)
                                                                  ⊤

SOME p IN parts (p.code = red)                          (BOOLEAN)

Items{EACH p IN oldparts: p.price<1000} <= parts        (BOOLEAN)
thispart IN parts                                       (BOOLEAN)
                                                                  ⊥
```

9 Statements

Statements denote actions. There are elementary and structured statements. Elementary statements are not composed of any parts that are themselves statements. They are the assignment, the procedure call, and the return and exit statements. Structured statements are composed of parts that are themselves statements. These are used to express sequencing, and conditional, selective, and repetitive execution.

```
$ statement = [assignment | ProcedureCall |
$         IfStatement | CaseStatement | WhileStatement |
$         RepeatStatement | LoopStatement | ForStatement |
$         WithStatement | EXIT | RETURN [expression]].
```

A statement may also be empty, in which case it denotes no action. The empty statement is included in order to relax punctuation rules in statement sequences.

9.1 Assignments

The assignment serves to replace the current value of a variable by a new value indicated by an expression. The assignment operator is written as ":=" and pronounced as "becomes".

$ assignment = designator UpdateOperator expression.
$ UpdateOperator = ":+" | ":–" | ":&" | ":=".

The designator to the left of the assignment operator denotes a variable. After an assignment is executed, the variable has the value obtained by evaluating the expression. The old value is lost (overwritten). The type of the variable must be assignment compatible with the type of the expression. Operand types are said to be *assignment compatible*, if either they are compatible or both are INTEGER or CARDINAL or subranges with base types INTEGER or CARDINAL.

A string of length n1 can be assigned to a string variable of length n2>n1. In this case, the string value is extended with a null character (0C). A string of length 1 is compatible with the type CHAR.

A constructor of type T1 can be assigned to a constructor of type T2 iff their formal parameter types and result types agree. The same holds for selectors and selector variables with the additional constraint, that the access restrictions of T2 have to be a restriction of the access restrictions of T1.

Assignments that update a relation variable rel of type RType by a relation expression rex, using one of the relation update operators, :+, :–, :&, are equivalent to assignments using the assignment operator, :=, and a more complicated relation expression.

Relation insertion:

```
rel:+ rex
```

is equivalent to

```
rel:= RType{EACH r IN rel: TRUE,
            EACH x IN rex: NOT SOME r IN rel (x.key=r.key)}.
```

Relation deletion:

```
rel:- rex
```

is equivalent to

```
rel:= RType{EACH r IN rel: NOT SOME x IN rex (r.key=x.key)}
```

Relation replacement:

```
rel:& rex
```

is equivalent to

```
rel:= RType{EACH r IN rel: NOT SOME x IN rex (r.key=x.key),
            EACH x IN rex: SOME r IN rel (x.key=r.key)}
```

The expressions x.key=y.key here stand for a conjunction of comparisons of the respective key attributes.

The execution of an assignment to a selected relation variable (see 8.2)

```
R[SP]:= rex
```

is intended to meet the following two constraints: The value of the selected relation variable after the assignment becomes equal to the relation expression, rex, while the value of the non-selected rest of variable R remains unchanged. Note that, due to constraint violations, a selective assignment may not be executable.

The assignment to a selected element variable (see 8.4.1) is a special case of selective relation assignment. ⊥

Examples of assignments:

```
i:= k
p:= i = j
j:= log2(i+j)
F:= log2
s:= {2,3,5,7,11,13}
a[i]:= (i+j) * (i-j)
t .key:= i
w[i+1].ch:= "A"
```

⊤

```
adjacent:= DirConn
parts := Items{EACH p IN oldparts: p.price>k}
parts :- Items{EACH p IN parts: p.code=red}
parts :& Items{thispart}
parts[RedItems] := oldparts
[InRange(oldparts)(0,k)] := Items{}
oldparts := Items{}
```

⊥

9.2 Procedure calls

A procedure call serves to activate a procedure or transaction. The procedure call may contain a list of actual parameters which are substituted in place of their corresponding formal parameters defined in the procedure declaration (see Ch. 12). The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. There exist two kinds of parameters: *variable* and *value parameters*.

In the case of variable parameters, the actual parameter must be a designator denoting a variable. If it designates a component of a structured variable, the selector is evaluated when the formal/actual parameter substitution takes place, i.e. before the execution of the procedure. If the parameter is a value parameter, the corresponding actual parameter must be an expression. This expression is evaluated prior to the procedure activation, and the resulting value is assigned to the formal parameter which now constitutes a local variable. The types of corresponding actual and formal parameters must be compatible in the case of variable parameters and assignment compatible in the case of value parameters.

$ ProcedureCall = designator [ActualParameters].

Examples of procedure calls:

```
Read(i)
Write(j*2+1,6)
INC(a[i])
```

9.3 Statement sequences

Statement sequences denote the sequence of actions specified by the component statements which are separated by semicolons.

$ StatementSequence = statement {";" statement}.

9.4 If statements

$ IfStatement = IF expression THEN StatementSequence
$ {ELSIF expression THEN StatementSequence}
$ [ELSE StatementSequence] END.

The expressions following the symbols IF and ELSIF are of type BOOLEAN. They are evaluated in the sequence of their occurrence, until one yields the value TRUE. Then its associated statement sequence is executed. If an ELSE clause is present, its associated statement sequence is executed if and only if all Boolean expressions yielded the value FALSE.

Example:

```
IF (ch>="A") & (ch<="Z") THEN ReadIdentifier
ELSIF (ch>="0") & (ch<="9") THEN ReadNumber
ELSIF ch = '"' THEN ReadString ('"')
ELSIF ch = "'" THEN ReadString ("'")
ELSE SpecialCharacter
END
```

9.5 Case statements

Case statements specify the selection and execution of a statement sequence according to the value of an expression. First the case expression is evaluated, then the statement sequence is executed whose case label list contains the obtained value. The type of the case expression must be a basic type (except REAL), an enumeration type, or a subrange type, and all labels must be compatible with that type. Case labels are constants, and no value must occur more than once. If the value of the expression does not occur as a label of any case, the statement sequence following the symbol ELSE is selected.

