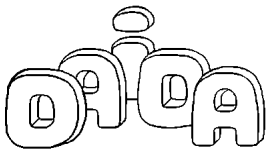

ESPRIT PROJECT 892, DAIDA

DEVELOPMENT OF ADVANCED INTERACTIVE DATA-INTENSIVE APPLICATIONS

Title: # DBPL Language and System Manual

Authors: Joachim W. Schmidt
Florian Matthes
Fachbereich Informatik, Universität Hamburg,
Schlüterstraße 70, D-2000 Hamburg 13, FRG

Project leader: Joachim W. Schmidt

Workpackage: Month 48, MAP 2.3

Status: initial document

Distribution: Partners, CEC

Date: April 1990

Keywords: D.3.0 [Programming Languages, General] *DBPL, Modula-2*
D.3.3 [Programming Languages, Constructs] *data types and data structures, modules, packages*
D.3.4 [Processors] *compilers, run-time environments*
H.2.3 [Database Management, Languages] *DDL, DML*

Partners: BIM S.A.I.N.V. (Belgium)
Groupe Française d'Informatique (France)
SCS Technische Automation und Systeme (FRG)
Universität Hamburg (FRG)
Universität Passau (FRG)
Cretan Computer Institute (Greece)
BP Research Center (UK)

Abstract

The database programming language DBPL [?] is a successor of Pascal/R [?] and integrates a set- and predicate-oriented view of relational database modelling into the system programming language Modula-2 [?]. Based on integrated programming language and database technology it offers a uniform framework for the implementation and maintenance of database application software.

This document describes the rationale for the design of DBPL, defines the elements of the DBPL language, and explains the use of the system software available for VAX/VMS systems. A small, self-contained application program is presented in chapter ?? to illustrate fundamental aspects of application programming in DBPL (data definition, data manipulation, modularization, user-interaction).

Chapter ?? gives a detailed description of the syntax and semantics of DBPL. It 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 in the margins ($\top$, $\bot$).

Intended Audience

This manual is written for readers who have a general knowledge of procedural programming languages and of the relational data model. Users of the DBPL system software should read the installation instructions (Chapter ??) first. Chapter ?? and ?? introduce the main language concepts and should be of general interest. The example program in Chapter ?? illustrates the spirit of modular program development using DBPL. Chapter ?? is intended as a reference for programmers and language implementors.

Related Documents

Data- and Rule-Based Database Programming in DBPL [?]

The Type System of DBPL [?]

DBPL System: The Prototype and its Architecture [?]

VAX-11 Modula-2 Users Guide

VAX-11 Symbolic Debugger Reference Manual

VAX-11 Linker Reference Manual

Revision History

April 1990:
: Initial version, corrections to DBPL report as of November 1988. Installation instructions for DBPL VAX/VMS software version 2.03.

November 1990:
: Installation instructions for DBPL VAX/VMS software version 2.1.

Contents

Chapter 1

Rationale of DBPL

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. The database programming language DBPL presented in this manual concentrates on the last issue to offer a uniform and consistent framework for the *efficient implementation* of database applications.

As a first cut, data-intensive applications may be characterized by their needs to specify and manipulate *heavily constrained* data that are *long-lived* and may be *shared* by a user community. These requirements are direct consequences of the fact that those data serve as (partial) representations of some organizational unit or physical structure that exist in their own 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, those representational data may become extremely voluminous – in current data-intensive applications up to $O(10^9)$ or even higher.

In addition to the need for global management of large amounts of *representational data*, typical database 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 this 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 application that determines the daily life of a database programmer.

DBPL [?] is a successor of Pascal/R [?] and addresses the above issues by extending a well-understood system programming language (namely Modula-2 [?]) by a small set of abstraction mechanisms needed for database application programming [?]. 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, recovery, and concurrency control.

The design of the DBPL language was influenced by requirements of the DAIDA environment for database application development [?, ?, ?] that asks for a target language that

- makes extensive use of sets and first-order predicates, and
- provides complete separation of typing and persistence.

An essential guideline for the design of DBPL can be characterized by the slogan "power through orthogonality". Instead of designing a new language (with its own naming, binding, scoping and typing rules) from scratch, DBPL extends an existing language and puts particular emphasis on the interoperability of the new "database" concepts with the given programming language concepts. 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,
- 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 utilization 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* (see **??**) (i.e. first-order logic abstractions of bulk data access) in typical DBPL programs facilitates such an approach.

Following the spirit of Modula-2, DBPL violates in some cases the design principle of "orthogonality of language concepts" in order to enable a well-engineered implementation of the language. 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 [?, ?].

Modula-2 was chosen to be the basis for DBPL as it is characterized by a clear module concept and a strict type system. It is further excelled by its balance between simplicity and expressive power.

In contrast to some persistent programming languages [?, ?, ?, ?], the evolution of database schemata and of application programs is outside the scope of the language DBPL and is delegated to specific components of the DAIDA environment [?, ?, ?, ?].

To summarize, the most prominent feature of the DBPL 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.

Chapter 2

Language Concepts for Database Programming

The following subsections highlight the central new language concepts of DBPL. Programmers familiar with a "traditional" programming language like Modula-2 only need to understand these few additional features to successfully utilize the database capabilities of DBPL. Due to its orthogonal language design, there are no *special* rules governing the naming, typing, binding, scoping, sequencing etc. of these Modula-2 extensions.

In particular, it should be noted that every correct Modula-2 program is also a correct DBPL program, i.e., DBPL is a fully upward compatible Modula-2 extension.

2.1 Data Type Relation

A relation type declaration specifies a structure consisting of a set of elements of the same type. The cardinality of this set is basically unlimited. Relations extend the notion of sets by allowing to specify a key, i.e. a substructure of the set element that is guaranteed to have a unique value within a relation and, therefore, provides (associative) identification.

