DATABASE MODELS AND RETRIEVAL LANGUAGES

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE TECHNISCHE WETENSCHAPPEN AAN DE TECHNISCHE HOGESCHOOL EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF. DR. S. T. M. ACKERMANS, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VAN DEKANEN IN HET OPENBAAR TE VERDEDIGEN OP VRIJDAG 16 MAART 1984 TE 16.00 UUR

DOOR

ENGBERT OENE DE BROCK

GEBOREN TE GRONINGEN

Dit proefschrift is goedgekeurd
door de promotoren

Prof.dr. W. Peremans

en

Prof.dr. F.E.J. Kruseman Aretz

Druk: Dissertatie Drukkerij Wibro, Helmond, Telefoon 04920-23981.

Aan mijn ouders

CONTENTS

GENERAL INTRODUCTION AND SUMMARY

As databases become more and more complex, the need for a mathematical theory of databases becomes stronger and stronger. In recent years several attempts to formalization emerged, but only a few of them meet standards of mathematical rigour. Furthermore, the rigorous proposals together cover only a few topics, most of which are at the conceptual level. (Some popular topics are the relational model and the so-called dependencies.) A comprehensive mathematical theory of databases, however, should provide for models at various levels of specification. Examples of models at different levels of specification are the relational model and, at a lower level, the network model (see [Ul 80], [Da 81], or [Re 84]). The network model, however, is hardly formalized.

This thesis contains a mathematical theory of databases that accounts for models at three different levels of specification. A *type 1 model* corresponds, more or less, to a relational model in which all (relevant) static integrity constraints are included. An important notion in terms of this model is that of a *DB function*, roughly speaking, a function that "links" two tables in a type 1 model. A *type 2 model* can now be described as a type 1 model extended with (names for) a "selected" set of DB functions. Both models are defined in chapter 1. A *sequential storage structure* (see chapter 2) is a model at a third level of specification and can be used to describe the semantics of those statements that express "direct-sequential" access to databases.

The fore-mentioned models are introduced in Part I of this thesis. The definitions are of a purely set theoretical nature and, hence, not based on vaguely defined concepts like "entity" or "atomic value". Instead, the notions of *set* and *function* will play a central role. Chapter 0 contains the basic set theoretical notions used in this thesis.

In Part II, three classes of retrieval languages are considered, namely, programming languages, conceptual languages, and fragments of a natural language. These classes of languages are described by means of two-level grammars. Syntax-directed translations (based on these grammars) are given from the natural language fragments into the conceptual languages and from the conceptual languages into the programming languages.

The semantics of the languages can be described in terms of the models introduced in Part I. Thus, although Part II is interesting in its own right, it also illustrates the usefulness of the theory developed in Part I.

The class of conceptual languages is introduced in chapter 4 and contains both languages in the style of "relational calculus" and languages in the style of "relational algebra" (and intermediate forms as well). Special attention is paid to conceptual languages that are "fit for" a type 2 model.

The programming languages are "PASCAL-like" (see chapter 5), but they also contain a small set of primitive "database statements". The semantics of these statements is explained in terms of sequential storage structures.

Translations from the non-procedural retrieval languages from chapter 4 into the procedural retrieval languages from chapter 5 are given in chapter 6. These translations also take into account the typical database problem of currency conflicts.

The general structure of queries in English - the natural language treated here - is described by means of a two-level grammar (see chapter 7). Per considered application, the grammar has to be extended with production rules that introduce the words and phrases that are characteristic for that application.

A syntax-directed translation from the structures presented in chapter 7 into those of chapter 4 is given in chapter 8. The translation satisfies and preserves the (structural) conditions on the form of the translation result that were explicated in section 8.1. These conditions should serve as a guideline for defining the translation of the application-dependent phrases.

The following scheme summarizes in which chapters the various languages and translations are presented. NL, CL, and PL stand for natural, conceptual, and programming language, respectively:

$$\begin{tikzcd} NL \arrow[rr] & & CL \arrow[rr] & & PL \\ 7 & 8 & 4 & 6 & 5 \end{tikzcd}$$

The *modularity* of the translation system is due to the interposition of the conceptual language. This might become clear when we visualize the situation that *several* (fragments of) natural languages are connected with the same database:

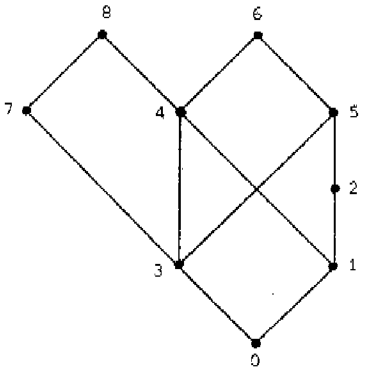
$$\begin{tikzcd}
NL_1 \arrow[dr] & & \\
NL_2 \arrow[r] & CL \arrow[r] & PL \\
NL_3 \arrow[ur] & &
\end{tikzcd}$$

or that, moreover, several programming languages are used (say, in course of time):

$$\begin{tikzcd}
& & PL_1 \\
NL_1 \arrow[dr] & & PL_2 \\
NL_2 \arrow[r] & CL \arrow[uur, dashed] \arrow[ur, dashed] \arrow[dr] \arrow[ddr, dashed] & \\
NL_3 \arrow[ur] & & PL_3 \\
& & PL_4
\end{tikzcd}$$

Thus, with n natural languages and m programming languages, only n + m translations are needed.

The interdependence of the chapters is as follows:



Since the well-known suppliers-parts-projects and employees-departments examples are too simple to illustrate some of the more intriguing database problems, the appendix contains a nontrivial example of a type 1 model and a type 2 model (for some fictitious hospital) in order to show the usefulness of our theory in practice. For the same reason, the appendix also contains a grammar for a fragment of English relevant to the hospital concerned. This grammar is an example of an application-dependent extension of the application-independent grammar from chapter 7. Finally, the fragment of English is translated in agreement with the conditions mentioned in section 8.1.

We finally note that the symbol □ is used to indicate the end of an example, that "iff" stands for "if and only if", and that the symbols $\overset{D}{\Leftrightarrow}$ and $\overset{D}{=}$ stand for "is by definition".

PART I. DATABASE MODELS

0. PRELIMINARIES

The purpose of this chapter is to settle our basic terminology and notations.

R is a <u>relation</u> $\overset{D}{\Leftrightarrow}$ R is a set of ordered pairs. If R is a relation then:

<u>dom(R)</u> $\overset{D}{=} \{x \mid (x;y) \in R\}$, called *the domain of* R;

<u>rng(R)</u> $\overset{D}{=} \{y \mid (x;y) \in R\}$, called *the range of* R;

$\underline{R^{-1}} \overset{D}{=} \{(y;x) \mid (x;y) \in R\}$, called *the inverse of* R.

F is a <u>function</u> $\overset{D}{\Leftrightarrow}$ F is a relation and for every $(x;y) \in F$ and $(x;y') \in F$ we have $y = y'$. Sometimes we use the word *tuple* as a synonym for "function". A function is a special kind of relation, so each notion defined for relations also applies to functions. If F is a function and $x \in \mathrm{dom}(F)$ then we denote the unique y for which $(x;y) \in F$ by <u>F(x)</u>, as usual, or sometimes by $\underline{F_x}$. By *the pre-image of* y *under* F we mean the set $\overline{\{x \in \mathrm{dom}(F) \mid F(x) = y\}}$. We note that $\emptyset$, *the empty set*, is also a function and that $\mathrm{dom}(\emptyset) = \mathrm{rng}(\emptyset) = \emptyset$.

If f and g are functions then:

$\underline{g \circ f} \overset{D}{=} \{(x;g(f(x))) \mid x \in \mathrm{dom}(f) \text{ and } f(x) \in \mathrm{dom}(g)\}$,
called *the composition of* g *after* f.

If f is a function and A is a set then:

$\underline{f \upharpoonright A} \overset{D}{=} \{(x;y) \in f \mid x \in A\}$, i.e., f *restricted to* A.

If T is a set of functions and A is a set then:

$\underline{T \Upharpoonright A} \overset{D}{=} \{f \upharpoonright A \mid f \in T\}$, i.e., T *projected on* A.

If A is a set then:

f is a *function over* A $\overset{D}{\Leftrightarrow}$ f is a function and $\mathrm{dom}(f) = A$;
f is a *function into* A $\overset{D}{\Leftrightarrow}$ f is a function and $\mathrm{rng}(f) \subseteq A$;
f is a *function onto* A $\overset{D}{\Leftrightarrow}$ f is a function and $\mathrm{rng}(f) = A$.

If A and B are sets then:
$\underline{A \to B} \overset{D}{=} \{f \mid f$ is a function and $\mathrm{dom}(f) = A$ and $\mathrm{rng}(f) \subseteq B\}$,
known as the set of all functions from A into B.

F is an $\underline{\text{injection}}$ $\overset{D}{\Leftrightarrow}$ F is a function and F^{-1} is a function. Thus, F is an injection if and only if F is a function and
$\forall x \in \mathrm{dom}(F): \forall x' \in \mathrm{dom}(F)$: if $F(x) = F(x')$ then $x = x'$. An injection is also called a *one-to-one* function.

If $n \in \mathbb{N}$ then:
F is an $\underline{n\text{-tuple}}$ $\overset{D}{\Leftrightarrow}$ F is a function over $\{k \in \mathbb{N} \mid k < n\}$. Here $\mathbb{N}$ denotes the set of natural numbers, i.e., including 0. Notation: $\langle x \rangle$ denotes the 1-tuple F defined by $F(0) = x$, $\langle x;y \rangle$ denotes the 2-tuple G defined by $G(0) = x$ and $G(1) = y$, etc.

F is a $\underline{\text{sequence}}$ $\overset{D}{\Leftrightarrow}$ $\exists n \in \mathbb{N}$: F is an n-tuple. We note that if F is a sequence then there is exactly *one* $n \in \mathbb{N}$ such that F is an n-tuple; this natural number is called the *length of* F. A sequence is a special kind of function, so each notion defined for functions also applies to sequences! We note that $\emptyset$ is also a sequence, *the empty sequence*.

If f and g are sequences then we define $\underline{f \,\&\, g}$, *the concatenation of* f *and* g, as follows:
When n is the length of f and m is the length of g then f & g is the function over $\{k \in \mathbb{N} \mid k < n+m\}$ defined by

$$f \,\&\, g(k) = \begin{cases} f(k) & \text{if } 0 \le k < n \,, \\ g(k-n) & \text{if } n \le k < n+m \,. \end{cases}$$

We define *the generalized concatenation* of a *sequence of* sequences recursively on the length of such a sequence:

$\mathrm{Gconc}(\emptyset) = \emptyset$;

$\mathrm{Gconc}(\gamma \,\&\, \langle q \rangle) = \mathrm{Gconc}(\gamma) \,\&\, q$ where q is a sequence and γ is a sequence of sequences.

If A is a set then:
$\underline{A^*} \overset{D}{=} \{f \mid f$ is a sequence and $\mathrm{rng}(f) \subseteq A\}$; $\underline{A^+} \overset{D}{=} A^* - \{\emptyset\}$;
F is an $\underline{\text{enumeration of } A}$ $\overset{D}{\Leftrightarrow}$ F is a sequence and $\mathrm{rng}(F) = A$ and F is one-to-one. We note that if there is an enumeration of A then A is a *finite* set (and conversely).

Finally we give some miscellaneous definitions including some generalizations of (more) familiar ones.

If W is a set of sets(1) then:
$\underline{\bigcup W} \overset{D}{=} \{x \mid \exists A \in W: x \in A\}$, called *the generalized union of* W.

F is a <u>set function</u> $\overset{D}{\Leftrightarrow}$ F is a function and $\forall x \in \mathrm{dom}(F)$: F(x) is a set(1).

If F is a set function then:
$\underline{\prod(F)} \overset{D}{=} \{f \mid f \text{ is a function over } \mathrm{dom}(F) \text{ and } \forall x \in \mathrm{dom}(f): f(x) \in F(x)\}$, called *the generalised product of* F.

F is a <u>function-valued function</u> $\overset{D}{\Leftrightarrow}$ F is a function and $\forall x \in \mathrm{dom}(F)$: F(x) is a function.

If F and G are function-valued functions then:
$\underline{G \circledcirc F} \overset{D}{=} \{(x; G(x) \circ F(x)) \mid x \in \mathrm{dom}(F) \cap \mathrm{dom}(G)\}$, called *the generalized composition of* G *after* F. In other words, $G \circledcirc F$ is the function over $\mathrm{dom}(F) \cap \mathrm{dom}(G)$ defined by $G \circledcirc F(x) = G(x) \circ F(x)$ for every $x \in \mathrm{dom}(F) \cap \mathrm{dom}(G)$.

(1) For readers familiar with axiomatic set theory we remark that we use a naive set theory in which we do not presuppose that everything is a set.

1. TWO CONCEPTUAL DATABASE MODELS

1.1. Type 1 models

D1.1: If A is a set then:

T is a <u>table over A</u> $\overset{D}{\Leftrightarrow}$ T is a set of functions over A.

Example 1.1: Figure 1.1(a) shows a table T1 over A1 = {DPT,NAME,NR,SAL,SEX} and figure 1.1(b) shows a table T2 over A2 = {DNR,NAME,MAN}. The table T2, for instance, consists of the two functions t = {(DNR;5),(MAN;7),(NAME;planning)} and t' = {(MAN;9),(DNR;7),(NAME;production)}. Indeed dom(t) = dom(t') = A2, as required by the definition. Furthermore, t(DNR) = 5, t(MAN) = 7, and so forth. (We note that MAN stands for "manager".)

NR	NAME	SAL	SEX	DPT
8	Smith	1200	♂	7
7	Jones	1309	♀	5
9	Brown	1300	♂	7

(a)

DNR	NAME	MAN
5	planning	7
7	production	9

(b)

Figure 1.1. An employee table and a department table. □

If figure 1.1 shows all of the information relevant to a certain (small) company at a particular moment, then this "snapshot" can be represented formally by a function v1 over, say, {EMPL,DEP} defined by v1(EMPL) = T1 and v1(DEP) = T2, thus distinguishing the employee table from the department table. If g1 is the function over {DEP,EMPL} defined by g1(EMPL) = A1 and g1(DEP) = A2, then v1 is what we call a *DB snapshot over* g1. We define this notion for arbitrary set functions g:

D1.2: If g is a set function then:

v is a <u>DB snapshot over g</u> $\overset{D}{\Leftrightarrow}$ v is a function over dom(g) and $\forall E \in$ dom(g): v(E) is a table over g(E).

If v is a DB snapshot over a set function g then <g;v> is called a *type 1 snapshot*.

In our example, v1 represents the state of affairs of the company at one particular moment. The state of affairs at an other moment will be represented by a(n other) function v2; v2 must also be a function over {DEP,EMPL} such that v2(DEP) is a table over A2 and v2(EMPL) is a table over A1. In other words, v2 must also be a DB snapshot over g1. Indeed, each possible state of affairs of our company can be represented by a DB snapshot over g1. On the other hand, not every DB snapshot over g1 - in the sense of D1.2 - will represent an allowed state of affairs for the company. The set of all states which are allowed - to be determined by the people of that company of course - is an example of what we call a *DB universe over* g1. Our definition of this notion is rather general:

D1.3: If g is a set function then:

U is a <u>DB universe over g</u> $\overset{D}{\Leftrightarrow}$ U is a set of DB snapshots over g.

For a (stepwise) definition of a nontrivial DB universe we refer the reader to the appendix.

A *type 1 model*, or *conceptual model*, consists of a set function and a DB universe over that set function:

D1.4: <g;U> is a <u>type 1 model</u> $\overset{D}{\Leftrightarrow}$ g is a set function and U is a DB universe over g.

If <g;U> is a type 1 model then g is called *the conceptual skeleton of* <g;U>. By a *table index of* <g;U> we mean an element of dom(g). If E is a table index of <g;U> then g(E) is called *the heading of* E *in* <g;U> and each element of g(E) is called an *attribute of* E *in* <g;U>.

A set B (of attributes) is called *uniquely identifying* (or *u.i.*) for a table T iff different elements of T have different values for at least one attribute in B:

D1.5: If A is a set and B ⊆ A and T is a table over A, then:

<u>B is u.i. for T</u> $\overset{D}{\Leftrightarrow}$ $\forall t \in T$: $\forall t' \in T$: if $t \upharpoonright B = t' \upharpoonright B$ then $t = t'$.

D1.6: If $<g;U>$ is a type 1 model and $E \in dom(g)$ then:

B is a <u>key for E in $<g;U>$</u> $\overset{D}{\Leftrightarrow}$ $B \subseteq g(E)$ and

$\forall v \in U$: B is u.i. for $v(E)$.

Unlike some other authors, we do not require "nonredundancy" (or "minimality") for being a key, i.e., we allow that a proper subset of B also has the property described in D1.6.

1.2. <u>Type 2 models</u>

Example 1.2: Let U1 be a DB universe over the set function g1 (introduced just before D1.2) such that

(1) {DNR} is a key for DEP in $<g1;U1>$ (i.e., at each "moment" every department has a unique department number), and

(2) for every $v \in U1$, every DPT-value in the table v(EMPL) also appears as a DNR-value in v(DEP) (i.e., at each moment every employee belongs to an "actual" department);

then this induces for every v in U1 a function F1(v) from v(EMPL) into v(DEP), assigning to each employee tuple in "state" v the tuple of his department. We deliberately use the function notation F1(v) because we will consider F1 itself as a function too, a (function-valued) function over U1. F1 is an example of what we call a DB function, in this case a DB function within $<g1;U1>$ for the ordered pair (EMPL;DEP). □

D1.7: If $<g;U>$ is a type 1 model and $(M;D) \in dom(g) \times dom(g)$ then:

F is a <u>DB function within $<g;U>$ for (M;D)</u> $\overset{D}{\Leftrightarrow}$

F is a function over U and $\forall v \in U$: $F(v) \in v(M) \rightarrow v(D)$.

DB functions are essential for databases: it is their formal existence and their (correct) implementation that makes a database more than a mere set of "files"!

Note that we allow DB functions to be "reflexive", i.e., in D1.7 we allow that M = D.

Example 1.3: Assume that

(1) {NR} is a key for EMPL in $<g1;U1>$ (from example 1.2) and

(2) for every $v \in U1$ the "MAN-column" $\{t(MAN) \mid t \in v(DEP)\}$ in $v(DEP)$ is a subset of the "NR-column" $\{t(NR) \mid t \in v(EMPL)\}$ in $v(EMPL)$;

then there is a DB function F2 within $<g1;U1>$ for (DEP;EMPL), namely the one for which F2(v) assigns to each department tuple the tuple of its manager in "state" $v \in U1$. But then there also is a DB function F3 within $<g1;U1>$ for (EMPL;EMPL), namely the one for which F3(v) assigns to each employee tuple in "state" v the tuple of the manager of his department. F3 is an example of a "reflexive" DB function. □

Other examples of DB functions can be found in the appendix. Our DB functions F1 and F2 are instances of the following general situation, which covers many cases of DB functions that occur in practice: If $<g;U>$ is a type 1 model, $(M;D) \in dom(g) \times dom(g)$, $a \in g(M)$, $a' \in g(D)$, and

(C1) $\{a'\}$ is a key for D in $<g;U>$ and

(C2) $\{t(a) \mid t \in v(M)\} \subseteq \{t'(a') \mid t' \in v(D)\}$ for every $v \in U$

then the function F over U defined by

$$F(v) = \{(t;t') \in v(M) \times v(D) \mid t(a) = t'(a')\} \quad \text{for every } v \in U$$

is a DB function within $<g;U>$ for (M;D)!

The proof is almost trivial: According to D1.7 we still have to check that $F(v) \in v(M) \to v(D)$; well, F(v) is a function because of (C1), $dom(F(v)) = v(M)$ because of (C2), and $rng(F(v)) \subseteq v(D)$ is trivial.

If both (C1) and (C2) hold then $\{a\}$ is sometimes called a *foreign key*, see for example [Da 81] or [Re 84].

The DB functions F3 in example 1.3 and Ihsp(PT-ADM) and Ihsp(REL-ADM) in the appendix are examples of DB functions that are not covered by the situation mentioned above.

We can make "new" DB functions out of given ones by generalized composition as stated by the following lemma.

L1.1: If $<g;U>$ is a type 1 model, $\{M,D,D'\} \subseteq dom(g)$,
F is a DB function within $<g;U>$ for (M;D), and
G is a DB function within $<g;U>$ for (D;D')
then $G \bullet F$ is a DB function within $<g;U>$ for (M;D').

Again, the proof is simple: Clearly, $G \circledast F$ is a function (see chapter 0), and $dom(F) \cap dom(G) = U \cap U = U$. Furthermore, $F(v) \in v(M) \rightarrow v(D)$ and $G(v) \in v(D) \rightarrow v(D')$ for every $v \in U$; thus $G \circledast F(v) = G(v) \circ F(v) \in v(M) \rightarrow v(D')$. According to D1.7, this completes the proof.

An example of such a generalized composition of two DB functions is F3 in our type 1 model <g1;U1> on employees and departments: $F3 = F2 \circledast F1$.

It will be convenient to have a name for some of the DB functions within a type 1 model, in order to be able to refer to them inside formal languages (for instance retrieval languages). Once we have names for two DB functions, a name for their generalized composition is usually superfluous. Furthermore, we only need names for "relevant" DB functions. (The relevance of a DB function has to be determined by the users of the database concerned.) In general, within a type 1 model a subset of all its DB functions should be chosen and the corresponding names should be specified. This can be represented formally by an "interpretation function" - I in D1.9 - that assigns to each *new* name the corresponding DB function. By "new" (in the previous sentence) we mean that these names for DB functions are to be distinct from the table indices! We also need a "typing function" - h in D1.9 - that assigns to each new name C a "matching" pair of table indices, i.e., if $h(C) = (M;D)$ then the DB function corresponding to C will be a DB function for (M;D).

A type 1 model extended with a "typing function" and an "interpretation function" will be called a *type 2 model*, cf. D1.9. The first component of the type 1 model together with the typing function constitutes a so-called *type 2 skeleton*.

D1.8: <g;h> is a <u>type 2 skeleton</u> $\overset{D}{\Leftrightarrow}$ g is a set function and
h is a function into $dom(g) \times dom(g)$ and
$dom(g) \cap dom(h) = \emptyset$.

D1.9: <g;h;U;I> is a <u>type 2 model</u> $\overset{D}{\Leftrightarrow}$
<g;h> is a type 2 skeleton and
U is a DB universe over g and
I is a function over dom(h) and
$\forall C \in dom(h)$: I(C) is a DB function within <g;U> for h(C).

If <g;h> is a type 2 skeleton, C $\in$ dom(h), and h(C) = (M;D) then we call M *the source index of* C *in* <g;h> and D *the target index of* C *in* <g;h>. Each element of dom(h) is called a *connector index under* <g;h>.

Note that

(a) we allow that the target index of a connector index is the same as its source index,

(b) we permit that different connector indices refer to different DB functions for the same pair of table indices, and

(c) we do not forbid that different connector indices refer to the same DB functions.

Example 1.4: Let h1 be the function over {DEPOF,MANAGEROF} defined by h1(DEPOF) = (EMPL;DEP) and h1(MANAGEROF) = (DEP;EMPL); then <g1;h1> is a type 2 skeleton (where g1 is the set function introduced just before D1.2). In figure 1.2(a), the typing function h1 is depicted. The complete type 2 skeleton is depicted in figure 1.2(b).

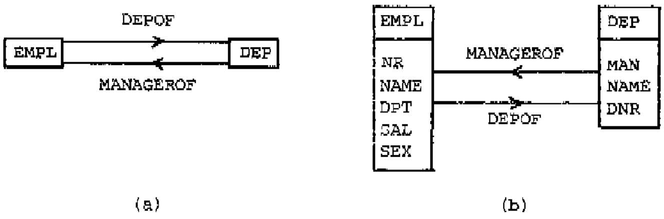


Figure 1.2. (a) Picture of h1. (b) Picture of <g1;h1>.

The connector index DEPOF is intended to refer to the DB function F1 and MANAGEROF is intended to refer to F2. The interpretation function I1 over {DEPOF,MANAGEROF} defined by I1(DEPOF) = F1 and I1(MANAGEROF) = F2 formally represents that interpretation. Now <g1;h1;U1;I1> is an example of a type 2 model. □

If F is a DB function (within a type 1 model <g;U>) for the ordered pair (M;D) and v $\in$ U then F(v) is what we call a *connector for* (M;D) *wrt.* (with respect to) v. In general:

D1.10: If v is a set function and $(M;D) \in \text{dom}(v) \times \text{dom}(v)$ then:

f is a <u>connector for (M;D) wrt. v</u> $\overset{D}{\Leftrightarrow}$ $f \in v(M) \to v(D)$.

With this terminology it is easy to formulate what the essential ingredients of a "snapshot" of a type 2 model <g;h;U;I> are, namely a DB snapshot v over g and for each $C \in \text{dom}(h)$ the connector I(C)(v) for the ordered pair h(C) wrt. v, where I(C)(v) is the DB function I(C) applied to the "state" v. Together with <g;h> these ingredients constitute a *type 2 snapshot*:

D1.11: <g;h;v;w> is a <u>type 2 snapshot</u> $\overset{D}{\Leftrightarrow}$

<g;h> is a type 2 skeleton and

v is a DB snapshot over g and

w is a function over dom(h) and

$\forall C \in \text{dom}(h)$: w(C) is a connector for h(C) wrt. v.

2. SEQUENTIAL STORAGE STRUCTURES

For an understanding of *sequential* programs for *stored* databases we need the notion of a *sequential storage structure*. The purpose of this chapter is to define this nontrivial notion, cf. D2.5.

The first component of a sequential storage structure is a so-called *arrangement*. An arrangement of a DB snapshot v associates a "position" or "location" (whatever that may be) with every tuple $t \in v(E)$, for all table indices $E \in dom(v)$. More precisely (and without the noise in the previous sentence):

D2.1: If v is a set function then:

μ is an <u>arrangement of v</u> $\overset{D}{\Leftrightarrow}$ μ is a function over dom(v) and

$\forall E \in dom(v)$: $\mu(E)$ is a one-to-one function onto v(E).

For "positions" (i.e., elements of $dom(\mu(E))$ for any $E \in dom(v)$) we may think of relative or absolute addresses, or so-called "database key values" (in which cases $dom(\mu(E))$ and $dom(\mu(E'))$ will be disjoint for $E \neq E'$), but also of natural numbers enumerating the elements of v(E). In this special case we speak of a *sequential arrangement of* v.

D2.2: If v is a set function then:

μ is a <u>sequential arrangement of v</u> $\overset{D}{\Leftrightarrow}$

μ is a function over dom(v) and

$\forall E \in dom(v)$: $\mu(E)$ is an enumeration of v(E).

In the general case, i.e., when μ is not necessarily sequential, we will also need an *ordering function for* μ as one of the components of a sequential storage structure:

D2.3: If μ is a function-valued function then:

r is an <u>ordering function for μ</u> $\overset{D}{\Leftrightarrow}$

r is a function over $dom(\mu)$ and

$\forall E \in dom(\mu)$: r(E) is an enumeration of $dom(\mu(E))$.

We note that if μ is an arrangement of a DB snapshot v and r is an ordering function for μ then, after all, the generalized composition of μ after r, i.e., the function $\{(E;\mu(E)\circ r(E)) \mid E \in dom(\mu)\}$, constitutes a *sequential* arrangement of v.

If f is a connector for a pair (M;D) wrt. a DB snapshot v and μ is an arrangement of v then we have the situation as depicted in figure 2.1. (We recall from chapter 0 that we may write μ_M instead of $\mu(M)$ - which will be convenient here - and from D2.1 that the function μ_D is one-to-one and *onto* v(D).)

$$\begin{tikzcd}
dom(\mu_M) \arrow[r, "\mu_M"] \arrow[d, dashed, "(\mu_D)^{-1}\circ f\circ\mu_M"'] & v(M) \arrow[d, "f"] \\
dom(\mu_D) \arrow[r, "\mu_D"'] & v(D)
\end{tikzcd}$$

Figure 2.1.

A *location link for* (M;D) *based on* f *and* μ determines for every "location" $p \in dom(\mu_D)$ an enumeration of the locations of those tuples in v(M) that are mapped to $\mu_D(p) \in v(D)$ by the function f:

D2.4: If v is a set function and $(M;D) \in dom(v) \times dom(v)$ and f is a connector for (M;D) wrt. v and μ is an arrangement of v then:

ℓ is a <u>location link for (M;D) based on f and μ</u> $\overset{D}{\Leftrightarrow}$

ℓ is a function over $dom(\mu_D)$ and

$\forall p \in dom(\mu_D)$: $\ell(p)$ is an enumeration of

$$\{p' \in dom(\mu_M) \mid f(\mu_M(p')) = \mu_D(p)\}.$$

We note that the set $\{p' \in dom(\mu_M) \mid f(\mu_M(p')) = \mu_D(p)\}$ mentioned in D2.4 is the pre-image of p under the (composite) function $(\mu_D)^{-1} \circ f \circ \mu_M \in dom(\mu_M) \to dom(\mu_D)$, cf. figure 2.1. Motivated by D1.10 and D2.4 we will call the function $\mu_D^{-1} \circ f \circ \mu_M$ the *location connector for* (M;D) *based on* f *and* μ; it maps the location of each $t \in v(M)$ to the location of f(t).

In addition to an arrangement and an ordering function, a sequential storage structure for a type 2 snapshot <g;h;v;w> will also contain for every connector index $C \in \mathrm{dom}(h)$ a location link for the pair $h(C)$, based on the corresponding connector $w(C)$, cf. D1.11, and the arrangement concerned. More precisely:

D2.5: If $<g;h;v;w>$ is a type 2 snapshot then:

$<\mu;r;K>$ is a <u>sequential storage structure for $<g;h;v;w>$</u> $\overset{D}{\Leftrightarrow}$

μ is an arrangement of v and

r is an ordering function for μ and

K is a function over $\mathrm{dom}(h)$ and

$\forall C \in \mathrm{dom}(h)$: $K(C)$ is a location link for $h(C)$ based on $w(C)$ and μ.

In conclusion we recall the purpose of each component in D2.5:

- μ accounts for the distinction between tuples and their "locations" in a stored database;
- r delivers, indirectly via μ, for each table index E an enumeration of the set $v(E)$;
- K, also indirectly via μ, delivers, for each connector index $C \in \mathrm{dom}(h)$, *per* tuple t in the "target" table of C an enumeration of those elements in the "source" table of C which are mapped to t by $w(C)$, the function C refers to in "state" v. (When we deal with a type 2 model $<g;h;U;I>$ then $w(C)$ will be $I(C)(v)$, see the paragraph following D1.10.)

In section 5.2 these concepts will be used to explain the effect of our standard procedures for "files" and "links".

PART II. SOME RETRIEVAL LANGUAGES FOR DATABASES

3. GRAMMARS

In this chapter we present the basic notions concerning formal languages. We first introduce the concept of a *quasi-cfg*, which is a generalization of the well-known concept of a context-free grammar (cfg).

D3.1: $\langle V;N;P;S\rangle$ is a <u>quasi-cfg</u> $\overset{D}{\Leftrightarrow}$ V is a set and $N \subseteq V$ and
$P \subseteq N \times V^+$ and $S \in N$.

If G is a quasi-cfg, say $G = \langle V;N;P;S\rangle$, then the following additional terminology and notations will be used:

(a1) by <u>$Voc(G)$</u> we mean V, called the *vocabulary of G*;

(a2) by <u>$Nv(G)$</u> we mean N, called the *nonterminal vocabulary of G*;

(a3) by <u>$Tv(G)$</u> we mean V-N, called the *terminal vocabulary of G*;

(a4) by <u>$Rs(G)$</u> we mean P, called the *rule set of G*;

(a5) by <u>$St(G)$</u> we mean S, called the *start symbol of G*;

(b1) A is a *symbol of G*	iff $A \in Voc(G)$;
(b2) A is a *nonterminal of G*	iff $A \in Nv(G)$;
(b3) A is a *terminal of G*	iff $A \in Tv(G)$;
(b4) A is a *production rule of G*	iff $A \in Rs(G)$.

We note that we do not allow the "right hand side" of a production rule to be the empty sequence.

A *cfg* or *context-free grammar* is a special kind of quasi-cfg:

D3.2: G is a <u>cfg</u> $\overset{D}{\Leftrightarrow}$ G is a quasi-cfg and
$Voc(G)$ and $Rs(G)$ are finite sets.

In the specification of concrete grammars, terminals will be written in bold type, nonterminals will begin with the bracket "<" and end with the bracket ">", and production rules will be written in the so-called *Backus-Naur Form* (BNF): In BNF, a production rule $(\alpha;\beta)$ is written as $\alpha ::= \beta'$ (when β' denotes the juxtaposition of the components of the sequence β) and, for instance, $\alpha ::= \varphi|\psi|\delta$ stands for the set $\{\alpha ::= \varphi,\ \alpha ::= \psi,\ \alpha ::= \delta\}$.

Example 3.1: An interesting example is the grammar with start symbol <int.>, terminal vocabulary {**0,1,2,3,4,5,6,7,8,9,-**}, nonterminal vocabulary {<int.>,<digit>,<pos.int.>,<nz.digit>}, and the following 16 production rules:

```
<int.>      ::= - <pos.int.>|0|<pos.int.>             (1)
<pos.int.> ::= <nz.digit>                              (2)
                |<pos.int.><digit>                     (3)
<digit>     ::= 0|<nz.digit>                           (4)
<nz.digit> ::= 1|2|3|4|5|6|7|8|9                      . (5)
```

The suggestive names for the nonterminals of this cfg only play a mnemonic role, of course, and no formal role other than to tell the nonterminals apart. □

We note that the concept of a quasi-cfg is, in its general form, not a "finitary" concept, because the vocabulary or the rule set can be infinite. However, many quasi-cfg's with an infinite vocabulary or an infinite rule set can still be defined in a "finitary" way. For this purpose several formalisms are available from the literature, for instance, *Van Wijngaarden grammars* (VWgs), used in [vW 75] for the definition of ALGOL68 (see also [vW 65]), *affix grammars*, see [Ko 71], or *attribute grammars*, introduced in [Kn 68] (see [He 84] for a definition devoid of implementation aspects). For an overview and other references we refer the reader to [MLB 76] and [BE 76].

In each of these formalisms, a (possibly infinite) set of production rules is obtained from a finite set of so-called *rule forms*. Loosely speaking, a rule form is a production rule containing "parameters" (known as *attribute variables* in the context of attribute grammars, *metanotions* in the context of VWgs, and *nonterminal affixes* in the context of affix grammars). A production rule is obtained from a rule form by replacing each parameter uniformly by a value that is allowed for that parameter. For the formal details, which vary per formalism, we refer the reader to the literature mentioned before; the essential common characteristic, however, is that these formalisms in fact all result in a quasi-cfg.(2). From this intermediate stage on, each formalism defines the important concepts (such as *derivation tree* and *the generated language*) in exactly the same way. Later on, these concepts will be defined for quasi-cfg's in general.