```
$ CaseStatement = CASE expression OF case {"|" case}
$        [ELSE StatementSequence] END.
$ case = [CaseLabelList ":" StatementSequence].
```

Example:

```
CASE i OF
 0:p:= p OR q; x:= x+y|
 1:p:= p OR q; x:= x-y|
 2:p:= p AND q; x:= x*y
END
```

9.6 While statements

While statements specify the repeated execution of a statement depending on the value of a Boolean expression. The expression is evaluated before each subsequent execution of the statement sequence. The repetition stops as soon as this evaluation yields the value FALSE.

```
$ WhileStatement = WHILE expression DO StatementSequence END.
```

Examples:

```
WHILE j>0 DO
  j:= j DIV 2; i:= i+1
END
WHILE i#j DO
  IF i>j THEN i:= i-j
  ELSE j:= j-i
  END
END
WHILE (t#NIL) & (t .key#i) DO
  t:= t .left
END
```

9.7 Repeat statements

Repeat statements specify the repeated execution of a statement sequence depending on the value of a Boolean expression. The expression is evaluated after each execution of the statement sequence, and the repetition stops as soon as it yields the value TRUE. Hence, the statement sequence is executed at least once.

$ RepeatStatement = REPEAT StatementSequence UNTIL expression.

Example:

```
REPEAT
  k:= i MOD j; i:= j; j:= k
UNTIL j = 0
```

9.8 For statements

The for statement indicates that a statement sequence is to be repeatedly executed while a progression of values is assigned to a variable. This variable is called the *control variable* of the for statement. It cannot be a component of a structured variable, it cannot be imported, ⊤ nor can it be a parameter. Its value should not be changed by the statement sequence.

$ ForStatement = FOR ControlSection DO StatementSequence END.
$ ControlSection = ident ":=" expression TO expression [BY ConstExpression] |
$ SelectiveAccessExpression | ident ":" selector.
⊥

The for statement

```
FOR v := A TO B BY C DO SS END
```

expresses repeated execution of the statement sequence SS with v successively assuming the values A, A+C, A+2C, ..., A+nC, where A+nC is the last term not exceeding B. v is called the control variable, A the starting value, B the limit, and C the increment. A and B must be assignment compatible with v; C must be a constant of type INTEGER or CARDINAL. ⊤ If no increment is specified, it is assumed to be 1.

If the control section is given by a selective access expression, the element variable is called control element variable. If a selector is used, its element variable is renamed by the identifier preceding the selector; the selector has to be unparameterized. The scope of a control element variable is the subsequent statement sequence.

The control section is evaluated only once to determine all elements e1, e2,..., en in the range relation that fulfill the selection expression. The iteration order of these elements is system defined.

The control element variable obeys the same rules as a selected relation element variable (see 8.4.1). The value of the control element variable may be changed if the access restriction of the range relation variable contains the access right :&.

Examples:

```
FOR i := 1 TO 80 DO j:= j+a[i] END
FOR i := 80 TO 2 BY -1 DO a[i] := a[i-1] END
FOR EACH o IN orders:
  SOME p IN parts ((p.itemname=o.itemname) AND (p.code=red)) DO
    i := i + o.quantity;
END;
FOR EACH item: RedItems(parts) DO
  item.price:= item.price DIV 2;
END
```

⊥

9.9 Loop statements

A loop statement specifies the repeated execution of a statement sequence. It is terminated by the execution of any exit statement within that sequence.

$ LoopStatement = LOOP StatementSequence END.

Example:

```
LOOP
  IF t1 .key> x THEN t2:= t1 .left; p:= TRUE
  ELSE t2:= t1 .right; p:= FALSE
  END;
  IF t2 = NIL THEN
    EXIT
  END;
  t1:= t2
END
```

While, repeat, and for statements can be expressed by loop statements containing a single exit statement. Their use is recommended as they characterize the most frequently occurring situations where termination depends either on a single condition at either the beginning or end of the repeated statement sequence, or on reaching the limit of an arithmetic progression. The loop statement is, however, necessary to express the continuous repetition of cyclic processes, where no termination is specified. It is also useful to express situations exemplified above. Exit statements are contextually, although not syntactically bound to the loop statement which contains them.

9.10 With statements

The with statement specifies a record variable and a statement sequence. In these statements the qualification of field identifiers may be omitted, if they are to refer to the variable specified in the With clause. If the designator denotes a component of a structured variable, the selector is evaluated once (before the statement sequence). The with statement opens a new scope.

$ WithStatement = WITH designator DO StatementSequence END.

Example:

```
WITH t  DO
  key:= 0; left:= NIL; right:= NIL
END
```

9.11 Return and exit statements

A return statement consists of the symbol RETURN, possibly followed by an expression. It indicates the termination of a procedure (or a module body), and the expression specifies the value returned as result of a function procedure. Its type must be assignment compatible with the result type specified in the procedure heading (see Ch. 12).

Function procedures require the presence of a return statement indicating the result value. There may be several, although only one will be executed. In proper procedures, a return statement is implied by the end of the procedure body. An explicit return statement therefore appears as an additional, probably exceptional termination point.

An exit statement consists of the symbol EXIT, and it specifies termination of the enclosing loop statement and continuation with the statement following that loop statement (see 9.9).

10 Selector declarations

Selector declarations introduce selectors. They consist of a heading defining the name and the type of the selector and a body containing a selective access expression with a designator as its range relation. Selectors are used to define value-based constraints on relation variables or to restrict access rights on relations.

If the ON-parameter is omitted, the selector is bound to the global relation variable of the reference type, which is given by the range relation in the selector body. If the ON-parameter is specified, it has to be used as the range relation. The WITH-parameters have to be value parameters and can be substituted to derive new specialized selectors.

A selector is called unparameterized, if neither ON- nor WITH-parameters are specified. Access restrictions are explained in 6.10 and are part of the selector type. The scope rules for selector declarations are the same as for procedure declarations.

$ SelectorDeclaration = SelectorHeading ";" SelectorBlock ident.
$ SelectorHeading = SELECTOR ident [OnParameter]
$ [WITH ParameterList] AccessRestriction [":" qualident].
$ SelectorBlock = BEGIN EACH ident IN designator ":" expression END.
$ OnParameter = ON "(" ident ":" FormalType ")".
$ ParameterList = "(" FPSection {";" FPSection} ")".

Examples of selector declarations:

```
SELECTOR RedParts: Items;
BEGIN EACH r IN parts: r.code=red END RedParts

SELECTOR RedItems ON (rel: Items);
BEGIN EACH r IN rel: r.code=red END RedItems

SELECTOR ColoredItems ON (rel: Items) WITH (color: Color);
BEGIN EACH r IN rel: r.code=color END ColoredItems

SELECTOR OfParts ON (orders: Deliveries) WITH (s: String);
BEGIN EACH o IN orders: o.itemname=s END OfParts
```

11 Constructor declarations

Constructor declarations introduce constructors. They consist of a heading defining the name and the type of the constructor and a body consisting of a list of access expressions. Constructors are used to intentionally define relations. Constructor declarations can be recursive (with a fixed-point semantic).

ON-parameters have to be of type relation. They can be substituted by (selected) relation variables or unparameterized constructors, i.e. their substitution yields a new constructor with references to global or persistent relations. The WITH-parameters have to be value parameters and can be substituted to derive new specialized constructors (see 8.3).

If neither WITH- nor ON-parameters are defined, the constructor is called unparameterized. Unparameterized constructors can be evaluated through relation expressions (see 8.4.3).

The scope rules for constructor declarations are the same as for procedure declarations.

```
$ ConstructorDeclaration = ConstructorHeading ";" ConstructorBlock ident.
$ ConstructorHeading = CONSTRUCTOR ident [ON ParameterList]
$       [WITH ParameterList] ":" qualident.
$ ConstructorBlock = BEGIN AccessExpressionList END.
```

Examples of constructors:

```
CONSTRUCTOR DirConn ON (P: Items): Connections;
BEGIN
  {a.itemname, c} OF
    EACH a IN P, EACH c IN a.connectedTo: TRUE
END DirConn

CONSTRUCTOR TransConn ON (P: Items): Connections;
BEGIN
  DirConn(P),
  {a.from, b.to} OF
    EACH a IN Connections{DirConn(P)},
```

```
        EACH b IN Connections{TransConn(P)}: a.to = b.from
    END TransConn
```

12 Procedure declarations

Procedure declarations consist of a *procedure heading* and a block which is said to be the *procedure body*. The heading specifies the procedure identifier and the *formal parameters*. The block contains declarations and statements. The procedure identifier is repeated at the end of the procedure declaration.

Procedures can be divided into two classes, namely *ordinary procedures* and *transaction procedures*. The latter are marked by the symbol TRANSACTION instead of PROCEDURE.

There are two kinds of procedures, namely *proper procedures* and *function procedures*. The latter are activated by a function designator as a constituent of an expression, and yield a result that is an operand in the expression. Proper procedures are activated by a procedure call. The function procedure is distinguished in the declaration by indication of the type of its result following the parameter list. Its body must contain a RETURN statement which defines the result of the function procedure.

All constants, variables, types, modules and procedures declared within the block that constitutes the procedure body are *local* to the procedure. The values of local variables, including those defined within a local module, are undefined upon entry to the procedure. Since procedures may be declared as local objects too, procedure declarations may be nested. Every object is said to be declared at a certain *level* of nesting. If it is declared local to a procedure at level k, it has itself level k+1. Objects declared in the module that constitutes a compilation unit (see Ch. 16) are defined to be at level 0.

In addition to its formal parameters and local objects, also the objects declared in the environment of the procedure are known and accessible in the procedure (with the exception of those objects that have the same name as objects declared locally).

The use of the procedure identifier in a call within its declaration implies recursive activation of the procedure.