In DBPL, the definition of a relation type may contain any nesting of records, arrays, variant records and relations. The only restriction is that on each level of nesting there has to be at least one non-relational attribute and that key attributes of non-first normal-form relations (NF^2 relations) are non-relational.

```
TYPE
  Nodes  = RELATION OF CARDINAL;
  Edges  = RELATION OF RECORD a,b: CARDINAL; END;
  NF2Rel = RELATION a,b OF
             RECORD
               a,b: CARDINAL;
               subrel: Edges;
             END;
VAR
```

```
  N, NWE: Nodes;
  E: Edges;
  P: NF2Rel;
```

In the above example, `Nodes` and `Edges` are true set types, whereas `NF2Rel` is a nested relation type with key attributes `a` and `b`.

Relation types are "orthogonal" type constructors, i.e. they coan can occur as elements of arbitrary structures like arrays or records,

```
VAR
  Table: ARRAY[0..10] OF Nodes;
```

and relation variables can appear in arbitrary scopes and extents, for example, in objects of other data types, i.e.

- as persistent, global, local, dynamic variables, and
- as value and reference parameters.

```
PROCEDURE SwapNodes (VAR n1, n2: Nodes);
  VAR
    n: Nodes;
BEGIN
  n:= n1; n1;= n2; n2:= n;
END SwapNodes;
```

Note, that relations in the "classical" Relational Data Model are simply relations with elements of "flat" record types, i.e. records that only contain fields of the base types (INTEGER, REAL, ...).

Modula-2 identifies elements of arrays by index expressions and record components by field names. Analogically, relation elements are identified by their key values. Therefore, substructures of hierarchical objects are uniformly accessed by a sequence of the previously mentioned identification mechanisms.

```
  P[1,5].subrel := Edges{}
```

This assignment replaces the value of the `subrel` attribute of an element `p` with the key values `p.a=1` and `p.b=5` in the relation `P` by the empty relation of type `Edges`.

2.2 Aggregates

DBPL allows the denotation of structured values by means of aggregates, which are simply an enumeration of (possibly structured) values prefixed by the name of the type to be constructed.

```
N:= Nodes{1,2,3,4};
E:= Edges{ {1,2}, {2,3}, {3,4}};
P:= NF2Rel{ {1,4,E} };
```

As illustrated by the example above, the type identifier can be omitted in nested expressions where the aggregate type can be deduced from its context.

2.3 First-Order Predicates

Any boolean expression in DBPL can contain first-order predicates with existential (`SOME`) and universal (`ALL`) quantifiers ranging over relation expressions. The use of (nested) predicates often allows to avoid explicit iterations over relations and thereby facilitates access optimization to be performed by the DBPL system:

```
IF SOME n IN N (NOT SOME e IN E (e.a = n)) THEN
  WriteString("There is a node without outgoing edges")
END;
IF NOT ALL e IN E (
  SOME n IN N (e.a = n) AND
  SOME n IN N (e.b = n)) THEN
  WriteString("There are edges from/to non-existent nodes")
END;
```

First-order predicates play a central role in the formulation of access expressions which are the subject of the next paragraphs.

2.4 Access Expressions

Access Expressions are selection and construction rules for relations. They generalize and unify the concepts of set-oriented retrieval, element-oriented iteration and (updateable) views found in relational database systems.

Selctive access expressions define subrelations of existing relation variables. The following expressions select the elements `e` of the relation variable `E`, fulfiling the selection predicate `e.a=5` respectively `SOME n IN N (e.a =n)`:

```
EACH e IN E: e.a = 5
EACH e IN E: SOME n IN N (e.a = n)
```

The first selection expression selects all edges having 5 as their start node, whereas the second selection expression selects all edges starting from a node contained in the set `N`.

Constructive access expressions denote relation expressions that are derived from combinations of existing relations:

```
{n1,n2} OF EACH n1 IN N, EACH n2 IN N: TRUE
{e1.a,e2.b} OF EACH e1 IN E, e2 IN E: (e1.b = e2.a)
```

The first expression constructs the cartesian product N×N. The second constructive expression builds a set of edges, containing an edge for each pair of nodes that are connected by a path of length 2.

Please note that access expressions (like boolean expressions) have to be used in a conforming context. There are three possible contexts for access expressions in DBPL:

Relation constructors build a new relation containing (copies of) the elements denoted by an access expression: (set-at-a-time operations)

```
Edges{EACH e IN E: e.a = 5}
Edges{EACH e IN E: SOME n IN N (e.a = n)}
Edges{{n1,n2} OF EACH n1 IN N, EACH n2 IN N: TRUE}
```

Relation iterators iterate over a subrelation denoted by a selective access expression: (element-at-a-time operations)

```
FOR EACH e IN E: e.a = 5 DO WriteCard(e.b, 1) END;
FOR EACH e IN E: SOME n IN N (e.a = n) DO WriteCard(e.b, 1) END;
```

Selector and constructor declarations allow to name and parameterize a given access expression (see section ?? and ??). (Lambda abstraction)

```
SELECTOR StartAt WITH(x: Node): Edges;
BEGIN EACH e IN E: e.a = x END StartAt;

CONSTRUCTOR AllPossibleEdges: Edges;
BEGIN EACH n1 IN N, EACH n2 IN N: TRUE END Edges;
```

As quantifiers may be nested and the range expression of the inner quantifier may depend on the quantified variable of the outer scope, it is possible to "descend into nested relations:

```
E:= Edges{e OF EACH p IN P, EACH e IN p.subrel: TRUE}
```

On the other hand, the nesting of aggregates and relational constructors in the projection list of a construction predicate facilitates the construction of nested relations.

```
NF2Rel { { e.a, e.b, {e}} OF EACH e IN E: TRUE }
```

To summarize, access expressions and first-order predicates form a relationally complete query formalism, exceeding the power of existing relational DBMS.