Before we present our formal definition of *derivation tree* - in other papers variously called generation tree, syntax tree, parse tree, analysis tree, phrase structure tree, or structural description - we first define the notion of a *labelled ordered tree over* V, for any set V. (Henceforth we simply say *tree* instead of labelled ordered tree.) For technical reasons a "one node" tree with label A will be formalized as the ordered pair $(A;\emptyset)$ and not simply as A.

D3.3: If V is a set then:

(a) $\underline{Lot(V)}$ is the smallest set Y such that $\forall A \in V: \forall q \in Y^*: (A;q) \in Y$;

(b) T is a $\underline{\text{tree over V}} \overset{D}{\Leftrightarrow} T \in Lot(V)$.

In other words, nothing is a tree over V except as required by (1) and (2) in the following (trivial) lemma.

L3.1: If V is a set then:

(1) if $A \in V$ then $(A;\emptyset)$ is a tree over V;

(2) if $A \in V$ and q is a nonempty sequence of trees over V then $(A;q)$ is a tree over V.

Thus, each tree T over V is an ordered pair. The first component of T is called *the root label of* T and will be denoted by $\underline{Rl(T)}$. We see that if T is a tree over V then $Rl(T) \in V$.

If V is a set then Fr_V is the function over $Lot(V)$ that assigns to each tree T over V the sequence consisting of its "leaf labels";

(2) In the context of VWgs, the nonterminals of the resulting quasi-cfg are called *notions* and in the case of attribute grammars they are called *attributed nonterminals*.

this sequence is called *the frontier of* T. For a "one node" tree $(A;\emptyset)$ this will be the 1-tuple $\langle A\rangle$, for a tree $(A;q)$ with q being a nonempty sequence of trees this will be the generalized concatenation of the frontiers of all trees $q(k)$, $k \in dom(q)$. Formally $\underline{\mathcal{F}r_V}$ is defined, recursively, by:

$$\mathcal{F}r_V((A;\emptyset)) = \langle A\rangle ,$$

$$\mathcal{F}r_V((A;q)) = \mathcal{G}conc(\mathcal{F}r_V \circ q) \quad \text{if } q \neq \emptyset .$$

If T is a tree over V as well as over V' then $\mathcal{F}r_V(T) = \mathcal{F}r_{V'}(T)$. Therefore the subscript V will often be omitted from now on.

If q is a sequence of trees then we denote the corresponding sequence of root labels by $\mathcal{R}\ell s(q)$. Formally:

D3.4: If V is a set and $q \in \mathcal{L}ot(V)^*$ then:

$$\underline{\mathcal{R}\ell s(q)} \overset{D}{=} \{(k;\mathcal{R}\ell(q(k))) \mid k \in dom(q)\} .$$

Note that if $q \in \mathcal{L}ot(V)^*$ then $\mathcal{R}\ell s(q) \in V^*$.

The following definition of *derivation tree* is an immediate formalization of the idea that a derivation tree consists of a root label together with an ordered set of "corresponding" subtrees, that is, corresponding to one of the production rules of the quasi-cfg concerned. Our definition is a generalization of an idea found in [Ba 82].

D3.5: If $\mathcal{G}$ is a quasi-cfg then:

(a) $\underline{\mathcal{D}tr(\mathcal{G})}$ is the smallest set Y such that
$(A;\emptyset) \in Y$ for every terminal A of $\mathcal{G}$, and
$(A;q) \in Y$ for every nonterminal A of $\mathcal{G}$ and every $q \in Y^*$ for which $(A;\mathcal{R}\ell s(q)) \in \mathcal{R}s(\mathcal{G})$;

(b) T is a $\underline{\text{derivation tree based on } \mathcal{G}} \overset{D}{\leftrightarrow} T \in \mathcal{D}tr(\mathcal{G})$.

In other words, nothing is a derivation tree based on $\mathcal{G}$ except as required by (1) and (2) in the following lemma.

L3.2: If $\mathcal{G}$ is a quasi-cfg then:

(1) if A is a terminal of $\mathcal{G}$ then $(A;\emptyset)$ is a derivation tree based on $\mathcal{G}$;

(2) if (A;r) is a production rule of G and q is a (nonempty) sequence of derivation trees based on G for which Rls(q) = r then (A;q) is a derivation tree based on G;

(3) $\mathit{Dtr}(G) \subseteq \mathit{Lot}(\mathit{Voc}(G))$.

If A is an element of the vocabulary of a quasi-cfg G then $\mathit{Drl}(G,\mathrm{A})$ will denote the set of derivation trees based on G with root label A, and $\mathit{Fsd}(G,\mathrm{A})$ will denote the set of frontiers of all those derivation trees. If A is the start symbol of G then we obtain $\mathit{DL}(G)$, called the *disambiguated language generated by* G, and $L(G)$, called the *language generated by* G. Formally:

D3.6: If G is a quasi-cfg and A $\in$ $\mathit{Voc}(G)$ then:

(a) $\underline{\mathit{Drl}(G,\mathrm{A})} \overset{D}{=} \{\mathrm{T} \mid \mathrm{T} \in \mathit{Dtr}(G) \text{ and } \mathit{Rl}(\mathrm{T}) = \mathrm{A}\}$;

(b) $\underline{\mathit{Fsd}(G,\mathrm{A})} \overset{D}{=} \{\mathit{Fr}(\mathrm{T}) \mid \mathrm{T} \in \mathit{Dtr}(G) \text{ and } \mathit{Rl}(\mathrm{T}) = \mathrm{A}\}$;

(c) $\underline{\mathit{DL}(G)} \overset{D}{=} \mathit{Drl}(G,\mathit{St}(G))$;

(d) $\underline{L(G)} \overset{D}{=} \mathit{Fsd}(G,\mathit{St}(G))$.

A quasi-cfg G is called *unambiguous* iff different derivation trees having the start symbol of G as their root label always have different frontiers; otherwise G is called *ambiguous*.

D3.7: If G is a quasi-cfg then:

G is $\underline{\text{unambiguous}} \overset{D}{\Leftrightarrow} \forall \mathrm{T}\in\mathit{DL}(G) : \forall \mathrm{T}'\in\mathit{DL}(G) :$ if $\mathit{Fr}(\mathrm{T}) = \mathit{Fr}(\mathrm{T}')$ then $\mathrm{T} = \mathrm{T}'$.

In other words, G is unambiguous iff Fr restricted to $\mathit{DL}(G)$ is a one-to-one function.

4. CONCEPTUAL LANGUAGES

4.0. Introduction and summary

In this chapter we present a class of conceptual (or "logical") languages that can serve as (higher level) retrieval languages for various sets of data, in particular for databases.

Each CL (conceptual language) is uniquely determined by a so-called *CL-basis*, a hotch-potch of "basic symbols". The more specific notion of a *CL-basis fit for* a type 2 skeleton proves to be useful for database applications. Both concepts are defined in section 4.1.

The sets of all *well-formed expressions* and all *well-formed queries* based on a CL-basis B are defined in section 4.2. It is also shown how these sets can be defined by means of a quasi-cfg. Finally, the important subsets of all *closed expressions* and all *closed queries* are defined.

Some typical database examples of closed queries are given in section 4.3.

4.1. CL-bases

The formal definition of the notion of a CL-basis will be followed by some (suggestive) terminology concerning the various ingredients of a CL-basis. Further explanation is given after example 4.1.

D4.1: <T;H> is a <u>CL-basis</u> $\overset{D}{\Leftrightarrow}$ T is a set and H is a 7-tuple such that
H_0, H_1, and H_2 are set functions over T,
H_3 and H_4 are set functions over T × T, and
H_5 and H_6 are set functions over (T × T) × T.

If B is a CL-basis, say B = <T;H>, and τ, τ', and σ are elements of T then the following notations will often be used:

Typ_B	will denote the set T,
$\mathrm{Plh}_B(\sigma)$	will denote the set $H_0(\sigma)$,
$\mathrm{Con}_B(\sigma)$	will denote the set $H_1(\sigma)$,
$\mathrm{Int}_B(\sigma)$	will denote the set $H_2(\sigma)$,
$\mathrm{Unop}_B(\tau,\sigma)$	will denote the set $H_3((\tau;\sigma))$,
$\mathrm{Arg}_B(\tau,\sigma)$	will denote the set $H_4((\tau;\sigma))$,
$\mathrm{Binop}_B(\tau,\tau',\sigma)$	will denote the set $H_5(((\tau;\tau');\sigma))$, and
$\mathrm{Det}_B(\tau,\tau',\sigma)$	will denote the set $H_6(((\tau;\tau');\sigma))$.

With respect to a CL-basis B we will call

Typ_B	the set of *types* of B,
$\mathrm{Plh}_B(\sigma)$	its set of *placeholders* of type σ,
$\mathrm{Con}_B(\sigma)$	its set of *constants* of type σ,
$\mathrm{Int}_B(\sigma)$	its set of *intensional constants* of type σ,
$\mathrm{Unop}_B(\tau,\sigma)$	its set of *unary operation symbols* with *operand* type τ and *result* type σ,
$\mathrm{Arg}_B(\tau,\sigma)$	its set of *arguments* with *operator* type τ and result type σ,
$\mathrm{Binop}_B(\tau,\tau',\sigma)$	its set of *binary operation symbols* with *first* operand type τ, *second* operand type τ', and result type σ, and
$\mathrm{Det}_B(\tau,\tau',\sigma)$	its set of *determiners* with *domain* type τ, *range* type τ', and result type σ.

Example 4.1: Well-known examples of formal languages are *first-order languages* (see, e.g., [Sh 67] or [BM 77]). We will show what kind of CL-bases are needed for first-order languages with nullary, unary and binary function and predicate symbols:

- Typ_B will be a set consisting of $\mathbf{t}$ and one other element, say $\mathrm{Typ}_B = \{e,\mathbf{t}\}$.
- $\mathrm{Plh}_B(e)$ will be the set of the *individual variables* of the intended first-order language, and $\mathrm{Plh}_B(\mathbf{t}) = \emptyset$.

- $\mathrm{Con}_B(\mathrm{e})$ will be the set of *individual constants* and $\mathrm{Con}_B(\mathbf{t})$ will be the set of *proposition symbols.*
- There are no intensional constants: $\mathrm{Int}_B(\mathrm{e}) = \mathrm{Int}_B(\mathbf{t}) = \emptyset$.
- $\mathrm{Unop}_B(\mathrm{e},\mathrm{e})$ will be the set of *unary function symbols,* $\mathrm{Unop}_B(\mathrm{e},\mathbf{t})$ will be the set of *unary predicate symbols,* $\mathrm{Unop}_B(\mathbf{t},\mathbf{t}) = \{\neg\}$, and $\mathrm{Unop}_B(\mathbf{t},\mathrm{e}) = \emptyset$.
- There are no arguments: $\mathrm{Arg}_B(\tau,\sigma) = \emptyset$ for every τ and σ in Typ_B.
- $\mathrm{Binop}_B(\mathrm{e},\mathrm{e},\mathrm{e})$ will be the set of *binary function symbols,* $\mathrm{Binop}_B(\mathrm{e},\mathrm{e},\mathbf{t})$ will be the set of *binary predicate symbols* (often containing $=$, the *equality symbol*), $\mathrm{Binop}_B(\mathbf{t},\mathbf{t},\mathbf{t}) = \{\wedge,\vee,\Rightarrow,\Leftrightarrow\}$, and $\mathrm{Binop}_B(\tau,\tau',\sigma) = \emptyset$ in the other five cases.
- $\mathrm{Det}_B(\mathrm{e},\mathbf{t},\mathbf{t}) = \{\forall,\exists\}$ and $\mathrm{Det}_B(\tau,\tau',\sigma) = \emptyset$ in the other seven cases.

Note that we gave a *class of* CL-bases, each CL-basis being a basis for one first-order language. In order to obtain a *particular* CL-basis, we still have to specify e and the seven sets Plh(e), Con(σ), Unop(e,σ), and Binop(e,e,σ), where $\sigma \in \{\mathrm{e},\mathbf{t}\}$.

Aside: A popular choice for Plh(e) is the set $\{\mathbf{x1},\mathbf{x2},\mathbf{x3},\ldots\}$, more precisely (and in line with the grammar following D4.5), the language generated by the grammar with start symbol <P;e> and the rule set consisting of

<P;e> ::= **x** <pos.int.>

and the last 13 production rules mentioned in the grammar in example 3.1. But also the smaller rule sets

<P;e> ::= **x**|<P;e>'

and, with more variety,

<P;e> ::= **x**|**y**|**z**|<P;e>'

give a suitable (infinite) set of individual variables. □

The central ingredients of a CL-basis are its *types*. In practice, the set of types of a CL-basis B is often defined as the language generated by a small unambiguous cfg G_0; i.e., $\mathrm{Typ}_B = L(G_0)$. In example 4.2 the set of types will be defined in this way. As another example of an infinite (!) set of types, we consider the set of types of Montague's language of *intensional logic* ([Mo 73]), a well-known language in formal linguistics and logic; the set of types can be described by the grammar with start symbol <Ty> and the following 4 production rules:

<Ty> ::= $\mathbf{t}$|$\mathbf{e}$|$\mathbf{s}$<Ty>|(<Ty><Ty>)

In example 4.1 a *finite* set of types was used.

Semantically, a type σ can be thought of as a dummy denoting a set $\mathcal{D}_B(\sigma)$ where $\mathcal{D}_B$ is a set function over Typ_B. In the sequel, $\mathbf{t}$ and $\mathbf{int}$ will be thought of as types with a standard denotation and if τ and σ are types then the 1-tuple $<\tau>$ and the ordered pair $(\tau;\sigma)$ will be used as types with a standard denotation in terms of $\mathcal{D}_B(\tau)$ and $\mathcal{D}_B(\sigma)$, see below. We note that it is not necessary that these types always occur in a CL-basis. We often write $\mathbf{so}\lfloor\tau\rfloor$ instead of $<\tau>$ and $\mathbf{fc}\lfloor\tau;\sigma\rfloor$ instead of $(\tau;\sigma)$. Finally, also $\mathbf{p}\lfloor\tau;\sigma\rfloor$ will be used as a type with a standard denotation. The standard denotations are:

$\mathcal{D}_B(\mathbf{t})$	$= \{0,1\}$,	i.e., the set of "truth values";
$\mathcal{D}_B(\mathbf{int})$	$= \mathbb{Z}$,	i.e., the set of integers;
$\mathcal{D}_B(\mathbf{so}\lfloor\tau\rfloor)$	$= \mathcal{P}(\mathcal{D}_B(\tau))$,	i.e., the power set of the set denoted by τ;
$\mathcal{D}_B(\mathbf{fc}\lfloor\tau;\sigma\rfloor)$	$= \mathcal{D}_B(\tau) \rightarrow \mathcal{D}_B(\sigma)$,	i.e., the set of all functions from $\mathcal{D}_B(\tau)$ into $\mathcal{D}_B(\sigma)$;
$\mathcal{D}_B(\mathbf{p}\lfloor\tau;\sigma\rfloor)$	$= \mathcal{D}_B(\tau) \times \mathcal{D}_B(\sigma)$,	i.e., the cartesian product of $\mathcal{D}_B(\tau)$ and $\mathcal{D}_B(\sigma)$.

We continue with some familiar examples of *constants* and of *unary* and *binary* operation symbols. The role of *intensional constants* and *arguments* (in connection with databases) will be illustrated in example 4.2 and examples of *determiners* other than $\forall$ and $\exists$ will be given in the comments following L4.1 in section 4.2.

The *logical connectives* $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ will be typical elements of $\mathrm{Binop}_B(\mathbf{t},\mathbf{t},\mathbf{t})$ for those CL-bases B in which they occur at all. The only useful unary operation symbol with operand type $\mathbf{t}$ and result type $\mathbf{t}$ is $\neg$, *the negation symbol*. We will use the symbols $\perp$ (for "false") and $\top$ (for "true") as constants of type $\mathbf{t}$.

The equality symbol $=$ can be put in $\text{Binop}_B(\sigma,\sigma,\mathbf{t})$ for all or, if desirable, only some types σ. Also the symbol $\neq$ can be included.

The well-known symbol $\in$ typically belongs to $\text{Binop}_B(\sigma,\mathbf{so}\lfloor\sigma\rfloor,\mathbf{t})$. Also the symbol $\notin$ can be included. Other well-known symbols from set theory are $\cup$ (for union), $\cap$ (for intersection), and $\setminus$ (for set difference); they typically belong to $\text{Binop}_B(\mathbf{so}\lfloor\sigma\rfloor,\mathbf{so}\lfloor\sigma\rfloor,\mathbf{so}\lfloor\sigma\rfloor)$. The symbol $\emptyset$ (for the empty set) can be put in $\text{Con}_B(\mathbf{so}\lfloor\sigma\rfloor)$ for any type σ. We will use **sngl** as a unary operation symbol with operand type σ and result type $\mathbf{so}\lfloor\sigma\rfloor$, denoting singleton formation.

It is useful to have a symbol in $\text{Binop}_B(\tau,\sigma,\mathbf{p}\lfloor\tau;\sigma\rfloor)$ denoting the formation of ordered pairs. We will use the symbol $;$ for that purpose.

An interesting candidate for $\text{Con}_B(\mathbf{int})$ would be the language generated by the grammar given in example 3.1. (This candidate is interesting because then *every* element of $\mathbb{Z}$, the set the type **int** is supposed to be denoting, is represented by *exactly one* constant of type **int**.) Both $\text{Unop}_B(\mathbf{int},\mathbf{int})$ and $\text{Binop}_B(\mathbf{int},\mathbf{int},\mathbf{int})$ could contain the symbols $+$ and $-$. Other typical elements of the latter set are $\times$ and $\div$ (for integer division). Typical elements of $\text{Binop}_B(\mathbf{int},\mathbf{int},\mathbf{t})$ are (besides $=$) the "relational" symbols $<$, $\geq$, $\leq$, and $>$.

We recall that the type $\mathbf{fc}\lfloor\tau;\sigma\rfloor$ can be thought of as denoting the set of all functions from the set denoted by τ into the set denoted by σ. Therefore it is useful to have a symbol in $\text{Binop}_B(\mathbf{fc}\lfloor\tau;\sigma\rfloor,\tau,\sigma)$ denoting function application. We will use the symbol @ for that purpose. For our database applications it is also useful to have a symbol in $\text{Binop}_B(\mathbf{fc}\lfloor\tau;\sigma\rfloor,\sigma,\mathbf{so}\lfloor\tau\rfloor)$ denoting the formation of the pre-image - see chapter 0 - of a "σ-object" under a function (expression) of type $\mathbf{fc}\lfloor\tau;\sigma\rfloor$. For that purpose we will use **inv**.

In order to express the queries that are relevant to a database based on a type 2 skeleton $<g;h>$, we need a CL-basis B that contains all table indices, connector indices, and attributes of that skeleton as "basic symbols". These basic symbols are to be classified and "typed" as follows:

(a) Every table index E in dom(g) should be an intensional constant. (Intensional constants will correspond to variables in the sense of computer science, see section 4.2.) Its type should be $\mathbf{so}\lfloor E\rfloor$. As a consequence, both $\mathbf{so}\lfloor E\rfloor$ and E should be types of B.

(b) Every connector index C in dom(h) should be an intensional constant of type h(C). (Thus, if M denotes the source index of C and D its target index, i.e., if h(C) = (M;D), then the type of the intensional constant C can be written as **fc**⌊M;D⌋.) As a consequence, h(C) should be a type of B.

(c) For every table index E in dom(g), each attribute of E should be an argument with operator type E and, moreover, there should be no other arguments with operator type E.

(d) Finally, each argument should have only one result type per operator type.

A CL-basis meeting all these requirements will be called a *CL-basis fit for* <g;h>:

D4.2: If <g;h> is a type 2 skeleton then:

B is a <u>CL-basis fit for <g;h></u> $\overset{D}{\Leftrightarrow}$

B is a CL-basis and

$[\forall E \in dom(g): \{E, <E>\} \subseteq Typ_B$ and $E \in Int_B(<E>)]$ and

$[\forall C \in dom(h): h(C) \in Typ_B$ and $C \in Int_B(h(C))]$ and

$[\forall E \in dom(g): g(E) = \bigcup \{Arg_B(E,\sigma) \mid \sigma \in Typ_B\}]$ and

$\forall \tau,\sigma,\sigma' \in Typ_B$: (if $\sigma \neq \sigma'$ then $Arg_B(\tau,\sigma) \cap Arg_B(\tau,\sigma') = \emptyset$).

We note that the last-mentioned requirement is of greater generality than the others; it does not bear upon the particular type 2 skeleton.

Example 4.2: We will check what it means for a CL-basis to be fit for <g1;h1>, the type 2 skeleton presented in example 1.4. We recall that dom(g1) = {DEP,EMPL}, dom(h1) = {DEPOF,MANAGEROF}, g1(DEP) = = {DNR,MAN,NAME}, g1(EMPL) = {DPT,NAME,NR,SAL,SEX}, h1(DEPOF) = = (EMPL;DEP), and h1(MANAGEROF) = (DEP;EMPL); see also figure 1.2(b).

By requirements (a) and (b), Typ_B must contain **DEP**, **EMPL**, **so**⌊**DEP**⌋, **so**⌊**EMPL**⌋, **fc**⌊**EMPL;DEP**⌋, and **fc**⌊**DEP;EMPL**⌋ as elements. One of the candidates for Typ_B is, for instance, the language generated by the grammar with start symbol <type> and the following 8 production rules:

<type> ::= **DEP**|**EMPL**|**t**|**int**|**str**|**g**
|**so**⌊<type>⌋
|**fc**⌊<type>**;**<type>⌋

(Here the type **str** is intended to denote the set of all sequences of characters and the "genus type" **g** is intended to denote a set with exactly two elements, say {0,1}. Furthermore, we like $\text{Con}_B(\mathbf{g})$ to be the set {♀,♂}.)

There must be at least 4 intensional constants in B (again, by (a) and (b)): $\text{Int}_B(\mathbf{so}\lfloor\mathbf{DEP}\rfloor)$ must contain **DEP**, $\text{Int}_B(\mathbf{so}\lfloor\mathbf{EMPL}\rfloor)$ must contain **EMPL**, $\text{Int}_B(\mathbf{fc}\lfloor\mathbf{EMPL};\mathbf{DEP}\rfloor)$ must contain **DEPOF**, and $\text{Int}_B(\mathbf{fc}\lfloor\mathbf{DEP};\mathbf{EMPL}\rfloor)$ must contain **MANAGEROF**.

The following (reasonable) choice for the collection of arguments with a table index as operator type is in accordance with the requirements (c) and (d):

$\text{Arg}_B(\mathbf{EMPL},\mathbf{int}) = \{\mathbf{NR},\mathbf{SAL},\mathbf{DPT}\}$ $\quad\quad$ $\text{Arg}_B(\mathbf{DEP},\mathbf{int}) = \{\mathbf{DNR},\mathbf{MAN}\}$

$\text{Arg}_B(\mathbf{EMPL},\mathbf{str}) = \{\mathbf{NAME}\}$ $\quad\quad$ $\text{Arg}_B(\mathbf{DEP},\mathbf{str}) = \{\mathbf{NAME}\}$

$\text{Arg}_B(\mathbf{EMPL},\mathbf{g}) = \{\mathbf{SEX}\}$ □

Examples of the use of intensional constants and arguments within queries will be given in section 4.3.

4.2. Queries

We start this section with a recursive definition of a relation Rwe(B), for each CL-basis B. If $(\alpha;\sigma) \in \text{Rwe}(B)$ then we say that α *is a well-formed expression of type* σ over B. L4.1 contains an alternative description of this notion. Further explanation is given after L4.1.

D4.3: If B is a CL-basis then:

<u>Rwe(B)</u> is the smallest set Y such that for every τ, τ', and σ in Typ_B:

(1) if $\alpha \in \text{Plh}_B(\sigma)$ then $(\alpha;\sigma) \in Y$,

(2) if $\alpha \in \text{Con}_B(\sigma)$ then $(\alpha;\sigma) \in Y$,

(3) if $\alpha \in \text{Int}_B(\sigma)$ then $({}^{\vee}\alpha;\sigma) \in Y$,

(4) if $\alpha \in \text{Unop}_B(\tau,\sigma)$ and $(\beta;\tau) \in Y$ then $(\alpha\beta;\sigma) \in Y$,

(5) if $\alpha \in \text{Binop}_B(\tau,\tau',\sigma)$, $(\beta;\tau) \in Y$, and $(\gamma;\tau') \in Y$ then $((\beta\alpha\gamma);\sigma) \in Y$,

(6) if $\alpha \in \text{Arg}_B(\tau,\sigma)$ and $(\beta;\tau) \in Y$ then $((\beta\bullet\alpha);\sigma) \in Y$,

(7) if $\alpha \in \mathrm{Plh}_B(\tau)$, $(\beta;\tau) \in Y$, and $(\gamma;\sigma) \in Y$ then $([\alpha \leftarrow \beta]\gamma;\sigma) \in Y$,

(8) if $\alpha \in \mathrm{Det}_B(\tau,\tau',\sigma)$, $\beta \in \mathrm{Plh}_B(\tau)$, $(\gamma;\mathbf{so}\lfloor\tau\rfloor) \in Y$, $(\varphi;\mathbf{t}) \in Y$, and $(\delta;\tau') \in Y$ then $(\alpha\beta \in \gamma \wedge \varphi : \delta;\sigma) \in Y$, and

(9) if $\beta \in \mathrm{Plh}_B(\tau)$, $(\gamma;\mathbf{so}\lfloor\tau\rfloor) \in Y$, $(\varphi;\mathbf{t}) \in Y$, and $(\delta;\sigma) \in Y$ then $(\sigma\beta \in \gamma \wedge \varphi : \delta;\sigma) \in Y$.

The set of all well-formed expressions of type σ (over B) is denoted by $\mathrm{We}_B(\sigma)$:

D4.4: If B is a CL-basis and $\sigma \in \mathrm{Typ}_B$ then:
$\underline{\mathrm{We}_B(\sigma)} \overset{D}{=} \{\alpha \mid (\alpha;\sigma) \in \mathrm{Rwe}(B)\}$.

The idea of avoiding a "concurrent" recursive definition of all the sets $\mathrm{We}_B(\sigma)$, $\sigma \in \mathrm{Typ}_B$, by using a "single" recursive definition of the set of all those pairs $(\alpha;\sigma)$ instead, is borrowed from Montague ([Mo 73], footnote 7).

An alternative description of the set of all well-formed expressions of type σ is obtained by saying that nothing is in $\mathrm{We}_B(\sigma)$, for any $\sigma \in \mathrm{Typ}_B$, except as required by (1)-(9) in the following lemma.

L4.1: If B is a CL-basis, $\tau \in \mathrm{Typ}_B$, $\tau' \in \mathrm{Typ}_B$, and $\sigma \in \mathrm{Typ}_B$ then:

(1) if $\alpha \in \mathrm{Plh}_B(\sigma)$ then $\alpha \in \mathrm{We}_B(\sigma)$;

(2) if $\alpha \in \mathrm{Con}_B(\sigma)$ then $\alpha \in \mathrm{We}_B(\sigma)$;

(3) if $\alpha \in \mathrm{Int}_B(\sigma)$ then ${}^{\vee}\alpha \in \mathrm{We}_B(\sigma)$;

(4) if $\alpha \in \mathrm{Unop}_B(\tau,\sigma)$ and $\beta \in \mathrm{We}_B(\tau)$ then $\alpha\beta \in \mathrm{We}_B(\sigma)$;

(5) if $\alpha \in \mathrm{Binop}_B(\tau,\tau',\sigma)$, $\beta \in \mathrm{We}_B(\tau)$, and $\gamma \in \mathrm{We}_B(\tau')$ then $(\beta\alpha\gamma) \in \mathrm{We}_B(\sigma)$;

(6) if $\alpha \in \mathrm{Arg}_B(\tau,\sigma)$ and $\beta \in \mathrm{We}_B(\tau)$ then $(\beta\bullet\alpha) \in \mathrm{We}_B(\sigma)$;

(7) if $\alpha \in \mathrm{Plh}_B(\tau)$, $\beta \in \mathrm{We}_B(\tau)$, and $\gamma \in \mathrm{We}_B(\sigma)$ then $[\alpha \leftarrow \beta]\gamma \in \mathrm{We}_B(\sigma)$;

(8) if $\mathbf{t} \in \mathrm{Typ}_B$, $\mathbf{so}\lfloor\tau\rfloor \in \mathrm{Typ}_B$, $\alpha \in \mathrm{Det}_B(\tau,\tau',\sigma)$, $\beta \in \mathrm{Plh}_B(\tau)$, $\gamma \in \mathrm{We}_B(\mathbf{so}\lfloor\tau\rfloor)$, $\varphi \in \mathrm{We}_B(\mathbf{t})$, and $\delta \in \mathrm{We}_B(\tau')$ then $\alpha\beta \in \gamma \wedge \varphi : \delta \in \mathrm{We}_B(\sigma)$;

(9) if $\mathbf{t} \in \mathrm{Typ}_B$, $\mathbf{SO}\lfloor\tau\rfloor \in \mathrm{Typ}_B$, $\beta \in \mathrm{Plh}_B(\tau)$, $\gamma \in \mathrm{We}_B(\mathbf{SO}\lfloor\tau\rfloor)$, $\varphi \in \mathrm{We}_B(\mathbf{t})$, and $\delta \in \mathrm{We}_B(\sigma)$ then $\boldsymbol{\sigma}\beta\ \boldsymbol{\in}\ \gamma \wedge \varphi\ \mathbf{:}\ \delta \in \mathrm{We}_B(\sigma)$.

We follow with some comments on these 9 clauses.

ad (1), (2), and (3): Roughly speaking, *placeholders* correspond to variables in the sense of logic (as in example 4.1), while *intensional constants* correspond to (external) variables in the sense of computer science: if α is an intensional constant then $^{\vee}\alpha$ can be read as "the current value of the variable α". The main difference between a *constant* and an *intensional constant* is that the "value" of an intensional constant does depend on the "actual state" (or "actual DB snapshot") while the "value" of a constant does not.

ad (6): In section 4.1 the symbol **@** was introduced to represent function application concerning functions for which the result type is the same for all of its arguments. Clause (6) accounts for function application concerning functions for which the result type depends on the argument concerned. (To some readers, such functions are maybe better known as *records* or as elements of *generalized products*.) Expressions denoting such functions confront us with a problem regarding types: What are the types of these expressions to look like? Or more precisely: How can we introduce types for these expressions without getting a laborious type calculus or type administration? In each practical application only a small number of such functions is necessary and each function has only finitely many arguments. Therefore, the following solution is feasible: Per application, some "primitive" types are introduced for such functions, for instance, the types **DEP** and **EMPL** in example 4.2. Furthermore, if τ_0 is such a "function type" then $\mathrm{Arg}(\tau_0,\sigma)$ must be the set of all arguments for which σ is the result type of application of a function (expression) of type τ_0 to that argument. Typical examples of such "function types" will be the table indices in a database system (hence the requirement $\mathrm{dom}(g) \subseteq \mathrm{Typ}_B$ in D4.2).

ad (7): This clause accounts for an *abbreviation* facility: $[\alpha \leftarrow \beta]\gamma$ may be read as "γ where $\alpha = \beta$".

ad (8): Familiar examples of determiners are the symbols $\forall$ and $\exists$ in Det(τ,**t**,**t**) as well as Σ (for general addition) and Π (for general multiplication) in, for instance, Det(τ,**int**,**int**).

$\forall\beta\in\gamma\wedge\varphi:\delta$ is usually written as $\forall\beta(((\beta\in\gamma)\wedge\varphi)\Rightarrow\delta)$,

$\exists\beta\in\gamma\wedge\varphi:\delta$ usually as $\exists\beta(((\beta\in\gamma)\wedge\varphi)\wedge\delta)$,

$\Sigma\beta\in\gamma\wedge\varphi:\delta$ sometimes as $\sum\limits_{\beta\in\gamma\wedge\varphi}\delta$, and

$\Pi\beta\in\gamma\wedge\varphi:\delta$ as $\prod\limits_{\beta\in\gamma\wedge\varphi}\delta$.

For later purpose, we will add the symbol $\nexists$ as a useful (though superfluous) element of Det(τ,**t**,**t**). The expression $\nexists\beta\in\gamma\wedge\varphi:\delta$ will be equivalent to the expression $\neg\exists\beta\in\gamma\wedge\varphi:\delta$. (We note that the word *equivalent* will be used in its informal sense, i.e., two expressions will be called equivalent if they have the same intended meaning.)

As other useful candidates in Det(τ,**t**,**t**) we introduce the determiners $(\exists\vartheta\eta)$ where $\vartheta \in \{=,<,\geq,\leq,>\}$ and $\eta \in$ Con(**int**).[3] The expression $(\exists{=}\eta)\beta\in\gamma\wedge\varphi:\delta$ should be read as "there are exactly η elements β in γ for which φ and δ hold". If $=$ is replaced by $<$, $\geq$, $\leq$, or $>$, respectively, then "exactly" should be replaced by "less than", "at least", "at most", or "more than", respectively. In other words, the expression $(\exists\vartheta\eta)\beta\in\gamma\wedge\varphi:\delta$ is equivalent to the expression $(\Sigma\beta\in\gamma\wedge(\varphi\wedge\delta):1\vartheta\eta)$.

We also want to treat the common way of set formation, usually expressed by means of $\{..\mid ....\}$, as a determiner. For that purpose we will use the symbol \$ (for \$et) and \$ can be placed in Det(τ,τ',**so**$\lfloor\tau'\rfloor$). In traditional notation, our expression $\$\beta\in\gamma\wedge\varphi:\delta$ would read as $\{\delta \mid \beta\in\gamma\wedge\varphi\}$ or, rather, as $\{\alpha \mid \exists\beta\in\gamma\wedge\varphi: (\alpha = \delta)\}$ where α is any "fresh" placeholder of type τ', i.e., $\alpha \neq \beta$ and α does not occur in any of the expressions γ, φ, or δ.

[3] It would be more general to allow $\eta \in$ We(**int**), but this implies that the set of determiners, i.e., a part of the basis, and the set of well-formed expressions have to be defined concurrently. In the presence of clause (7), however, it is sufficiently general to allow $\eta \in$ Con(**int**) $\cup$ Plh(**int**), or just $\eta \in$ Plh(**int**). In that case we would have to adapt D4.6, case (8).