```
$ ProcedureDeclaration = ProcedureHeading ";" block ident.
$ ProcedureHeading = (PROCEDURE | TRANSACTION) ident [FormalParameters].
$ block = {declaration} [USING IntentionList]
$         [BEGIN StatementSequence] END.
$ declaration = CONST {ConstantDeclaration ";"} |
$         TYPE {TypeDeclaration ";"} |
$         VAR {VariableDeclaration ";"} |
$         SelectorDeclaration ";" | ConstructorDeclaration ";" |
$         ProcedureDeclaration ";" | ModuleDeclaration ";".
$ IntentionList = Intention { ";" Intention}.
$ Intention = expression {"," expression} AccessRestriction.
```

Transaction procedures are the only means of interacting with the database. They consist of a sequence of operations on persistent variables and have to be regarded as atomic with respect to their effects on the database. The property of atomicity does not hold for variables other than persistent variables.

Under some scheduling strategies transaction procedures are subject to automatic restarts. A restart does not reset the global variables and parameters used in the transaction procedures. In contrast to ordinary procedures, nested and recursive calls of transaction procedures are inadmissable.

An intention list may be used to denote the set of persistent variables accessed during execution of a transaction procedure. The intention list is a means for passing additional information about the transaction's behavior to the compiler. The compiler may use this information for some optimization.

Access to a persistent object is defined by its designator and (a superset of) the needed access rights. If a designator is denoted by a selector of type T, the access restriction in the intention list has to be a restriction of T. An unparameterized constructor C may be used to request read access to all (sub-)relations needed to evaluate C. ⊥

12.1 Formal parameters

Formal parameters are identifiers which denote actual parameters specified in the procedure call. The correspondence between formal and actual parameters is established when the procedure is called. There are two kinds of parameters, namely *value* and *variable parameters*. The kind is indicated in the formal parameter list. Value parameters stand for local variables to which the result of the evaluation of the corresponding actual parameter is assigned as initial value. Variable parameters correspond to actual parameters that are variables, and they stand for these variables. Variable parameters are indicated by the symbol VAR, value parameters by the absence of the symbol VAR.

Formal parameters are local to the procedure, i.e. their scope is the program text which constitutes the procedure declaration.

```
$ FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" qualident].
$ FPSection = [VAR] IdentList ":" FormalType.
$ FormalType = [ARRAY OF] qualident.
```

The type of each formal parameter is specified in the parameter list. In the case of variable parameters it must be compatible with its corresponding actual parameter (see 9.2), in the case of value parameters the formal type must be assignment compatible with the actual type (see 9.1). If the parameter is an array, the form

```
ARRAY OF T
```

may be used, where the specification of the actual index bounds is omitted. The parameter is then said to be an *open array parameter*. T must be the same as the element type of the actual array, and the index range is mapped onto the integers 0 to N–1, where N is the number of elements. The formal array can be accessed elementwise only, or it may occur as actual

parameter whose formal parameter is without specified index bounds. A function procedure without parameters has an empty parameter list. It must be called by a function designator whose actual parameter list is empty too.

Restriction: If a formal parameter specifies a procedure type, then the corresponding actual parameter must be either a procedure declared at level 0 or a variable (or parameter) of that procedure type. It cannot be a standard procedure.

Examples of procedure declarations:

```
PROCEDURE Read(VAR x: CARDINAL);
  VAR i: CARDINAL; ch: CHAR;
BEGIN i:= 0;
  REPEAT ReadChar(ch)
  UNTIL (ch>= "0") & (ch<= "9");
  REPEAT i:= 10*i + (ORD(ch)-ORD("0"));
    ReadChar(ch)
  UNTIL (ch<"0") OR (ch>"9");
  x:= i
END Read

PROCEDURE Write(x,n: CARDINAL);
  VAR i: CARDINAL;
    buf: ARRAY[1..10] OF CARDINAL;
BEGIN i:= 0;
  REPEAT INC(i); buf[i]:= x MOD 10; x:= x DIV 10
  UNTIL x = 0;
  WHILE n>i DO
    WriteChar(" "); DEC(n)
  END;
  REPEAT WriteChar(CHR(buf[i] + ORD("0")));
    DEC(i)
  UNTIL i = 0;
END Write

PROCEDURE log2(x: CARDINAL): CARDINAL;
  VAR y: CARDINAL; (* assume x>0 *)
BEGIN x: = x-1; y: = 0;
  WHILE x>0 DO
    x:= x DIV 2; y:= y+1
  END;
  RETURN y
END log2
```

T

```
TRANSACTION AveragePrice (irel: Items): INTEGER;
  VAR sum: INTEGER;
USING irel FOR (=);
```

```
BEGIN
  sum := 0;
  FOR EACH item IN irel: TRUE DO sum := sum + item.price END;
  RETURN sum DIV CARD(irel)
END AveragePrice;