2.5 Relation Operations

In addition to relation iteration and relation assignment as illustrated above, DBPL has predefined relation operations for set-oriented insertion, deletion and update:

```
E:= E2; E:+ E2; E:- E2; E:& E2
```

Furthermore, there are the usual infix operators (`=, <, >, <=, >=, <>`) to compare two relations of the same type on equality, (strict) set containment, or inequality. Therefore, one can write more succintly `E <= E2` instead of

```
ALL e IN E SOME e2 IN E2 ((e.a=e2.a) AND (e.b=e2.b))
```

2.6 Databases and Persistence

Any module (separate compilation unit) can be declared as a database module by prefixing it with the keyword `DATABASE`. All variables declared in a database module are persistent, i.e. their lifetime is not restricted to a single program execution and exceeds the lifetime of all programs importing them. Furthermore, they are *shared* variables, i.e. they can be accessed by several programs (concurrently). Procedure variables and pointers are not allowed within a database module. The declaration of databases is not restricted to definition modules, which permits the definition of persistent abstract data types.

Persistent variables may only be accessed during transaction execution (see section ??). The DBPL runtime system will detect all violations to this rule.

The initialization of persistent variables has to be accomplished before they are accessed for the first time. This is done by extending the original semantics of the module initialization code of Modula-2. Every database module may contain a supplementary initialization code, which is executed only once at the beginning of the existence of a persistent variable.

```
DATABASE MODULE PersistentGraph;

TYPE
  Nodes  = RELATION OF CARDINAL;
  Edges  = RELATION OF RECORD a,b: CARDINAL; END;
VAR
  N : Nodes;
  E : Edges;

TRANSACTION InitDB;
(* this transaction will be called only one during DB lifetime *)
BEGIN
  N:= Nodes{1};
  E:= Edges{{1,1}};
END InitDB;
```

```
DATABASE
  InitDB;
BEGIN
  (* standard Modula-2 initialization code *)
END PersistentGraph.
```

2.7 Transactions

As the unit of concurrency control and recovery, a transaction comprises a sequence of actions and is regarded atomic concerning its effect on the database. Transaction in DBPL can be named and parameterized. In case of nested or recursive calls of transactions, the inner calls are treaded as ordinary procedure calls, i.e. DBPL only provides a "'flat transaction model.

The following transactions deletes all nodes in `N` that are reachable through a path of arbitrary length of edges in `E` from a giving starting node `n`. It uses a standard depth-first search traversal strategy.

```
TRANSACTION DeleteSuccessors(n: Node);
(* iterative solution *)
VAR
  visited: Nodes;

  PROCEDURE VisitNode(n: Node);
  BEGIN
    visited:+ Nodes{n};
    FOR EACH succ IN N:
      SOME e IN E ((e.a = n) AND (e.b = succ))
      AND NOT succ IN visited DO
       VisitNode(succ);
    END;
  END VisitNode;

BEGIN
  visited:= Nodes{};
  VisitNode(n);
  N:- visited;
END DeleteSuccessors;
```

If there are other transactions reading or updating the database variables `N` or `E` simultaneously, the DBPL system will execute them together with `DeleteSuccessors` in a serializable schedule.

2.8 Selectors

Selectors are means to describe value-based constraints on relation variables. The application of a selector defines a selected relation variable, which is equivalent to an updatable view in database systems.

The declaration of a selector has the following general structure:

```
SELECTOR sp ON (R : RelType)
            WITH ( formal parameters )
            FOR ( access restrictions )
BEGIN
  selective access expression
END sp;
```

The ON-parameter allows selectors to be bound to a relation type and not only to a specific relation variable.

```
SELECTOR NodesWithEdges ON (X: Nodes);
BEGIN
  EACH n IN X: SOME e IN E ((e.a = n) OR (e.b = n))
END NodesWithEdges;
```

By declaring access restrictions in the signature of a selector (FOR-parameter), it is possible to restrict the operations allowed on relation variables. The access restrictions are connected to the selector type.

```
SELECTOR ReadNodesWithEdges ON (X: Nodes) FOR (=);
BEGIN
  EACH n IN X: SOME e IN E ((e.a = n) OR (e.b = n))
END ReadNodesWithEdges;
```

Within an expression, the value of a selected relation variable is equal to the value of a relation created by a relational query expression, based on the selective access expression of the selector.

```
NWE:= N[NodesWithEdges]
```

Informally, the semantics of updates to selected variables is defined as follows:

> updates are executed if and only if a new relation value can be constructed, such that the non-selected part of the new relation is equal to the non-selected part of the relation before the update, and the selected part of the new relation is equal to the right-hand-side of the update statement.

With the current selector semantics, any violation to this constraint will cause the update operation not to be executed at all.

```
N[NodesWithEdges]:= Nodes{1,2,5};
```

As node 5 does not belong to any edge in E, the assignment above will not be executed.

By virtue of these semantics of updates on selected relation variables, selectors can be used to enforce value-based integrity constraints on relation variables. Using the optional access restrictions in a selector heading one can constrain access to selected relation variables even further, namely to those kinds of operations explicitely listed in the heading of the selector. Access restrictions are statically verified by the compiler:

```
N[ReadNodesWithEdges]:= N;
```

Since the declaration of ReadNodesWithEdges restricts the use of the selected relation variable to read operations, the compiler disallows the assignment above.

2.9 Constructors

Constructors are named and paramaterized construction rules for relations. They are used to define derived relations by a list of constructive access expressions. These expressions refer to relation variables or to other derived relations. One can show an equivalence between constructors in DBPL and certain models of deductive database programs, which have been shown to correspond to Horn-clause logic. Constructors are therefore considerably more expressive than the relational query formalisms.