The symbol $\mathbf{U}$ (denoting generalized union) can be treated as an element of Det(τ,**so**$\lfloor\tau'\rfloor$,**so**$\lfloor\tau'\rfloor$). We note that $\mathbf{\$}\beta\in\gamma\wedge\varphi{:}\delta$ is equivalent to $\mathbf{U}\beta\in\gamma\wedge\varphi{:}\,\mathbf{sngl}\ \delta$.

Also the symbol $\boldsymbol{\lambda}$ (for functional abstraction) could be treated as a determiner. However, treating $\boldsymbol{\lambda}$ as an element of Det(τ,τ',**fc**$\lfloor\tau;\tau'\rfloor$), as might be expected, creates a problem: The type **fc**$\lfloor\tau;\tau'\rfloor$ represents the set of all "total" functions over $\mathcal{D}(\tau)$, i.e., the set denoted by τ, while $\boldsymbol{\lambda}\beta\in\gamma\wedge\varphi{:}\delta$ denotes a "partial" function over the set denoted by τ. There are two reasons for this "partiality": (1) γ denotes, in general, just a subset of the set denoted by τ, and (2) φ acts as a further restriction on this set. We are inclined to treat $\boldsymbol{\lambda}$ as an element of Det(τ,τ',**so**$\lfloor$**p**$\lfloor\tau;\tau'\rfloor\rfloor$), in accordance with the usual treatment of functions in set theory. As a consequence, function application (denoted by the symbol **@**) does not work for $\boldsymbol{\lambda}$-expressions since **@** only works for expressions of type **fc**$\lfloor\tau;\tau'\rfloor$. This is not a serious consequence, however, because $\boldsymbol{\lambda}$-expressions in our query languages are only meant to *construct* functions, without actually having to *apply* these functions. Moreover, the expression $(\boldsymbol{\lambda}\beta\in\gamma\wedge\varphi{:}\delta\,@\,\alpha)$ would be equivalent to $[\beta\leftarrow\alpha]\delta$ and hence, for our purposes, superfluous.

We follow with some general remarks concerning determiners.

For each α in Det(τ,τ',σ), We(σ) might contain an expression e_α such that for every φ in We(**t**) and every δ in We(τ'), the expression $\alpha\beta\in\emptyset\wedge\varphi{:}\delta$ is equivalent to e_α or, in terms of axiom systems, such that $(\alpha\beta\in\emptyset\wedge\varphi{:}\delta = e_\alpha)$ can be chosen as an axiom. Loosely speaking, the expression e_α, which is independent of φ and δ, describes the "result" of the determiner α when applied to the empty set. For each determiner α in the first column of the table below, e_α is given in the second column.

α	e_α	d_α
$\forall$	$\top$	$\wedge$
$\exists$	$\bot$	$\vee$
Σ	0	$+$
Π	1	$\times$
$\nexists$	$\top$	
$(\exists\vartheta\eta)$	$(0\vartheta\eta)$	
$\$$	$\emptyset$	$\dot{\cup}$
$\cup$	$\emptyset$	$\cup$
λ	$\emptyset$	

Table 4.1.

For some determiners α in table 4.1, the third column contains a binary operation symbol d_α of which that determiner is a generalization in the following sense: If $\alpha \in \mathrm{Det}(\tau,\tau',\sigma)$ then $d_\alpha \in \mathrm{Binop}(\sigma,\tau',\sigma)$ and for every β in $\mathrm{Plh}(\tau)$, every γ in $\mathrm{We}(\mathbf{so}\lfloor\tau\rfloor)$, every φ in $\mathrm{We}(\mathbf{t})$, every δ in $\mathrm{We}(\tau')$, and every β' in $\mathrm{Plh}(\tau) - \{\beta\}$ that does not occur in any of the expressions γ, φ, or δ, the expression

$$\forall\beta'\in\gamma\wedge[\beta\leftarrow\beta']\varphi\colon (\alpha\beta\in\gamma\wedge\varphi\colon\delta = (\alpha\beta\in(\gamma\backslash\mathbf{sngl}\ \beta')\wedge\varphi\colon\delta\cdot d_\alpha[\beta\leftarrow\beta']\delta))$$

can be chosen as an axiom, an axiom that reduces quantification over γ to quantification over a smaller set. (As an exercise, the reader should check the well-formedness of the expression above.) This "reduction axiom" and the axiom $(\alpha\beta\in\emptyset\wedge\varphi\colon\delta = e_\alpha)$ together "define" α for those γ that denote finite sets. (In chapter 6, these axioms can be used in proving the correctness of the translation of expressions that contain determiners.)

In table 4.1, the symbol $\dot{\cup}$ is an element of $\mathrm{Binop}(\mathbf{so}\lfloor\tau'\rfloor,\tau',\mathbf{so}\lfloor\tau'\rfloor)$ denoting union with a singleton, in other words, $(\mu\ \dot{\cup}\ \delta)$ is equivalent to $(\mu \cup \mathbf{sngl}\ \delta)$.

We finally note that it turns out to be convenient to use $\alpha\beta\in\gamma\colon\delta$ as shorthand for $\alpha\beta\in\gamma\wedge\mathbf{T}\colon\delta$, a special case which occurs very frequently in practice. This could be sanctioned by adding another clause to D4.3:

(8') if $\alpha \in \mathrm{Det}_B(\tau,\tau',\sigma)$, $\beta \in \mathrm{Plh}_B(\tau)$, $(\gamma;\mathbf{so}\lfloor\tau\rfloor) \in Y$, and $(\delta;\tau') \in Y$ then $(\alpha\beta\in\gamma\colon\delta;\sigma) \in Y$.

ad (9): The expression $\sigma\beta\in\gamma\wedge\varphi:\delta$ should be read as "δ where β is such that $((\beta\in\gamma)\wedge\varphi)$ holds". Syntactically, clause (9) resembles clause (8), but semantically there is a difference: Clause (9) is a clause with a *presupposition*, namely the presupposition that there is *at least one* β such that $((\beta\in\gamma)\wedge\varphi)$ holds. (Therefore, clause (8) and clause (9) will have different translations in chapter 6, where $\sigma\beta\in\gamma\wedge\varphi:\delta$ will be translated into a program in which the presupposition will also be checked.)

We note that clause (9) introduces nondeterminism in our languages (although chapter 6 contains a deterministic implementation). In our applications of clause (9), β will denote a table element and φ will prescribe the value of one of the keys (key in the sense of D1.6); in such a case there is at most one β such that $((\beta\in\gamma)\wedge\varphi)$ holds.

ad(7) and (9): Although $[\alpha \leftarrow \beta]\gamma$ is equivalent to $\sigma\alpha\in \mathbf{sngl}\ \beta\wedge\mathbf{T}:\gamma$, the former expression will have a completely different (and in fact a more "efficient") translation in chapter 6.

After this commentary on L4.1 (and hence on D4.3), we continue our subject with the notion of a *well-formed query*, i.e., a well-formed expression followed by the symbol **?**. If B is a CL-basis then Wq_B is the set of all well-formed queries based on B:

D4.5: If B is a CL-basis then:

$$\underline{Wq_B} \overset{D}{=} \{\alpha\mathbf{?} \mid \exists\sigma\in Typ_B\colon \alpha \in We_B(\sigma)\}.$$

A CL-basis and the corresponding sets of well-formed expressions and queries can be described by means of two grammars, namely as follows:

For the description of the set of types we use a quasi-cfg G_0; the set of types shall be $L(G_0)$. In practice, a *cfg* G_0 suffices, like in example 4.2.

Next, let $P_0(G_0)$ be the rule set consisting of all production rules of one of the following 10 forms (where τ, τ', and σ can vary over $L(G_0)$):

(0) <Query> ::= <E;σ> ?

(1) <E;σ> ::= <P;σ>

(2) | <C;σ>

(3) | ∀<I;σ>

(4) | <U;τ;σ><E;τ>

(5) | {<E;τ><B;τ;τ';σ><E;τ'>)

(6) | (<E;τ> • <A;τ;σ>)

(7) | [<P;τ> ← <E;τ>]<E;σ>

(8) | <D;τ;τ';σ><P;τ> ∈ <E;so⌊τ⌋> ∧ <E;t> : <E;τ'>

(9) | σ <P;τ> ∈ <E;so⌊τ⌋> ∧ <E;t> : <E;σ>

To this rule set $P_0(G_0)$ we have to add a rule set P_1 containing zero or more production rules for each of the nonterminals <P;σ>, <C;σ>, <I;σ>, <U;τ;σ>, <B;τ;τ';σ>, <A;τ;σ>, and <D;τ;τ';σ> (for any τ, τ', and σ in $L(G_0)$, the intended set of types). We require that the non-terminals <Query> and <E;σ>, for all σ in $L(G_0)$, do not occur in any production rule of P_1 in order to rule out the possibility of defining the ingredients of the intended CL-basis concurrently with the sets of well-formed expressions and queries.

The additional clause (8') can be incorporated by adding the following rule form to the other 10 rule forms:

(8') <E;σ> ::= <D;τ;τ';σ><P;τ> ∈ <E;so⌊τ⌋> : <E;τ'>

By means of G_0 and the quasi-cfg G_1 that has $P_0(G_0) \cup P_1$ as its rule set and <Query> as its start symbol, we arrive at a CL-basis, namely the CL-basis <$L(G_0)$;H> where H is the 7-tuple defined as follows (see D4.1 and D3.6): For all τ, τ', and σ in $L(G_0)$

$H_0(\sigma)$ = $Fsd(G_1$,<P;σ>) ,

$H_1(\sigma)$ = $Fsd(G_1$,<C;σ>) ,

$H_2(\sigma)$ = $Fsd(G_1$,<I;σ>) ,

$H_3((\tau;\sigma))$ = $Fsd(G_1$,<U;τ;σ>) ,

$H_4((\tau;\sigma))$ = $Fsd(G_1$,<A;τ;σ>) ,

$H_5(((\tau;\tau');\sigma))$ = $Fsd(G_1$,<B;τ;τ';σ>) ,

$H_6(((\tau;\tau');\sigma))$ = $Fsd(G_1$,<D;τ;τ';σ>) .

Finally, let B_1 be $\langle L(G_0);H\rangle$; then $We_{B_1}(\sigma)$ is just $Fsd(G_1,\langle E;\sigma\rangle)$ for every σ in Typ_{B_1} (thus in $L(G_0)$), and Wq_{B_1} is just $L(G_1)$!

After this grammatical intermezzo, we define the set $\underline{FP_B(\psi)}$, for each CL-basis B and for all well-formed expressions ψ based on B; $FP_B(\psi)$ is called the set of *free placeholders* in ψ. The definition is by recursion on ψ. According to D4.3 (or L4.1), there are 9 cases:

D4.6: If B is a CL-basis then:

$$\begin{array}{lll}
(1) & FP_B(\alpha) & = \{\alpha\}\ , \\
(2) & FP_B(\alpha) & = \emptyset\ , \\
(3) & FP_B(\check{}\alpha) & = \emptyset\ , \\
(4) & FP_B(\alpha\beta) & = FP_B(\beta)\ , \\
(5) & FP_B(\pmb{(}\beta\alpha\gamma\pmb{)}) & = FP_B(\beta) \cup FP_B(\gamma)\ , \\
(6) & FP_B(\pmb{(}\beta\cdot\alpha\pmb{)}) & = FP_B(\beta)\ , \\
(7) & FP_B(\pmb{[}\alpha \leftarrow \beta\pmb{]}\gamma) & = FP_B(\beta) \cup [FP_B(\gamma) - \{\alpha\}]\ , \\
(8) & FP_B(\alpha\beta\pmb{\in}\gamma\pmb{\wedge}\varphi\pmb{:}\delta) & = FP_B(\gamma) \cup [[FP_B(\varphi) \cup FP_B(\delta)] - \{\beta\}]\ , \\
(9) & FP_B(\pmb{\sigma}\beta\pmb{\in}\gamma\pmb{\wedge}\varphi\pmb{:}\delta) & = FP_B(\gamma) \cup [[FP_B(\varphi) \cup FP_B(\delta)] - \{\beta\}]\ .
\end{array}$$

A *closed expression* is a well-formed expression without free placeholders and a *closed query* - the kind of query we are interested in (!) - is a closed expression followed by the symbol **?**.

D4.7: If B is a CL-basis and $\sigma \in Typ_B$ then:

$$\begin{array}{ll}
(a) & \underline{Cle_B(\sigma)} \overset{D}{=} \{\alpha \in We_B(\sigma) \mid FP_B(\alpha) = \emptyset\}\ ; \\
(b) & \underline{Clq_B} \overset{D}{=} \{\alpha\pmb{?} \mid \exists\tau\in Typ_B\colon \alpha \in Cle_B(\tau)\}\ .
\end{array}$$

4.3. Examples of queries

In order to express the queries that are relevant to a database based on a certain type 2 skeleton, we need a CL-basis *fit for* that type 2 skeleton (see D4.2). In this section we present some general forms of queries over an arbitrary CL-basis fit for some type 2 skeleton <g;h> as well as some concrete examples of queries with respect to the sample type 2 skeleton <g1;h1> (see example 4.2). For some of these queries, we also give account of the rule forms (as presented after D4.5) used.

In the remainder of this section, <g;h> will be an arbitrary type 2 skeleton, E will be a table index (i.e., $E \in \mathrm{dom}(g)$), a and b will be attributes of E (i.e., $\{a,b\} \subseteq g(E)$), and B will be a CL-basis fit for <g;h>. From D4.2 we now conclude that E and $\mathbf{SO}\lfloor E \rfloor$ are types of B, that E is an intensional constant of type $\mathbf{SO}\lfloor E \rfloor$, and that there is one type σ_0 and one type σ_1 such that $a \in \mathrm{Arg}_B(E,\sigma_0)$ and $b \in \mathrm{Arg}_B(E,\sigma_1)$. Finally, x will be a placeholder of type E.

The request for the a-values of all E-tuples is expressed by the (closed) query $\mathbf{\$}x \in {}^{\mathbf{v}}E \,\mathbf{:}\mathbf{(}x\mathbf{.}a\mathbf{)?}$ which is built up from clauses (3), (1), (6), (8'), and finally (0). In particular, if a = **NR** and E = **EMPL** then the set of all employee numbers was asked for.

Often, a request is of the form "Give all E-tuples for which the a-value is w" where w is a particular closed expression of type σ_0, i.e., $w \in \mathrm{Cle}_B(\sigma_0)$. This request is expressed by
$\mathbf{\$}x \in {}^{\mathbf{v}}E \wedge \mathbf{((}x\mathbf{.}a\mathbf{)} = w\mathbf{)} \,\mathbf{:}\, x\mathbf{?}$, a closed query that is built up by using the clauses (3), (1), (6), (5), again (1), (8), and (0). In particular, if E = **EMPL**, a = **SEX**, w = **♀**, and x = **e** then the set of all female employees was asked for. The request for all employees that do *not* belong to department 5 - an example that will be used in chapter 6 - has a similar form: $\mathbf{\$e \in {}^{v}EMPL \wedge ((e.DPT) \neq 5) : e?}$

If <g;U> is a type 1 model in which a is a key for E (see D1.6) then one often asks for the b-value of *the* (possibly absent (!)) E-tuple for which the a-value is w. This can be expressed by the query $\boldsymbol{\sigma}x \in {}^{\mathbf{v}}E \wedge \mathbf{((}x\mathbf{.}a\mathbf{)} = w\mathbf{)} \,\mathbf{:}\mathbf{(}x\mathbf{.}b\mathbf{)?}$ which at the same time shows an application of clause (9). In particular, the request for the name of department 5 is of this form: take E = **DEP**, a = **DNR**, w = **5**, and b = **NAME**.

We now illustrate the use of connector indices as intensional constants (see one of the requirements in D4.2). There are two main uses of these language elements, one in combination with **@** and the other in combination with **inv**. We explain these uses below.

Let C be a connector index under <g;h>, i.e., $C \in \mathrm{dom}(h)$. By D1.8, h(C) will be a pair of table indices, say the pair (M;E). From

D4.2 we conclude that (M;E) is a type of B and that C is an intensional constant of that type. We recall that our usual notation for the type (M;E) is **fc**⌊M;E⌋.

If β is an expression of type M then **(**$^{\vee}$C **@** β**)** is an expression of type E which denotes the result of application of the DB function denoted by C to the M-tuple denoted by β. For instance, **(**$^{\vee}$**DEPOF @** β**)** denotes the department tuple of the employee table denoted by β, and **(**$^{\vee}$**MANAGEROF @ (**$^{\vee}$**DEPOF @** β**))** denotes the manager tuple of that department tuple.

If γ is an expression of type E then **(**$^{\vee}$C **inv** γ**)** is an expression of type **so**⌊M⌋ which denotes the pre-image of the E-tuple denoted by γ under the DB function denoted by C. For instance, **(**$^{\vee}$**DEPOF inv** γ**)** denotes the set of all employee tuples belonging to the department tuple denoted by γ.

We finally illustrate the fore-mentioned constructs more concretely by the following two examples of closed queries, both expressing the question whether or not there is an employee who earns more than the manager of his (or her) department:

(a) **$\exists e \in {}^{\vee}$EMPL: ((e.SAL) > ((**$^{\vee}$**MANAGEROF @ (**$^{\vee}$**DEPOF @ e)).SAL))?**

(b) **$\exists d \in {}^{\vee}$DEP: [n ← ((**$^{\vee}$**MANAGEROF @ d).SAL)] $\exists e \in$(**$^{\vee}$**DEPOF inv d):**
((e.SAL) > n)?

The latter query shows an application of clause (7) and also two applications of clause (8').

5. A CLASS OF PROGRAMMING LANGUAGES

5.0. Introduction and summary

In section 5.1 a set consisting of 44 production rules and 25 rule forms is presented. Together with <program> as start symbol, this set can be used to make various quasi-cfg's that describe programming languages into which the conceptual languages of chapter 4 can be translated. In order to get such quasi-cfg's, the details of the production rules for variable identifiers, field identifiers, and constants of type $\mathbb{Z}$ have to be added to the syntax given in section 5.1; and for more specific applications, the "infrastructure" in section 5.1 might be extended with extra production rules for, e.g., types, constants, unary operators, binary operators, and standard procedures.

After the specification of the common syntax of our quasi-cfg's (with <program> as start symbol), various comments on the grammar are given in section 5.2. In this section we also mention some context-sensitive requirements a program should satisfy, requirements which are not expressed in the grammar itself.

In order to indicate the way in which our programming languages can be simulated in existing programming languages for data processing, we will compare parts of them with two important languages intended for data handling, namely, with PASCAL and with COBOL extended with some typical database statements, the so-called DBTG proposal. These comparisons can be found in section 5.3 and are primarily meant for readers acquainted with one of these languages.

5.1. The common syntax

```
(00)  <program>          ::= <block>.
(01)  <block>            ::= begin <var.decl.part>; <stat.list> end
(02)  <var.decl.part>    ::= <var.decl.>
(03)                        |<var.decl.part>, <var.decl.>
(04)  <stat.list>        ::= <stat.>
(05)                        |<stat.list>; <stat.>
(06)  <stat.>            ::= skip
(07)                        |halt
(08)                        |<assignm.stat.>
(09)                        |if <cond.> then <stat.list> else <stat.list> fi
(10)                        |if <cond.> then <stat.list> fi
(11)                        |while <cond.> do <stat.list> od
(12)                        |<block>
(13)                        |<stand.proc.stat.>

(14)  <type>             ::= 𝔹
(15)                        |ℤ
(16)                        |file of <type>
(17)                        |product <field decl.part> end
(18)  <field decl.part>  ::= <field decl.>
(19)                        |<field decl.part>, <field decl.>
(20)  <field decl.>      ::= <field id.>: <type>

(21)  <var.decl.>        ::= <var.id.;T>: T
(22)  <assignm.stat.>    ::= <var.;T> := <expr.;T>
(23)  <var.;T>           ::= <var.id.;T>
(24)                        |<var.;T'> . <field id.>
(25)  <expr.;T>          ::= <var.;T>
(26)                        |<const.;T>
(27)                        |<un.opr.;T1,T><expr.;T1>
(28)                        |(<expr.;T1><bin.opr.;T1,T2,T><expr.;T2>)
(29)  <cond.>            ::= <expr.;𝔹>
```

```
(30)  <const.;𝔹>              ::= true | false
(31)  <un.opr.;𝔹,𝔹>           ::= not
(32)  <un.opr.;ℤ,ℤ>           ::= -
(33)  <bin.opr.;𝔹,𝔹,𝔹>        ::= and|or| =
(34)  <bin.opr.;ℤ,ℤ,ℤ>        ::= +|-|*|div|mod
(35)  <bin.opr.;ℤ,ℤ,𝔹>        ::= <|≥|≤|>|=

(36)  <stand.proc.stat.>      ::= return(<expr.;T>)
(37)                              |mess(<const.;ℤ>)
(38)                              |<file gener.ps.>
(39)                              |<file insp.ps.>
(40)                              |<link insp.ps.>

(41)  <fv.;T>                 ::= <var.;file of T>
(42)  <expr.;𝔹>               ::= in(<fv.;T>)
(43)  <file gener.ps.>        ::= rewrite(<fv.;T>)
(44)                              |write(<fv.;T>; <expr.;T>)

(45)  <var.decl.>             ::= <var.id.;ℙ> : ℙ
(46)  <assignm.stat.>         ::= <var.id.;ℙ> := <fv.;T> ↓
(47)  <file insp.ps.>         ::= fdp(<fv.;T>; <var.id.;ℙ>)
(48)                              |fdc(<fv.;T>; <avd.part;T>)
(49)                              |fff(<fv.;T>)
(50)                              |fnf(<fv.;T>)
(51)                              |read(<fv.;T>; <var.;T>)
(52)  <avd.part;T>            ::= <avd.;T>
(53)                              |<avd.part;T>, <avd.;T>
(54)  <avd.;T>                ::= <field id.> : <expr.;T1>

(55)  <link insp.ps.>         ::= fte(<var.id.;li T to T1>)
(56)                              |ffl(<var.id.;li T to T1>)
(57)                              |fnl(<var.id.;li T to T1>)
```

5.2. Commentary on the syntax

This section contains some explanation and nomenclature concerning the syntax given in section 5.1.

ad (00): A *program* is a *block* followed by a period.

ad {(02),(03),(04),(05)}: A *variable declaration part* consists of one or more variable declarations separated by commas, whereas a *statement list* consists of one or more statements separated by semicolons. (Thus, a statement list can consist of a *single* statement.)

Within a variable declaration part of a block, each variable may be declared at most once. We note that this is a context-sensitive requirement that is not expressed in the grammar itself.

ad {(03),(19),(53)}: We use the comma as a separator if, from a conceptual point of view, the order of the separated items is irrelevant. For instance, variable declaration parts consisting of the same variable declarations in a possibly different order should be considered equivalent.

ad {(01),(05),(44),(47),(48),(51)}: We use the semicolon as a separator if the order of the separated items can be relevant (e.g., the order of the statements in a statement list).

ad (10): The statement **if** α **then** β **fi** is equivalent to the statement **if** α **then** β **else skip fi** and, hence, superfluous. Nevertheless, it is a convenient abbreviation.

ad (14): Instead of $\mathbb{B}$, the type identifier **Boolean** is commonly used.

ad (15): Instead of $\mathbb{Z}$, the type identifier **integer** is commonly used.

ad (16): We call α the *base type* of the type **file of** α.

ad {(17),(18),(19),(20)}: A *product type* contains a *field declaration part* which, in turn, contains one or more field declarations. A *field declaration* consists of a field identifier together with a type. A product type corresponds to what is often known as a *record type*; such a type "denotes" a generalized product (see chapter 0). We recall that production rules for field identifiers still have to be added.

Within a product type each field identifier may be declared only once. This is another context-sensitive requirement not expressed in the grammar itself.

Field declaration parts consisting of the same field declarations in a possibly different order should be considered equivalent (see ad {(03),(19),(53)}).

If α is a field identifier declared in a product type β then β is called an *operator type of* α and the type mentioned in the (unique) declaration of α in β is called *the result type of* α *within* β or, conversely, *the component type of* β *associated with* α.

We note that a field identifier can have several operator types and several result types, but per operator type only one result type, because of the context-sensitive requirement mentioned above.

ad {(14),(15),(16),(17)}: These types and type constructs are sufficient for the general structure of the translations given in chapter 6. In specific applications, however, various extra types (e.g., a "string type" $\mathbb{S}$) and type constructs (e.g., the array construct) might be useful.

By a *type in the wider sense* we mean a terminal string of the nonterminal <tws> when the following production rules are added to the syntax of section 5.1:

```
<tws> ::= <type>
        | ℙ
        | li <type> to <type>
```

The *position type* $\mathbb{P}$ will be used in (45)-(47) and the so-called *link types* in (55)-(57).

ad (21) and following: Our programming languages are *typed* (or *attributed* or *two-level*) languages, just as the languages in chapter 4. For instance, each *variable*, *constant*, and *expression* "has" some type, as the phrase goes. (By a *type* we mean a terminal string of the nonterminal <type>.) Furthermore, each *unary operator* has an *operand* type and a *result* type, and each *binary operator* has a *first* operand type, a *second* operand type, and a result type.

Each of the 25 lines (21) up to and including (28), (36), (41), (42), (43), (44), and (46) up to and including (57), contains a *rule form* instead of a production rule (see the discussion on rule forms in chapter 3). Each rule form r stands for a *set* of production rules: Every function f associating a type with each of the "parameters" T, T', T_1, and T_2 (in so far as present in r), gives rise to one production rule, namely, the one obtained from r by replacing all occurrences of a parameter $\sigma \in \{T,T',T_1,T_2\}$ by the corresponding type $f(\sigma)$. For example, one of the production rules obtained from the rule form in line (27) is

<expr.;$\mathbb{B}$> ::= <un.opr.;$\mathbb{Z}$,$\mathbb{B}$><expr.;$\mathbb{Z}$>

using the function f for which $f(T) = \mathbb{B}$ and $f(T_1) = \mathbb{Z}$. A reasonable (though superfluous) example of a unary operator with operand type $\mathbb{Z}$ and result type $\mathbb{B}$ is the operator **even** where the Boolean expression **even** β is equivalent to the expression $((\beta \ \mathbf{mod}\ 2) = 0)$.

ad (21): The rule form in this line represents infinitely many production rules for the nonterminal <var.decl.>, one production rule for every type. Two examples are:

<var.decl.> ::= <var.id.;$\mathbb{B}$> **:** $\mathbb{B}$

<var.decl.> ::= <var.id.;**file of** $\mathbb{Z}$> **;** **file of** $\mathbb{Z}$

We recall that production rules for variable identifiers still have to be added.

Note that line (45) contains another production rule for <var.decl.>.

ad (22): This rule form states that the variable and the expression in an *assignment statement* must have the same type. Again, there are *several* production rules for *one* nonterminal.

Note that line (46) contains another rule form for assignment statements.

ad {(23,(25),(26),(41),(52),(53)}: With the rule form in each of these lines, *several* nonterminals are possible on the "left hand side", but per possible nonterminal there is only *one* production rule.

ad {(24),(27),(28),(54)}: In each of these cases *several* nonterminals are possible on the left and, moreover, per possible nonterminal there are *several* production rules.

ad {(23),(24)}: When other type constructs are introduced (see the first paragraph ad {(14),(15),(16),(17)}) then extra rule forms for <var.;T> might be useful.

ad (24): This rule form does not express the following context-sensitive requirements:

(a) for T' a product type ψ' must be chosen,

(b) for T a component type ψ of ψ' must be chosen, and

(c) the field identifier must be a field of ψ' having ψ as its result type within ψ'.

ad {(25),(26),(27),(28)}: Our programming languages are representative examples of the kind of programming languages that still prevail in practice. This particularly holds for the limited possibilities for making *expressions*. The present rule forms for expressions resemble those in section 4.2 only to some extent: there are no analogues to the rule forms (7), (8), and (9) of the "conceptual grammars". Therefore, we cannot translate such conceptual languages into the present kind of programming languages in the obvious way, i.e., by translating expressions to *expressions*.

ad (26): In line (30) we specify the constants of type $\mathbb{B}$. The details of a specification of constants of type $\mathbb{Z}$ are not given in section 5.1. By the way, there are several possibilities. For instance, each of the following two possibilities (using the sample grammar given in chapter 3) will do:

<const.;$\mathbb{Z}$> ::= **0** | <pos.int.>

<const.;$\mathbb{Z}$> ::= <digit> | <const.;$\mathbb{Z}$><digit>

In the latter case, different constants, such as **7** and **007**, can denote the same integer. In both cases, not every integer can be denoted by a *constant* but, by means of the production rule in line (32), every integer can be denoted by an *expression*.

The set of constants of other types varies per programming language. A reasonable structure for the constants of an additional "string type" **S** would be

```
<const.;S> ::= '<char.list>'
<char.list> ::= <char.>
              |<char.list><char.>
```

In other words, a constant of type **S** is a *character list* enclosed by a *begin quote* and an *end quote* respectively.

As a more specific example, for our employees-and-departments database presented in chapter 1, we could introduce a "genus type" **G** with two constants, one for "male" and one for "female":

```
<const.;G> ::= M|F
```

ad {(27),(28)}: We have only a few (important) "standard" unary and binary operators; cf. the 15 production rules in the lines (31) up to and including (35). However, our programming languages can have other unary and binary operators as well; e.g., for concatenation of strings:

```
<bin.opr.;S,S,S> ::= &
```

ad (29): A *condition* is a Boolean expression.

ad {(31),(33)}: These operators are usually called *Boolean* or *logical operators*.

ad {(32),(34)}: These operators are known as *arithmetic operators*.

ad (35): These five operators are usually called *relational operators*.

ad (36): By execution of the standard procedure statement **return**(α), the value of the expression α will be delivered to the appropriate "output device" (whatever that may be).

ad (37): The standard procedure **mess** is used to deliver various special messages such as "Division by zero" and "Existence condition not fulfilled".

ad {(38),(39),(40)}: For database retrieval, we need some standard procedures for inspection of files and links and some standard procedures for generation of (internal) files, in order to record intermediate and final results.

ad (41): <fv.;T> is introduced only as an abbreviation, in behalf of the rule forms to come. <fv.;ψ> is the nonterminal for file variables with *base* type ψ.

ad (42): At any time *at most one* component per file R (then called the "current" component of R) is "directly accessible". The standard Boolean expression **in**(R) holds when indeed some component of R is directly accessible, and **in**(R) does not hold when no component of R is directly accessible.

ad (43) and following: In the explanation of the remaining statements, we give for each statement its so-called *pre-condition* (telling when the statement can be used) and its so-called *post-assertion* (describing the situation after its execution). If prior to the execution of a statement its pre-condition is not fulfilled, the program should be interrupted.

ad {(43),(44)}: We charge no special pre-conditions for the statements **rewrite**(R) and **write**(R;β). For both statements the post-assertion is that **in**(R) does not hold.

If **&** denotes concatenation of sequences with components of type T, **< >** denotes the empty sequence, and **<**β**>** denotes the one-place sequence (i.e., 1-tuple) with the value of the expression β as its component, then the statement

rewrite(R) is equivalent to R **:=** **< >** and
write(R;β) is equivalent to R **:=** R **&** **<**β**>** .

ad {(46),(47),(48),(49),(50),(51)}: As already suggested by the explanation of the file generation procedures, the value of a file variable R can best be conceived of as a sequence. Such a sequence, say s, will be considered as the composition of an enumeration e of the "positions" (or "addresses" or "locations") occupied by the components of R, and an "occupation function" b assigning to each occupied position the R-component concerned. In a formula: $s = b \circ e$ where $rng(e) = dom(b)$. And in a figure:

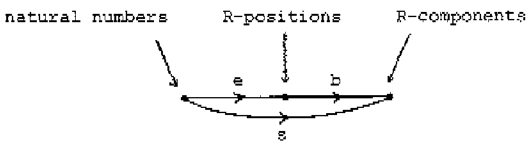


Figure 5.1.

The functions b and e associated with the value of R will be used to explain the effect of the file inspection procedures when applied to R.

If $<\mu;r;K>$ is a sequential storage structure for a type 2 snapshot $<g;h;v;w>$ and R is the database file corresponding to some table index $E \in \text{dom}(g)$ then b will be $\mu(E)$ and e will be $r(E)$; cf. chapter 2.

ad {(45),(46),(47)}: With each file variable R there is associated a variable $R\downarrow$ of type $\mathbb{P}$, where $\mathbb{P}$ is a special type called the *position type* or the *address type*. The variable $R\downarrow$ is called the *position variable of* R (or also the *currency variable of* R). We recall that **in**(R) holds when and only when some component of R is directly accessible. When **in**(R) holds then the position of the "current" component of R will be the value of $R\downarrow$. (The latter sentence can in fact be seen as the *definition* of "current".) The value of $R\downarrow$ will be an element of dom(b).

Sometimes a "current position" in a file is needed again at a later stage, i.e., after an update of the currency variable of that file. This so-called *currency problem* can be solved in an elegant manner as follows:

By (45) we can declare auxiliary variables of type $\mathbb{P}$ which can be used, by (46), to "remember" the current position of a file R. By means of (47) an "old position" can be re-assigned to the currency variable of R, for the statement **fdp**(R;y) is equivalent to the assignment statement $R\downarrow := y$, a "dual" of (46).

ad (46): The pre-condition for $y := R\downarrow$ is that **in**(R) holds. In that case, $R\downarrow$ will have a value $p \in \text{dom}(b)$. The corresponding post-assertion is that **in**(R) still holds and that both y and R$\downarrow$ have the value p.

ad (47): The pre-condition for **fdp**(R;y) is that y has a value p which is in dom(b). The corresponding post-assertion is that **in**(R) holds and that both y and R$\downarrow$ have the value p.

ad {(48),(52),(53),(54)}: While **fdp** accounts for *direct access by position*, **fdc** accounts for *direct access by contents*. **fdc** only applies to file variables with a product type as base type; and in practice, **fdc** applies only to some of these variables. These rule forms do not express the context-sensitive requirements that

(a) for T a product type ψ must be chosen;

(b) within each *attribute/value declaration* (see (54)), a field identifier α of ψ must be mentioned and, moreover, for T_1 the result type of α within ψ must be chosen;

(c) within the *attribute/value declaration part* β of the procedure statement **fdc**(R;β), each field identifier may be declared at most once.

Because of these requirements, β represents a function t_β, consisting of attribute/value pairs. In other words, β represents a "tuple fragment" t_β.