TRANSACTION DropItem (s: String);
USING orders[OfParts()(s)] FOR (=);
      parts FOR (=, :-) ;
BEGIN
  IF SOME p IN parts (p.itemname=s) AND
     (orders[OfParts(s)] = Deliveries{})
  THEN parts :- Items{parts[s]}
  END
END DropItem;
```

⊥

12.2 Standard procedures

Standard procedures are predefined. Some are *generic* procedures that cannot be explicitly declared, i.e. they apply to classes of operand types or have several possible parameter list forms. Standard procedures are

ABS(x)	absolute value; result type = argument type.
CAP(ch)	if ch is a lower case letter, the corresponding capital letter; if ch is a capital letter, the same letter.
CHR(x)	the character with ordinal number x. CHR(x) = VAL(CHAR,x)
FLOAT(x)	x of type CARDINAL represented as a value of type REAL.
HIGH(a)	high index bound of array a.
MAX(T)	the maximum value of type T.
MIN(T)	the minimum value of type T.
ODD(x)	x MOD 2 # 0.
ORD(x)	ordinal number (of type CARDINAL) of x in the set of values defined by type T of x. T is any enumeration type, CHAR, INTEGER, or CARDINAL.
SIZE(T)	the number of storage units required by a variable of type T, or the number of storage units required by the variable T.
TRUNC(x)	real number x truncated to its integral part (of type CARDINAL).
VAL(T,x)	the value with ordinal number x and with type T. T is any enumeration type, CHAR, INTEGER, or CARDINAL. VAL(T,ORD(x)) = x, if x of type T.

⊤

DEC(x)	x:= x–1
DEC(x,n)	x:= x–n
EXCL(s,i)	s:= s–{i} for sets, s:–{i} for relations
HALT	terminate program execution
INC(x)	x:= x+1
INC(x,n)	x:= x+n
INCL(s,i)	s:= s+{i} for sets, s:+{i} for relations

The procedures INC and DEC also apply to operands x of enumeration types and of type CHAR. In these cases they replace x by its (n-th) successor or predecessor.

The five relation handling procedures LOWEST, NEXT, THIS, HIGHEST and PRIOR select at most one element from the possibly selected relation variable rel given as the first parameter. If the element exists it is assigned to the second parameter r which must be a variable of the element type of the first parameter, and EOR(rel) becomes FALSE; if the element does not exist EOR(rel) becomes TRUE and r remains unchanged.

The procedures assume an order on the elements of the relation rel. If a relation key was specified in the declaration of rel, then this order is given by the lexicographic order on the value sets of the types of the key components. If the key list was empty, a system dependent order will be used.

LOWEST(rel, r)	selects the first element in rel.
HIGHEST(rel, r)	selects the last element in rel.
PRIOR(rel, r)	selects the predecessor of r in rel.
NEXT(rel, r)	selects the successor of r in rel.
THIS(rel, r)	selects the element in rel, that has the same ordinal value as r.
EOR()	returns, whether the last execution of any of the above listed operations selected an element r IN rel.
CARD(rex)	rex is a relation expression of any relation type and the result is the actual number of relation elements in rex; the result type is CARDINAL.

13 Modules

A module constitutes a collection of declarations and a sequence of statements. They are enclosed in the brackets MODULE and END. The module heading contains the module identifier, and possibly a number of *import lists* and an *export list*. The former specify all identifiers of objects that are declared outside but used within the module and therefore have to be imported. The export-list specifies all identifiers of objects declared within the module and used outside. Hence, a module constitutes a wall around its local objects whose transparency is strictly under control of the programmer.

Objects local to a module are said to be at the same scope level as the module. They can be considered as being local to the procedure enclosing the module but residing within a more restricted scope.

$ ModuleDeclaration = MODULE ident [priority] ";" {import} [export] block ident.
$ priority = "[" ConstExpression "]".
$ export = EXPORT [QUALIFIED] IdentList ";".
$ import = [FROM ident] IMPORT IdentList ";".

The module identifier is repeated at the end of the declaration.

The statement sequence that constitutes the *module body* is executed when the procedure to which the module is local is called. If several modules are declared, then these bodies are executed in the sequence in which the modules occur. These bodies serve to initialize local variables and must be considered as prefixes to the enclosing procedure's statement part.

If an identifier occurs in the import (export) list, then the denoted object may be used inside (outside) the module as if the module brackets did not exist. If, however, the symbol EXPORT is followed by the symbol QUALIFIED, then the listed identifiers must be prefixed with the module's identifier when used outside the module. This case is called *qualified export*, and is used when modules are designed which are to be used in coexistence with other modules not known a priori. Qualified export serves to avoid clashes of identical identifiers exported from different modules (and presumably denoting different objects).

A module may feature several import lists which may be prefixed with the symbol FROM and a module identifier. The FROM clause has the effect of unqualifying the imported identifiers. Hence they may be used within the module as if they had been exported in normal, i.e. non-qualified mode.

If a record type is exported, all its field identifiers are exported too. The same holds for the constant identifiers in the case of an enumeration type.

Examples of module declarations:

The following module serves to scan a text and to copy it into an output character sequence. Input is obtained characterwise by a procedure inchr and delivered by a procedure outchr. The characters are given in the ASCII code; control characters are ignored, with the exception of LF (line feed) and FS (file separator). They are both translated into a blank and cause the Boolean variables eoln (end of line) and eof (end of file) to be set respectively. FS is assumed to be preceded by LF.

```
MODULE LineInput;
   IMPORT inchr, outchr;
   EXPORT read, NewLine, NewFile, eoln, eof, lno;
   CONST LF = 12C; CR = 15C; FS = 34C;

   VAR lno: CARDINAL;(* line number *)
     ch: CHAR; (* last character read *)
     eof, eoln: BOOLEAN;

   PROCEDURE NewFile;
   BEGIN
     IF NOT eof THEN
       REPEAT inchr(ch) UNTIL ch = FS;
```

```
    END;
    eof: = FALSE; eoln:= FALSE; lno:= 0
  END NewFile;

  PROCEDURE NewLine;
  BEGIN
    IF NOT eoln THEN
      REPEAT inchr(ch) UNTIL ch = LF;
      outchr(CR); outchr(LF)
    END;
    eoln: = FALSE; INC(lno)
  END NewLine;

  PROCEDURE read(VAR x: CHAR);
  BEGIN (* assume NOT eoln AND NOT eof *)
    LOOP inchr(ch); outchr(ch);
      IF ch>= " " THEN
        x:= ch; EXIT
      ELSIF ch = LF THEN
        x:= " "; eoln:= TRUE; EXIT
      ELSIF ch = FS THEN
        x:= " "; eoln:= TRUE; eof:= TRUE; EXIT
      END
    END
  END read;

BEGIN eof:= TRUE; eoln:= TRUE
END LineInput.
```

The next example is a module which operates a disk track reservation table, and protects it from unauthorized access. A function procedure NewTrack yields the number of a free track which is becoming reserved. Tracks can be released by calling procedure ReturnTrack.