2.9.1 Declaration

The signature of a constructor defines its name, formal parameters and result type. Similar to selectors, constructors are first-order objects, which are typed according to their signature. The body of a constructor declaration is a list of relational expressions, defining a relation of the result type.

```
CONSTRUCTOR NonDirectedEdges: Edges;
BEGIN
  EACH e IN E: TRUE,
  {e.b, e.a} OF EACH e IN E: TRUE
END NonDirectedEdges;
```

2.9.2 Application

The application of constructors is restricted to relational expressions (i.e. update operations on derived relations are inadmissible). The substitution of actual parameters is executed according to the rules defined for selectors (see **??**). The value of a constructor application is obtained by an evaluation of the predicates within the constructor body, which may contain further constructors.

```
E:= Edges{NonDirectedEdges}
```

2.9.3 Recursive Constructors

A set of constructor declarations may contain cyclic references. The semantics of the application of such a recursive constructor is defined by the least fixed point of a system of relation-valued functions. The existence and uniqueness of this fixed point is guaranteed for a (syntactically) restricted class of constructors.

```
CONSTRUCTOR Closure ON (X: Edges): Edges;
BEGIN
  EACH e IN X: TRUE,
  {e1.a, e2.b} OF EACH e1 IN X, EACH e2 IN Edges{Closure(X)}: e1.b = e2.a
END Closure;
```

The above constructor defines a relation that contains an edge for all node pairs that are connected by a (directed) path of arbitrary length.

Using recursive constructors one can express the transaction of section ?? that deletes all transitive successors of a given node more declaratively as

```
TRANSACTION DeleteSuccessors(n: Node);
BEGIN
  N:- Nodes{EACH succ IN N: SOME e IN Edges{Closure(E)}
              ((e.a=n) AND (e.b=succ))}
END DeleteSuccessors;
```

This version of the transaction should be as least as efficient as the previous iterative solution and it leaves even more possibilities for access optimizations to be performed by the DBPL run-time system.

Chapter 3

DBPL Language Report

3.1 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.

3.2 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 **??**).

```
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.

3.3 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 **??**) 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. **??**), it extends to all those units which import M1.

4. Field identifiers of a record declaration (see **??**) 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. **??**) 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	(??)	LONGINT	(??)
BITSET	(??)	LONGREAL	(??)
BOOLEAN	(??)	LOWEST	(??)
CAP	(??)	MAX	(??)
CARD	(??)	MIN	(??)
CARDINAL	(??)	NEXT	(??)
CHAR	(??)	NIL	(??)
CHR	(??)	ODD	(??)
DEC	(??)	ORD	(??)
EOR	(??)	PRIOR	(??)
EXCL	(??)	PROC	(??)
FALSE	(??)	REAL	(??)
FLOAT	(??)	RESTRICTED	(??)
HALT	(??)	THIS	(??)
HIGH	(??)	SIZE	(??)
HIGHEST	(??)	TRUE	(??)
INC	(??)	TRUNC	(??)
INCL	(??)	VAL	(??)
INTEGER	(??)		

3.4 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. **??**).

Examples of constant declarations are

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

3.5 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 = [1..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)
```

3.5.1 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).

3.5.2 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)
```

3.5.3 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]
```

3.5.4 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
```

3.5.5 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 **??**) 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.

3.5.6 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]
```

3.5.7 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.

3.5.8 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 [FormalTypeList].
$ FormalTypeList = "(" [[VAR] FormalType
$          {"," [VAR] FormalType}] ")" [":" qualident].
```

The standard type PROC denotes a parameterless procedure:

```
PROC = PROCEDURE
```

3.5.9 Relation types

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 AddressClass 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;
```

3.5.10 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. **??**). 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. **??**). 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 ON (Items) FOR (=)
```

⊥

3.5.11 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
```

⊥

3.6 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. ??):

```
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
suppliers: Companies
orders: Deliveries
InRange: ItemsInRange
adjacent: ConnectionRule
```

3.7 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.

3.7.1 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. ⊥

3.7.2 Selectors

⊤

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

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

$ 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 ??.
- 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. ?? together with their types in Ch. ?? and ??):

```
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
```

⊥

3.7.3 Constructors

⊤

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

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.

⊥

3.7.4 Operands

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

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 **??**).

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

This identifier may possibly be qualified by module identifiers (see Ch. **??** and **??**), 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 **??**). 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 **??**).

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. **??**).

Examples of designators (see examples in Ch. **??**):

```
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)
```

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)
```

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 **??**,
- 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,100,p.connectedTo} OF
      EACH p IN oldparts: p.price<1000}          (Items)
Connections{TransConn(parts)}                    (Connections)
```

⊥

3.7.5 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.

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)

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. ⊥

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)
```

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. **??**), 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 compare values of relation type, and then denote relation equality or inclusion. 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. ??):

```
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)
[RedParts] <=
Items{EACH p IN parts: p.price < 100}   (BOOLEAN)
thispart IN parts                       (BOOLEAN)
                                                    ⊥
```

3.8 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.

3.8.1 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 **??**)

```
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 **??**) is a special case of selective relation assignment.

The standard procedure RESTRICTED (see **??**) allows to detect violations of a selector or a key constraint that occured during relation insertion, deletion, replacement or 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{}
```

⊥

3.8.2 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. **??**). 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])
```

3.8.3 Statement sequences

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

$ StatementSequence = statement {";" statement}.

3.8.4 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
```

3.8.5 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
```

3.8.6 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
```

3.8.7 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
```

3.8.8 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 ??). 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 + p.quantity;
END;
FOR EACH item: RedItems(parts) DO
  item.price:= item.price DIV 2;
END
```

⊥

3.8.9 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.

3.8.10 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
```

3.8.11 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. **??**).

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 **??**).