There is no special pre-condition for **fdc**(R;β). The post-assertion is that **in**(R) holds iff there is a tuple $t \in rng(b)$ such that $t_\beta \subseteq t$, and that *if* **in**(R) holds then R$\downarrow$ will have a value $p \in dom(b)$ such that $t_\beta \subseteq b(p)$. In other words, if there is an R-component having the attribute values as described by β then **in**(R) holds and the variable R$\downarrow$ will "point to" such an R-component; otherwise **in**(R) does not hold.

ad (49): There is no special pre-condition for the statement **fff**(R). The post-assertion is that **in**(R) holds iff $0 \in dom(e)$, and that if **in**(R) holds then R$\downarrow$ will have the value e(0). In other words, if the value of the file variable R is not empty then **in**(R) holds and R$\downarrow$ will point to the first component of R; otherwise **in**(R) does not hold. Mnemonics: **fff** stands for "find first of file".

ad (50): The pre-condition for **fnf**(R) is that **in**(R) holds. In that case, R$\downarrow$ will have a value $p \in$ dom(b). We recall that dom(b) = rng(e). Let j be $e^{-1}(p)$, i.e., the serial number of p under the enumeration e. The corresponding post-assertion is that **in**(R) holds iff $j+1 \in$ dom(e), and that if **in**(R) holds then R$\downarrow$ will have the value e(j+1). In other words, if there was a next component then **in**(R) holds afterwards and R$\downarrow$ will then point to that next component; otherwise **in**(R) does not hold afterwards. Mnemonics: **fnf** stands for "find next of file".

ad (51): The pre-condition for **read**(R;x) is that **in**(R) holds. In that case, R$\downarrow$ will have a value $p \in$ dom(b). The corresponding post-assertion is that **in**(R) still holds, R$\downarrow$ still has the value p, and the variable x will have the value b(p). In other words, **read**(R;x) assigns the current component of R to x, without changing the current position.

ad {(55),(56),(57)}: The effect of the link inspection procedures will be explained in terms of a sequential storage structure $<\mu;r;K>$ for a type 2 snapshot <g;h;V;w>, cf. D2.5. In the explanation below, C will be a connector index, M will be the source index of C, and D will be the target index of C; in other words, $C \in$ dom(h) and h(C) = (M;D). Furthermore, ρ_C will denote the location connector for (M;D) based on w(C) and μ, i.e., the function $(\mu_D)^{-1} \circ w(C) \circ \mu_M$ from dom(μ_M) into dom(μ_D), cf. figure 2.1.

The connector index C will be treated as a variable of type **li** M **to** D, where **li** M **to** D is a special type, a so-called *link type*.

We recall that the source index and the target index of a connector index might be the same. For our link inspection procedures, this will not be tangling since, by definition, **ffl** will operate on the current *target* position, while **fnl** and **fte** will operate on the current *source* position, also by definition.

Mnemonics: **fte** stands for "find target element", **ffl** for "find first in link", and **fnl** for "find next in link".

ad (55): The precondition for **fte(C)** is that **in(M)** holds. In that case, M $\downarrow$ will have a value $m \in \mathrm{dom}(\mu_M)$. The corresponding post-assertion is that **in(D)** holds and that D $\downarrow$ will have the value $\rho_C(m)$. In other words, afterwards D $\downarrow$ will point to the "C-target" of the M-component that was current beforehand.

We note that the currency variable of the source index is not updated if the source index and the target index are *not* the same, in casu, if $M \neq D$. However, if they are the same then also the currency variable of the source index is affected (because it is the target index as well).

ad (56): The pre-condition for **ffl(C)** is that **in(D)** holds. In that case, D $\downarrow$ will have a value $d \in \mathrm{dom}(\mu_D)$. The corresponding post-assertion is that **in(M)** holds iff $0 \in \mathrm{dom}(K_C(d))$, and that if **in(M)** holds then M $\downarrow$ will have the value $K_C(d)(0)$. In other words, if at least one M-component is mapped to the current D-component by the function w(C) then **in(M)** holds and M $\downarrow$ will point to the first M-component in the enumeration $K_C(d)$ of the M-components corresponding to the D-component that was current beforehand; otherwise **in(M)** does not hold.

ad (57): The pre-condition for **fnl(C)** is that **in(M)** holds. In that case, M $\downarrow$ will have a value $m \in \mathrm{dom}(\mu_M)$. Let a be the enumeration $K_C(\rho_C(m))$ and j be $a^{-1}(m)$. The corresponding post-assertion is that **in(M)** holds iff $j+1 \in \mathrm{dom}(a)$, and that if **in(M)** holds then M will have the value $a(j+1)$. In other words, if there was a next component in the enumeration of the M-components with the same "C-target" as the M-component that was current beforehand then **in(M)** holds afterwards and M $\downarrow$ will then point to that next component; otherwise **in(M)** does not hold afterwards.

5.3. A comparison with PASCAL and with the DBTG proposal

In order to indicate the way in which our programming languages can be simulated in existing languages for data processing, we will compare parts of them with two important languages intended for data handling, namely, with COBOL extended with some typical database statements (the DBTG proposal, cf. [CC 81]), and with PASCAL.

We start with a comparison with PASCAL.

ad (06): **skip** corresponds to the *empty statement* in PASCAL.

ad (07): By adding the label declaration "**label 0;**" in the program block and replacing "**end.**" at the end of a PASCAL program by "**0: end.**", the statement **halt** can be simulated in a systematic way by the PASCAL statement **goto 0**.

ad {(09),(10),(11)}: The statement lists can be replaced by a so-called *compound statement*, and the terminators **fi** and **od** can be deleted.

ad (12): After adding a name to each block in the original program, the statement in (12) can be replaced by PASCAL's *procedure statement*.

ad (36): If α is an expression of type $\mathbb{B}$, $\mathbb{Z}$, or **$** then **return(α)** corresponds to the statement **write(output,α)**. However, the latter statement is not defined if the type of α is a *structured type*, e.g., a file type. In a conversion of our programming languages into PASCAL, the conversion of the **return**-statement should be defined recursively on the structure of our types.

ad (37): Each specific application of **mess** is equivalent to some **return**-statement using an expression of type **$**; and this case is treated in "ad (36)".

ad (42): **in(R)** corresponds to **not eof(R)**.

ad {(43),(44)}: Our file generation procedures **rewrite** and **write** are almost the same as those in PASCAL, the differences being that (1) our procedures allow *any* type as a base type (of the file type), and (2) our procedure **write** does *not* have a pre-condition. Compare, for instance, [HW 73].

ad (49): **fff**(R) corresponds to **reset**(R).

ad (50): **fnf**(R) corresponds to **get**(R).

ad (51): Our **read**(R;x) is equivalent to PASCAL's x := R ↑ whereas PASCAL's **read**(R,x) is equivalent to our statement list **read**(R;x); **fnf**(R). In PASCAL, files are meant for *sequential* processing only, and therefore it was natural to combine copying of a component with advancing the file position. For databases, however, other "navigation" possibilities are feasible as well. This accounts for the two different choices made for **read**.

ad (45) and following: Of all facilities presented from line (45) on, only those in (49), (50), and (51) are explicitly available in standard PASCAL!! For instance, PASCAL lacks procedures for direct access by contents as well as procedures for direct access by position. Procedures for link inspection, which are very useful in databases, are not available either.

We continu with a comparison with the DBTG proposal, using COBOL as the so-called "host language".

ad (01): The variable declaration parts of *all* blocks occurring in a program must be placed in the so-called DATA DIVISION of the corresponding COBOL program. In order to avoid a *clash of variables*, renaming of (local) variables might be necessary.

ad (08): An assignment statement can be simulated by a COMPUTE statement in COBOL.

ad {(09),(10),(11),(12)}: We have to add a (procedure) name to each "block statement" and to each statement list occurring in a **while** - or **if** - statement. In the original program each such block and statement list must be replaced by the corresponding procedure name; and the body of each procedure, preceded by its name, must be specified *after* the body of the main program, i.e., after the statement STOP RUN. Furthermore, **while** c **do** pn **od** becomes PERFORM pn UNTIL NOT c. (Note that the systematic replacement of each statement list in an IF-statement by a procedure call also avoids the problems concerning the period in COBOL.)

ad (36): **return**(α) corresponds to the statement DISPLAY α if the (grammatical) construction of the type of α does not make use of the production rule in (16). Returning a *file* must be written out explicitly (using a PERFORM-UNTIL statement).

ad (37): Each specific application of **mess** is equivalent to some **return**-statement using an expression of type **S** and such a statement is equivalent to a DISPLAY-statement.

ad (47) and following: Our "find statement" (statements beginning with an **f**) do not operate on the current of run-unit but on the current of some *file*. The same holds for our **read**. In our treatment, we can dispense with the currents of set type, area, and run-unit. We can also dispense with the SUPPRESS-statement, which we consider harmful, to paraphrase [Dk 68]. (Such a statement obscures the structure of a program.)

ad (47): This corresponds to the *format* 1 of the FIND-statement in the DBTG proposal.

ad {(48),(52),(53),(54)}: Conversion to DBTG will result in some MOVE statements - one for every attribute/value declaration - followed by a FIND statement of format 2. The conversion is determined by the following rules (using three auxiliary functions C_1, C_2, and C_3):

```
(48)  C_1(fdc(R;α))  = C_2(α)
                       FIND ANY R.

(52)  C_2(α)         = C_3(α)

(53)  C_2(β,γ)       = C_2(β)
                       C_3(γ)

(54)  C_3(δ:φ)       = MOVE φ TO δ.
```

ad (51): Our **read**(R;x) is equivalent to GET R. MOVE R TO x, provided that the current of run-unit has the same value as the current of R.

ad {(55),(56),(57)}: Our extra type **li** M **to** D corresponds more or less to a DBTG set type with member M and owner D.

ad (55): This corresponds to the FIND-OWNER-statement, i.e., format 6.

ad {(56),(57)}: These are FIND statements of format 4; (56) more or less corresponds to the FIND-FIRST-statement, (57) to the FIND-NEXT-statement.

ad {(49),(50)}: We note that a file variable is often treated as a so-called "singular set" in DBTG. In that case, (49) falls under (56) and (50) falls under (57).

6. TRANSLATING CONCEPTUAL LANGUAGES INTO PROGRAMMING LANGUAGES

6.0. Introduction and summary

Let B be a CL-basis in the sense of chapter 4 and let P be a programming language in the sense of chapter 5. We want to assign a program $Z(q)$ of P to each closed query q based on B, i.e., to each $q \in Clq_B$. (The purpose of the program $Z(q)$ is to compute the answer to the query q.) The translation function Z over Clq_B will be defined in section 6.3. In order to define the function Z, we need some auxiliary functions.

To begin with, we need a function F which assigns to each type of B a "corresponding" type of P. This function will be introduced in section 6.1.

In order to translate expressions containing determiners, we need two auxiliary functions over the set of determiners of B. Both functions are introduced in section 6.2.

The most important auxiliary function is a function V which assigns a statement (list) $V(R,\alpha)$ to a well-formed expression α over B and a variable R of P. The post-assertion of $V(R,\alpha)$ is that $(R \doteq \alpha)$ holds (provided that all presuppositions of α - see chapter 4 - are fulfilled). In other words, after execution of $V(R,\alpha)$ the variable R will have the value of α in the "current state" (provided that the value of α is defined in that state). V is introduced in section 6.3. This section also contains the definition of Z.

In section 6.4, an example of a closed query will be translated step by step.

In section 6.5, we present some important improvements on the straightforward translation given in section 6.3. As an illustration, the sample query used in section 6.4 will be translated once again, this time by using our alternative translation function.

In section 6.6, some special problems concerning the translation of connector indices are treated.

6.1. Translating types

Given a CL-basis B and a programming language P, we want to define a function, say $\mathcal{F}$, from the set of types of B into the set of types of P (or, rather, into its set of types "in the wider sense", see section 5.2). If the set of types of B is defined recursively then $\mathcal{F}$ is usually defined recursively as well.

Natural choices for the types **t** and **int** are $\mathcal{F}(\mathbf{t}) = \mathbb{B}$ and $\mathcal{F}(\mathbf{int}) = \mathbb{Z}$. For other types σ of B there is usually some freedom in the choice of $\mathcal{F}(\sigma)$. In this thesis, we let $\mathcal{F}(\mathbf{so}\lfloor\tau\rfloor)$ be the type **file of** $\mathcal{F}(\tau)$, because a set is usually implemented as a sequence (preferably as a one-to-one sequence, i.e., a sequence without "duplicates"). Furthermore, we choose $\mathcal{F}(\mathbf{fc}\lfloor\tau;\tau'\rfloor)$ to be **li** $\mathcal{F}(\tau)$ **to** $\mathcal{F}(\tau')$, a type " in the wider sense".

If τ_0 is a primitive type such as discussed in section 4.2, ad (6), then we define $\mathcal{F}(\tau_0)$ as the product type in P of which the field declaration part consists of exactly those field declarations $\alpha : \mathcal{F}(\sigma)$ for which σ is a type of B and α is an element of $\mathrm{Arg}_B(\tau_0,\sigma)$. An instance of such a primitive type is the type **DEP** introduced in example 4.2. If we choose $\mathcal{F}(\mathbf{str}) = \mathbb{S}$ (see section 5.2, ad (26)) then it follows from our definition of $\mathcal{F}$ that

```
F(DEP) = product DNR : ℤ,
                 MAN : ℤ,
                 NAME : 𝕊
         end
```

A good choice for the "genus type" **g** of example 4.2 is $\mathcal{F}(\mathbf{g}) = \mathbb{G}$; cf. section 5.2, ad (26).

6.2. Two auxiliary functions for determiners

In clause (8) of the translations in section 6.3 and section 6.5 we make use of a function $\mathcal{I}$ which assigns a statement list $\mathcal{I}(\alpha,\mathrm{R})$ to a determiner α and a variable R. More precisely, if $\alpha \in \mathrm{Det}_B(\tau,\tau',\sigma)$ then R is a variable of type $\mathcal{F}(\sigma)$. The purpose of $\mathcal{I}(\alpha,\mathrm{R})$ is to initialize the variable R to the value of the expression $\alpha\beta\mathbf{C}\emptyset\wedge\varphi;\delta$. We recall from section 4.2 that for a determiner α there is an expression

e_α such that $(\alpha\beta\in\emptyset\wedge\varphi:\delta = e_\alpha)$ is an axiom schema, whereas e_α is independent of φ and δ. For each determiner α in the first column of the table below, e_α is given in the second column and the initialization $I(\alpha,R)$ is given in the fourth column.

In clause (8) we also use a function O which assigns a statement list $O(\alpha,R,S)$ to each determiner α in combination with a variable R and an expression S of the programming language. More precisely, if $\alpha \in \mathrm{Det}_B(\tau,\tau',\sigma)$ then R is of type $F(\sigma)$ and S is of type $F(\tau')$. If α denotes a generalization - in the sense of section 4.2, ad (8) - of some binary operation symbol d_α then the purpose of $O(\alpha,R,S)$ is to assign to the variable R the result of applying the binary operation denoted by d_α to the current value of R and the value of S. The third column of the table below contains the binary operation symbols concerned.

For determiners of the form $(\exists\vartheta\eta)$ we recall from section 4.2 that $(\exists\vartheta\eta)\beta\in\gamma\wedge\varphi:\delta$ is equivalent to $(\Sigma\beta\in\gamma\wedge(\varphi\wedge\delta):1\ \vartheta\eta)$. In translations concerning those determiners, we need an extra variable of type $\mathbb{Z}$ in order to compute the value of $\Sigma\beta\in\gamma\wedge(\varphi\wedge\delta):1$. In the table below, this variable is denoted by k. A better translation for determiners of the form $(\exists\vartheta\eta)$ will be given in section 6.5.

α	e_α	d_α	$I(\alpha,R)$	$O(\alpha,R,S)$
Σ	**0**	$+$	R **:= 0**	R **:= (**R **+** S**)**
Π	**1**	$\times$	R **:= 1**	R **:= (**R ***** S**)**
$\exists$	$\bot$	$\vee$	R **:= false**	R **:= (**R **or** S**)**
$\forall$	$\top$	$\wedge$	R **:= true**	R **:= (**R **and** S**)**
\$	$\emptyset$	$\dot{\cup}$	**rewrite(**R**)**	**write (**R**;**S**)**
$(\exists\vartheta\eta)$	$(0\vartheta\eta)$		k **:= 0;** R **:= (**k$\vartheta\eta$**)**	**if** S **then** k **:= (**k **+ 1);** R **:= (**k$\vartheta\eta$**)** **fi**

6.3. Translating queries into programs

A translation function $\mathcal{Z}$ from the conceptual language generated by a CL-basis B into a programming language P is defined in clause (0) below. $\mathcal{Z}$ is defined in terms of a function $\mathcal{V}$ which assigns a statement list $\mathcal{V}(R,\alpha)$ to each well-formed expression α over B in combination with a variable R of P. More precisely, if α is of type σ then R is of type $\mathcal{F}(\sigma)$. The definition of $\mathcal{V}$ will be by recursion on the structure of the well-formed expression (i.e., the second component): The 9 clauses below correspond to those in section 4.2. Execution of the statement list $\mathcal{V}(R,\alpha)$ will terminate if all "set-valued" subexpressions of α (i.e., all subexpressions with a type of the form $\mathbf{SO}\lfloor\tau\rfloor$) denote *finite* sets. In that case, the variable R will have the value of the expression α in the "current state", provided that the value of α is defined in that state. In a database application, the current state will be the current *DB snapshot*; cf. D1.2. If the value of α is not defined then an appropriate message will be returned; see for instance the translation of clause (9). It can be proved that if $\mathcal{V}(R,\alpha)$ is used within the translation of a closed query then the resulting program will contain a declaration of the variable R, and the type of R will be $\mathcal{F}(\sigma)$ if the type of α is σ.

The straigtforward translation presented in this section is uniform and easy to comprehend, but it will produce inefficient programs. Therefore, some (very effective) refinements will be presented in section 6.5.

We start with the translation rules sec and then we give some comments, first on some common features and then on the individual features of the translation rules.

Strictly speaking, not the expressions themselves are translated, but their derivation trees (based on the grammar proposed after D4.5 in section 4.2). The reason is that the *types* of the (sub)expressions will play a role in the translation and these types do not occur in the expressions themselves but in their derivation trees.

```
(0)  Z(α?)           - begin R : F(σ); V(R,α); return(R) end.

(1)  V(R,α)          = R := α

(2)  V(R,α)          = $S_0$(α,R)

(3)  V(R,$^{\vee}$α)          = R := α

(4)  V(R,αβ)         = begin H : F(τ); V(H,β); $S_1$(α,R,H) end

(5)  V(R,(βαγ))      = begin H : F(τ), H' : F(τ');
                               V(H,β); V(H',γ);
                               $S_2$(α,R,H,H')
                        end

(6)  V(R,(β.α))      - begin H : F(τ); V(H,β); R := H.α end

(7)  V(R,[α←β]γ)     = begin α : F(τ); V(α,β); V(R,γ) end

(8)  V(R,αβ∈γ∧φ:δ) =

     begin β : F(τ), H : F(so⌊τ⌋), B : 𝔹, H' : F(τ');                   (8.1)
           I(α,R); V(H,γ); fff(H);                                      (8.2)
           while in(H)                                                  (8.3)
              do read(H;β);                                             (8.4)
                 V(B,φ); if B then V(H',δ); O(α,R,H') fi;               (8.5)
                 fnf(H)                                                 (8.6)
              od                                                        (8.7)
     end                                                                (8.8)

(9)  V(R,σβ∈γ∧φ:δ) =

     begin β : F(τ), H : F(so⌊τ⌋), B : 𝔹;                               (9.1)
           B := false; V(H,γ); fff(H);                                  (9.2)
           while (in(H) and not B)                                      (9.3)
              do read(H;β);                                             (9.4)
                 V(B,φ);                                                (9.5)
                 if not B then fnf(H) fi                                (9.6)
              od;                                                       (9.7)
           if B then V(R,δ)                                             (9.8)
                else mess(1); halt                                      (9.9)
           fi                                                           (9.10)
     end                                                                (9.11)
```

We follow with some comments on the translation rules just introduced.

ad {(0),(4),(5),(6),(8),(9)}: In translations using these clauses, new (local) variables are introduced. (In the translation rules above they are denoted by the meta-variables R, H, H', and B.) In each application of one of these rules within the (recursive) translation of one and the same query, we must use "fresh" variables, i.e., variables that are not already introduced elsewhere within the translation of the query concerned.

ad {(7),(8),(9)}: The variable declared in the (first) variable declaration in the block produced in (7), (8), and (9), respectively, is not "fresh", but is taken over from the "source expression". (We denoted these program variables by the meta-variables α, β, and β, respectively.) As a consequence, a placeholder of type τ in the conceptual language should also occur in the programming language, namely as a variable of type $F(\tau)$.

ad {(2),(4),(5)}: In these clauses, $S_0(\alpha,R)$, $S_1(\alpha,R,H)$, and $S_2(\alpha,R,H,H')$ are statement lists that have to be specified per constant, per unary operation symbol, and per binary operation symbol, respectively. The intended post-assertion for $S_0(\alpha,R)$ is that $(R = \alpha)$ holds, for $S_1(\alpha,R,H)$ it is that $(R = \alpha H)$ holds, and for $S_2(\alpha,R,H,H')$ the intended post-assertion is that $(R = (H\alpha H'))$ holds. In other words, $S_0(\alpha,R)$ is equivalent to "R := α", $S_1(\alpha,R,H)$ is equivalent to "R := αH", and $S_2(\alpha,R,H,H')$ is equivalent to "R := (HαH')". Examples will be given further on, when we treat each of the clauses individually.

ad {(1),(3)}: Intensional constants correspond to so-called *external variables*, i.e., variables that already exist outside our programs; together, they constitute the *information system* we want to ask questions about. Placeholders, on the other hand, translate to variables that are local to our programs.

ad (0): We want to translate "expression languages" (as presented in chapter 4) into "statement languages" (as presented in chapter 5). Although these two kinds of languages have a completely different nature, clause (0) shows us a simple structure for the translation of expression languages into statement languages: A query is translated to a program in which an auxiliary variable is used to compute the answer to that query and then the result, i.e., the value of that variable, is returned.

ad (2): $S_0(\alpha,R)$ needs to be specified for each constant α separately; it has to be a statement (list) equivalent to "R := α". Examples are:

$S_0(\emptyset,R)$ = **rewrite**(R)

$S_0(\bot,R)$ = R := **false**

$S_0(\top,R)$ = R := **true**

$S_0(\alpha,R)$ = R := α   for every $\alpha \in$ Con(**int**)

ad (4): $S_1(\alpha,R,H)$ needs to be specified for each unary operation symbol α separately; it has to be a statement (list) equivalent to "R := αH". Examples are:

S_1(**sngl**,R,H) = **rewrite**(R); **write**(R;H)

$S_1(\neg,R,H)$ = R := **not** H

$S_1(-,R,H)$ = R := − H

$S_1(+,R,H)$ = R := H

ad (5): $S_2(\alpha,R,H,H')$ needs to be specified for each binary operation symbol α separately; it has to be a statement (list) equivalent to "R := (HαH')". Examples are:

```
S2(≠,R,H,H') = R := not(H = H')
S2(⇒,R,H,H') = if H then R := H' else R := true fi
S2(÷,R,H,H') = if (H' = 0) then mess(0); halt
                           else R := (H div H')
               fi
```

Hence, if α is the binary operation symbol ÷ and the denominator turns out to be zero then the standard procedure **mess** is called and after that the execution of the surrounding program will be terminated. The call **mess(0)** will return an appropriate message, say "Division by zero". A more refined message handling is also feasible, cf. ad (9).

Furthermore, if α is $\wedge$, $\vee$, $\Leftrightarrow$, or $\times$ then there is a "corresponding" symbol α' in the programming language (**and**, **or**, $=$, and $\neq$, respectively), and $S_2(\alpha,R,H,H')$ will simply be $R := (H\alpha'H')$. For each $\alpha \in \{+,-,<,\geq,\leq,>,=\}$ the corresponding symbol is α itself.

ad (6): We conclude from section 6.1 that the argument α will also be a field of $F(\tau)$, the type of H.

ad (7): Before the value of γ is computed, the variable (and "former placeholder") α will be assigned the value of the expression β.

ad (8): We have several comments on this clause:

1) We obtain a translation rule for the additional clause (8') in section 4.2 by replacing line (8.5) by

$$V(H',\delta);\ O(\alpha,R,H'); \qquad (8'.5)$$

In this case the declaration of the auxiliary variable B in line (8.1) can be deleted.

2) If α is of the form $(\exists\vartheta\eta)$ then an extra auxiliary variable of type $\mathbb{Z}$ has to be declared in line (8.1); see also section 6.2 and section 6.5.

3) If the expression **init**(H) denotes the set of file components up to (but *excluding*) the current component of H whenever **in**(H) holds, and **init**(H) denotes the set of *all* file components of H when **not in**(H) holds, then

$$(R = \alpha\beta \in \textbf{init}(H) \wedge \varphi : \delta)$$

is a so-called *invariant* of the **while**-loop (see, e.g., [AA 78]).

ad (9): There are two important invariants of this **while**-loop:

(a) $\neg\exists\beta\in\textbf{init}(H):\varphi$

(b) **if** B **then** φ

After termination of the **while**-statement, (B **or not in**(H)) holds. If B holds then by (b) φ holds for the current value of β (for which $(\beta \in \gamma)$ also holds); then δ is computed for this value of β. If β does not hold then **not in**(H) holds. In that case, **init**(H) denotes the value of H; this value will be the value of γ, thanks to the preceding statement list $V(H,\gamma)$. Hence, if B does not hold then $\neg\exists\beta\in\gamma:\varphi$ holds by (a) and, consequently, the *presupposition* $\exists\beta\in\gamma:\varphi$ (of the source expression $\sigma\beta\in\gamma\wedge\varphi:\delta$) does not hold. Therefore, a message will be returned in this case and the execution of the surrounding program will be terminated. An appropriate message would be "Existence condition not fulfilled".

A more technical reply (expressed in the conceptual language itself) would be to return the text $\neg\exists\beta\in\gamma:\varphi$, i.e., the juxtaposition of the symbol $\neg$, the symbol $\exists$, the text our *meta-variable* β *stands for* (!), the symbol $\in$, the text γ stands for, etc. This can be accomplished by replacing **mess(1)** in line (9.9) by **return**('$\neg\exists\beta\in\gamma:\varphi$').

Similarly, **mess(0)**, in one of the examples of clause (5), could be replaced by **return**('γ = 0').

6.4. An example

A simple example of a closed query concerning our employees-and-departments database (see example 4.2) is

$$\$e \in {}^{\mathsf{Y}}\mathrm{EMPL} \wedge ((e.\mathrm{DPT}) \neq 5):\ e?$$

This query, asking for the set of all employees that do not belong to department 5, will be translated step by step. A number above an equality sign will indicate the translation rule applied there. For convenience, $\$e \in {}^{\mathsf{Y}}\mathrm{EMPL} \wedge ((e.\mathrm{DPT}) \neq 5):\ e$ is abbreviated by $\alpha 1$, $((e.\mathrm{DPT}) \neq 5)$ by $\varphi 1$, and $(e.\mathrm{DPT})$ by $\beta 1$. The type of $\alpha 1$ is **so**$\lfloor$EMPL$\rfloor$ and from section 6.1 it follows that

```
F(EMPL) = product DPT : ℤ,
                  NAME: 𝕊,
                  NR  : ℤ,
                  SAL : ℤ,
                  SEX : 𝔾
          end
```

F(**EMPL**) will not be written out again hereafter. As auxiliary variables we will successively choose **X0**, **X1**, **X2**, etc.

After this explanation, we are ready to present the translation of our query:

```
Z(α1?) $\overset{(0)}{=}$ begin X0: F(so⌊EMPL⌋);
                  V(X0,α1);
                  return(X0)
              end.
```

where F(**so⌊EMPL⌋**) = **file of** F(**EMPL**)

and V(**X0**,α1) = V(**X0**,**\$e** ∈ ˅**EMPL** ∧ φ1**:** ***e***) $\overset{(8)}{=}$

```
begin e: F(EMPL), X1: F(so⌊EMPL⌋), X2: 𝔹, X3: F(EMPL);
    I($,X0); V(X1,˅EMPL); fff(X1);
    while in(X1)
       do read(X1;e);
          V(X2,φ1); if X2 then V(X3,e); O($,X0,X3) fi;
          fnf(X1)
       od
end
```

From section 6.2 we conclude that I(**\$**,**X0**) = **rewrite(X0)** and O(**\$**,**X0**,**X3**) = **write(X0;X3)**. Furthermore,

```
V(X1,˅EMPL)  $\overset{(3)}{=}$  X1 := EMPL

V(X3,e)      $\overset{(1)}{=}$  X3 := e

V(X2,φ1)      =   V(X2,(β1 ≠ 5))
             $\overset{(5)}{=}$  begin X4: F(int), X5: F(int);
                         V(X4,β1); V(X5,5);
                         s₂(≠,X2,X4,X5)
                     end
```

where s_2(≠,**X2**,**X4**,**X5**) = **X2 := not(X4 = X5)**

and V(**X5**,**5**) $\overset{(2)}{=}$ s_0(**5**,**X5**)

= **X5 := 5**

and V(**X4**,β1) = V(**X4**,(**e.DPT**)) $\overset{(6)}{=}$

```
    begin X6: F(EMPL); V(X6,e); X4 := X6.DPT end
```

Finally, $V(X6,e) \overset{(1)}{=}$ X6 := e

The complete program (in which, however, F(EMPL) is not written out) is given below. The resulting program is obviously very inefficient. In section 6.5, however, we present a more refined (though less uniform) function V' with which much better (i.e., more efficient) programs are obtained. For the result of the translation of our sample query according to V', we refer the reader to the end of section 6.5; the result according to V is:

```
    Z($e ∈ ᵛEMPL ∧ ((e.DPT) ≠ 5): e?) =

    begin X0: file of F(EMPL);
       begin e: F(EMPL), X1: file of F(EMPL), X2: 𝔹, X3: F(EMPL);
             rewrite(X0); X1 := EMPL; fff(X1);
             while in(X1)
                do read(X1;e);
                   begin X4: ℤ, X5: ℤ;
                      begin X6: F(EMPL);
                            X6 := e; X4 := X6.DPT
                      end;
                      X5 := 5;
                      X2 := not(X4 = X5)
                   end;
                   if X2 then X3 := e; write(X0;X3) fi;
                   fnf(X1)
                od
       end;
       return(X0)
    end.
```

6.5. Some improvements

In this section we present some improvements on the straightforward translation set out in section 6.3. More specifically, instead of V we will present a function V' with which we obtain better (i.e., more efficient) programs. On the other hand, V' is more laborious than V (because of the case analysis introduced). Z', the associated translation function, is defined in terms of V' in the same way as Z was defined in terms of V:

$Z'(\alpha ?)$ = **begin** R: $F(\sigma)$; $V'(R,\alpha)$; **return**(R) **end**.

Before we describe V', we introduce the notion of a *simple* expression of a conceptual language. Informally, an expression of a conceptual language is simple if there is an equivalent *expression* in each programming language of the form described in chapter 5. Formally, the following recursive definition determines for each expression of a conceptual language (in the sense of chapter 4) whether it is simple or not:

- an expression that arises from clause (1) or (3) of D4.3 is simple;
- α, from (2), is simple iff α is of type **t** or α is of type **int**;
- $\alpha\beta$, from (4), is simple iff β is simple and $\alpha \in \{\neg, -, +\}$;
- $(\beta\alpha\gamma)$, from (5), is simple iff β and γ are simple and
  either $\alpha \in \{\wedge, \vee, \Leftrightarrow, \times, +, -, <, \geq, \leq, >\}$
  or $\alpha \in \{=, \neq\}$ and β is of type **int**;
- $(\beta . \alpha)$, from (6), is simple iff β is a so-called *136-expression*, i.e., an expression that is built up by using only clauses (1), (3), and (6);
- an expression that arises from (7), (8), or (9) is not simple.

We leave it to the reader to prove that all 136-expressions are simple and that if an expression is simple then all of its subexpressions are simple as well.

For each simple expression e an equivalent expression $W(e)$ - belonging to each of our programming languages - is defined recursively as follows:

(1) $\mathcal{W}(\alpha) = \alpha$

(3) $\mathcal{W}(^{\vee}\alpha) = \alpha$

(2) $\mathcal{W}(\bot) =$ **false**

$\mathcal{W}(\top) =$ **true**

$\mathcal{W}(\alpha) = \alpha$ for every $\alpha \in Con($**int**$)$

(4) $\mathcal{W}(\neg\beta) =$ **not** $\mathcal{W}(\beta)$

$\mathcal{W}(-\beta) = -\mathcal{W}(\beta)$

$\mathcal{W}(+\beta) = \mathcal{W}(\beta)$

(5) $\mathcal{W}((\beta \neq \gamma)) =$ **not**$(\mathcal{W}(\beta) = \mathcal{W}(\gamma))$

$\mathcal{W}((\beta\alpha\gamma)) = (\mathcal{W}(\beta)\ \alpha'\ \mathcal{W}(\gamma))$

if α is $\wedge$, $\vee$, $\leftrightarrow$, $\times$, $+$, $-$, $<$, $\geq$, $\leq$, $>$, or $=$, and

α' is **and**, **or**, $=$, $*$, $+$, $-$, $<$, $\geq$, $\leq$, $>$, or $=$,

respectively

(6) $\mathcal{W}((\beta.\alpha)) = \mathcal{W}(\beta).\alpha$

An example of a simple expression is the expression **((e.DPT) ≠ 5)** which we used in section 6.4. By the rules above,

$\mathcal{W}($**((e.DPT) ≠ 5)**$) =$ **not(e.DPT = 5)**

We note that for each 136-expression e, $\mathcal{W}(e)$ is a variable of the programming language; cf. section 6.3, ad {(1),(3)}, and line (24) in section 5.1.

By means of the function $\mathcal{W}$, which is defined over the set of simple expressions, we can give several improvements on the translation set out in section 6.3. To start with, we give 6 of the 9 clauses of the definition of $\mathcal{V}'$. The clauses (5), (8), and (9) will be treated later. In the following survey, the definition of our new translation function Z' - in terms of the not yet completely defined function $\mathcal{V}'$ - will also be repeated.