```
MODULE TrackReservation;
  EXPORT NewTrack, ReturnTrack;
  CONST ntr = 1024; (* no. of tracks *)
    w = 16;   (*word size*)
    m = ntr DIV w;

  VAR i: CARDINAL;
    free: ARRAY [0..m-1] OF BITSET;

  PROCEDURE NewTrack(): INTEGER;
  (* reserves a new track and yields its index as result,
     if a free track is found, and -1 otherwise *)
    VAR i,j: CARDINAL; found: BOOLEAN;
  BEGIN found:= FALSE; i:= m;
```

```
    REPEAT DEC(i); j:= w;
      REPEAT DEC(j);
        IF j IN free[i] THEN found:= TRUE END
      UNTIL found OR (j=0)
    UNTIL found OR (i=0);
    IF found THEN EXCL(free[i],j); RETURN i*w+j
    ELSE RETURN -1
    END
  END NewTrack;

  PROCEDURE ReturnTrack(k: CARDINAL);
  BEGIN (* assume 0<=k<ntr *)
    INCL(free[k DIV w], k MOD w)
  END ReturnTrack;

BEGIN (* mark all tracks free *)
  FOR i:= 0 TO m-1 DO free[i]:= {0..w-1} END
END TrackReservation.
```

14 System-dependent facilities

DBPL offers certain facilities that are necessary to program low-level operations referring directly to objects particular of a given computer and/or implementation. These include for example facilities for accessing devices that are controlled by the computer, and facilities to break the data type compatibility rules otherwise imposed by the language definition. Such facilities are to be used with utmost care, and it is strongly recommended to restrict their use to specific modules (called low-level modules). Most of them appear in the form of data types and procedures imported from the standard module SYSTEM. A low-level module is therefore explicitly characterized by the identifier SYSTEM appearing in its import list.

Note: Because the objects imported from SYSTEM obey special rules, this module must be known to the compiler. It is therefore called a pseudo-module and need not be supplied as a separate definition module (see Ch. 16).

The facilities exported from the module SYSTEM are specified by individual implementations. Normally, the types WORD and ADDRESS, and the procedures ADR, TSIZE, NEWPROCESS, TRANSFER are among them (see also Ch. 15).

The type WORD represents an individually accessible storage unit. No operation except assignment is defined on this type. However, if a formal parameter of a procedure is of type WORD, the corresponding actual parameter may be of any type that uses one storage word in the given implementation. If a formal parameter has the type ARRAY OF WORD, its corresponding actual parameter may be of any type; in particular it may be a record type to be interpreted as an array of words.

The type ADDRESS is defined as

```
ADDRESS = POINTER TO WORD
```

It is compatible with all pointer types, and also with the type CARDINAL. Therefore, all operators for integer arithmetic apply to operands of this type. Hence, the type ADDRESS can be used to perform address computations and to export the results as pointers. The following example of a primitive storage allocator demonstrates a typical usage of the type ADDRESS.

```
MODULE Storage;
  FROM SYSTEM IMPORT ADDRESS;
  EXPORT Allocate;

  VAR lastused: ADDRESS;

  PROCEDURE Allocate(VAR a: ADDRESS; n: CARDINAL);
  BEGIN a:= lastused; lastused:= lastused + n
  END Allocate;

BEGIN lastused:= 0
END Storage
```

The function ADR(x) denotes the storage address of the variable x and is of type ADDRESS. TSIZE(T) is the number of storage units assigned to any variable of type T. TSIZE is of an arithmetic type depending on the implementation.

Examples:

```
ADR(lastused)   TSIZE(Node)
```

Besides those exported from the pseudo-module SYSTEM, there are two other facilities whose characteristics are system-dependent. The first is the possibility to use a type identifier T as a name denoting the *type transfer function* from the type of the operand to the type T. Evidently, such functions are data representation dependent, and they involve no explicit conversion instructions.

The second non-standard facility is used in variable declarations. It allows to specify the absolute address of a variable and to override the allocation scheme of a compiler. This facility is intended for access to storage locations with specific purpose and fixed address, such as e.g. device registers on computers with "memory-mapped I/O". This address is specified as a constant integer expression enclosed in brackets immediately following the identifier in the variable declaration. The choice of an appropriate data type is left to the programmer.

15 Processes

Being a Modula-2 extension, DBPL is designed primarily for implementation on a conventional single-processor computer. For multiprogramming it offers only some basic facilities which allow the specification of quasi-concurrent processes and of genuine concurrency for peripheral devices. The word *process* is here used with the meaning of *coroutine*. Coroutines are processes that are executed by a (single) processor one at a time.

15.1 Creating a process and transfer of control

A new process is created by a call to

```
PROCEDURE NEWPROCESS(P: PROC; A: ADDRESS; n: CARDINAL;
                     VAR p1: ADDRESS)
```

P denotes the procedure which constitutes the process,
A is the base address of the process' workspace,
n is the size of this workspace,
p1 is the result parameter.

A new process with P as program and A as workspace of size n is assigned to p1. This process is allocated, but not activated. P must be a parameterless procedure declared at level 0.

A transfer of control between two processes is specified by a call to