3.9 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 **??** 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
```

3.10 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 **??**).

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

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

⊥

3.11 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. **??**) 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 the case of nested or recursive calls of transaction procedures, the entire transaction is "flattened", i.e. inner calls are treaded as ordinary procedure calls.

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. ⊥

3.11.1 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 **??**), in the case of value parameters the formal type must be assignment compatible with the actual type (see **??**). 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
```

```
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 ORD(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[OfPart()(s)] = Deliveries{})
  THEN parts :- Items{parts[s]}
  END
END DropItem;
```

3.11.2 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.
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.
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.
RESTRICTED()	a value of type BOOLEAN that indicates whether an update operation on a (selected) relation was not executed due to a violation of a key or a selector constraint.
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 (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() becomes FALSE; if the element does not exist EOR() becomes TRUE and r remains unchanged.

The procedures provide an ordered access to the elements of the relation rel. This order is implied by the lexicographic order on the value sets of the types of the key components.

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 a value of type BOOLEAN that indicates whether the last application of one of the above operations selected an element r IN rel.

3.12 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.
```

3.13 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. ??).

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. ??).

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.

3.14 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.

3.14.1 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.

3.14.2 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.

3.15 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 ";" | ConstructorHeading ";" |
$         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. ⊥

Chapter 4

DBPL Installation Guide

The DBPL distribution tape contains the DBPL system (a compiler and a runtime system) as well as a Modula-2 compiler, both running under VAX/VMS and developed at the University of Hamburg. Both compilers are based on the Modula-2 compiler M2RT11 developed at the "Institut für Informatik, Eidgenössische Technische Hochschule Zürich".

The DBPL compiler is able to utilize the symbolfiles produced by the Modula-2 compiler. Using Foreign-Definition modules (see the Modula-2 User's Guide on tape), it is furthermore possible to develop arbitrary mixed language systems with the DBPL compiler.

The current implementation of the DBPL system is still incomplete, please consult section ??) for details.

4.1 First Installation of DBPL

Before proceeding with the installation process, please make sure that you have enough disk space available. Approximately 8000 disc blocks are needed to install both compilers and their libraries.

1. Mount the tapedrive by

   `$ mount/foreign` *tape_drive*

   where *tape_drive* is the name of the used tape drive, e.g. `mua0:`.

2. Define the directory on which the top-directory of the DBPL system should reside, as the default directory.

3. Copy the system from tape or TK-50 by

   `$ backup/log` *tape_drive* `[...]*.*.*`

4. Go to `[.dbpl]` and load the file *install_dbpl.com* with an editor. The first line of this file is

   ```
   $ define $dbpll  "''f$trnlnm("$your_disk")'[path_to.dbpl.]"/trans=(conc,term)
   ```

   On the first line of this file, change `$your_disk` to the (logical) name of your actual disc and `path_to` to the actual path of the installation directory.

5. Execute `@install_dbpl.com`.
6. Dismount the tape:

   `$ dismount` *tape_drive*

4.2 User Access to DBPL

1. Every user of DBPL has to execute `install_dbpl.com` in the directory `[.dbpl]`(e.g. in his `login.com` file).
2. Make sure, that the file protection of the DBPL system files and system directories allows read-access (respectively execute-access for `*.exe` files) for the designated users.
3. Every DBPL user needs an open file quota of at least 100. This is necessary for the DBPL runtime system to handle persistent and temporary relations. The standard settings for the open file quota will lead to problems *even for small DBPL applications*!

4.3 Using DBPL

A step-by step example is given in the next section.

1. Create the DBPL source files.
2. Compile them with `xdb/`*command_qualifier file_name*. Look at the file `$DBPL:xdb.cld` for allowed command qualifiers. *DATABASE MODULE's* have to be compiled with the qualifier `/create` to create an (uninitialized) database. The other command qualifiers are described in the VAX-11 Modula-2 User's Guide. The first application program accessing that database will execute the user-definable database initialization code for that database.
3. Link your objectfiles with `link` $file_1.obj, \ldots, file_n.obj$. File$_1$ should be the main program module. It is not necessary to name the object libraries of the Modula-2 and DBPL system explicitely.
4. Start your application with `run` $file_1.exe$.

.

For Modula-2 programs you can use the Modula-2 compiler in the same way by compiling this files with `mod` *command_qualifier file_name*. Look at the file `$dbpl:mod.cld` for allowed command qualifiers. They are explained in the VAX-11 Modula-2 User's Guide.

4.4 An Example Application

Go to the directory $DOC:

```
$ set def $doc
```

The example application consists of the following files

1. `gtypes.def` and `gtypes.dbp`
2. `gdisplay.def` and `gdisplay.dbp` both importing from `gtypes.def`.
3. `gdemo.dbp`, the main program module and the only database module, importing from `gtypes.def` and `gdisplay.def`.

Compile these modules with

```
$ xdb/log/debug gtypes.def
$ xdb/log/debug gdisplay.def
$ xdb/log/debug gtypes.dbp
$ xdb/log/debug gdisplay.dbp
$ xdb/log/debug/create gdemo.dbp
```

Please note, that the order of these operations is significant, since the compiler has to read the symbol files of all imported modules. By using the qualifier `/create` the persistent database variables will be created. You will therefore find some new files with extensions `.dat, .i00, .dic` in the current default directory. These files are needed during subsequent executions of the DBPL application. Now link your application with

```
$ link/debug gdemo.obj,gtypes.obj,gdisplay.obj
```

Note, that the main module `gdemo` is at the first position. If you use this link command, the VAX-Debugger will be automatically activated when you start your application. If you do not need this feature, omit the `/debug` qualifier in the link-command. Now you can start the application with

```
$ run gdemo
```

If you have used the "debug version" you should type `go` after the debugger prompt or start your application simply with