(0) $Z'(\alpha?)$ = **begin** R**:** $F(\sigma)$**;** $\mathcal{V}'(R,\alpha)$**; return**(R) **end.**

(1) $\mathcal{V}'(R,\alpha) = R := \alpha$

(2) $\mathcal{V}'(R,\alpha) = S_0(\alpha,R)$

(3) $\mathcal{V}'(R,{}^{\vee}\alpha) = R := \alpha$

```
(4)  $\mathcal{V}'(R,\alpha\beta)$      = $S_1(\alpha,R,\mathcal{W}(\beta))$          if $\beta$ is simple
     $\mathcal{V}'(R,\alpha\beta)$      = begin $H$: $\mathcal{F}(\tau)$;
                               $\mathcal{V}'(H,\beta)$;
                               $S_1(\alpha,R,H)$
                         end                           if $\beta$ is not simple

(6)  $\mathcal{V}'(R,(\beta.\alpha))$   = $R := \mathcal{W}(\beta).\alpha$            if $(\beta.\alpha)$ is simple
     $\mathcal{V}'(R,(\beta.\alpha))$   = begin $H$: $\mathcal{F}(\tau)$;
                               $\mathcal{V}'(H,\beta)$;
                               $R := H.\alpha$
                         end                           if $(\beta.\alpha)$ is not simple

(7)  $\mathcal{V}'(R,[\alpha\leftarrow\beta]\gamma)$ = begin $\alpha$: $\mathcal{F}(\tau)$; $\mathcal{V}'(\alpha,\beta)$; $\mathcal{V}'(R,\gamma)$ end
```

For clause (5) we have a more elaborate case analysis. Just as there are 2 cases in (4), i.e., for a *unary* operation symbol, there are 4 cases for each binary operation symbol α in (5):

(C11) both β and γ are simple,

(C10) β is simple and γ is not simple,

(C01) β is not simple and γ is simple,

(C00) both β and γ are not simple.

(In general, for an n-ary operation symbol there would be 2^n cases.) If both β and γ are simple then we treat all binary operation symbols in the same way, but in the other three cases we will distinguish between

(a) $\alpha \notin \{\wedge,\vee,\rightarrow\}$ and (b) $\alpha \in \{\wedge,\vee,\rightarrow\}$.

We start with (C11) and the (a)-versions of (C10), (C01), and (C00), respectively:

```
(5)  $\mathcal{V}'(R,(\beta\alpha\gamma))$ = $S_2(\alpha,R,\mathcal{W}(\beta),\mathcal{W}(\gamma))$
     $\mathcal{V}'(R,(\beta\alpha\gamma))$ = begin $H'$: $\mathcal{F}(\tau')$; $\mathcal{V}'(H,\gamma)$; $S_2(\alpha,R,\mathcal{W}(\beta),H')$ end
     $\mathcal{V}'(R,(\beta\alpha\gamma))$ = begin $H$: $\mathcal{F}(\tau)$; $\mathcal{V}'(H,\beta)$; $S_2(\alpha,R,H,\mathcal{W}(\gamma))$ end
     $\mathcal{V}'(R,(\beta\alpha\gamma))$ = begin $H$: $\mathcal{F}(\tau)$, $H'$: $\mathcal{F}(\tau')$;
                           $\mathcal{V}'(H,\beta)$; $\mathcal{V}'(H',\gamma)$;
                           $S_2(\alpha,R,H,H')$
                       end
```

As an example of the difference between V and V', we note that V'(**X2**, ((**e.DPT**) $\neq$ **5**)) is the single statement **X2 := not(e.DPT = 5)** whereas V(**X2**, ((**e.DPT**) $\neq$ **5**)) is a block containing 4 statements and 3 extra auxiliary variables (see the 6th up to and including the 12th line of the program at the end of section 6.4).

It is easily (albeit recursively) proved that if α is simple then $V'(\mathrm{R},\alpha)$ will be R := $W(\alpha)$!

We continu with the (b)-versions of (C10) and (C00):

(5) $V'(\mathrm{R},(\beta \wedge \gamma)) = V'(\mathrm{R},\beta)$; **if** R **then** $V'(\mathrm{R},\gamma)$ **fi**

$V'(\mathrm{R},(\beta \vee \gamma)) = V'(\mathrm{R},\beta)$; **if not** R **then** $V'(\mathrm{R},\gamma)$ **fi**

$V'(\mathrm{R},(\beta \Rightarrow \gamma)) = V'(\mathrm{R},\beta)$; **if** R **then** $V'(\mathrm{R},\gamma)$ **else** R := **true fi**

Thanks to these rules, the computation of the value of γ is sometimes avoided in the resulting computer program. This can be very profitable if this computation is expected to be laborious.

Finally, the (b)-version of (C01) reads:

(5) $V'(\mathrm{R},(\beta \wedge \gamma)) = V'(\mathrm{R},\gamma)$; **if** R **then** $V'(\mathrm{R},\beta)$ **fi**

$V'(\mathrm{R},(\beta \vee \gamma)) = V'(\mathrm{R},\gamma)$; **if not** R **then** $V'(\mathrm{R},\beta)$ **fi**

$V'(\mathrm{R},(\beta \Rightarrow \gamma)) = V'(\mathrm{R},\gamma)$; **if not** R **then** $V'(\mathrm{R},\beta)$; R := **not** R **fi**

Here the computation of the value of β is sometimes avoided[4]. Moreover, we note that in all (b)-versions no auxiliary variables are needed.

We point out that if a presupposition of γ (respectively β) is not fulfilled then the first (respectively the second) set of alternatives for $\wedge$, $\vee$, and $\Rightarrow$ might give other responses than the original proposals in section 6.3. The first set of alternatives implements the *conditional* version of each of the three symbols, and the second alternative for $\wedge$ respectively $\vee$ implements the conditional version of

(4) As a consequence, this alternative would sometimes be better for the case (C00) as well, for instance if on certain grounds, maybe on mere *syntactic* grounds, the computation of β is expected to be much more laborious than the computation of γ, and (on "semantic" grounds) the "situation" γ seems to be unlikely when dealing with $\wedge$ (or rather likely when dealing with $\vee$ or $\Rightarrow$).

$(\gamma \wedge \beta)$ respectively $(\gamma \vee \beta)$. The original proposals, on the other hand, preserve the "symmetry" (or *commutativity*) of $\wedge$ and of $\vee$.

One of the reasons that we payed extra attention to improvements on the translation of the symbols $\wedge$, $\vee$, and $\Rightarrow$ is that those symbols occur rather frequently in practice, notably the symbol $\wedge$.

Also for the clauses (8) and (8') we have several improvements on the translation given in section 6.3. For referential purposes, we name them (a), (b), (c), etc.

(a) First of all, $\mathcal{V}'(R, \alpha\beta\epsilon\gamma\wedge\phi : \delta)$ will not be expressed in terms of $\mathcal{V}$ but in terms of $\mathcal{V}'$, an improvement in itself.

(b) Furthermore, for some determiners the condition **in**(H) in line (8.3) can be replaced by a stronger one. For the determiners Π, $\exists$, and $\forall$ a stronger condition is given in the table below. (In (d), the determiners of the form $(\exists\theta\eta)$ will be treated separately.)

α	a stronger condition wrt. α
Π	(**in**(H) **and not**(R = 0))
$\exists$	(**in**(H) **and not** R)
$\forall$	(**in**(H) **and** R)

(c) If α is the determiner $\forall$ and the stronger condition wrt. α is used, then $\mathcal{O}(\alpha, R, H')$, i.e., the assignment statement R := (R **and** H'), is equivalent to R := H'. Consequently, the fragment "$\mathcal{V}(H', \delta)$; $\mathcal{O}(\alpha, R, H')$" occurring in line (8.5) and in line (8'.5), can be simplified to "$\mathcal{V}'(R, \delta)$". Moreover, the variable H' and its declaration in line (8.1) can be dispensed with.

If α is the determiner $\exists$ then the fragment "$\mathcal{V}(H', \delta)$; $\mathcal{O}(\alpha, R, H')$" can also be replaced by "$\mathcal{V}'(R, \delta)$", by a similar argument.

(d) We recall from section 6.2 that in the translation concerning a determiner of the form $(\exists\theta\eta)$ an extra variable of type F(**int**) will be used. If k denotes that variable then the condition **in**(H) in line (8.3) can be replaced by the stronger condition (**in**(H) **and** (k $\leq$ η)) because, informally speaking, the definite value for the Boolean variable R is known as soon as (k $>$ η) holds. (If θ is the symbol $<$ or $\geq$ then even (**in**(H) **and** (k $<$ η)) would do!) Furthermore, it is sufficient to perform the assignment statement R := (k $\theta\eta$) only once, namely, *after* the **while**-statement, and not *before* and *with every* *iteration of* the body of the **while**-statement, as in the original proposal. These considerations lead us to the following translation scheme:

```
(8)  $V'(R,(\exists\vartheta\eta)\beta\in\gamma\wedge\varphi:\delta)$ =

     begin $\beta$: $F(\tau)$, $H$: $F(\mathbf{so}\lfloor\tau\rfloor)$, $B$: $\mathbb{B}$, $H'$: $\mathbb{B}$, $k$: $\mathbb{Z}$;
            $k$ := 0; $V'(H,\gamma)$; fff($H$);
            while (in($H$) and ($k \leq \eta$))
               do read ($H$;$\beta$);
                  $V'(B,\varphi)$;
                  if $B$ then $V'(H',\delta)$;
                            if $H'$ then $k$ := ($k$ + 1) fi
                  fi;
                  fnf($H$)
               od;
            $R$ := ($k\vartheta\eta$)
     end
```

The expression $(k = \Sigma\beta \in \mathbf{init}(H) \wedge (\varphi \wedge \delta): 1)$ is an invariant of the **while**-loop. In this connection, we recall that $(\exists\vartheta\eta)\beta\in\gamma\wedge\varphi:\delta$ is equivalent to $(\Sigma\beta \in \gamma \wedge (\varphi\wedge\delta): 1\ \vartheta\eta)$. The main advantage of the translation given above over the translation associated with the latter expression is the stronger "**while**-condition", namely, $(\mathbf{in}(H)$ **and** $(k \leq \eta))$ versus $\mathbf{in}(H)$.

We note that the extra variable B of type $\mathbb{B}$ can be saved by using H' instead.

(e) The next improvement applies if φ or δ is simple:

- if φ in clause (8) is simple then the fragment "$V(B,\varphi)$; **if** B **then**" in the original proposal (and "$V'(B,\varphi)$; **if** B **then**" in our proposal for $(\exists\vartheta\eta)$ in (d)) can be replaced by "**if** $W(\varphi)$ **then**", and the declaration of B can be omitted.
- if δ in clause (8), or in clause (8'), is simple and $\alpha \notin \{\forall,\exists\}$ then the fragment "$V(H',\delta)$; $O(\alpha,R,H')$" in the original proposal - see line (8.5) and line (8'.5) - can be replaced by "$O(\alpha,R,W(\delta))$", and the fragment "$V'(H',\delta)$; **if** H' **then**" in our latest proposal for $(\exists\vartheta\eta)$ can be replaced by "**if** $W(\delta)$ **then**". In both cases, the declaration of H' can be omitted.

For $\alpha \in$ {**∀**,**∃**} we already introduced a better proposal in (c), namely, to replace "$\mathcal{V}(H',\delta)$**;** $\mathcal{O}(\alpha,R,H')$" by "$\mathcal{V}'(R,\delta)$". This proposal even applies if δ is not simple.

Both improvements mentioned in (e) apply to our example used in section 6.4. The improved version of line (8.5) results in

```
if not(e.DPT = 5) then write (X0;e) fi;
```

whereas line (8.5) originally resulted in 8 lines of code; see section 6.4.

The following (important) improvement applies to each of the clauses (8), (8'), and (9). If γ is a 136-expression then γ is a simple expression and $\mathcal{W}(\gamma)$ is a variable. Therefore, it is feasible to choose for H the variable $\mathcal{W}(\gamma)$ itself. In this case, the declaration of H and the fragment "$\mathcal{V}(H,\gamma)$**;**" can be omitted, and the other (four) occurrences of H must then be replaced by "$\mathcal{W}(\gamma)$".

We note that this improvement is very effective since the omitted statement list $\mathcal{V}(H,\gamma)$ represented (needless) copying of a complete file!

A possible consequence of the fore-mentioned improvement is that it can introduce a *currency problem*: Unlike the "new" variable H, the "old" variable $\mathcal{W}(\gamma)$ can play a role in the program fragment that arises from line (8.5), (8'.5), or (9.5), respectively. In particular, the currency variable $\mathcal{W}(\gamma)\downarrow$ could be affected in that program fragment:

- by one of the file inspection procedures of line (47), (48), (49), or (50) of section 5.1, namely, if the f̲ile v̲ariable concerned is $\mathcal{W}(\gamma)$,
- by a link inspection procedure of the form **fte(**C**)**, namely, if $\mathcal{W}(\gamma)$ happens to be the *target index* of C, or
- by **ffl(**C**)** or **fnl(**C**)** if $\mathcal{W}(\gamma)$ happens to be the *source index* of C.

On account of the statement **fnf(**$\mathcal{W}(\gamma)$**)** in the sixth line of the translation scheme, the original value of $\mathcal{W}(\gamma)\downarrow$ has to be "remembered" when such a currency problem is menacing. We can do this by introducing an auxiliary variable β' of type **ℙ** and enclosing the fifth line of the translation scheme by the "dual" statements β' **:=** $\mathcal{W}(\gamma)\downarrow$ and **fdp(**$\mathcal{W}(\gamma)$**;** β'**)**, respectively. For clause (8), for instance, this results in the following translation scheme:

```
(8)  V'(R,αβ∈γ∧φ:δ) =

     begin β: F(τ), β': ℙ, B: 𝔹, H': F(τ');
            I(α,R); fff(W(γ));
            while in(W(γ))
               do read(W(γ);β); β' := W(γ)↓;
                  V'(B,φ); if B then V'(H',δ); O(α,R,H') fi;
                  fdp(W(γ);β'); fnf(W(γ))
               od
     end
```

In our example used in section 6.4, $^{\vee}$**EMPL** is a 136-expression, so the fore-mentioned improvement applies; it does not introduce any currency problem. When we use the translation function Z' (and hence the function V') then we get a considerably better program, indeed, a program as good as a "hand-made" one:

```
     Z'($e ∈ ˅EMPL ∧ ((e.DPT) ≠ 5): e?) =

     begin X0: file of F(EMPL);
           begin e: F(EMPL);
                 rewrite(X0); fff(EMPL);
                 while in(EMPL)
                    do read(EMPL;e);
                       if not(e.DPT = 5) then write (X0;e) fi;
                       fnf(EMPL)
                    od
           end;
           return(X0)
     end.
```

6.6. Translation rules concerning connector indices

In database applications, the binary operation symbols **@** and **inv** are useful in combination with connector indices, see section 4.3. The translation of these special combinations is given below.

With the rule forms

$<AUX;\sigma> ::= <P;\sigma>$ (RF1)

$|(^{\vee}<I;\mathbf{fc}\lfloor\tau;\sigma\rfloor> \ \textbf{@} \ <AUX;\tau>)$ (RF2)

the set of all terminal strings of the nonterminal $<AUX;\sigma_0>$ is a subset of the set of all well-formed expressions of type σ_0. In the context of a CL-basis B, this special subset will be denoted by $\underline{Spdb_B(\sigma_0)}$.

In the remainder of this section, <g;h> will be a type 2 skeleton, B will be a CL-basis fit for <g,h> (see D4.2), C will be a connector index under <g;h>, and h(C) will be (M;D). From D4.2 we conclude that C is an intensional constant of type $\mathbf{fc}\lfloor M;D\rfloor$ within the CL-basis B.

For each $\psi \in Spdb_B(M)$ and each variable R of type $F(D)$, where F is the function introduced in section 6.1, we define the following translation rule:

$$V'(R,(^{\vee}C \ \textbf{@} \ \psi)) = A_M(\psi)\mathbf{;}\ \mathbf{fte}(C)\mathbf{;}\ \mathbf{read}(D;R)$$

where $A_M(\psi)$ is a statement list to be defined later on. The intended post-assertion of $A_M(\psi)$ is that $\mathbf{in}(M)$ holds and that the value of the "position variable" $M\downarrow$ will be the position of the M-component described by ψ. With this post-assertion of $A_M(\psi)$, the post-assertion of the fragment "$A_M(\psi)\mathbf{;}\ \mathbf{fte}(C)$" implies that the value of $D\downarrow$ will be the position of the "C-target" of the M-component described by ψ; cf. section 5.2, ad (55). Finally, it follows from section 5.2, ad (51), that the post-assertion of $V'(R,(^{\vee}C \ \textbf{@} \ \psi))$ implies that $(R \equiv (^{\vee}C \ \textbf{@} \ \psi))$ holds, as it should be.

Another special case of interest for database applications is the case that the well-formed expression γ in clause (8), (8'), or (9) in section 4.2 is of the form $(^{\vee}C\ \mathbf{inv}\ \psi')$ where $\psi' \in Spdb_B(D)$. The following translation scheme for clause (8) also accounts for the possibility that the currency variable of M could be affected, the only currency problem that could arise in this case.

(8) $V'(R,\alpha\beta \in (^{\vee}C\ \mathbf{inv}\ \psi') \wedge \varphi\mathbf{;}\ \delta) =$

```
begin $\beta$: $F(M)$, $\beta'$: **P**, $B$: **B**, $H'$: $F(\tau')$;
    $I(\alpha,R)$; $A_D(\psi')$; ffl(C);
    while in(M)
       do read(M;$\beta$); $\beta'$ := M$\downarrow$;
          $V'(\beta,\varphi)$; if B then $V'(H',\delta)$; $O(\alpha,R,H')$ fi;
          fdp(M;$\beta'$); fnl(C)
       od
end
```

In no currency problem can arise then the statements β' **:=** M$\downarrow$ and **fdp(**M**;**β'**)** can be omitted, just as the declaration of β'. The translation schemes for the other clauses are similar.

We note that the post-assertion of $A_D(\psi')$ is that **in(**D**)** holds and that the value of D$\downarrow$ will be the position of the D-component described by ψ'. For an explanation of the statements **ffl(**C**)** and **fnl(**C**)**, we refer the reader to section 5.2, ad (56) and ad (57).

We still have to define a function A over dom(g), the set of table indices of the type 2 skeleton <g;h>, such that A_E is a function over $\mathrm{Spdb}_B(E)$ for each $E \in \mathrm{dom}(g)$. The definition will be by recursion on the rule forms (RF1) and (RF2) at the beginning of this section; the parameters τ and σ in these rule forms will vary over dom(g).

For each $\tau \in \mathrm{dom}(g)$, $\sigma \in \mathrm{dom}(g)$, $\psi \in \mathrm{Spdb}_B(\tau)$, and $X \in \mathrm{dom}(h)$ such that $h(X) = $ **fc**$\lfloor\tau$**;**$\sigma\rfloor$ we define:

(RF2) $\qquad A_\sigma(\langle^{\vee}X @ \psi\rangle) = A_\tau(\psi)$; **fte(**X**)**

For $\sigma \in \mathrm{dom}(g)$ and $\beta \in \mathrm{Plh}_B(\sigma)$, $A_\sigma(\beta)$ can be defined as

(RF1) $\qquad A_\sigma(\beta) = $ **fdc(**σ**;**a**:**β.a**)**

for any key a for σ. (We note that we used (48), (52), and (54) of section 5.1.) Depending on the context in which $A_\sigma(\beta)$ is used, other choices might be better. For instance, if it is used in a context in which there is a position variable β' of which the value happens to be the position of the σ-component described by β then the choice

(RF1) $\qquad A_\sigma(\beta) = $ **fdp(**σ**;**β'**)**

would do. For example, in the translation schemes for clause (8) given in this section and at the end of the previous section, $V'(B,\varphi)$ and

$V'(H',\delta)$ occur in such a context. In a context in which the value of $\sigma\downarrow$ is already the position of the σ-component described by β - a situation that occurs frequently in practice - $A_\sigma(\beta)$ can even be omitted.

As an illustration of the fore-mentioned translation rules, the two closed queries presented at the end of section 4.3 will be translated. These queries are repeated below. We recall that both queries express the question whether or not there is an employee who earns more than the manager of his department.

(a) **∃e∈ᵛEMPL: ((e.SAL) > ((ᵛMANAGEROF @ (ᵛDEPOF @ e)).SAL))?**

(b) **∃d∈ᵛDEP: [n ← ((ᵛMANAGEROF @ d).SAL)] ∃e∈(ᵛDEPOF inv d):**
((e.SAL) > n)?

In the following translation of (a), **(ᵛDEPOF @ e)** is abbreviated by $\psi 1$, **(ᵛMANAGEROF @** $\psi 1$**)** by $\beta 1$, **(**$\beta 1$**.SAL)** by $\gamma 1$, **(e.SAL)** by $\beta 2$, **(**$\beta 2$ **>** $\gamma 1$**)** by $\delta 1$, and **∃e∈ᵛEMPL:** $\delta 1$ by $\alpha 1$.

```
(a)  Z'(α1?) = begin X0: B; V'(X0,α1); return(X0) end.
     where V'(X0,α1) = V'(X0,∃e∈ᵛEMPL: δ1) (8')=

     begin e: F(EMPL), pe: P;
           I(∃,X0); fff(EMPL);
           while (in(EMPL) and not X0)
              do read(EMPL;e); pe := EMPL ↓;
                 V'(X0,δ1);
                 fdp(EMPL;pe); fnf(EMPL)
              od
     end
```

since **ᵛEMPL** is a 136-expression and the use of W(**ᵛEMPL**) will introduce a currency problem (see below). Furthermore, we applied improvement (c) mentioned in section 6.5 for clause (8').

F(**EMPL**) is written out in section 6.4. By section 6.2,

I(**∃,X0**) = **X0 := false**

Next,

$\mathcal{V}'(\mathbf{X0},\beta 1) = \mathcal{V}'(\mathbf{X0},(\beta 2 \mathbf{>} \gamma 1)) \overset{(5)}{=}$

$\mathbf{begin\ X1: \mathbb{Z};}\ \mathcal{V}'(\mathbf{X1},\gamma 1)\mathbf{;}\ S_2(\mathbf{>},\mathbf{X0},\mathcal{W}(\beta 2),\mathbf{X1})\ \mathbf{end}$

by the (a)-version of case (C10) for clause (5), see section 6.5. By section 6.3, ad (5),

$S_2(\mathbf{>},\mathbf{X0},\mathcal{W}(\beta 2),\mathbf{X1}) = \mathbf{X0 := (}\mathcal{W}(\beta 2)\ \mathbf{> X1)}$

$= \mathbf{X0 := (e.SAL > X1)}$

Furthermore,

$\mathcal{V}'(\mathbf{X1},\gamma 1) = \mathcal{V}'(\mathbf{X1},(\beta 1\mathbf{.SAL})) \overset{(6)}{=}$

$\mathbf{begin\ X2:}\ F(\mathbf{EMPL})\mathbf{;}\ \mathcal{V}'(\mathbf{X2},\beta 1)\mathbf{;\ X1 := X2.SAL\ end}$

where

$\mathcal{V}'(\mathbf{X2},\beta 1) = \mathcal{V}'(\mathbf{X2},(^{\mathbf{Y}}\mathbf{MANAGEROF\ @}\ \psi 1)) =$

$A_{\mathbf{DEP}}(\psi 1)\mathbf{;\ fte(MANAGEROF);\ read(EMPL;X2)}$

according to the first translation rule mentioned in this section. By the translation rule for (RF2),

$A_{\mathbf{DEP}}(\psi 1) = A_{\mathbf{DEP}}((^{\mathbf{Y}}\mathbf{DEPOF@}\,\mathbf{e})) = A_{\mathbf{EMPL}}(\mathbf{e})\mathbf{;\ fte(DEPOF)}$

and by the first alternative for (RF1),

$A_{\mathbf{EMPL}}(\mathbf{e}) = \mathbf{fdc(EMPL;NR:\ e.NR)}$

The second alternative for (RF1) can be applied as well, namely, by using the position variable **pe**. In fact, the third (and most desirable) alternative - omitting $A_{\mathbf{EMPL}}(\mathbf{e})$ in the translation result of $A_{\mathbf{DEP}}(\psi 1)$ - can also be applied here.

If we choose the "context-free alternative", i.e., the first one, then the final translation result is as follows (except for writing out $F(\mathbf{EMPL})$):

$Z'(\mathbf{\exists e\in {}^{Y}EMPL:\ ((e.SAL) > ((^{Y}MANAGEROF\ @\ (^{Y}DEPOF\ @\ e)).SAL))?}) =$

```
begin X0: 𝔹;
   begin e: F(EMPL); pe.ℙ;
         X0 := false; fff(EMPL);
         while (in(EMPL) and not X0)
            do read (EMPL;e); pe := EMPL ↓;
               begin X1: ℤ;
                  begin X2: F(EMPL);
                        fdc(EMPL;NR: e.NR);
                        fte(DEPOF);
                        fte(MANAGEROF);
                        read (EMPL;X2);
                        X1 := X2.SAL
                  end;
                  X0 := (e.SAL > X1)
               end;
               fdp(EMPL;pe); fnf(EMPL)
            od
   end;
   return(X0)
end.
```

In the following translation of (b) - the other closed query - ((e.SAL) > n) is abbreviated by δ3, ∃e∈(ᵛDEPOF inv d): δ3 by γ3, (ᵛMANAGEROF @ d) by β4, (β4.SAL) by β3, [n ← β3]γ3 by δ2, and ∃d∈ᵛDEP: δ2 by α2.

(b) $Z'(\alpha 2?)$ = begin X0: 𝔹; V'(X0,α2); return(X0) end.

where

V'(X0,α2) = V'(X0,∃d∈ᵛDEP: δ2) $\overset{(\mathrm{B}')}{=}$

```
begin d: F(DEP);
      I(∃,X0); fff(DEP);
      while (in(DEP) and not X0)
         do read (DEP;d);
            V'(X0,δ2);
            fnf(DEP)
         od
end
```

since $^{\vee}$**DEP** is a 136-expression and the use of W($^{\vee}$**DEP**) will not introduce a currency problem (see below). We also applied improvement (c) mentioned in section 6.5 for clause (8').

F(**DEP**) is written out in section 6.1. We recall that

$I(\exists,$**X0**$)$ = **X0 := false**

Furthermore,

V'(**X0**,$\delta 2$) = V'(**X0**,**[n** ← $\beta 3$**]**$\gamma 3$) $\overset{(7)}{=}$

begin n: ℤ; V'(**n**,$\beta 3$)**;** V'(**X0**,$\gamma 3$) **end**

First, we elaborate V'(**n**,$\beta 3$):

V'(**n**,$\beta 3$) = V'(**n**,($\beta 4$**.SAL**)) $\overset{(6)}{=}$

begin X1: F(**EMPL**)**;** V'(**X1**,$\beta 4$)**; n := X1.SAL end**

since **(**$\beta 4$**.SAL)** is not simple; cf. section 6.5.

V'(**X1**,$\beta 4$) = V'(**X1**,($^{\vee}$**MANAGEROF @ d**)) =

$A_{\mathbf{DEP}}$(**d**)**; fte(MANAGEROF); read(EMPL;X1)**

and

$A_{\mathbf{DEP}}$(**d**) = **fdc(DEP;DNR: d.DNR)**

by the first alternative for (RF1). The second alternative for (RF1) does not apply here but, fortunately, the third one does (since the variable **DEP** ↓ still "points to" the current value of the variable **d**).

Next, we elaborate V'(**X0**,$\gamma 3$):

```
V'(X0,γ3) = V'(X0,∃e∈(ᵛDEPOF inv d): δ3) =
begin e: F(EMPL);
      I(∃,X0); A_DEP(d); ffl(DEPOF);
      while (in(EMPL) and not X0)
         do read (EMPL;e);
            V'(X0,δ3);
            fnl(DEPOF)
         od
end
```

by the (8')-variant of the translation scheme for **inv** mentioned in this section, improvement (c) mentioned in section 6.5, and the fact that V'(**X0**,$\delta 3$) will not introduce a currency problem.

Finally, by case (C11) for clause (5) and the definitions of S_2 and W,

$$V'(XO,\delta 3) = V'(XO,((e.SAL) > n))$$
$$\overset{(5)}{=} S_2(>,XO,W((e.SAL)),W(n))$$
$$= XO := (e.SAL > n)$$

The resulting program is as follows:

```
Z'(∃d∈ᵛDEP: [n ← ((ᵛMANAGEROF @ d).SAL)]
    ∃e∈(ᵛDEPOF inv d): ((e.SAL) > n)?) =

begin XO: 𝔹;
   begin d: F(DEP);
         XO := false; fff(DEP);
         while (in(DEP) and not XO)
            do read (DEP;d);
               begin n: ℤ;
                  begin X1: F(EMPL);
                        fdc(DEP;DNR: d.DNR);
                        fte(MANAGEROF);
                        read (EMPL;X1);
                        n := X1.SAL
                  end;
                  begin e: F(EMPL);
                        XO := false;
                        fdc(DEP;DNR: d.DNR);
                        ffl(DEPOF);
                        while (in(EMPL) and not XO)
                           do read (EMPL;e);
                              XO := (e.SAL > n);
                              fnl(DEPOF)
                           od
                  end
               end;
               fnf(DEP)
            od
   end;
   return(XO)
end.
```

7. THE STRUCTURE OF QUERIES IN ENGLISH

7.0. Introduction and summary

In this chapter we present a set of general production rules reflecting the structure of certain fragments of English that are useful for formulating requests. Translations to our conceptual languages will be given in chapter 8. These translations are independent of any particular application. Per considered application, the set of general rules must be extended with production rules that introduce the words and phrases that are characteristic for that application. The appendix contains an example of such an extension (for a hospital database).

In section 7.1 the general rule forms are presented and in section 7.2 some comments on these rule forms are given. Of course, these rule forms only account for *some* of the syntactic structures that are possible in English. Therefore, in practice, the general framework presented in section 7.1 might have to be extended with other general rule forms.

In section 7.3 attention is paid to an intermediate sort of rules, namely, general syntactic rules that have application-dependent (or, rather, representation-dependent) translations.

7.1. The general syntax

The grammar presented in this section will contain *rule forms* (i.e., production rules containing "parameters"), just like the grammars presented at the end of section 4.2 and in section 5.1. We use two sorts of parameters, which are shortly explained below.

As in the case of the languages in chapter 4 and chapter 5, our fragments are *typed* (or *attributed* or *affixed*). Typing can be used to exclude meaningless (or unwanted) combinations, such as **female departments** or **the planning department works for employee 12.** In our grammar, the parameters τ, τ', σ, and σ' will vary over the set of types relevant to the (database) application concerned, just as in chapter 4.

The parameters μ and μ' indicate the feature of *number* (of noun phrases and verb phrases for instance). They will vary over the (application-*in*dependent) set {si,pl}, where si stands for *singular* and pl for *plural*. The feature of *person* can be ignored because we only use the third person (which is sufficient in information retrieval). The feature of *gender* will also be ignored; unlike number, the feature of gender would hardly have any disambiguating effect in our fragments.

In this chapter, no rules are given for nonterminals of the form <BN;μ;σ>, <DJ;σ>, <PJ;σ>, <ET;μ;σ;σ';τ>, <AV;σ>, <FU;σ;σ'>, <AR;σ;σ'>, <PE;σ>, or <CE;σ>. Production rules for these nonterminals should be added per application. (We note that application-dependent production rules for the *other* nonterminals might have to be added as well.) Examples of such production rules can be found in the appendix..

The start symbol of the grammar presented below is <RQ> (for request). Further explanation will follow in section 7.2.

For each nonterminal used in the general syntax, the *occurrence table* on page 84 indicates in which rule forms that nonterminal occurs on the left hand side and in which rule forms it occurs on the right hand side.

```
1    <RQ>            ::= Give <EX;τ>.
2                       |Is it true that <ST> ?
3    <ST>            ::= <NP;μ;τ><VP;μ;τ>
4    <NP;μ;σ>        ::= <DT;μ><CN;μ;σ>
5    <NP;si;σ>       ::= <PN;σ>
6    <CN;μ;σ>        ::= <SN;μ;σ>
7                       |<SN;μ;σ><RC;μ;σ>
8    <SN;μ;σ>        ::= <BN;μ;σ>
9                       |<DJ;σ><SN;μ;σ>
10                      |<SN;μ;σ><PJ;σ>
11   <RC;μ;σ>        ::= <SC;μ;σ>
12                      |<CC;μ;σ>
13   <CC;μ;σ>        ::= <SC;μ;σ><A><SC;μ;σ>
14                      |<SC;μ;σ>,<CC;μ;σ>
15   <SC;μ;σ>        ::= <RP><VP;μ;σ>
16                      |whose <FN;σ;τ><VP;si;τ>
17   <VP;μ;σ>        ::= <IV;μ;σ>
18                      |<CI;μ;σ>
19   <CI;μ;σ>        ::= <IV;μ;σ><A><IV;μ;σ>
20                      |<IV;μ;σ>,<CI;μ;σ>
21   <IV;μ;σ>        ::= <EI;μ;σ;τ>[<AL;τ>][<AD>]
22                      |<ET;μ;σ;τ';τ><NP;μ';τ'>[<AL;τ>][<AD>]
23                      |<C;μ><DJ;σ>
24   <IV;si;σ>       ::= <C;si><QN;σ>
25   <IV;pl;σ>       ::= <C;pl><CN;pl;σ>
26   <EI;μ;σ;σ'>     ::= <ET;μ;σ;σ;σ'><RF;μ>
27   <AL;σ>          ::= <AV;σ>
28                      |<AV;σ><AL;σ>
29   <AD>            ::= <ND;pl> times
30                      |<CO> once

31   <EX;σ>          ::= <SE;σ>
32                      |<CF;τ;σ><QS;τ>
33   <EX;so[σ]>      ::= <CF;τ;σ><DD;μ><CN;μ;τ>
34   <CF;σ;σ'>       ::= <FE;σ;σ'>
35                      |<FL;σ;σ'>
36                      |<CF;τ;σ'><FE;σ;τ>
37   <FE;σ;σ'>       ::= the  <FN;σ;σ'><PPR>
```

```
38   <FL;σ;p⌊τ;τ'⌋>  ::= <FN;σ;τ><A><FN;σ;τ'><PPR>
39                       |<FN;σ;τ>,<FL;σ;τ'>
40   <FN;σ;σ'>       ::= <FU;σ;σ'>
41                       |<AR;σ;σ'>
42   <QS;σ>          ::= <SE;σ>
43                       |<PE;σ>
44   <SE;σ>          ::= <CE;σ>
45                       |the <GD;τ';σ><FN;τ;τ'> of all <CN;pl;τ>
46   <SE;int>        ::= the number of <CN;pl;τ>

47   <PN;σ>          ::= <QS;σ>
48                       |<FE;τ;σ><PN;τ>
49   <QN;σ>          ::= <IA><CN;si;σ>

50   <DT;μ>          ::= <BD;μ>
51                       |<RD;μ>
52   <BD;μ>          ::= <ND;μ>
53                       |<OD;μ>
54   <BD;si>         ::= <IA>
55   <ND;μ>          ::= <CO><CA;μ>
56                       |<CA;μ>

57   <CA;si>         ::= one
58   <CA;pl>         ::= zero
59                       |two
60                       |three
61   <CO>            ::= at least
62                       |at most
63                       |exactly
64                       |more than
65                       |less than
66   <GD;int;int>    ::= total
67                       |minimal
68                       |maximal
69   <GD;int;rat>    ::= average
70   <OD;μ>          ::= some
71                       |no

72   <RD;si>   ::= every|each
73   <RD;pl>   ::= all
74   <IA>      ::= a|an
75   <DD;si>   ::= every|each
76   <DD;pl>   ::= all|the
77   <A>       ::= and|,and
78   <C;si>    ::= is
79   <C;pl>    ::= are
80   <PPR>     ::= of|for|in
81   <RP>      ::= who|that
82   <RF;si>   ::= himself
83                 |herself
84                 |itself
85   <RF;pl>   ::= themselves
```

a nonterminal of the form	occurs on the *left* hand side in	occurs on the *right* hand side in
<A>	77	13,19,38
<AD>	29,30	21,22
<AL;σ>	27,28	21,22,28
<AR;σ;σ'>		41
<AV;σ>		27,28
<BD;μ>	52,53,54	50
<BN;μ;σ>		8
<C;μ>	78,79	23,24,25
<CA;μ>	57,58,59,60	55,56
<CC;μ;σ>	13,14	12,14
<CE;σ>		44
<CF;σ;σ'>	34,35,36	32,33,36
<CI;μ;σ>	19,29	18,20
<CN;μ;σ>	6,7	4,25,33,45,46,49
<CO>	61,62,63,64,65	30,55
<DD;μ>	75,76	33
<DJ;σ>		9,23
<DT;μ>	50,51	4
<EI;μ;σ;σ'>	26	21
<ET;μ;σ;σ';σ">		22,26
<EX;σ>	31,32,33	1
<FC;σ;σ'>	37	34,36,48
<FL;σ;σ'>	38,39	35,39
<FN;σ;σ'>	40,41	16,37,38,38,39,45
<FU;σ;σ'>		40
<GD;σ;σ'>	66,67,68,69	45
<IA>	74	49,54
<IV;μ;σ>	21,22,23,24,25	17,19,19,20
<ND;μ>	55,56	29,52
<NP;μ;σ>	4,5	3,22
<OD;μ>	70,71	53
<PK;σ>		43
<PJ;σ>		10
<PN;σ>	47,48	5,48
<PPR>	80	37,38
<QN;σ>	49	24
<QS;σ>	42,43	32,47
<RC;μ;σ>	11,12	7
<RD;μ>	72,73	51
<RF;μ>	82,83,84,85	26
<RP>	81	15
<RQ>	1,2	
<SC;μ;σ>	15,16	11,12,13,14
<SE;σ>	44,45,46	31,42
<SN;μ;σ>	8,9,10	6,7,9,10
<ST>	3	2
<VP;μ;σ>	17,18	3,15,16

7.2. Commentary on the syntax

In this section, some explanation and some related (linguistic) terminology is given.

ad {(1),(2)}: A *request* is either an expression of arbitrary type preceded by the imperative **Give** and followed by a period, or a sentence preceded by the text **Is it true that** and followed by a question mark.