```
PROCEDURE TRANSFER(VAR p1, p2: ADDRESS)
```

This call suspends the current process, assigns it to p1, and resumes the process designated by p2. Evidently, p2 must have been assigned a process by an earlier call to either NEWPROCESS or TRANSFER. Both procedures must be imported. A program terminates, when control reaches the end of a procedure which is the body of a process.

Note: assignment to p1 occurs after identification of the new process p2; hence, the actual parameters may be identical.

15.2 Device processes and interrupts

If a process contains an operation of a peripheral device, then the processor may be transferred to another process after the operation of the device has been initiated, thereby leading to a concurrent execution of that other process with the *device process*. Usually, termination of the device's operation is signalled by an interrupt of the main processor. In terms of DBPL, an interrupt is a transfer operation. This interrupt transfer is (in Modula-2 implemented on the PDP-11) preprogrammed by and combined with the transfer after device initiation. This combination is expressed by a call to

```
PROCEDURE IOTRANSFER(VAR p1, p2: ADDRESS; va: CARDINAL)
```

In analogy to TRANSFER, this call suspends the calling device process, assigns it to p1, resumes (transfers to) the suspended process p2, and in addition causes the interrupt transfer occurring upon device completion to assign the interrupted process to p2 and to resume the device process p1. va is the interrupt vector address assigned to the device. The procedure IOTRANSFER must be imported from the module SYSTEM, and should be considered as PDP-11 implementation-specific.

It is necessary that interrupts can be postponed (disabled) at certain times, e.g. when variables common to the cooperating processes are accessed, or when other, possibly time-critical

operations have priority. Therefore, every module is given a certain priority level, and every device capable of interrupting is given a priority level. Execution of a program can be interrupted, if and only if the interrupting device has a priority that is greater than the priority level of the module containing the statement currently being executed. Whereas the device priority is defined by the hardware, the priority level of each module is specified by its heading. If an explicit specification is absent, the level in any procedure is that of the calling program. IOTRANSFER must be used within modules with a specified priority only.

16 Compilation units

A text which is accepted by the compiler as a unit is called a *compilation unit*. There are three kinds of compilation units: main modules, definition modules, and implementation modules. A main module constitutes a main program and consists of a so-called *program module*. In particular, it has no export list. Imported objects are defined in other (separately compiled) program parts which themselves are subdivided into two units, called definition module and implementation module.

The *definition module* specifies the names and properties of objects that are relevant to clients, i.e. other modules which import from it. The *implementation module* contains local objects and statements that need not be known to a client. In particular the definition module contains the export list, constant, type, and variable declarations, and specifications of procedure, selector and constructor headings. The corresponding implementation module contains the complete procedure declarations, and possibly further declarations of objects not exported. Definition and implementation modules exist in pairs. Both may contain import lists, and all objects declared in the definition module are available in the corresponding implementation module without explicit import.

Compilation units prefixed with the symbol DATABASE are called *database modules*. Variables declared within a database module which reside at the same scope level as the modules are called persistent variables. Their lifetime is longer than that of any program importing the database module. The set of all persistent variables of a database module constitutes a database. Persistent variables are shared objects, i.e. they can be accessed by several programs simultaneously. An access to a persistent variable must be part of a transaction execution. During the transaction execution the calling program is guaranteed to be the only one accessing the respective persistent variables; the values of these variables upon entry to the transaction however cannot be deduced from the program.

```
$ CompilationUnit = [DATABASE] (DefinitionModule | ImplementationModule |
$       ProgramModule). $ DefinitionModule = DEFINITION MODULE ident ";"
$       {import} {definition} END ident ".".
$ definition = CONST {ConstantDeclaration ";"} |
$       TYPE {ident ["=" type] ";"} |
$       VAR {VariableDeclaration ";"} |
$       SelectorHeading ";" | SelectorDeclaration ";" |
$       ConstructorHeading ";" | ConstructorDeclaration ";" |
$       ProcedureHeading ";".
$ ImplementationModule = IMPLEMENATION MODULE ident [priority] ";"
```