```
$ run/nodebug gdemo
```

This avoids the activation of the debugger, too.

You can use the module-management facilites (MMS) of the VAX operating system to keep track of your module dependencies and simplify the generation of an executable image (see MMS user's guide for further informations). There is a description file `gdemo.mms` for the module dependencies of the demo application and the default rules for the generation of DBPL code and DBPL symbol files. Using MMS it is sufficient to type

```
$ mms/description=gdemo.mms
```

4.5 IO-Facilities

Many DBPL applications need heavy user-interactions. Therefore you will find some predefined modules with simple facilities for menu-driven selection and window-oriented in-/output in the installation directory $IO. These facilities are offered by the following modules:

- Form
- Selector
- Window

For their functionality look at the corresponding definition module's in the directory $IOSOURCES.

4.6 Directory Structure

The top level directory is **$DBPL**. It contains the DBPL-compiler (`dvcpublic.exe`), the Modula-2 compiler (`modula.exe`), the Modula-2 object library (`modula.olb`), the DBPL run time system object library (`dbplrts.olb`) and files necessary for the installation of DBPL. It contains three subdirectories:

$DOC contains the DVI-files and the latex-files of the LaTeX-Versions of the DBPL-Report (`dbplreport.latex/dvi`)and this installation guide (`dbpl_installation.latex/dvi`). Furthermore, it contains the example application (see section **??**).

$SYM contains the symbolfile of the interface to the DBPL run time system (`dbplrts.sym`) (only for internal use by the DBPL compiler) and some Modula-2 symbol files. Furthermore, it contains the subdirectory **$MODSOURCES** with the definition modules for the standard modules.

$IO contains the symbolfiles and the object library for the IO-Facilities (see section **??**). and a subdirectory **$IOSOURCES** with the definition modules of the symbolfiles and the implementation modules of the objectfiles in the IO object library.

4.7 System Restrictions

Restrictions of the current release of the DBPL-compiler:

- Access expressions must not contain function calls (detected during body analysis).
- Aggregates must not contain relations or predicates (detected during code generation).
- The `USING` list within transactions is not implemented (detected during code generation, HALT statement during compilation).
- The assembler listing (selected with the command qualifier /LIST/MACHINE) contains minor syntactic errors, if DBPL specific language constructs are used.
- The internal codetable during code generation may overflow if there are many constructor or selector declarations within a single module (detected during code generation). This problem can be avoided by introducing additional local modules and distributing the selectors and constructors among them.
- The Modula-2 standard functions ABS, CAP, CHR, etc. must not be used within constant expressions. Their evaluation has to take place at runtime.

Restrictions of the current release of the DBPL run time system:

- Updates on selected relation variables are not implemented, i.e. the following program would abort at run time:

```
SELECTOR sp ON ... WITH ...
...

BEGIN
   Rel[sp] := ...
END;
```

- All files representing database variables are created (during compilation) and looked up (during execution) in the default directory. Name conflicts between multiple databases are therefore not avoided and can lead to strange error messages. This problem can be avoided by giving unique variable names within all database modules. A counter example would be

```
DATABASE MODULE A;
VAR
  x: RELATION OF INTEGER;
END A;

DATABASE MODULE B;
VAR
  x: RELATION OF CHAR;
END B;
```

Chapter 5

A Complete DBPL Example

The following small application is taken from [?] and [?] where it serves as an example for the DAIDA methodology of database programming by formal refinement of conceptual designs.

The task is to model research companies that are engaged in research projects and that are supported by grants from third parties. Furthermore they can hire employees to work on some of their own projects.

The following DBPL program addresses all aspects of such a prototypical data-intensive application:

- Representation of the data objects of the application domain (projects, employees, companies) with their relationships (gets grant from, works on, belongs to).
- Maintenance of the application-specific integrity constraints during database updates (a company is not allowed to hire employees that work on projects the company is not engaged in).
- Interaction with the user to display the database contents, to initiate database transactions and to accept transaction parameters.

This database application is broken down into several manageable and reusable modules. Each module consists of a specification called DEFINITION MODULE and an implementation part called IMPLEMENTATION MODULE:

RCDemo is the main module. It controls the global program execution according to the users requests using a simple menu and from interface. It calls (predefined) transactions that are imported from other modules.

RCOps exports a single transaction to hire an employee for a set of projects. Furthermore it contains the definition of the database schema.

RCTypes exports type declarations needed for the database schema definition. All application programs accessing the research company database will import these types to declare local variables or parameters of procedures and transactions.

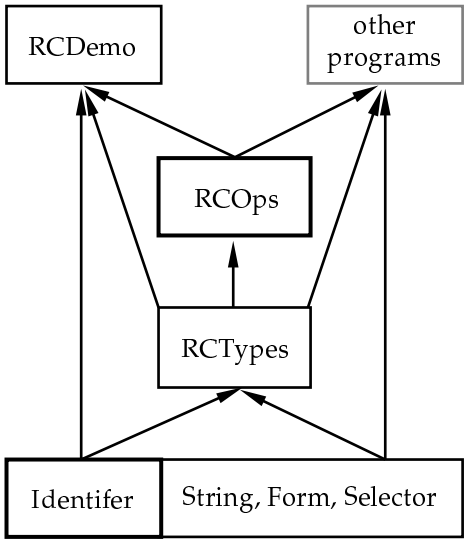

Figure 5.1: Modules and Import Relationships of the Application Example

Identifer encapsulates a mechanism to generate database-wide unique identifiers. This particular implementation simply demonstrates the use of scalar database variables and does not scale up for real database applications.

String, Form and Selector are modules from a standard library that comes with the DBPL system. They provide string handling procedures and a simple user interface based on (overlapping) text-windows.

Figure **??** shows the module relationships of this particular example. An arrow from a box A to a box B indicates that the definition or the implementation of module A uses types, variables or operations exported by the definition module of B.

In the following listings, compilation units are separated by a horizontal bar.