We give two examples of requests (concerning our employees-and-departments database):

Give name and number of each department.
Is it true that no employee works for department 12 ?

The second alternative for <RQ> is our standard form for yes/no questions and the first one is our counterpart to so-called wh-questions (i.e., questions starting with **When**, **Where**, **Which**, **Who**, **Why**, etc.). Our standard form for yes/no questions avoids the well-known confusion concerning short yes/no answers to negative questions. (Barbara Partee: "Didn't you make your homework yet?" Her son: "Yes.") Our alternative for wh-questions avoids certain scope ambiguities which may arise in the case of genuine wh-questions. (For a discussion of this problem, we refer the reader to, e.g., [Sc 83], Ch.II, section 6.)

ad (3): A *sentence* is a noun phrase followed by a verb phrase of the same number and type.

ad {(4),(5)}: A *noun phrase* is either a common noun phrase of the same number and type preceded by a determiner of the same number, or a proper noun (phrase) of the same type. In the latter case, the noun phrase is singular.

Examples of noun phrases are **every employee**, **exactly three departments** and **the planning department**. The latter is a proper noun phrase.

ad {(6),(7)}: A *common noun phrase* is a simple noun phrase possibly followed by a restrictive relative clause, all of the same number and type.

ad {(8),(9),(10)}: A *simple noun phrase* is a basic noun of the same number and type preceded by zero or more adherent adjectival phrases of the same type, and followed by zero or more appositive adjectival phrases, also of the same type. The production rules for basic nouns as well as for adherent and appositive adjectival phrases should be added *per* application. Various examples of such production rules are given in the appendix.

We note that the present rule forms give rise to *syntactic* ambiguities. For instance, **female employee from Amsterdam** can be analyzed as ((**female employee**) **from Amsterdam**) but also as (**female** (**employee from Amsterdam**)). A disambiguating alternative for these three rule forms would be

```
<SN;μ;σ>   ::= <VSN;μ;σ>
              |<SN;μ;σ><PJ;σ>

<VSN;μ;σ>  ::= <BN;μ;σ>
              |<DJ;σ><VSN;μ;σ>
```

ad {(11),(12),(13),(14),(77)}: A *restrictive relative clause* is either a simple clause or a compound clause. A *compound clause* consists of two or more simple clauses; the last two of these simple clauses are separated by **and** or **, and** and the others by a comma. All clauses have the same number and type. (Strictly speaking, too many combinations are allowed: **and** should be used iff there are *two* simple clauses, and **, and** should be used iff there are three or more simple clauses. We did not exclude the other combinations, however.)

An example of a (singular) restrictive relative clause containing two simple clauses is

who earns at least 3650 guilders and whose department has more than 100 employees

ad {(15),(16),(81)}: In the present fragment, a *simple clause* can be a verb phrase of the same number and type preceded by an appropriate relative pronoun - this is the only place where (81) is used - but it can also be a singular verb phrase preceded by the relative pronoun **whose** and a functional noun whose so-called *operand* type (denoted by σ in rule form (16)) is equal to the type of the simple clause and whose *result* type (τ in rule form (16)) is equal to the type of the verb phrase. Each of the two alternatives is used in the construction of the restrictive relative clause mentioned above.

It is likely that general rule forms for other cases of simple clauses are also needed in practice.

ad {(17),(18),(19),(20),(77)}: Here a similar construction as in {(11),(12),(13),(14)} is used: "RC" is replaced by "VP", "SC" by "IV", and "CC" by "CI". We recall that "VP" stands for *verb phrase*; furthermore, "IV" stands for intransitive verb phrase and "CI" for compound intransitive verb phrase. It is left to the reader to give verbal descriptions of these rule forms.

ad {(21),(22),(23),(24),(25)}: In the present fragment we have five general rule forms for *intransitive verb phrases*, three for arbitrary number, one for singular number only, and one for plural number only.

ad {(21),(22)}: We used square brackets to indicate that the part within the brackets is optional. Thus, together, (21) and (22) stand for 8 rule forms. We could use this shorthand in other places too (for instance, in (6) + (7)). In chapter 8, however, each rule form has to be translated separately.

An intransitive verb phrase (i.v.p.) can be an existential intransitive verb phrase of the same number but it can also be an existential transitive verb phrase of the same number followed by a noun phrase of arbitrary number; in each case, an adverbial list may follow and, independently, an adverbial of degree may come at the end.

The type of the i.v.p. is equal to the *subject type* of the existential (in)transitive verb phrase (σ in these rule forms), and the type of the adverbial list is equal to the *underlying type* of the existential (in)transitive verb phrase (τ in both rule forms). In (22), the type of the noun phrase is equal to the *object type* of the existential transitive verb phrase.

Loosely speaking, an *existential* (in)transitive verb phrase is an (in)transitive verb phrase for which an adverbial of degree - see (29) and (30) - makes sense. Further explanation of this "semantics-generated" notion will follow in chapter 8. (This explanation will be in terms of connector indices and table indices.)

Production rules for existential transitive verb phrases should be added per application. Some examples can be found in the appendix.

ad {(23),(24),(25),(78),(79)}: These three rule forms for intransitive verb phrases are simple but useful. They say that an intransitive verb phrase can be an adherent adjectival phrase of the same type, a quasi-proper noun phrase of the same type, or a plural common noun phrase of the same type, in each case preceded by the appropriate copula.

Other production rules for intransitive verb phrases might be added per application.

ad {(26),(82),(83),(84),(85)}: Strictly speaking, production rules for existential intransitive verb phrases should also be added application. Nevertheless, in order to illustrate the translation of reflexive pronouns (in chapter 8), the following special (but application-independent) rule form for existential i.v.p.'s is inserted: An existential intransitive verb phrase can be a *reflexive* existential transitive verb phrase (that is, an e.t.v.p. whose *object* type is equal to its *subject* type) followed by a reflexive pronoun. (In other words, here the reflexive e.t.v.p. is also *used* reflexively.)

ad {(27),(28)}: An *adverbial list* consists of one or more adverbial phrases, all of the same type. Production rules for adverbial phrases should be added *per* application.

ad {(29),(30)}: An *adverbial of degree* is either a (plural) numeral determiner followed by the word **times** or a comparative followed by the word **once**. In (61)-(65), some comparatives are listed.

ad {(31),(32),(33),(75),(76)}: An *expression* can be a simple expression of the same type or a quasi-simple expression preceded by a composite functional expression whose *operand* type is equal to the type of the quasi-simple expression and whose *result* type is equal to the type of the expression.

For "set-valued" expressions there is another possibility as well, namely the possibility that they consist of a common noun phrase preceded by a definite determiner of the same number and a composite functional expression whose operand type is equal to the type of the common noun phrase and whose result type is equal to the "base type" of the (set) type of the expression.

An example using rule form (33) is the request

Give name, number, and department number of the manager of each department.

concerning our ubiquitous employees-and-departments database. By (75), **each** is the definite determiner. This typical example of a database request also accounts for the next few rule forms.

ad {(34),(35),(36)}: A *composite functional expression* is a functional expression or a functional noun list, followed by zero or more functional expressions; the *operand* type of the composite functional expression is equal to the operand type of the last "functional" in the series and its *result* type is equal to the result type of the first "functional" in the series. Furthermore, within this series of "functionals", the result type of each functional expression with a predecessor - see τ in (36) - must be equal to the operand type of the preceding "functional".

The sample request mentioned above contains one functional noun list, namely **name, number, and department number of**, and one functional expression, namely **the manager of**.

ad {(37),(80)}: A *functional expression* is a functional noun with the same operand type and with the same result type, preceded by the definite article and followed by a possessive preposition.

ad {(38),(39)}: A *functional noun list* consists of two or more functional nouns, the last one followed by a possessive preposition, the last two separated by **and** or **, and**, and the others separated by a comma. All functional nouns have the same operand type as the functional noun list has. The result type of the functional noun list is the "pairing type" of the successive result types of its immediate constituents. The sample functional noun list above consists of three functional nouns.

We note that a similar construction as in {(13),(14)} is used and that, similarly, too many combinations are allowed.

ad {(35),(39)}: Although these rule forms are formulated for arbitrary result types of the functional noun list on the right hand side, we conclude from {(38),(39)} that we only encounter functional noun lists of which the result type is a "pairing type", i.e., a type of the form $\underline{p}\lfloor\tau;\tau'\rfloor$.

ad {(40),(41)}: Depending on the underlying model, a *functional noun* acts either as an argument noun or as a function noun. As a consequence, argument nouns and function nouns have to be incorporated *per* application.

In our employees-and-departments model, **name**, **number**, and **department number** are argument nouns and **manager** is a function noun.

ad {(42),(43)}: A *quasi-simple expression* is either a simple expression of the same type or a presuppositional expression of the same type.

Typical examples of presuppositional expressions are constructions such as **employee 13** and **the linguistics department.** The characteristic feature of a presuppositional expression is that it depends on the "actual" DB snapshot whether it has a "denotation" or not.

Production rules for presuppositional expressions should be added *per* application.

ad {(44),(45),(46)}: In the present fragment we have three rule forms for *simple expressions*, two for simple expressions of arbitrary type and one more for "integer-valued" simple expressions. The three rule forms are discussed below.

ad (44): A simple expression can be a constant expression of the same type. The actual choice of constant expressions is delegated to the application-dependent part of the grammar. (The characteristic feature of a constant expression is that its "denotation" does not depend on the "actual" DB snapshot.)

ad (45): A simple expression can also be a functional noun preceded by the definite article and an aggregate determiner, and followed by the text **of all** and a plural common noun phrase. The type of the simple expression is equal to the *result* type of the aggregate determiner, the result type of the functional noun is equal to the *range* type of the aggregate determiner, and the operand type of the functional noun is equal to the type of the common noun phrase. (The last mentioned type could be called the *domain* type of the construction on hand, in line with the terminology concerning determiners that was introduced after D4.1 in section 4.1.) In (66)-(69), some aggregate determiners are listed.

An example of a simple expression of this form is **the total salary of all employees**, cf. production rule (66).

ad (46): We can obtain an "integer-valued" simple expression also by placing the text **the number of** before any plural common noun phrase.

ad {(47),(48)}: A *proper noun phrase* is a quasi-simple expression preceded by zero or more functional expressions.

ad {(49),(74)}: A *quasi-proper noun phrase* is a singular common noun phrase of the same type preceded by an indefinite article. We note that it is not dictated by the grammar that the common noun phrase is preceded by the *right* indefinite article.

ad {(50),(51),(52),(53),(54)}: For three reasons we had to distinguish various subsets of the set of all determiners that are allowed to be used in rule form (4). One reason for this differentiation comes from production rule (29), another one from rule form (49). Furthermore, we have to distinguish those determiners δ that can be used in popular (application-dependent) appositive adjectival phrases such as

with δ **employee(s)**

of type DEP in our employees-and-departments application (see, e.g., figure 1.2). Such a determiner δ will be called an *absolute determiner*. (The other determiners are called relative determiners.) Each numeral determiner and each indefinite article is an example of an absolute determiner, which is reflected in (52) and (54). Examples of determiners can be found in (70)-(74).

ad {(55),(56)}: A *numeral determiner* is a cardinal of the same number, possibly preceded by a comparative.

Some examples of cardinals are given in (57), (58), (59), and (60) but it is, of course, inevitable to add, say, the grammar presented in example 3.1, together with the rule form <CA;μ> ::= <int.> or <CA;μ> ::= <pos.int.>.

ad (57) and following: These are *lexical* rules, i.e., rules without a nonterminal on the right hand side. There are 34 production rules and 2 rule forms, namely (70) and (71). Each of these rule forms has 2 instances.

7.3. Intermediate forms

In section 7.1 we gave examples of application-independent rule forms of which the translation will also be application-independent (see chapter 8). In the appendix, we will give examples of application-dependent production rules of which the translation will of course also be application-dependent. However, there are also many application-independent production rules of which the translation is in fact application-dependent or, rather, *representation*-dependent, such as the rules for adjectival and adverbial phrases of time. For instance, in some applications **March 16, 1984** should be translated to **840316**, in others to **19840316**, etc. Moreover, several "semi-general" phrases are irrelevant to certain applications. For instance, phrases dealing with hours and minutes (such as **16:00 p.m.**) are irrelevant to information systems that do not keep track of such data. For the above-mentioned reasons, these "semi-general" phrases were not included in the general syntax presented in section 7.1. As an example, we shall present a cluster of production rules for such phrases. We chose for the "JV-phrases" of time, i.e., phrases that can be used both as adjectival and as adverbial phrases of time.

In the rules below, "PT" stand for preposition of time and date is a type. As a consequence, the rule form (44) in section 7.1 provides for the production rule <SE;date> ::= <CE;date> which is needed to bridge the gap between the rules SG01, SG02, SG03 and the rules SG07, SG08. The 10 production rules for <digit> are not specified. (We note that a "day" such as **32** is not ruled out by this grammar.)

For a concrete application of "JV-phrases" we refer the reader to the appendix.

```
SG01  <JV;date>  ::= <PT><SE;date>
SG02                 |between <SE;date> and <SE;date>
SG03                 |in the period <SE;date> to <SE;date>
SG04                 |in the period <month><day> to <day>,<year>
SG05                 |in <month><year>
SG06                 |in <year>
SG07  <CE;date>  ::= <month><day>,<year>
SG08                 |<D2><D2><D2>
SG09  <year>     ::= <D2><D2>
SG10  <day>      ::= <D2>
SG11                 |<digit>
SG12  <D2>       ::= <digit><digit>

SG13  <PT>       ::= on
SG14                 |before
SG15                 |after
SG16  <month>    ::= January
SG17                 |February
SG18                 |March
SG19                 |April
SG20                 |May
SG21                 |June
SG22                 |July
SG23                 |August
SG24                 |September
SG25                 |October
SG26                 |November
SG27                 |December
```

8. TRANSLATING FRAGMENTS OF ENGLISH INTO CONCEPTUAL LANGUAGES

8.0. Introduction and summary

In this chapter, we give translation rules for the grammar rules presented in chapter 7. In each particular application, the target language will be some conceptual language of chapter 4.

In particular, we want to assign a *closed query* (see D4.7) to each *disambiguated request*, i.e., to each derivation tree with root label <RQ>. Thus, we want to map derivation trees to "strings". For a suitable parsing algorithm, i.e., an algorithm that constructs the derivation tree(s) for a given "input string", we refer the reader to [Ea 70].

The result of a translation of a derivation tree has to satisfy certain minimal conditions which (only) depend on the root label of that derivation tree. These conditions are presented in section 8.1. It can be proved that the translation rules given in section 8.2 satisfy and preserve these conditions. Furthermore, per application, these conditions can serve as a guideline for defining the translation of the application-dependent phrases.

8.1. On the form of the translation

Let G be a quasi-cfg containing the nonterminals presented in chapter 7 and describing some fragment of English. A translation function M from the set of derivation trees of G into the conceptual language based on a CL-basis B has to satisfy at least the following conditions:

If α is a derivation tree based on G (see D3.5) and the root label of α is:

- <RQ> then $M(\alpha) \in \mathrm{Clq}_B$, i.e., the translation of α is a closed query (see D4.7);

- <ST> then $\mathcal{M}(\alpha) \in \mathrm{Cle}_B(\mathbf{t})$, i.e., the translation of α is a closed expression of type $\mathbf{t}$.

If the root label of α is of the form:

- <EX;σ> or <SE;σ> then $\mathcal{M}(\alpha) \in \mathrm{Cle}_B(\sigma)$;
- <FU;σ;σ'> then $\mathcal{M}(\alpha) \in \mathrm{Cle}_B(\mathbf{fc}\lfloor\sigma;\sigma'\rfloor)$;
- <AR;σ;σ'> then $\mathcal{M}(\alpha) \in \mathrm{Arg}_B(\sigma,\sigma')$;
- <CE;σ> then $\mathcal{M}(\alpha) \in \mathrm{Con}_B(\sigma)$.

In order to formulate further conditions, we introduce some auxiliary definitions:

$\underline{\mathrm{Ro}} \overset{D}{=} \{=,<,\geq,\leq,>\}$;

$\underline{\mathrm{Dro}_B} \overset{D}{=} \{(\exists\vartheta\eta) \mid \vartheta \in \mathrm{Ro} \text{ and } \eta \in \mathrm{Con}_B(\mathbf{int})\}$.

We require of B that $\mathrm{Ro} \subseteq \mathrm{Binop}_B(\mathbf{int},\mathbf{int},\mathbf{t})$ and $\mathrm{Dro}_B \subseteq \mathrm{Det}_B(\tau,\mathbf{t},\mathbf{t})$ for each $\tau \in \mathrm{Typ}_B$.

Further conditions on the translation of a derivation tree α are:

If the root label of α is of the form:

- <GD;σ;σ'> then $\mathcal{M}(\alpha) \in \mathrm{Det}_B(\tau,\sigma,\sigma')$;
- <OD;μ> then $\mathcal{M}(\alpha) \in \mathrm{Det}_B(\tau,\mathbf{t},\mathbf{t})$;
- <CA;μ> then $\mathcal{M}(\alpha) \in \mathrm{Con}_B(\mathbf{int})$;
- <CO> then $\mathcal{M}(\alpha) \in \mathrm{Ro}$;
- <ND;μ> then $\mathcal{M}(\alpha) \in \mathrm{Dro}_B$;
- <AD> then $\mathcal{M}(\alpha) \in \mathrm{Dro}_B$;
- <BD;μ> then $\mathcal{M}(\alpha) \in \mathrm{Det}_B(\tau,\mathbf{t},\mathbf{t})$;
- <DT;μ> then $\mathcal{M}(\alpha) \in \mathrm{Det}_B(\tau,\mathbf{t},\mathbf{t})$.

Some derivation trees must be translated in combination with a placeholder (of the appropriate type). Here we have a similar situation as in chapter 6, where a well-formed expression of the conceptual language, i.e., the "source" language, had to be translated in combination with a variable of the programming language, i.e., the "target" language.

If the root label of α is of the form

<CF;σ;σ'>, <FE;σ;σ'>, <FL;σ;σ'>, or <FN;σ;σ'>

then α must be translated in combination with a placeholder of type σ: If $x \in \mathrm{Plh}_B(\sigma)$ then we require that $M(x,\alpha) \in \mathrm{We}_B(\sigma')$ and $\mathrm{FP}_B(M(x,\alpha)) \subseteq \{x\}$, i.e., $M(x,\alpha)$ is a well-formed expression of type σ' in which only the placeholder x might be free.

If the root label of α is of the form

<AL;σ>, <AV;σ>, <CC;μ;σ>, <CI;μ;σ>, <DJ;σ>, <IV;μ;σ>, <PJ;σ>, <QN;σ>, <RC;μ;σ>, <SC;μ;σ>, or <VP;μ;σ> and $x \in \mathrm{Plh}_B(\sigma)$

then $M(x,\alpha) \in \mathrm{We}_B(\mathbf{t})$ and $\mathrm{FP}_B(M(x,\alpha)) \subseteq \{x\}$.

If the root label of α is of the form

<BN;μ;σ>, <CN;μ;σ>, or <SN;μ;σ>

then $M(x,\alpha) \in \mathrm{We}_B(\mathbf{t})$ and $\mathrm{FP}_B(M(x,\alpha)) \subseteq \{x\}$. More specifically, $M(x,\alpha)$ should be a terminal string of the auxiliary nonterminal <RE;x> having the following production rules (in combination with the grammar presented after D4.5 in section 4.2):

<RE;x> ::= $\mathbf{(}$x $\boldsymbol{\in}$ <E;$\mathbf{so}\lfloor\sigma\rfloor$>$\mathbf{)}$ (PR1)

| $\mathbf{(}$<RE;x> $\boldsymbol{\wedge}$ <E;$\mathbf{t}$>$\mathbf{)}$ (PR2)

The set of all terminal strings of the nonterminal <RE;x> will be denoted by $\mathrm{Restr}_B(x)$. Thus, $\mathrm{Restr}_B(x)$ is a subset of $\mathrm{We}_B(\mathbf{t})$ and the fore-mentioned requirement of $M(x,\alpha)$ is that $M(x,\alpha) \in \mathrm{Restr}_B(x)$. In other words, $M(x,\alpha)$ is a conjunction of well-formed expressions of type $\mathbf{t}$ such that the first conjunct is of the form $\mathbf{(}x \boldsymbol{\in} \gamma\mathbf{)}$, for some $\gamma \in \mathrm{We}_B(\mathbf{so}\lfloor\sigma\rfloor)$.

If the root label of the derivation tree α is of the form <EI;μ;σ;σ'> then α must be translated in combination with a placeholder x in $\mathrm{Plh}_B(\sigma)$ and a placeholder x' in $\mathrm{Plh}_B(\sigma')$. We stipulate that $M(x,x',\alpha) \in \mathrm{Restr}_B(x')$ and that $\mathrm{FP}_B(M(x,x',\alpha)) \subseteq \{x,x'\}$.

If the root label of α is of the form <ET;μ;σ;σ';σ''> then α must be translated in combination with three placeholders, one from $\mathrm{Plh}_B(\sigma)$, one from $\mathrm{Plh}_B(\sigma')$, and one from $\mathrm{Plh}_B(\sigma'')$; for $x \in \mathrm{Plh}_B(\sigma)$, $x' \in \mathrm{Plh}_B(\sigma')$, and $x'' \in \mathrm{Plh}_B(\sigma'')$ we stipulate that $M(x,x',x'',\alpha) \in \mathrm{Restr}_B(x'')$ and that $\mathrm{FP}_B(M(x,x',x'',\alpha)) \subseteq \{x,x',x''\}$.

We note that each $\psi \in \mathrm{Restr}_B(x)$, for any placeholder x, is not exactly of the form that is required between the determiner and the colon in the clauses (8) and (8') of chapter 4: If ψ is built up by means of (PR1) only, then the outer parentheses should be omitted. If ψ is built up by means of (PR2) then it is a conjunction of which the conjuncts are associated to the left while it would have been better if they were associated to the right. Furthermore, the outer parentheses and the parentheses of the first conjunct, which is of the form $(x \in \gamma)$, should be omitted. We will ignore this (harmless) difference from now on.

If the root label of α is of the form $\langle NP;\mu;\sigma\rangle$, $\langle PE;\sigma\rangle$, $\langle PN;\sigma\rangle$, or $\langle QS;\sigma\rangle$ then α must be translated in combination with a placeholder $x \in \mathrm{Plh}_B(\sigma)$.

If the root label of α is of the form $\langle NP;\mu;\sigma\rangle$ then we stipulate that $M(x,\alpha)$ followed by any well-formed expression of type $\mathbf{t}$ together constitute a well-formed expression of type $\mathbf{t}$ again. More precisely: $\forall e \in \mathrm{We}_B(\mathbf{t}): M(x,\alpha)\ \&\ e \in \mathrm{We}_B(\mathbf{t})$, where $M(x,\alpha)\ \&\ e$ denotes the concatenation of the sequences (of "characters") $M(x,\alpha)$ and e (cf. chapter 0). Moreover, we stipulate that $\mathrm{FP}_B(M(x,\alpha)\ \&\ e) \subseteq \mathrm{FP}_B(e) - \{x\}$.

Similarly, if the root label of α is of the form $\langle PE;\sigma\rangle$, $\langle PN;\sigma\rangle$, or $\langle QS;\sigma\rangle$ then we stipulate that $M(x,\alpha)$ followed by any well-formed expression of any type τ together constitute a well-formed expression of type τ again. More precisely: $\forall \tau \in \mathrm{Typ}_B: \forall e \in \mathrm{We}_B(\tau): M(x,\alpha)\ \&\ e \in \mathrm{We}_B(\tau)$. Furthermore, we stipulate that $\mathrm{FP}_B(M(x,\alpha)\ \&\ e) \subseteq \mathrm{FP}_B(e) - \{x\}$.

If the root label of α is of the form $\langle A\rangle$, $\langle C;\mu\rangle$, $\langle DD;\mu\rangle$, $\langle IA\rangle$, $\langle PPR\rangle$, $\langle RD;\mu\rangle$, $\langle RF;\mu\rangle$, or $\langle RP\rangle$ then no translation rule is needed for α. This can be concluded from the translation rules given for those rule forms that contain one of these nonterminals on their right hand side: In each of these translation rules, the (sub)tree in question disappears on the right hand side. For this reason, no translation rules will be given for the production rules (72)-(85) of chapter 7.

In summary, we require that if $x \in \mathrm{Plh}_B(\sigma)$, $x' \in \mathrm{Plh}_B(\sigma')$, $x'' \in \mathrm{Plh}_B(\sigma'')$, and α is a derivation tree with a root label of the form:

- $\langle AD\rangle$ then $M(\alpha) \in \mathrm{Dro}_B$;
- $\langle AR;\sigma;\sigma'\rangle$ then $M(\alpha) \in \mathrm{Arg}_B(\sigma,\sigma')$;
- $\langle BD;\mu\rangle$ then $M(\alpha) \in \mathrm{Det}_B(\tau,\mathbf{t},\mathbf{t})$;
- $\langle CA;\mu\rangle$ then $M(\alpha) \in \mathrm{Con}_B(\mathbf{int})$;
- $\langle CE;\sigma\rangle$ then $M(\alpha) \in \mathrm{Con}_B(\sigma)$;
- $\langle CO\rangle$ then $M(\alpha) \in \mathrm{Ro}$;
- $\langle DT;\mu\rangle$ then $M(\alpha) \in \mathrm{Det}_B(\tau,\mathbf{t},\mathbf{t})$;
- $\langle EX;\sigma\rangle$ then $M(\alpha) \in \mathrm{Cle}_B(\sigma)$;
- $\langle FU;\sigma;\sigma'\rangle$ then $M(\alpha) \in \mathrm{Cle}_B(\mathbf{fc}[\sigma;\sigma'])$;
- $\langle GD;\sigma;\sigma'\rangle$ then $M(\alpha) \in \mathrm{Det}_B(\tau,\sigma,\sigma')$;
- $\langle ND;\mu\rangle$ then $M(\alpha) \in \mathrm{Dro}_B$;
- $\langle OD;\mu\rangle$ then $M(\alpha) \in \mathrm{Det}_B(\tau,\mathbf{t},\mathbf{t})$;
- $\langle RQ\rangle$ then $M(\alpha) \in \mathrm{Clq}_B$;
- $\langle SE;\sigma\rangle$ then $M(\alpha) \in \mathrm{Cle}_B(\sigma)$;
- $\langle ST\rangle$ then $M(\alpha) \in \mathrm{Cle}_B(\mathbf{t})$;
- $\langle CF;\sigma;\sigma'\rangle$, $\langle FE;\sigma;\sigma'\rangle$, $\langle FL;\sigma;\sigma'\rangle$, or $\langle FN;\sigma;\sigma'\rangle$
  then $M(x,\alpha) \in \mathrm{We}_B(\sigma')$ and $\mathrm{FP}_B(M(x,\alpha)) \subseteq \{x\}$;
- $\langle AL;\sigma\rangle$, $\langle AV;\sigma\rangle$, $\langle CC;\mu;\sigma\rangle$, $\langle CI;\mu;\sigma\rangle$, $\langle DJ;\sigma\rangle$, $\langle IV;\mu;\sigma\rangle$, $\langle PJ;\sigma\rangle$, $\langle QN;\sigma\rangle$, $\langle RC;\mu;\sigma\rangle$, $\langle SC;\mu;\sigma\rangle$, or $\langle VP;\mu;\sigma\rangle$
  then $M(x,\alpha) \in \mathrm{We}_B(\mathbf{t})$ and $\mathrm{FP}_B(M(x,\alpha)) \subseteq \{x\}$;
- $\langle BN;\mu;\sigma\rangle$, $\langle CN;\mu;\sigma\rangle$, or $\langle SN;\mu;\sigma\rangle$
  then $M(x,\alpha) \in \mathrm{Restr}_B(x)$ and $\mathrm{FP}_B(M(x,\alpha)) \subseteq \{x\}$;
- $\langle EI;\mu;\sigma;\sigma'\rangle$
  then $M(x,x',\alpha) \in \mathrm{Restr}_B(x')$ and $\mathrm{FP}_B(M(x,x',\alpha)) \subseteq \{x,x'\}$;
- $\langle ET;\mu;\sigma;\sigma';\sigma''\rangle$
  then $M(x,x',x'',\alpha) \in \mathrm{Restr}_B(x'')$ and $\mathrm{FP}_B(M(x,x',x'',\alpha)) \subseteq \{x,x',x''\}$;
- $\langle NP;\mu;\sigma\rangle$
  then $M(x,\alpha)\ \&\ d \in \mathrm{We}_B(\mathbf{t})$ and $\mathrm{FP}_B(M(x,\alpha)\ \&\ d) \subseteq \mathrm{FP}_B(d) - \{x\}$ for each $d \in \mathrm{We}_B(\mathbf{t})$;
- $\langle PE;\sigma\rangle$, $\langle PN;\sigma\rangle$, or $\langle QS;\sigma\rangle$
  then $M(x,\alpha)\ \&\ d \in \mathrm{We}_B(\tau)$ and $\mathrm{FP}_B(M(x,\alpha)\ \&\ d) \subseteq \mathrm{FP}_B(d) - \{x\}$ for each $\tau \in \mathrm{Typ}_B$ and each $d \in \mathrm{We}_B(\tau)$.

8.2. Translating the general rules

In the translation rules given below, x will be an arbitrary placeholder in $\mathrm{Plh}_B(\sigma)$ and x' an arbitrary placeholder in $\mathrm{Plh}_B(\sigma')$.

A placeholder y of type τ appears on the right hand side of the translation rules for the rule forms (3), (16), (21), (22), (32), (33), (36), (45), (46), and (48); and for (22), another placeholder y' of type τ' is needed as well. In each application of one of these rule forms within the (recursive) translation of one and the same request, we must use "fresh" placeholders, i.e., placeholders that are not already introduced elsewhere within the translation of the request concerned.

We recall from chapter 7 that both (21) and (22) stand for 4 rule forms. For each of these rule forms, a separate translation will be given. In these translation rules, δ will always denote the derivation tree with root label <AD> and γ always the derivation tree with a root label of the form <AL;τ>.

In all translation rules below, α, β, γ, and δ denote derivation trees. In (3), for instance, α and β denote derivation trees with a root label of the form <NP;μ;τ> and <VP;μ;τ>, respectively. As it stands, the left hand side does not contain a derivation tree: For the sake of simplicity, we wrote $M(\alpha\beta)$, where we should write $M((\text{<ST>};\text{<}\alpha;\beta\text{>}))$, according to L3.2(2) and the notation for 2-tuples introduced in chapter 0. A similar remark holds for the other translation rules. Notably, a seemingly circular translation such as in (5) should be read as $M((\text{<NP};\underline{\text{si}};\sigma\text{>};\text{<}\alpha\text{>})) = M(\alpha)$.