```
$       {import} declaration
$       [DATABASE DEFINITON StatementSequence]
$       [DATABASE IMPLEMENTATION StatementSequence]
$       [BEGIN StatementSequence] END ident ".".
$ ProgramModule = MODULE ident [priority] ";" {import} declaration
$       [DATABASE StatementSequence] [BEGIN StatementSequence] END ident ".".
```

⊥

The definition module evidently represents the interface between the implementation module on one side and its clients on the other side. The definition module contains those declarations which are relevant to the client modules, and presumably no other ones. Hence, the definition module acts as the implementation module's (extended) export list, and all its declared objects are exported.

Definition modules imply the use of qualified export. Type definitions may consist of the full specification of the type (in this case its export is said to be transparent), or they may consist of the type identifier only. In this case the full specification must appear in the corresponding implementation module, and its export is said to be *opaque*. The type is known in the importing client modules by its name only, and all its properties are hidden. Therefore, procedures operating on operands of this type, and in particular operating on its components, must be defined in the same implementation module which hides the type's properties. Opaque export is restricted to pointers. Assignment and test for equality are applicable to all opaque types.

As in local modules, the body of an implementation module acts as an initialisation facility for its local objects. Before its execution, the imported modules are initialized in the order in which they are listed. If circular references occur among modules, their order of initialization is not defined. ⊤

The body of a database module may contain initialization statements for each kind of compilation unit. This initialization is executed only once during database lifetime, namly before any access to the persistent variables has been made.

It is also possible to define the body of a selector or constructor in the definition part of a module. These selectors and constructors must not be redefined in the implementation module. ⊥

References

[BJM+89] A. Borgida, M. Jarke, J. Mylopoulos, J.W. Schmidt, and Y. Vassiliou. The Software Development Environment as a Knowledge Base Management System. In J.W. Schmidt and C. Thanos, editors, *Foundations of Knowledge Base Management*, Topics in Information Systems. Springer-Verlag, 1989.

[BJS86] S. Böttcher, M. Jarke, and J.W. Schmidt. Adaptive Predicate Managers in Database Systems. In *Proc. of the 12th International Conference on VLDB*, Kyoto, 1986.

[Böt90] S. Böttcher. Improving the Concurrency of Integrity Checks and Write Operations. In *Proc. ICDT 90*, Paris, December 1990.

[Car88] L. Cardelli. Types for Data-Oriented Languages. In *Advances in Database Technology, EDBT '88*, volume 303 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1988.

[EEK+85] H. Eckhardt, J. Edelmann, J. Koch, M. Mall, and J.W. Schmidt. Draft Report on the Database Programming Language DBPL. DBPL-Memo 091-85, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1985.

[ERMS91] J. Eder, A. Rudloff, F. Matthes, and J.W. Schmidt. Data Construction with Recursive Set Expressions in DBPL. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

[JGL+88] W. Johannsen, L. Ge, W. Lamersdorf, K. Reinhard, and J.W. Schmidt. Database Application Support in Open Systems: Language Support and Implementation. In *Proc. IEEE 4th Int. Conf. on Data Engineering*, Los Angeles, USA, February 1988.

[JK83] M. Jarke and J. Koch. Range Nesting: A Fast Method to Evaluate Quantified Queries. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 196–206, May 1983.

[JLRS88] W. Johannsen, W. Lamersdorf, K. Reinhard, and J.W. Schmidt. The DURESS Project: Extending Databases into an Open Systems Architecture. In *Advances in Database Technology, EDBT '88*, volume 303 of *Lecture Notes in Computer Science*, pages 616–620. Springer-Verlag, 1988.

[JLS85] M. Jarke, V. Linnemann, and J.W. Schmidt. Data Constructors: On the Integration of Rules and Relations. In *11th Intern. Conference on Very Large Data Bases, Stockholm*, August 1985.

[JMW+90] M. Jeusfeld, M. Mertikas, I. Wetzel, Jarke. M., and J.W. Schmidt. Database Application Development as an Object Modelling Activity. In *Proc. 16th VLDB Conference*, Brisbane, Australia, August 1990.

[MS89] F. Matthes and J.W. Schmidt. The Type System of DBPL. In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, pages 255–260, June 1989.

[MS91] F. Matthes and J.W. Schmidt. Towards Database Application Systems: Types, Kinds and Other Open Invitations. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

[MS92] F. Matthes and J.W. Schmidt. DBPL User and System Manual. FIDE Technical Report FIDE/92/??, Fachbereich Informatik, Universität Hamburg, West Germany, 1992.

[MSS91] F. Matthes, G. Schröder, and J.W. Schmidt. VAX Modula-2 User's Guide; VAX DBPL User's Guide. DBPL Memo 121-91, Fachbereich Informatik, Universität Hamburg, West Germany, December 1991.

[Sch77] J.W. Schmidt. Some High Level Language Constructs for Data of Type Relation. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Toronto, Canada*, August 1977.

[SEM88] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 112-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1988.

[SGLJ89] J.W Schmidt, L. Ge, V. Linnemann, and M. Jarke. Integrated Fact and Rule Management Based on Database Technology. In J.W. Schmidt and C. Thanos, editors, *Foundations of Knowledge Base Management*, Topics in Information Systems. Springer-Verlag, 1989.

[SL85] J.W. Schmidt and V. Linnemann. Higher Level Relational Objects. In *Proc. 4th British National Conference on Databases (BNCOD 4)*. Cambridge University Press, July 1985.

[SM89] J.W. Schmidt and F. Matthes. Advances in Database Programming: On Concepts, Languages and Methodologies. In *Proc. 16th SOFSEM'89*, Ždiar, High Tatra, ČSSR, December 1989. Available through Hamburg University.

[SM90a] J.W. Schmidt and F. Matthes. DBPL Language and System Manual. Esprit Project 892 MAP 2.3, Fachbereich Informatik, Universität Hamburg, West Germany, April 1990.

[SM90b] J.W. Schmidt and F. Matthes. Language Technology for Post-Relational Data Systems. In A. Blaser, editor, *Database Systems of the 90s*, volume 466 of *Lecture Notes in Computer Science*, pages 81–114, November 1990.

[SM91a] J.W. Schmidt and F. Matthes. Modular and Rule-Based Database Programming in DBPL. FIDE Technical Report FIDE/91/15, Fachbereich Informatik, Universität Hamburg, West Germany, February 1991.

[SM91b] J.W. Schmidt and F. Matthes. Naming Schemes and Name Space Management in the DBPL Persistent Storage System. In *Proceedings of the Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts.* Morgan Kaufmann Publishers, January 1991.

[SWBM89] J.W. Schmidt, I. Wetzel, A. Borgida, and J. Mylopoulos. Database Programming by Formal Refinement of Conceptual Designs. *IEEE – Data Engineering*, September 1989.