```
DEFINITION MODULE Identifier;
(* Generates database-wide unique identifiers
*)
TYPE
  Type = CARDINAL;

PROCEDURE New () : Type;
(* Returns a unique identifier *)

PROCEDURE Nil () : Type;
(* Returns NIL-identifier *)

END Identifier.
```

```
DATABASE IMPLEMENTATION MODULE Identifier;
(* Implementation uses a database variable 'count' that is incremented
   for every New operation.
*)
VAR
  count: Type;

PROCEDURE New () : Type;
BEGIN
  INC(count);
  RETURN count;
END New;

PROCEDURE Nil () : Type;
BEGIN
  RETURN 0;
END Nil;

DATABASE IMPLEMENTATION
  count:= Nil ();
END Identifier.
```

```
DEFINITION MODULE RCTypes;
IMPORT Identifier, String;

(* global type declarations needed for the database schema and all
   transactions running against that schema *)

TYPE
  (* basic types *)
  EmpIds = Identifier.Type;
  Agencies = (ESPRIT, DFG, NSF);
  CompNames = String.Type;
  PersNames = String.Type;
  ProjNames = String.Type;

  (* One project and a set of projects *)
  ProjIdRecType = RECORD
                    ProjName : ProjNames;
                    GetsGrantFrom : Agencies;
                  END;
  ProjIdRelType = RELATION OF ProjIdRecType;

  ProjRecType = RECORD
                  ProjId : ProjIdRecType;
                  Consortium : RELATION OF CompNames; (* multivalued *)
                END;
  ProjRelType = RELATION ProjId OF ProjRecType;

  (* One employee and a set of employees *)
```

```
  EmpRecType = RECORD
                 EmpId : Identifier.Type; (* unique identifier *)
                 PersName : PersNames;
                 BelongsTo: CompNames;
                 WorksOn : ProjIdRelType; (* multivalued *)
               END;
  EmpRelType = RELATION EmpId OF EmpRecType;

  (* One company and a set of companies *)
  CompRecType = RECORD
                  CompName : CompNames; (* unique identifier *)
                  EngagedIn : ProjIdRelType; (* multivalued *)
                END;
  CompRelType = RELATION CompName OF CompRecType;

END RCTypes.
```

```
IMPLEMENTATION MODULE RCTypes;
(* Since the definition module RCTypes already contains all delcarations,
   the implementation part is empty
*)
END RCTypes.
```

```
DATABASE DEFINITION MODULE RCOps;
(* The keyword DATABASE indicates that all variables declared within
   this module are persistent.
*)
FROM RCTypes IMPORT
  CompRelType, EmpRelType, ProjRelType, ProjIdRelType, EmpIds, PersNames,
  CompNames;

VAR (* persistent database relations *)
  compRel : CompRelType;
  empRel : EmpRelType;
  projRel : ProjRelType;

TRANSACTION hireForProject (name : PersNames;
                            belongs : CompNames;
                            works : ProjIdRelType) : EmpIds;

(* Hires the employee described by 'name' for the set of projects 'works' in
   which the company with name 'belongs' is engaged in. In case of success a new
   unique identifier for the hired employee will be returned, (otherwise
   'Identifier.Nil').
*)
END RCOps.
```

```
IMPLEMENTATION MODULE RCOps;

IMPORT Identifier;
FROM RCTypes IMPORT
  PersNames, CompNames, ProjIdRelType, EmpIds, ProjRecType, CompRecType,
  EmpRecType, ProjIdRecType, Agencies;

TRANSACTION hireForProject (name    : PersNames;
                            belongs : CompNames;
                            works   : ProjIdRelType) : EmpIds;
  VAR emp : EmpRecType;
BEGIN
  IF SOME v1 IN compRel (v1.CompName = belongs) THEN
    IF ALL v2 IN works
       SOME v3 IN compRel[belongs].EngagedIn
         ((v2.ProjName = v3.ProjName) AND
          (v2.GetsGrantFrom = v3.GetsGrantFrom)) THEN
      (* referential integrity assured; employee can be hired *)
      WITH emp DO
        EmpId := Identifier.New ();
        PersName := name;
        BelongsTo := belongs;
        WorksOn := works;
      END;
      INCL (empRel, emp);
      RETURN emp.EmpId
    ELSE
      RETURN Identifier.Nil ()
    END
  ELSE
    RETURN Identifier.Nil ()
  END
END hireForProject;

END RCOps.
```

```
MODULE RCDemo;
(* Main program, interaction with the user, global program control
*)

IMPORT Conversions, Form, Selector, Identifier;
FROM RCTypes IMPORT
  EmpRecType, PersNames, Agencies, CompRecType, ProjIdRecType,
  ProjRecType, ProjIdRelType, CompNames, EmpIds;
FROM RCOps IMPORT
  empRel, compRel, projRel, hireForProject;

TRANSACTION ShowEmployees; (* displays the contents of empRel *)
...
END ShowEmployees;
```

```
TRANSACTION ShowProjects; (* displays the contents of projRel *)
...
END ShowProjects;

TRANSACTION ShowCcompanies; (* displays the contents of compRel *)
...
END ShowCompanies;

PROCEDURE TryToHireEmployee;
(* accepts parameters from the user and calls transaction hireForProject *)

  PROCEDURE BuildLeftSide (VAR form : Form.T);
  BEGIN
    form:= Form.Create("Hire Employee");
    Form.ConstField(0, 1, "Employee Name");
    Form.ConstField(1, 1, "Company");
    Form.ConstField(2, 1, "Works On");
    Form.ConstField(3, 1, "Grant From");
    Form.ConstField(4, 1, "Another Project");
  END BuildLeftSide;