1	$M(\textbf{Give}\ \alpha.)$	=	$M(\alpha)\,\mathbf{?}$
2	$M(\textbf{Is it true that}\ \alpha?)$	=	$M(\alpha)\,\mathbf{?}$
3	$M(\alpha\beta)$	=	$M(y,\alpha)M(y,\beta)$
4	$M(x,\alpha\beta)$	=	$M(\alpha)M(x,\beta)\,\mathbf{:}$
5	$M(x,\alpha)$	=	$M(x,\alpha)$
6	$M(x,\alpha)$	=	$M(x,\alpha)$
7	$M(x,\alpha\beta)$	=	$\big(M(x,\alpha)\ \wedge\ M(x,\beta)\big)$
8	$M(x,\alpha)$	=	$M(x,\alpha)$
9	$M(x,\alpha\beta)$	=	$\big(M(x,\beta)\ \wedge\ M(x,\alpha)\big)$
10	$M(x,\alpha\beta)$	=	$\big(M(x,\alpha)\ \wedge\ M(x,\beta)\big)$
11	$M(x,\alpha)$	=	$M(x,\alpha)$
12	$M(x,\alpha)$	=	$M(x,\alpha)$
13	$M(x,\alpha\beta\gamma)$	=	$\big(M(x,\alpha)\ \wedge\ M(x,\gamma)\big)$
14	$M(x,\alpha\mathbf{,}\beta)$	=	$\big(M(x,\alpha)\ \wedge\ M(x,\beta)\big)$
15	$M(x,\alpha\beta)$	=	$M(x,\beta)$
16	$M(x,\textbf{whose}\ \alpha\beta)$	=	$[y \leftarrow M(x,\alpha)]M(y,\beta)$
17	$M(x,\alpha)$	=	$M(x,\alpha)$
18	$M(x,\alpha)$	=	$M(x,\alpha)$
19	$M(x,\alpha\beta\gamma)$	=	$\big(M(x,\alpha)\ \wedge\ M(x,\gamma)\big)$
20	$M(x,\alpha\mathbf{,}\beta)$	=	$\big(M(x,\alpha)\ \wedge\ M(x,\beta)\big)$
21	$M(x,\alpha\gamma\delta)$	=	$M(\delta)M(x,y,\alpha)\,\mathbf{:}\ M(y,\gamma)$
	$M(x,\alpha\gamma)$	=	$\exists\ M(x,y,\alpha)\,\mathbf{:}\ M(y,\gamma)$
	$M(x,\alpha\delta)$	=	$M(\delta)M(x,y,\alpha)\,\mathbf{:}\ \top$
	$M(x,\alpha)$	=	$\exists\ M(x,y,\alpha)\,\mathbf{:}\ \top$
22	$M(x,\alpha\beta\gamma\delta)$	=	$M(y',\beta)M(\delta)M(x,y',y,\alpha)\,\mathbf{:}\ M(y,\gamma)$
	$M(x,\alpha\beta\gamma)$	=	$M(y',\beta)\ \exists\ M(x,y',y,\alpha)\,\mathbf{:}\ M(y,\gamma)$
	$M(x,\alpha\beta\delta)$	=	$M(y',\beta)M(\delta)M(x,y',y,\alpha)\,\mathbf{:}\ \top$
	$M(x,\alpha\beta)$	=	$M(y',\beta)\ \exists\ M(x,y',y,\alpha)\,\mathbf{:}\ \top$
23	$M(x,\alpha\beta)$	=	$M(x,\beta)$
24	$M(x,\alpha\beta)$	=	$M(x,\beta)$
25	$M(x,\alpha\beta)$	=	$M(x,\beta)$
26	$M(x,x',\alpha\beta)$	=	$M(x,x,x',\alpha)$
27	$M(x,\alpha)$	=	$M(x,\alpha)$
28	$M(x,\alpha\beta)$	=	$\big(M(x,\alpha)\ \wedge\ M(x,\beta)\big)$
29	$M(\alpha\ \textbf{times})$	=	$M(\alpha)$
30	$M(\alpha\ \textbf{once})$	=	$(\exists\ M(\alpha)\mathbf{1})$
31	$M(\alpha)$	=	$M(\alpha)$
32	$M(\alpha\beta)$	=	$M(y,\beta)M(y,\alpha)$
33	$M(\alpha\beta\gamma)$	=	$\mathbf{\$}\ M(y,\gamma)\,\mathbf{:}\ M(y,\alpha)$

34	$M(x,\alpha)$	$= M(x,\alpha)$
35	$M(x,\alpha)$	$= M(x,\alpha)$
36	$M(x,\alpha\beta)$	$= [y \leftarrow M(x,\beta)]M(y,\alpha)$
37	$M(x, \textbf{the } \alpha\beta)$	$= M(x,\alpha)$
38	$M(x,\alpha\beta\gamma\delta)$	$= (M(x,\alpha) ; M(x,\gamma))$
39	$M(x,\alpha\textbf{,}\beta)$	$= (M(x,\alpha) ; M(x,\beta))$
40	$M(x,\alpha)$	$= (M(\alpha) \mathbin{@} x)$
41	$M(x,\alpha)$	$= (x \cdot M(\alpha))$
42	$M(x,\alpha)$	$= [x \leftarrow M(\alpha)]$
43	$M(x,\alpha)$	$= M(x,\alpha)$
44	$M(\alpha)$	$= M(\alpha)$
45	$M(\textbf{the } \alpha\beta \textbf{ of all } \gamma)$	$= M(\alpha) M(\underline{y},\gamma) : M(y,\beta)$
46	$M(\textbf{the number of } \alpha)$	$= \Sigma\, M(\underline{y},\alpha) : \mathbf{1}$
47	$M(x,\alpha)$	$= M(x,\alpha)$
48	$M(x,\alpha\beta)$	$= M(\underline{y},\beta)[x \leftarrow M(y,\alpha)]$
49	$M(x,\alpha\beta)$	$= M(x,\beta)$
50	$M(\alpha)$	$= M(\alpha)$
51	$M(\alpha)$	$= \forall$
52	$M(\alpha)$	$= M(\alpha)$
53	$M(\alpha)$	$= M(\alpha)$
54	$M(\alpha)$	$= \exists$
55	$M(\alpha\beta)$	$= (\exists\ M(\alpha) M(\beta))$
56	$M(\alpha)$	$= (\exists \neq M(\alpha))$
57	$M(\textbf{one})$	$= \mathbf{1}$
58	$M(\textbf{zero})$	$= \mathbf{0}$
59	$M(\textbf{two})$	$= \mathbf{2}$
60	$M(\textbf{three})$	$= \mathbf{3}$
61	$M(\textbf{at least})$	$= \geq$
62	$M(\textbf{at most})$	$= \leq$
63	$M(\textbf{exactly})$	$= =$
64	$M(\textbf{more than})$	$= >$
65	$M(\textbf{less than})$	$= <$
66	$M(\textbf{total})$	$= \Sigma$
67	$M(\textbf{minimal})$	$= \textbf{MIN}$
68	$M(\textbf{maximal})$	$= \textbf{MAX}$
69	$M(\textbf{average})$	$= \textbf{AVE}$
70	$M(\textbf{some})$	$= \exists$
71	$M(\textbf{no})$	$= \nexists$

It can be proved that these translation rules satisfy and preserve the conditions summarized at the end of section 8.1. The proof is by induction on the structure of the derivation tree and consists of 77 cases (i.e., 71 plus 6, see (21) and (22)). Below, we sketch the proof of two illustrative cases, namely, (16) and (48). It is left to the reader to check the other cases.

(16): According to the conditions for a derivation tree with a root label of the form <SC;μ;σ>, we have to prove that $M(x,\mathbf{whose}\ \alpha\beta) \in We_B(\mathbf{t})$ and $FP_B(M(x,\mathbf{whose}\ \alpha\beta)) \subseteq \{x\}$. We can use the induction hypotheses for $M(x,\alpha)$ and $M(y,\beta)$, the subexpressions occurring on the right hand side of the translation rule. (We recall from the beginning of this section that $y \in Plh_B(\tau)$.) The root label of α is <FN;σ;τ> (see section 7.1, rule form (16)), so the IH (induction hypothesis) for $M(x,\alpha)$ is (by section 8.1):

(a) $M(x,\alpha) \in We_B(\tau)$ and

(b) $FP_B(M(x,\alpha)) \subseteq \{x\}$.

The root label of β is <VP;<u>si</u>;τ>, so the IH for $M(y,\beta)$ is:

(c) $M(y,\beta) \in We_B(\mathbf{t})$ and

(d) $FP_B(M(y,\beta)) \subseteq \{y\}$.

From (a), (c), translation rule (16), and L4.1(7) we conclude that $M(x,\mathbf{whose}\ \alpha\beta) \in We_B(\mathbf{t})$. Furthermore,

$FP_B(M(x,\mathbf{whose}\ \alpha\beta))$	
$= FP_B([y \leftarrow M(x,\alpha)]M(y,\beta))$	by translation rule (16)
$= FP_B(M(x,\alpha)) \cup [FP_B(M(y,\beta)) - \{y\}]$	by D4.6, clause (7)
$\subseteq \{x\} \cup [\{y\} - \{y\}]$	by (b) and (d)
$= \{x\}$	

(48): According to the condition associated with a derivation tree with a root label of the form <PN;σ>, we have to prove for each $\tau' \in Typ_B$ and each $d \in We_B(\tau')$ that $M(x,\alpha\beta)\ \&\ d \in We_B(\tau')$ and $FP_B(M(x,\alpha\beta)\ \&\ d) \subseteq FP_B(d) - \{x\}$. We can use the induction hypotheses for $M(y,\beta)$ and $M(y,\alpha)$, where $y \in Plh_B(\tau)$ and the root labels of α and β are $<FE;\tau;\sigma>$ and $<PN;\tau>$, respectively. The IH for $M(y,\alpha)$ is:

(a) $M(y,\alpha) \in We_B(\sigma)$ and

(b) $FP_B(M(y,\alpha)) \subseteq \{y\}$.

The IH for $M(y,\beta)$ is that for every $\tau' \in Typ_B$ and every $e \in We_B(\tau')$:

(c) $M(y,\beta)$ & $e \in We_B(\tau')$ and

(d) $FP_B(M(y,\beta)$ & $e) \subseteq FP_B(e) - \{y\}$.

The proof, for any $\tau' \in Typ_B$ and any $d \in We_B(\tau')$, now runs as follows: Since $x \in Plh_B(\sigma)$, it follows from (a) and L4.1(7) that $[x \leftarrow M(y,\alpha)]d \in We_B(\tau')$. Hence, by translation rule (48) and the induction hypothesis (c), $M(x,\alpha\beta)$ & $d =$
$= M(y,\beta)[x \leftarrow M(y,\alpha)]d \in We_B(\tau')$, q.e.d. Furthermore,

$$\begin{aligned}
& FP_B(M(x,\alpha\beta) \;\&\; d) \\
& = FP_B(M(y,\beta)[x \leftarrow M(y,\alpha)]d) && \text{by translation rule (48)} \\
& \subseteq FP_B([x \leftarrow M(y,\alpha)]d) - \{y\} && \text{by (d)} \\
& = (FP_B(M(y,\alpha)) \cup [FP_B(d) - \{x\}]) - \{y\} && \text{by D4.6, clause (7)} \\
& = (FP_B(M(y,\alpha)) - \{y\}) \cup ([FP_B(d) - \{x\}] - \{y\}) && \text{by } (\star)\text{, see below} \\
& = [FP_B(d) - \{x\}] - \{y\} && \text{using (b)} \\
& \subseteq FP_B(d) - \{x\}
\end{aligned}$$

Above, $(\star)$ denotes the right distributive law for set difference over union: $(X \cup Y) - Z = (X - Z) \cup (Y - Z)$.

We end this section with some remarks on existential transitive verb phrases.

We recall from chapter 7 that an (in)transitive verb phrase for which an adverbial of degree (such as **more than 50 times**) makes sense, is called *existential*. An example of an existential transitive verb phrase relevant to our hospital database (as presented in the appendix) is the phrase **has treated**. An example of its use in connection with an adverbial of degree is the request

Is it true that no specialist has treated some patient more than 50 times?

Intuitively, specialist x has treated patient x' more than 50 times "means" that there are more than 50 treatments of x' performed by x. In the hospital database given in the appendix, the table indices and connector indices relevant to this request are the following:

$$\begin{tikzcd} SP & & P \\ & PT \arrow[ul, "PT\text{-}SP"] \arrow[ur, "PT\text{-}P"'] & \end{tikzcd}$$

In our hospital database, **has treated** is a terminal string of the nonterminal <ET;<u>si</u>;SP;P;PT>. SP is called the *subject type* of the existential transitive verb phrase **has treated**, P is called its *object type*, and PT is called its *underlying type* (see chapter 7). Loosely speaking, M(x,x',x",**has treated**) should express that

x" **is a treatment of** x' **by** x

A correct choice would be

M(x,x',x",**has treated**) = x" ∈ ($^\vee$**PT-P inv** x') ∧ (($^\vee$**PT-SP @** x") = x)

In general, let <g;h> be a type 2 skeleton, {D,D',M} ⊆ dom(g), and {C,C'} ⊆ dom(h) such that h(C) = (M;D) and h(C') = (M;D'). In a figure:

$$\begin{tikzcd} D & & D' \\ & M \arrow[ul, "C"] \arrow[ur, "C'"'] & \end{tikzcd}$$

Furthermore, let B be a CL-basis fit for <g;h>, d ∈ Plh_B(D), d' ∈ Plh_B(D'), and m ∈ Plh_B(M). For a derivation tree α with root label <ET;μ;D;D';M>, M(d,d',m,α) is equal to

m ∈ ($^\vee$C' **inv** d') ∧ (($^\vee$C **@** m) = d)

or to an equivalent expression in $Restr_B$(m), provided that C and C' are the connector indices that α refers to. Various examples of existential transitive verb phrases can be found in the appendix.

8.3. Some examples

With the production rules

<BN;si;SP>	::=	**specialist**
<BN;si;P>	::=	**patient**
<ET;si;SP;P;PT>	::=	**has treated**
<CA;pl>	::=	**50**

the request near the end of the previous section can be completely analyzed. In the following translation, **specialist** is abbreviated by β1, **patient** by β4, the noun phrase **no specialist** by α2, **has treated** by α3, the noun phrase **some patient** by β3, the adverbial of degree **more than 50 times** by δ3, and the comparative **more than** by α4. Furthermore, the concatenation of α3, β3, and δ3 is abbreviated by β2. We also indicate the translation rules used.

$M(\textbf{Is it true that } \alpha 2\ \beta 2\textbf{?})$

$= M(\alpha 2\ \beta 2)\textbf{?}$	by 2
$= M(\mathbf{s},\alpha 2)M(\mathbf{s},\beta 2)\textbf{?}$	by 3
$= M(\textbf{no})M(\mathbf{s},\beta 1)\textbf{:}\ M(\mathbf{s},\beta 2)\textbf{?}$	by 4
$= \nexists\ M(\mathbf{s},\beta 1)\textbf{:}\ M(\mathbf{s},\alpha 3\ \beta 3\ \delta 3)\textbf{?}$	by 50,53, and 71
$= \nexists\ M(\mathbf{s},\beta 1)\textbf{:}\ M(\mathbf{p},\beta 3)M(\delta 3)M(\mathbf{s},\mathbf{p},\mathbf{t},\alpha 3)\textbf{:}\ \mathrm{T}\textbf{?}$	by 17 and 22
$= \nexists\ M(\mathbf{s},\beta 1)\textbf{:}\ \exists\ M(\mathbf{p},\beta 4)\textbf{:}\ M(\delta 3)M(\mathbf{s},\mathbf{p},\mathbf{t},\alpha 3)\textbf{:}\ \mathrm{T}\textbf{?}$	by 4,50,53, and 70
$= \nexists\ M(\mathbf{s},\beta 1)\textbf{:}\ \exists\ M(\mathbf{p},\beta 4)\textbf{:}\ (\exists > M(\mathbf{50}))M(\mathbf{s},\mathbf{p},\mathbf{t},\alpha 3)\textbf{:}\ \mathrm{T}\textbf{?}$	by 29,55, and 64

With the translation rules

$M(\textbf{s,specialist})$	$= \mathbf{s} \in {}^{\vee}\mathbf{SP}$
$M(\textbf{p,patient})$	$= \mathbf{p} \in {}^{\vee}\mathbf{P}$
$M(\textbf{s,p,t,has treated})$	$= \mathbf{t} \in ({}^{\vee}\mathbf{PT\text{-}P}\ \mathbf{inv}\ \mathbf{p}) \wedge ((\mathbf{t.SNR}) = (\mathbf{s.ENR}))$
$M(\mathbf{50})$	$= \mathbf{50}$

the final translation is as follows:

$M(\textbf{Is it true that no specialist has treated some patient more than 50 times?}) =$

$$\nexists\ \mathbf{s}\in{}^{\vee}\mathbf{SP}\textbf{:}\ \exists\ \mathbf{p}\in{}^{\vee}\mathbf{P}\textbf{:}\ (\exists > \mathbf{50})\mathbf{t} \in ({}^{\vee}\mathbf{PT\text{-}P}\ \mathbf{inv}\ \mathbf{p}) \wedge ((\mathbf{t.SNR}) = (\mathbf{s.ENR}))\textbf{:}\ \mathrm{T}\textbf{?}$$

As a second example, we translate the request

Give name, number, and department number of the manager of each department.

concerning our employees-and-departments database. In chapter 7, this request was already partly analyzed. With the production rules

<BN;si;DEP>	::=	**department**
<FU;DEP;EMPL>	::=	**manager**
<AR;EMPL;str>	::=	**name**
<AR;EMPL;int>	::=	**number**
		|**department number**

this analysis can be completed. In the following translation, **name** is abbreviated by α4, **number** by α5, **department number** by γ5, **manager** by α3, **department** by γ1, and **the manager of** by β2. Furthermore,
β4 stands for α5**,** **and** γ5 **of**
α2 stands for α4**,**β4
α1 stands for α2 β2, and
α0 stands for α1 **each** γ1.

$M(\mathbf{Give}\ \alpha0\mathbf{.})$	
$= M(\alpha0)\mathbf{?}$	by 1
$= \mathbf{\$}\ M(\mathbf{d},\gamma1)\mathbf{:}\ M(\mathbf{d},\alpha1)\mathbf{?}$	by 33
$= \mathbf{\$}\ M(\mathbf{d},\gamma1)\mathbf{:}\ [\mathbf{e} \leftarrow M(\mathbf{d},\beta2)]M(\mathbf{e},\alpha2)\mathbf{?}$	by 36
$= \mathbf{\$}\ M(\mathbf{d},\gamma1)\mathbf{:}\ [\mathbf{e} \leftarrow (M(\alpha3)\ @\ \mathbf{d})]M(\mathbf{e},\alpha2)\mathbf{?}$	by 37 and 40

where

$M(\mathbf{e},\alpha2)$	
$= (M(\mathbf{e},\alpha4)\mathbf{;}\ M(\mathbf{e},\beta4))$	by 35 and 39
$= ((\mathbf{e}.M(\alpha4))\mathbf{;}\ (M(\mathbf{e},\alpha5)\mathbf{;}\ M(\mathbf{e},\gamma5)))$	by 41 and 38
$= ((\mathbf{e}.M(\alpha4))\mathbf{;}\ ((\mathbf{e}.M(\alpha5))\mathbf{;}\ (\mathbf{e}.M(\gamma5))))$	by 41 and 41

With the translation rules

M(**d,department**)	=	$\mathbf{d} \in {}^{\vee}\mathbf{DEP}$
M(**manager**)	=	$^{\vee}\mathbf{MANAGEROF}$
M(**name**)	=	**NAME**
M(**number**)	=	**NR**
M(**department number**)	=	**DPT**

the final translation is as follows:

M(**Give name, number, and department number of the manager of each department.**) =

$ d∈$^{\vee}$DEP: [e ← ($^{\vee}$MANAGEROF @ d)]((e.NAME); ((e.NR); (e.DPT)))?

8.4. Translating the intermediate forms

We recall that the translation of the production rules presented in section 7.3 depends on the representation of dates that is chosen in the application concerned. The translation rules below apply when a date is represented by 6 digits in the usual way (where, e.g., October 5, 1953 is represented by **531005**).

If α is a derivation tree with root label <JV;date> then α must be translated in combination with a placeholder x of type **int**. We then require that $M(x,\alpha) \in We_B(\mathbf{t})$ and $FP_B(M(x,\alpha)) \subseteq \{x\}$.

If the root label of α is <digit> then $M(\alpha)$ will be the digit concerned or, formally, $M(\alpha) = Fr(\alpha)$, i.e., the frontier of the tree α (see chapter 3).

If the root label of α is <month>, <D2>, <day>, or <year> then $M(\alpha)$ will be a sequence (or "string") of 2 digits.

If the root label of α is <CE;date> then $M(\alpha)$ will be a string of 6 digits and, hence, an element of $Con_B(\mathbf{int})$.

If the root label of α is <SE;date> then we require that $M(\alpha) \in Cle_B(\mathbf{int})$.

Finally, if the root label of α is <PT> then we require that $M(\alpha) \in Binop_B(\mathbf{int},\mathbf{int},\mathbf{t})$.

We point out that no "new" (or "fresh") placeholders are needed in the translation rules below.

SG01	$M(x,\alpha\beta)$	$= (x\ M(\alpha)M(\beta))$
SG02	$M(x,$**b.** α **and** $\beta)$	$= ((M(\alpha) < x) \wedge (x < M(\beta)))$
SG03	$M(x,$**i.t.p.** α **to** $\beta)$	$= ((M(\alpha) \leq x) \wedge (x \leq M(\beta)))$
SG04	$M(x,$**i.t.p.** $\alpha\beta$ **to** γ**,** $\delta)$	$= ((M(\delta)M(\alpha)M(\beta) \leq x) \wedge (x \leq M(\delta)M(\alpha)M(\gamma)))$
SG05	$M(x,$**in** $\alpha\beta)$	$= ((x \div 100) = M(\beta)M(\alpha))$
SG06	$M(x,$**in** $\alpha)$	$= ((x \div 10000) = M(\alpha))$
SG07	$M(\alpha\beta,\gamma)$	$= M(\gamma)M(\alpha)M(\beta)$
SG08	$M(\alpha\beta\gamma)$	$= M(\alpha)M(\beta)M(\gamma)$
SG09	$M(\alpha\beta)$	$= M(\beta)$
SG10	$M(\alpha)$	$= M(\alpha)$
SG11	$M(\alpha)$	$= 0\ M(\alpha)$
SG12	$M(\alpha\beta)$	$= M(\alpha)M(\beta)$
SG13	$M($**on**$)$	$= =$
SG14	$M($**before**$)$	$= <$
SG15	$M($**after**$)$	$= >$
SG16	$M($**January**$)$	= **01**
SG17	$M($**February**$)$	= **02**
SG18	$M($**March**$)$	= **03**
SG19	$M($**April**$)$	= **04**
SG20	$M($**May**$)$	= **05**
SG21	$M($**June**$)$	= **06**
SG22	$M($**July**$)$	= **07**
SG23	$M($**August**$)$	= **08**
SG24	$M($**September**$)$	= **09**
SG25	$M($**October**$)$	= **10**
SG26	$M($**November**$)$	= **11**
SG27	$M($**December**$)$	= **12**

As an illustration, we translate the JV-phrase **in the period January 4 to 20, 1986**.

By SG04,

$M(x,$**in the period January 4 to 20, 1986**$)$ $\overset{04}{=}$

$((M($**1986**$)M($**January**$)M($**4**$) \leq x) \wedge (x \leq M($**1986**$)M($**January**$)M($**20**$)))$

where $M($**1986**$) \overset{09}{=} M($**86**$) \overset{12}{=} M($**8**$)M($**6**$) =$ **86**

and $M($**January**$) \overset{16}{=}$ **01** and $M(\mathbf{4}) \overset{11}{=} \mathbf{0}\, M(\mathbf{4}) = \mathbf{04}$

and $M(\mathbf{20}) \overset{10}{=} M(\mathbf{20}) \overset{12}{=} M(\mathbf{2})M(\mathbf{0}) = \mathbf{20}$

Thus,

$M(x, \textbf{in the period January 4 to 20, 1986}) =$

$((\mathbf{860104} \leq x) \wedge (x \leq \mathbf{860120}))$

As a second example, we translate **in January 1986**:

$M(x, \textbf{in January 1986})$

$\overset{05}{=} ((x \div \mathbf{100}) = M(\mathbf{1986})M(\textbf{January}))$

$= ((x \div \mathbf{100}) = \mathbf{8601})$

APPENDIX. A NONTRIVIAL EXAMPLE

One of the main problems of a real life database is, besides its mass (i.e., a very large number of tuples in the actual DB snapshot), also its complexity (i.e., many table indices, many attributes, and many relevant DB functions). Our tiny employees-and-departments database and the widely used suppliers-parts-projects example are too simple to illustrate some of the more intriguing database problems. For that reason we also present a *nontrivial* example of a database, in casu a hospital database. Although this example is still smaller than many real-life examples, it yet shows some of the consequences of the complexity of a database. Our example is a variation on the one in [Re 84], chapter 7.

A.1. A nontrivial type 1 model

We start with the definition of a conceptual skeleton Ghsp and then give the intuitive meaning of the table indices and attributes of Ghsp. The definition of a DB universe over Ghsp will follow later on.

The set function Ghsp is defined by

dom(Ghsp) = {P,EMP,NRS,SP,DPT,RM,ADM,REL,TM,PT,M,MP} and

Ghsp(P) = {PNR,NM,AD,CTY,BDT,BLG,RHF,SX}

Ghsp(EMP) = {ENR,NM,AD,CTY,SAL}

Ghsp(NRS) = {NNR,DNR,SVNR}

Ghsp(SP) = {ENR,DNR,LOC,NB,OND}

Ghsp(DPT) = {DNR,DNM,NNR,ENR}

Ghsp(RM) = {RNR,DNR,NB}

Ghsp(ADM) = {PNR,INDTE,SNR,RNR,ARCH,RSN}

Ghsp(REL) = {PNR,INDTE,RDTE,INVAM}

Ghsp(TM) = {TCD,TNM,TSRT,TAR}

Ghsp(PT) = {PNR,TCD,NT,SNR,ASNR,DTE,LENGTH,TRM}

Ghsp(M) = {MCD,MNM,MSRT,DGCD}

Ghsp(MP) = {PNR,MCD,DTE,SNR,LENGTH,FD,AMT} .

In other words, Ghsp is a conceptual skeleton with 12 table indices and a total number of 61 attributes.

The table indices are intended for patients (P), employees (EMP), departments (DPT), nursing rooms (RM), admittances (ADM), possible treatments (TM), specific applications of a treatment (PT), medicines (M), and medicin prescriptions (MP). For two subsets of employees there is an additional table index, namely for nurses (NRS) and for specialists (SP). The ADM tables are intended to be cumulative, i.e., containing both current and past admittances; for the subset of past admittances (or releases) there is an additional table index REL. (The connection between REL and ADM, and between NRS and SP on the one hand and EMP on the other hand, is known as *generalization* - see [SS 77] - or as *differentiation*.)

The relevant features for a patient are his (or her) patient number (PNR), name (NM), address (AD), city (CTY), date of birth (BDT), blood group (BLG), rhesus factor (RHF), and sex (SX).

Employee number (ENR), name (NM), address (AD), city (CTY), and salary (SAL) are relevant for all employees of our (imaginary) hospital. Not all employees belong to a fixed department, but nurses and specialists do. That is why Ghsp(NRS) and Ghsp(SP) contain an attribute DNR (for department number) and Ghsp(EMP) doesn't. For specialists, there are other relevant features as well, such as the number of beds available for that specialist (NB), the status of that specialist (OND), and the employee number of his (or her) so-called locum tenens (LOC). A feature that is relevant for nurses only is the employee number of his (or her) supervisor (SVNR). Note that the features sex and date of birth are considered relevant only for patients, not for employees.

The relevant features for a department are its number (DNR), its name (DNM), and the employee numbers of its head nurse (NNR) and its head specialist (ENR).

For a nursing room, its room number (RNR), the number of the department that nursing room belongs to (DNR), and its number of beds (NB) are relevant.

Reason for admittance (RSN) and date of admittance (INDTE) are relevant features for each admittance. Other relevant features are the patient number of the subject (PNR), the room number of the nursing room concerned (RNR), the employee number of the specialist responsible for that admittance (SNR), and an indication whether or not that admittance is finished (ARCH, which stands for "archive"). Additional features for releases are the date of release (RDTE) and the invoice amount (INVAM).

The relevant features for a treatment are code (TCD), name (TNM), sort (TSRT), and tariff (TAR).

Relevant features for an application of a treatment are treatment code (TCD), patient number of the subject (PNR),and serial number of this treatment to this subject (NT), i.e., the number of times the subject has undergone the treatment (code) concerned (the actual application included). The other relevant features are the employee numbers of the treating specialist and of the assistent (SNR, respectively ASNR), treatment date (DTE), treatment room (TRM), and length of treatment (LENGTH), in minutes.

The relevant features for a medicine are code (MCD), name (MNM), sort (MSRT), and danger code (DGCD).

The relevant features for a medicine prescription are number of the receiving patient (PNR), medicine code (MCD), starting date (DTE), number of days of that prescription (LENGTH), frequency, i.e., times a day (FD), number of units per time (AMT, for "amount"), and employee number of the prescribing specialist (SNR).

Note that the data of each specialist are thus spread over two tuples, namely over an SP tuple and over an EMP tuple. In order to relate each SP tuple to the corresponding EMP tuple, Ghsp(SP) contains the attribute ENR. Similarly, Ghsp(NRS) contains the attribute NNR. Also the data of each release will be spread over two tuples. The attributes PNR and INDTE in Ghsp(REL) will be used to relate each REL tuple to the corresponding ADM tuple.

Further on, we will define a DB universe over Ghsp that accounts for various requirements (or so-called static integrity constraints). Some of the simplest requirements are:

R1: The (legal) minimum for salaries is 1268 (guilders a month).

R2: An employee number consists of 3 or 4 digits.

R3: A nurse number consists of 4 digits.

R4: A specialist number consists of 3 digits.

R5: Some specialists may have no beds available.

R6: A nursing room may contain at most 15 beds.

R7: Some treatments may be free of charge (i.e., the tariff is 0).

R8: Danger codes can vary from 1 up to and including 20.

R9: A medicine can be prescribed for at most three months per prescription and the maximum frequency is 6 times a day; however, there is no restriction on the number of units per time.

The requirements above are requirements *per attribute*. Our DB universe will also account for the following requirements, which are requirements *between* different attributes of the same tuple:

R10: The locum tenens of a specialist must (of course) be someone else.

R11: Admittances last at least one night.

R12: The treating specialist and the assistent for a treatment are not the same persons. Although this might be trivial, it must nevertheless be stated explicitly in the definition of the DB universe.

Furthermore, we want to have the following keys:

{PNR} for P	{DNR} and {DNM} for DPT	{PNR,INDTE} for ADM
{ENR} for EMP	{TCD} and {TNM} for TM	{PNR,INDTE} for REL
{NNR} for NRS	{MCD} and {MNM} for M	{PNR,TCD,NT} for PT
{ENR} for SP		{PNR,MCD,DTE} for MP
{RNR} for RM		

These requirements are in fact requirements *per table*, i.e., between different tuples of the same table. Our DB universe will account for these 15 requirements per table as well as for the following:

R13: The salary of a specialist may not be higher than four times the average salary. (We note that a specialist has an employee number consisting of three digits. Later we also require that, conversely, specialists are the only employees with a 3-digit employee number. Therefore, it is possible to formulate this requirement as a requirement per EMP table.)

R14: The supervisor of a nurse is also a nurse and, moreover, this supervising nurse works for (i.e., belongs to) the same department.

R15: The locum tenens of a specialist is also a specialist (though not necessarily of the same department).

R16: There is at most one current admittance per patient.

R17: Different (past) admittances of the same patient do not overlap.

R18: Per patient different applications of the same treatment are numbered consecutively and chronologically, starting with 1.

Finally, our DB universe will also account for the following requirements, which are requirements *between* different tables of the same DB snapshot:

R19: The "P table" must contain all patient numbers mentioned in the ADM table, the PT table, or the MP table.

R20: The DPT table must contain all department numbers mentioned in the NRS table, the SP table, or the RM table.

R21: The SP table must contain all specialist numbers mentioned in the DPT table, the ADM table, the PT table (twice), or the MP table.

R22: The RM table must contain all room numbers mentioned in the ADM table.

R23: The TM table must contain all treatment codes mentioned in the PT table.

R24: The M table must contain all medicine codes mentioned in the MP table.

R25: The EMP table must contain all nurse numbers mentioned in the NRS table.

R26: The EMP table must contain all specialist numbers mentioned in the SP table; furthermore, specialists are the only employees with a 3-digit employee number.

R27: The REL table represents all past admittances (and no others).

R28: The head nurse of a department must be a nurse belonging to that department.

R29: Head nurses, and only those nurses, "supervise" themselves.

R30: A head specialist is head specialist of his own department (i.e., the department he formally belongs to) but possibly of other departments as well.

R31: A current and a past admittance of the same patient do not overlap.

R32: A treatment of a patient only happens during an admittance of that patient (but not on the day of release). In our hospital, that admittance is called the underlying admittance of that treatment.

We give some auxiliary definitions before we define our DB universe over Ghsp.

If $m \in \mathbb{N}$ and $n \in \mathbb{N}$ then:

$$[m..n] \overset{D}{=} \{k \in \mathbb{N} \mid m \leq k \text{ and } k \leq n\} ;$$

$$[m..) \overset{D}{=} \{k \in \mathbb{N} \mid m \leq k\} .$$

If A is a set, T is a table over A, and $a \in A$ then:

$$\underline{Cv(a,T)} \overset{D}{=} \{t(a) \mid t \in T\} .$$

Thus, Cv(a,T) is the set of "column values" in the "a-column" of T.

The generalized product of a set function - see chapter 0 - will be written in the form <u>PROD</u> ... <u>END</u> as in

<u>PROD</u> STRT : STRSET, HNR: [1..) <u>END</u>

which stands for $\Pi(F_0)$ where F_0 is the set function {(STRT;STRSET), (HNR; [1..))}.

In our DB universe, STRT stands for "street" and HNR for "house number". The set STRSET will be left unspecified but might be conceived of as some set of strings, e.g., the set of all sequences over some (character) set C.

On the following pages, we give a stepwise buildup of our non-trivial DB universe Uhsp over Ghsp. By D1.4, <Ghsp;Uhsp> is an example of a type 1 model.