  CONST (* valid strings for user inputs: *)
    YesNoStr = "|y|n";
    AgenciesStr = "|ESPRIT|DFG|NSF";
  TYPE
    YesNoType = (y, n);
  VAR
    form : Form.T;
    emp : PersNames;
    comp : CompNames;
    projid : ProjIdRecType;
    localproj : ProjIdRelType;
    yesno : YesNoType;
    edit : BOOLEAN;

BEGIN
  Form.Message ("Trying to a hire an Employee for some Projects", FALSE);
  (* get user inputs for attributes of an employee: *)
  BuildLeftSide (form);
  emp := ""; Form.TextField( 0, 19, emp);
  comp := ""; Form.TextField( 1, 19, comp);
  projid.ProjName := ""; Form.TextField( 2, 19, projid.ProjName);
  Form.EnumField (3, 19, AgenciesStr, projid.GetsGrantFrom);
  Form.EnumField (4, 19, YesNoStr, yesno);
  Form.Display (form, 1, 1);
  IF Form.Edit (form) THEN
    INCL (localproj, projid);
    Form.Erase (form); edit := TRUE;
    (* get values for the other projects the employee is engaged in: *)
    WHILE (yesno = y) AND edit DO
      BuildLeftSide (form);
      Form.ConstField (0, 19, emp);
```

```
      Form.ConstField (1, 19, comp);
      projid.ProjName := ""; Form.TextField( 2, 19, projid.ProjName);
      Form.EnumField (3, 19, AgenciesStr, projid.GetsGrantFrom);
      Form.EnumField (4, 19, YesNoStr, yesno);
      Form.Display (form, 1, 1);
      edit := Form.Edit (form);
      IF edit THEN
        INCL (localproj, projid)
      ELSE
        Form.Message ("The last form was ignored.", TRUE)
      END;
      Form.Erase (form);
    END;
    (* call transaction: *)
    Form.Message ("Cheking integrity constraints...", FALSE);
    IF hireForProject (emp, comp, localproj) # Identifier.Nil() THEN
      IF CARD (localproj) = 1 THEN
        Form.Message ("The Employee has been hired for the Project.", FALSE)
      ELSE (* >= 2 *)
        Form.Message ("The Employee has been hired for the Projects.", FALSE);
      END;
      Employees.ShowEmployees;
    ELSE
      Form.Message ("Integrity constraints violated - no Employee hired!", TRUE)
    END
  ELSE (* first form has not been edited *)
    Form.Erase (form);
    Form.Message ("You just abandoned hiring another Employee", TRUE);
  END;
END TryToHireEmployee;

PROCEDURE MainMenu;
  CONST MenuStr = "Your Choice ?|Hire an Employee|ShowEmployees"
       "|ShowCompanies|ShowProjects|ShowAll|QuitProgram";
  VAR quit : BOOLEAN; choice : CARDINAL;
BEGIN
  quit := FALSE;
  REPEAT
    choice := Selector.Select (1, 2, MenuStr);
    CASE choice OF
      1 : TryToHireEmployee;
    | 2 : Employees.ShowEmployees;
    | 3 : Companies.ShowCompanies;
    | 4 : Projects.ShowProjects;
    | 5 : Employees.ShowEmployees;
          Companies.ShowCompanies;
          Projects.ShowProjects;
    | 6 : quit := TRUE
    END
  UNTIL quit
END MainMenu;

BEGIN (* this is the main program: *)
```

```
  Employees.ShowEmployees;
  Companies.ShowCompanies;
  Projects.ShowProjects;
  MainMenu;
END RCDemo.
```

Bibliography

[ACC81] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-Algol: An Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7), July 1981.

[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.

[BMSW89] A. Borgida, J. Mylopoulos, J.W. Schmidt, and I. Wetzel. Support for Data-Intensive Applications: Conceptual Design and Software Development. In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 1989.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *ACM-SIGMOD International Conference on Management of Data*, pages 316–325, Boston, Ma., June 1984.

[DCBM89] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88 – A Database Programming Language? In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 1989.

[Dea89] A. Dearle. Environments: a flexible binding mechanism to support system evolution. In *Proc. HICSS-22, Hawaii*, volume II, pages 46–55, January 1989.

[Gro87] Persistent Programming Research Group. PS-algol Reference Manual. PPRR 12-87, University of Glasgow, Dept. of Comp. Science, 1987.

[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.

[JLRS88] W. Johannsen, W. Lamersdorf, K. Reinhardt, 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.

[MRS89] F. Matthes, A. Rudloff, and J.W. Schmidt. Data- and Rule-Based Database Programming in DBPL. Esprit Project 892 WP/IMP 3.b, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, March 1989.

[MS89] F. Matthes and J.W. Schmidt. The Type System of DBPL. In *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, pages 255–260, June 1989.

[NS87] P. Niebergall and J.W. Schmidt. Integrated DAIDA Environment, Part 2: DBPL-Use: A Tool for Language-Sensitive Programming. DAIDA Deliverable WP/IMP-2.c, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1987.

[SBK+88] J.W. Schmidt, M. Bittner, H. Klein, H. Eckhardt, and F. Matthes. DBPL System: The Prototype and its Architecture. Esprit Project 892 WP/IMP 3.2, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, November 1988.

[Sch77] J.W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3), September 1977.

[SEM88] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 111-88, 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.

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

[Tea88] DAIDA Team. Towards KBMS for Software Development: An Overview of the DAIDA Project. In *Advances in Database Technology, EDBT '88*, volume 303 of *Lecture Notes in Computer Science*, pages 572–577. Springer-Verlag, 1988.

[Wir83] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1983.

[WNS89] I. Wetzel, P. Niebergall, and J.W. Schmidt. A Mapping Assistant for Database Program Development. Esprit Project 892 WP/IMP 1.d, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, March 1989.