For referential purposes, the formal counterparts of the forementioned requirements (from the tenth one on) will be indicated by a corresponding number.

```
DPSET = [1..100];
PSET  = [1..100000];
D3    = [100..999];
ESET  = [100..9999];
D4    = [1000..9999];
DSET  = [700101..991231];
BDSET = [18500101..19991231];
ADDR  = PROD
           STRT: STRSET,
           HNR : [1..)
        END;

PRP   = PROD
           PNR: PSET,
           NM : STRSET,
           AD : ADDR,
           CTY: STRSET,
           BDT: BDSET,
           BLG: {0,A,B,AB},
           RHF: {+,-},
           SX : {♀,♂}
        END;
```

```
PREMP = PROD
           ENR : ESET,
           NM  : STRSET,
           AD  : ADDR,
           CTY : STRSET,
           SAL : [1268..)
        END;
PRNRS = PROD
           NNR : D4,
           DNR : DPSET,
           SVNR: D4
        END;
PRSP  = PROD
           ENR : D3,
           DNR : DPSET,
           LOC : D3,
           NB  : [0..),
           OND : [0..1]
        END;
PRDPT = PROD
           DNR : DPSET,
           DNM : STRSET,
           NNR : D4,
           ENR : D3
        END;
```

```
PRRM  = PROD
           RNR   : [1..1399],
           DNR   : DPSET,
           NB    : [1..15]
        END;
PRADM = PROD
           PNR   : PSET,
           INDTE : DSET,
           SNR   : D3,
           RNR   : [1..1399],
           ARCH  : [0..1],
           RSN   : STRSET
        END;
PRREL = PROD
           PNR   : PSET,
           INDTE : DSET,
           RDTE  : DSET,
           INVAM : [160..)
        END;
PRTM  = PROD
           TCD   : STRSET,
           TNM   : STRSET,
           TSRT  : STRSET,
           TAR   : [0..)
        END;
```

```
PRPT  = PROD
           PNR    : PSET,
           TCD    : STRSET,
           NT     : [1..),
           SNR    : D3,
           ASNR   : D3,
           DTE    : DSET,
           LENGTH : [1..),
           TRM    : STRSET
        END;
PRM   = PROD
           MCD    : STRSET,
           MNM    : STRSET,
           MSRT   : STRSET,
           DGCD   : [1..20]
        END;
PRMP  = PROD
           PNR    : PSET,
           MCD    : STRSET,
           DTE    : DSET,
           SNR    : D3,
           LENGTH : [1..90],
           FD     : [1..6],
           AMT    : [1..)
        END;
```

```
TUSSP = {t ∈ PRSP | t(LOC) ≠ t(ENR)};                                              (FR10)
TUSRL = {t ∈ PRREL | t(INDTE) < t(RDTE)};                                          (FR11)
TUSPT = {t ∈ PRPT | t(SNR) ≠ t(ASNR)};                                             (FR12)

PU    = {T ⊆ PRP | {PNR} is u.i. for T};
EMPU  = {T ⊆ PREMP | {ENR} is u.i. for T and
                     ∀t∈T: if t(ENR) ≤ 999 then t(SAL) ≤ 4 × (Σ_{t'∈T} t'(SAL)/#T)};   (FR13)
NRSU  = {T ⊆ PRNRS | {NNR} is u.i. for T and
                     Cv(SVNR,T) ⊆ Cv(NNR,T) and                                    (FR14)
                     ∀t∈T: ∀t'∈T: if t(NNR) = t'(SVNR) then t(DNR) = t'(DNR)};
SPU   = {T ⊆ TUSSP | {ENR} is u.i. for T and
                     Cv(LOC,T) ⊆ Cv(ENR,T)};                                       (FR15)
DPTU  = {T ⊆ PRDPT | {DNR} is u.i. for T and {DNM} is u.i. for T};
RMU   = {T ⊆ PRRM | {RNR} is u.i. for T};
ADMU  = {T ⊆ PRADM | {PNR,INDTE} is u.i. for T and
                     {PNR} is u.i. for {t ∈ T | t(ARCH) = 0}};                     (FR16)
RELU  = {T ⊆ TUSRL | {PNR,INDTE} is u.i. for T and
                     ∀t∈T: ∀t'∈T: if t(PNR) = t'(PNR) and t(INDTE) < t'(INDTE) then t(RDTE) ≤ t'(INDTE)};   (FR17)
TMU   = {T ⊆ PRTM | {TCD} is u.i. for T and {TNM} is u.i. for T};
PTU   = {T ⊆ TUSPT | {PNR,TCD,NT} is u.i. for T and
                     ∀t∈T: if t(NT) ≠ 1 then ∃t'∈T: t'↾{PNR,TCD} = t↾{PNR,TCD} and       (FR18)
                                      t'(NT) = t(NT) - 1 and t'(DTE) ≤ t(DTE)};

MU    = {T ⊆ PRM | {MCD} is u.i. for T and {MNM} is u.i. for T};
MPU   = {T ⊆ PRMP | {PNR,MCD,DTE} is u.i. for T};
```

```
PRHSP = PROD             | Uhsp = {v | v ∈ PRHSP and
        P  : PU,         |            Cv(PNR,v(P)) ⊇ Cv(PNR,v(ADM)) ∪ Cv(PNR,v(PT)) ∪ Cv(PNR,v(MP)) and               (FR19)
        EMP: EMPU,       |            Cv(DNR,v(DPT)) ⊇ Cv(DNR,v(NRS)) ∪ Cv(DNR,v(SP)) ∪ Cv(DNR,v(RM)) and              (FR20)
        NRS: NRSU,       |            Cv(ENR,v(SP)) ⊇ Cv(ENR,v(DPT)) ∪ Cv(SNR,v(ADM)) ∪ Cv(SNR,v(PT)) ∪               (FR21)
        SP : SPU,        |                           Cv(ASNR,v(PT)) ∪ Cv(SNR,v(MP)) and
        DPT: DPTU,       |            Cv(RNR,v(RM)) ⊇ Cv(RNR,v(ADM)) and                                               (FR22)
        RM : RMU,        |            Cv(TCD,v(TM)) ⊇ Cv(TCD,v(PT)) and                                                (FR23)
        ADM: ADMU,       |            Cv(MCD,v(M)) ⊇ Cv(MCD,v(MP)) and                                                 (FR24)
        REL: RELU,       |            Cv(ENR,v(EMP)) ⊇ Cv(NNR,v(NRS)) and                                              (FR25)
        TM : TMU,        |            Cv(ENR,v(SP)) = {t(ENR) | t ∈ v(EMP) and t(ENR) ≤ 999} and                       (FR26)
        PT : PTU,        |            v(REL) ↾ {PNR,INDTE} = {t ↾ {PNR,INDTE} | t ∈ v(ADM) and t(ARCH) = 1} and        (FR27)
        M  : MU,         |            v(DPT) ↾ {DNR,NNR} ⊆ v(NRS) ↾ {DNR,NNR} and                                      (FR28)
        MP : MPU         |            Cv(NNR,v(DPT)) = {t(NNR) | t ∈ v(NRS) and t(NNR) = t(SVNR)} and                  (FR29)
      END;               |            v(DPT) ↾ {DNR,ENR} ⊇ {t ↾ {DNR,ENR} | t ∈ v(SP) and t(ENR) ∈ Cv(ENR,v(DPT))} and (FR30)
                         |            (∀t∈v(ADM): ∀t'∈v(REL): if t(PNR) = t'(PNR) and t(ARCH) = 0                      (FR31)
                         |                                 then t'(RDTE) ≤ t(INDTE)) and
                         |            ∀t∈v(PT): ∃t'∈v(ADM): (t(PNR) = t'(PNR) and t'(INDTE) ≤ t(DTE) and               (FR32)
                         |                                (if t'(ARCH) = 1
                         |                                 then ∃r∈v(REL): (r ↾ {PNR,INDTE} = t' ↾ {PNR,INDTE}
                         |                                                 and t(DTE) < r(RDTE))))}.
```

A.2. A nontrivial type 2 model

We will "extend" the type 1 model <Ghsp;Uhsp> to a type 2 model with 21 distinguished DB functions. The function Hhsp introduces 21 connector indices - see section 1.2 - together with their source index and their target index:

Hhsp(NRS-EMP)	=	(NRS;EMP)	Hhsp(DPT-NRS)	=	(DPT;NRS)
Hhsp(NRS-DPT)	=	(NRS;DPT)	Hhsp(DPT-SP)	=	(DPT;SP)
Hhsp(NRS-NRS)	=	(NRS;NRS)	Hhsp(RM-DPT)	=	(RM;DPT)
Hhsp(SP-EMP)	=	(SP;EMP)	Hhsp(MP-P)	=	(MP;P)
Hhsp(SP-DPT)	=	(SP;DPT)	Hhsp(MP-SP)	=	(MP;SP)
Hhsp(SP-SP)	=	(SP;SP)	Hhsp(MP-M)	=	(MP;M)
Hhsp(ADM-P)	=	(ADM;P)	Hhsp(PT-P)	=	(PT;P)
Hhsp(ADM-SP)	=	(ADM;SP)	Hhsp(PT-TM)	=	(PT;TM)
Hhsp(ADM-RM)	=	(ADM;RM)	Hhsp(PT-SP)	=	(PT;SP)
Hhsp(REL-ADM)	=	(REL;ADM)	Hhsp(PT-AST)	=	(PT;SP)
			Hhsp(PT-ADM)	=	(PT;ADM)

Summarized informally: Hhsp(α-β) = (α;β), except for PT-AST.

According to D1.9 we still have to define an "interpretation function" Ihsp over dom(Hhsp). This function tells which DB function each connector index stands for. For 19 of the 21 connector indices C, the DB function Ihsp(C) is an instance of the "standard" situation described after example 1.3. For REL-ADM, however, the "foreign key" consists of *two* attributes and for PT-ADM there is not even a "foreign key". Nevertheless, the relation Ihsp(PT-ADM)(v), which is defined below, happens to be a *function* over v(PT). This can be proved by using the definition of Uhsp, more specifically, by using the formal requirements (FR32), (FR16), (FR17), and (FR31).

For any type 1 model <g;U>, $v \in U$, $M \in$ dom(g), $a \in g(M)$, $D \in$ dom(g), and $a' \in g(D)$, we define the following special variant of an *equi-join* (cf. [Ul 80]):

$$\underline{\text{Speq}(v,M,a,D,a')} \overset{D}{=} \{(t;t') \in v(M) \times v(D) \mid t(a) = t'(a')\} .$$

The definition of Ihsp now runs as follows: For every $v \in$ Uhsp we define

```
Ihsp(NRS-EMP)(v)  = Speq(v,NRS,NNR,EMP,ENR)

Ihsp(NRS-DPT)(v)  = Speq(v,NRS,DNR,DPT,DNR)

Ihsp(NRS-NRS)(v)  = Speq(v,NRS,SVNR,NRS,NNR)

Ihsp(SP-EMP)(v)   = Speq(v,SP,ENR,EMP,ENR)

Ihsp(SP-DPT)(v)   = Speq(v,SP,DNR,DPT,DNR)

Ihsp(SP-SP)(v)    = Speq(v,SP,LOC,SP,ENR)

Ihsp(ADM-P)(v)    = Speq(v,ADM,PNR,P,PNR)

Ihsp(ADM-SP)(v)   = Speq(v,ADM,SNR,SP,ENR)

Ihsp(ADM-RM)(v)   = Speq(v,ADM,RNR,RM,RNR)

Ihsp(REL-ADM)(v)  =
{(t;t') ∈ v(REL) × v(ADM) | t ↾ {PNR,INDTE} = t' ↾ {PNR,INDTE}}

Ihsp(DPT-NRS)(v)  = Speq(v,DPT,NNR,NRS,NNR)

Ihsp(DPT-SP)(v)   = Speq(v,DPT,ENR,SP,ENR)

Ihsp(RM-DPT)(v)   = Speq(v,RM,DNR,DPT,DNR)

Ihsp(MP-P)(v)     = Speq(v,MP,PNR,P,PNR)

Ihsp(MP-SP)(v)    = Speq(v,MP,SNR,SP,ENR)

Ihsp(MP-M)(v)     = Speq(v,MP,MCD,M,MCD)

Ihsp(PT-P)(v)     = Speq(v,PT,PNR,P,PNR)

Ihsp(PT-TM)(v)    = Speq(v,PT,TCD,TM,TCD)

Ihsp(PT-SP)(v)    = Speq(v,PT,SNR,SP,ENR)

Ihsp(PT-AST)(v)   = Speq(v,PT,ASNR,SP,ENR)

Ihsp(PT-ADM)(v)   =
{(t;t') ∈ v(PT) × v(ADM) | t(PNR) = t'(PNR) and
                           t'(INDTE) ≤ t(DTE) and
                           if t'(ARCH) = 1 then
                           ∃r∈v(REL):
                             (r ↾ {PNR,INDTE} = t' ↾ {PNR,INDTE} and
                             t(DTE) < r(RDTE))}
```

It follows from the definition of Hhsp and Ihsp that <Ghsp;Hhsp;Uhsp;Ihsp> is an example of a type 2 model!

The functions Ihsp(NRS-EMP)(v), Ihsp(SP-EMP)(v), and Ihsp(REL-ADM)(v) are *one-to-one* since the "foreign keys" {NNR}, {ENR}, and {PNR,INDTE} are *keys* too (for the source indices NRS, SP, and REL, respectively). We note that the concept of *generalization* applies here; see [SS 77].

We further note that the function Ihsp(DPT-NRS)(v) is also one-to-one. This is a consequence of the formal requirement (FR28). (Another consequence of (FR28) is that {NNR} is also a key for DPT in <Ghsp;Uhsp> even though this was not explicitly required!)

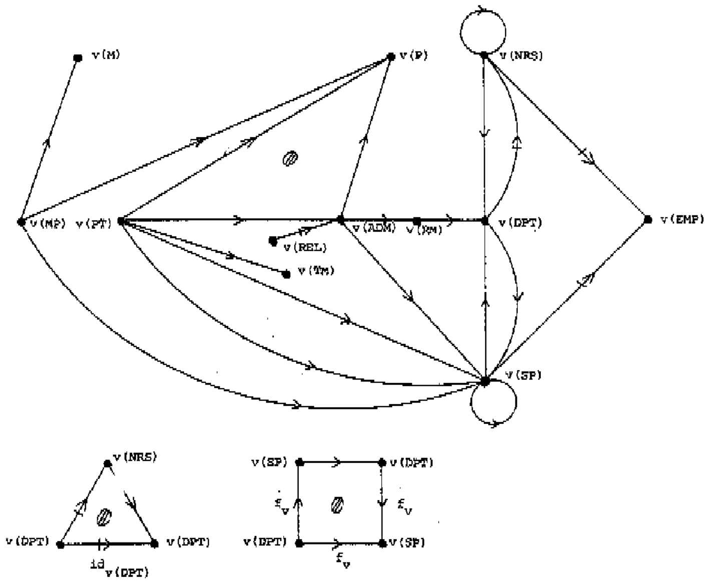
It is easily checked that Ihsp(PT-P)(v) is the composition of Ihsp(ADM-P)(v) after Ihsp(PT-ADM)(v). Furthermore, the composition of Ihsp(NRS-DPT)(v) after Ihsp(DPT-NRS)(v) is the identity function on v(DPT); this follows from (FR28) again.

Finally, let f_v be the function Ihsp(DPT-SP)(v); then $f_v = f_v \circ$ Ihsp(SP-DPT)(v) $\circ f_v$, because of (FR30).

The following pictures summarize much of the foregoing discussion. The first picture indicates the "direction" of the 21 distinguished DB functions; the four one-to-one functions are indicated by a "dashed" arrow ($\bullet\!\!\longrightarrow\!\!\!+\!\!\rightarrowtail\!\!\cdots\!\bullet$). The equation

Ihsp(PT-P)(v) = Ihsp(ADM-P)(v) $\circ$ Ihsp(PT-ADM)(v)

is indicated by the symbol $\oslash$ in the "triangle" concerned. The other two equations are represented by separate pictures. (In other branches of mathematics, these pictures are known as *commutative diagrams*.)



A.3. A nontrivial fragment of English

In this section we give a fragment of English relevant to the hospital database just introduced. This fragment merely serves as an illustration and is not intended to account for each and every word or phrase in English that might be relevant to this hospital. The application-dependent production rules presented here must, of course, be used in combination with the rule forms given in sections 7.1 and 7.3.

The translation of the present fragment is given in section A.4.

```
S01  <BN;pl;σ>      ::= <BN;si;σ>S

S02  <BN;si;P>      ::= patient|subject
S03  <BN;si;EMP>    ::= employee
S04  <BN;si;NRS>    ::= nurse
S05  <BN;si;SP>     ::= specialist
S06  <BN;si;DPT>    ::= department
S07  <BN;si;RM>     ::= nursing room
S08  <BN;si;ADM>    ::= admittance
S09  <BN;si;REL>    ::= release
S10  <BN;si;TM>     ::= treatment
S11  <BN;si;PT>     ::= [patient]treatment
S12  <BN;si;M>      ::= medicine
S13  <BN;si;MP>     ::= [medicine]prescription

S14  <DJ;P>         ::= female
S15                     |male
S16  <DJ;ADM>       ::= past
S17                     |current
S18  <DJ;M>         ::= dangerous

S19  <FU;NRS;DPT>   ::= department
S20  <FU;NRS;NRS>   ::= supervisor
S21  <FU;SP;DPT>    ::= department
S22  <FU;SP;SP>     ::= locum[tenens]
S23  <FU;DPT;NRS>   ::= head nurse
S24  <FU;DPT;SP>    ::= head specialist

S25  <FU;ADM;SP>    ::= responsible specialist
S26  <FU;ADM;P>     ::= subject
S27  <FU;MP;P>      ::= subject
S28  <FU;MP;SP>     ::= prescribing specialist
S29  <FU;MP;M>      ::= medicine
S30  <FU;PT;P>      ::= subject
S31  <FU;PT;SP>     ::= treating specialist
S32  <FU;PT;SP>     ::= assistent
S33  <FU;PT;ADM>    ::= underlying admittance

S34  <AR;P;int>     ::= number
S35                     |date of birth
S36  <AR;P;str>     ::= name
S37                     |address
S38                     |city
S39  <AR;P;g>       ::= sex
S40  <AR;P;bg>      ::= blood group
S41  <AR;P;r>       ::= rhesus factor
S42  <AR;EMP;int>   ::= number
S43                     |salary
S44  <AR;EMP;str>   ::= name
S45                     |address
S46                     |city
S47  <AR;NRS;int>   ::= number
S48                     |department number
S49                     |number of the supervisor
```

```
S50  <AR;SP;int>   ::= number
S51                    |department number
S52                    |number of the locum
S53                    |number of beds
S54                    |status
S55  <AR;DPT;int>  ::= number
S56                    |number of the head nurse
S57                    |number of the head specialist
S58  <AR;DPT;str>  ::= name
S59  <AR;RM;int>   ::= room number
S60                    |department number
S61                    |number of beds
S62  <AR;ADM;int>  ::= patient number
S63                    |number of the subject
S64                    |number of the responsible specialist
S65                    |room number
S66                    |date of admittance
S67  <AR;ADM;str>  ::= reason
S68  <AR;REL;int>  ::= patient number
S69                    |number of the subject
S70                    |date of admittance
S71                    |date of release
S72                    |invoice amount
S73  <AR;TM;int>   ::= tariff
S74  <AR;TM;str>   ::= code
S75                    |name
S76                    |sort
```

```
S77  <AR;PT;int>    ::= number of the subject
S78                     |number of the treating specialist
S79                     |number of the assistent
S80                     |[treatment]date
S81                     |length
S82  <AR;PT;str>    ::= [treatment]code
S83                     |treatment room
S84  <AR;M;int>     ::= danger code
S85  <AR;M;str>     ::= code
S86                     |name
S87                     |sort
S88  <AR;MP;int>    ::= starting date
S89                     |length
S90                     |frequency
S91  <AR;MP;str>    ::= code

S92  <CF;σ;σ>       ::= [all of] the data of

S93  <CE;int>       ::= <int.>
S94  <CE;str>       ::= '<char.list>'
S95                     |<char.list>
S96  <char.list>    ::= <char.>
S97                     |<char.list><char.>
S98  <CE;bg>        ::= O | A | B | AB

S99  <SE;int>       ::= the total number of beds
```

```
S100  <PE;P>    ::= <BN;si;P><CE;int>
S101               |the <BN;si;P> with number <CE;int>
S102  <PE;EMP> ::= <BN;si;EMP><CE;int>
S103  <PE;NRS> ::= <BN;si;NRS><CE;int>
S104  <PE;SP>  ::= <BN;si;SP><CE;int>
S105  <PE;RM>  ::= <BN;si;RM><CE;int>
S106  <PE;DPT> ::= <BN;si;DPT><CE;int>
S107               |the <CE;str><BN;si;DPT>
S108  <PE;TM>  ::= <BN;si;TM><CE;str>
S109               |the <BN;si;TM> with code <CE;str>
S110               |the <CE;str><BN;si;TM>
S111  <PE;M>   ::= <BN;si;M><CE;str>
S112               |the <BN;si;M><CE;str>
S113  <PE;ADM> ::= the <BN;si;ADM> of <PN;P> on <CE;date>
S114  <PE;REL> ::= the <BN;si;REL> of <PN;P> on <CE;date>

S115  <PJ;P>    ::= with blood group <CE;bg>
S116               |called <CE;str>
S117               |from <CE;str>
S118  <PJ;EMP> ::= called <CE;str>
S119               |from <CE;str>
S120  <PJ;NRS> ::= with department number <CE;int>
S121  <PJ;SP>  ::= with department number <CE;int>
S122               |with <CE;int> beds
S123  <PJ;RM>  ::= with department number <CE;int>
S124               |with <CE;int> beds
S125  <PJ;ADM> ::= for [reason of] <CE;str>
S126               |with patient number <CE;int>
```

```
S127  <PJ;ADM> ::= with specialist number <CE;int>
S128               |for room number <CE;int>
S129  <PJ;TM>  ::= of sort <CE;str>
S130  <PJ;PT>  ::= with code <CE;str>
S131               |with patient number <CE;int>
S132               |with specialist number <CE;int>
S133               |with assistent number <CE;int>
S134               |in treatment room <CE;str>
S135               |of <CE;int> minutes
S136  <PJ;M>   ::= of sort <CE;str>
S137               |with danger code <CE;int>
S138  <PJ;MP>  ::= of [medicine] code <CE;str>
S139               |for patient number <CE;int>
S140               |with specialist number <CE;int>
S141               |for <CE;int> days
S142               |for <CE;int> times a day
S143               |of <CE;int> units per time
S144               |of <CE;int> units a day
S145               |of <CE;int> units
S146  <PJ;DPT> ::= with <CE;int> beds
S147               |with <BD;p><CN;u;NRS>
S148               |with <BD;p><CN;u;SP>
S149               |with <BD;p><CN;u;RM>

S150  <PJ;NRS> ::= of <PN;DPT>
S151               |under supervision of <PN;NRS>
S152  <PJ;SP>  ::= of <PN;DPT>
S153  <PJ;RM>  ::= of <PN;DPT>
```

```
S154  <PJ;MP>   ::= of <PN;M>
S155                |by <PN;SP>
S156                |to <PN;P>
S157  <PJ;PT>   ::= of <PN;P>
S158                |by <PN;SP>
S159                |assisted by <PN;SP>
S160                |during <PN;ADM>
S161  <PJ;ADM>  ::= of <PN;P>
S162                |by <PN;SP>
S163                |to <PN;RM>
S164                |to <PN;DPT>
S165  <PJ;REL>  ::= of <PN;P>

S166  <PJ;ADM>  ::= <JV;date>
S167  <PJ;REL>  ::= <JV;date>
S168  <PJ;PT>   ::= <JV;date>
S169  <PJ;MP>   ::= <JV;date>

S170  <AV;ADM>  ::= for <CE;str>
S171                |to <PN;RM>
S172                |to <PN;DPT>
S173                |<JV;date>
S174  <AV;REL>  ::= <JV;date>
S175  <AV;PT>   ::= in treatment room <CE;str>
S176                |during <PN;ADM>
S177                |<JV;date>
S178  <AV;MP>   ::= for <CE;int> days
S179                |to <PN;P>
```

```
S180  <EI;μ;σ;σ'>      ::= <W;μ><PP;τ;σ;σ'>[by <PN;τ>]
S181  <ET;μ;τ;σ;σ'>    ::= <H;μ><PP;τ;σ;σ'>

S182  <W;si>           ::= was
S183  <W;pl>           ::= were
S184  <H;si>           ::= has
S185  <H;pl>           ::= have

S186  <PP;SP;P;ADM>    ::= admitted
S187  <PP;SP;P;PT>     ::= treated
S188  <PP;SP;SP;PT>    ::= assisted
S189  <PP;SP;TM;PT>    ::= performed
S190  <PP;P;TM;PT>     ::= undergone
S191  <PP;P;M;MP>      ::= used
S192  <PP;SP;M;MP>     ::= prescribed

S193  <EI;μ;P;REL>     ::= <W;μ> released

S194  <IV;μ;P>         ::= <L;μ> in <CE;str>
S195  <IV;μ;EMP>       ::= <L;μ> in <CE;str>
S196  <L;si>           ::= lives
S197  <L;pl>           ::= live

S198  <IV;μ;SP>        ::= <C;μ><PR2> duty
S199  <PR2>            ::= on
S200                       |off
```

A.4. Translating the nontrivial fragment of English

In this section we give translation rules for each of the production rules introduced in section A.3.

S01: $M(x,\alpha\mathbf{S}) = M(x,\alpha)$.

S02 - S13: In each of these cases, the root label of the derivation tree is of the form <BN;<u>si</u>;σ>. The translation of such a derivation tree α in combination with a placeholder x of type σ is as follows:

$M(x,\alpha) = x \in {}^{\mathbf{v}}\sigma$

S14: $M(x,$**female**$)$ = **((x.SX) = ♀)**

S15: $M(x,$**male**$)$ = **((x.SX) = ♂)**

S16: $M(x,$**past**$)$ = **((x.ARCH) = 1)**

S17: $M(x,$**current**$)$ = **((x.ARCH) = 0)**

S18: $M(x,$**dangerous**$)$ = **((x.DGCD) ≥ 16)**

S19 - S33: In each of these cases, the root label of the derivation tree is of the form <FU;σ;σ'>. The translation of such a derivation tree α is of the form

$M(\alpha) = {}^{\mathbf{v}}C$

where C is a connector index with source index σ and target index σ', i.e., C ε dom(Hhsp) and Hhsp(C) = (σ;σ'). For each of the 15 cases, the corresponding connector index is given below.

S19	**NRS-DPT**	S23	**DPT-NRS**	S27	**MP-P**	S30	**PT-P**
S20	**NRS-NRS**	S24	**DPT-SP**	S28	**MP-SP**	S31	**PT-SP**
S21	**SP-DPT**	S25	**ADM-SP**	S29	**MP-M**	S32	**PT-AST**
S22	**SP-SP**	S26	**ADM-P**			S33	**PT-ADM**

S34 - S91: The translation of a derivation tree α with a root label of the form <AR;σ;σ'> will result in an attribute of the table index σ, i.e., $M(\alpha) \in$ Ghsp(σ). For each of the 58 cases, $M(\alpha)$ is given below.

S34	**PNR**	S46	**CTY**	S58	**DNM**	S70	**INDTE**	S82	**TCD**
S35	**BDT**	S47	**NNR**	S59	**RNR**	S71	**RDTE**	S83	**TRM**
S36	**NM**	S48	**DNR**	S60	**DNR**	S72	**INVAM**	S84	**OGCD**
S37	**AD**	S49	**SYNR**	S61	**NB**	S73	**TAR**	S85	**MCD**
S38	**CTY**	S50	**ENR**	S62	**PNR**	S74	**TCD**	S86	**MNM**
S39	**SX**	S51	**ONR**	S63	**PNR**	S75	**TNM**	S87	**MSRT**
S40	**BLG**	S52	**LOC**	S64	**SNR**	S76	**TSRT**	S88	**DTE**
S41	**RHF**	S53	**NB**	S65	**RNR**	S77	**PNR**	S89	**LENGTH**
S42	**ENR**	S54	**OND**	S66	**INDTE**	S78	**SNR**	S90	**FD**
S43	**SAL**	S55	**DNR**	S67	**RSN**	S79	**ASNR**	S91	**MCD**
S44	**NM**	S56	**NNR**	S68	**PNR**	S80	**DTE**		
S45	**AD**	S57	**ENR**	S69	**PNR**	S81	**LENGTH**		

S92: $\mathcal{M}(x,\alpha) = x$

S93: The set of "integer-valued" constant expressions will be the language generated by the grammar presented in example 3.1. Strictly speaking, the 16 production rules of that grammar must be added here as well. The translation rule for S93 is

$\mathcal{M}(\alpha) = \mathcal{Fr}(\alpha)$

Thus, $\mathcal{M}(\alpha)$ will be the frontier of the tree α (see chapter 3), i.e., the original text.

S94: $\mathcal{M}(\text{‘}\alpha\text{’}) = \text{‘}\mathcal{Fr}(\alpha)\text{’}$

S95: $\mathcal{M}(\alpha) = \text{‘}\mathcal{Fr}(\alpha)\text{’}$

S96 and S97: The production rules for <char.> still have to be added. No translation rules are needed for S96 and S97. This follows from the translation rules for S94 and S95.

S98: For each of the 4 cases, the translation is

$\mathcal{M}(\alpha) = \mathcal{Fr}(\alpha)$

S99: $\mathcal{M}(\alpha) = \boldsymbol{\Sigma}\ y \in \check{\mathbf{RM}}\textbf{:}\ (y.\mathbf{NB})$

S100 - S112: Here, each derivation tree α with a root label of the form <PE;σ> has two direct subtrees, namely, a derivation tree β with root label <BN;<u>si</u>;σ> and a derivation tree γ with a root label of the form <CE;σ'>. The translation of α in combination with a placeholder x of type σ is

$M(x,\alpha) = \boldsymbol{\sigma}\, M(x,\beta) \wedge ((x.b) = M(\gamma))$**:**

where b, an attribute of the table index σ, is given below:

S100	**PNR**	S102	**ENR**	S105	**RNR**	S108	**TCD**	S111	**MCD**
S101	**PNR**	S103	**NNR**	S106	**DNR**	S109	**TCD**	S112	**MNM**
		S104	**ENR**	S107	**DNM**	S110	**TNM**		

S113: $M(x,$**the** α **of** β **on** $\gamma) =$

$M(y,\beta)\boldsymbol{\sigma}\, x \in (^{\vee}$**ADM-P inv** $y) \wedge ((x.$**INDTE**$) = M(\gamma))$**:**

S114: $M(x,$**the** α **of** β **on** $\gamma) =$

$\boldsymbol{\sigma}\, M(x,\alpha) \wedge ((x.$**RDTE**$) = M(\gamma)) \wedge ((x.$**PNR**$) = M(y,\beta)(y.$**PNR**$))$**;**

S115 - S143: Here, each derivation tree α with a root label of the form <PJ;σ> has one direct subtree, namely, a derivation tree β with a root label of the form <CE;σ'>. The translation of α in combination with a placeholder x of type σ is

$M(x,\alpha) = ((x.b) = M(\beta))$

where b, an attribute of the table index σ, is given below:

S115	**BLG**	S121	**DNR**	S127	**SNR**	S133	**ASNR**	S139	**PNR**
S116	**NM**	S122	**NB**	S128	**RNR**	S134	**TRM**	S140	**SNR**
S117	**CTY**	S123	**DNR**	S129	**TSRT**	S135	**LENGTH**	S141	**LENGTH**
S118	**NM**	S124	**NB**	S130	**TCD**	S136	**MSRT**	S142	**FD**
S119	**CTY**	S125	**RSN**	S131	**PNR**	S137	**DGCD**	S143	**AMT**
S120	**DNR**	S126	**PNR**	S132	**SNR**	S138	**MCD**		

S144: $M(x,$**of** α **units a day**$) = (((x.$**AMT**$) \times (x.$**FD**$)) = M(\alpha))$

S145: $M(x,$**of** α **units**$) = ((((x.$**AMT**$) \times (x.$**FD**$)) \times (x.$**LENGTH**$)) = M(\alpha))$

S146: $M(x,$**with** α **beds**$) = (M(\alpha) = \boldsymbol{\Sigma}\, y \in (^{\vee}$**RM-DPT inv** $x)$**:** $(y.$**NB**$))$

S147 - S149: Here, the translation is of the form

$M(x,$**with** $\alpha\beta) = M(\alpha) y \in (^{\vee}C$ **inv** $x)$**:** $M(y,\beta)$

where C is the "appropriate" connector index:

S147	**NRS-DPT**
S148	**SP-DPT**
S149	**RM-DPT**

S150 - S163: Here, each derivation tree α with a root label of the form <PJ;σ> has one direct subtree, namely, a derivation tree β with a root label of the form <PN;σ'>. A correct (but not "optimal") translation of α in combination with a placeholder x of type σ is

$$M(x,\alpha) = M(y,\beta)\mathbf{((}^{\vee}C \ @ \ x\mathbf{)} = y\mathbf{)}$$

where C is a connector index with source index σ and target index σ'. For each of the 14 cases, the corresponding connector index is given below.

S150	**NRS-DPT**	S154	**MP-M**	S157	**PT-P**	S161	**ADM-P**
S151	**NRS-NRS**	S155	**MP-SP**	S158	**PT-SP**	S162	**ADM-SP**
S152	**SP-DPT**	S156	**MP-P**	S159	**PT-AST**	S163	**ADM-RM**
S153	**RM-DPT**			S160	**PT-ADM**		

For 13 of the 14 rules - not for S160 - $\mathbf{((}^{\vee}C \ @ \ x\mathbf{)} = y\mathbf{)}$ can be replaced by $\mathbf{((}x.a\mathbf{)} = \mathbf{(}y.a'\mathbf{))}$ where a and a' are the following attributes (of the table indices σ and σ', respectively):

	a	a'
S150	**DNR**	**DNR**
S151	**SVNR**	**NNR**
S152	**DNR**	**DNR**
S153	**DNR**	**DNR**
S154	**MCO**	**MCO**
S155	**SNR**	**ENR**
S156	**PNR**	**PNR**

	a	a'
S157	**PNR**	**PNR**
S158	**SNR**	**ENR**
S159	**ASNR**	**ENR**
S161	**PNR**	**PNR**
S162	**SNR**	**ENR**
S163	**RNR**	**RNR**

S164: $M(x,\textbf{to}\ \beta) = M(y,\beta)\mathbf{(((}^{\vee}\mathbf{ADM\text{-}RM}\ @\ x\mathbf{).DNR)} = \mathbf{(}y\mathbf{.DNR))}$

S165: $M(x,\textbf{of}\ \beta) = M(y,\beta)\mathbf{((}x\mathbf{.PNR)} = \mathbf{(}y\mathbf{.PNR))}$

S166 - S169: Here, the translation is of the form

$$M(x,\alpha) = \mathbf{[}y \leftarrow \mathbf{(}x.b\mathbf{)]}M(y,\alpha)$$

where b is the "appropriate" attribute:

S166	**INDTE**
S167	**RDTE**
S168	**DTE**
S169	**DTE**

S170 - S179: We note that each of these adverbial phrases is *also* an appositive adjectival phrase. This follows from

S125, S163, S164, S166, S167, S134, S160, S168, S141, and S156,

respectively. The translation of each adverbial phrase will be the same as the translation of the corresponding appositive adjectival phrase. For instance, by S141

S178: $\mathcal{M}(x,$ **for** β **days**$) = $ **(**$(x.$**LENGTH**$) = \mathcal{M}(\beta)$**)**

S180 - S181: These rule forms are of a more general nature than the other production rules. S180, in combination with S182 and S183, says that an existential intransitive verb phrase can be a past participle preceded by **was** or **were** and possibly followed by **by** and a proper noun phrase. (This is the *past tense* in the *passive voice*.) S181, in combination with S184 and S185, says that an existential transitive verb phrase can be a past participle preceded by **has** or **have** (thus, the *present perfect* tense in the *active* voice).

Because the proper noun phrase is optional in S180, we had to split the translation of a past participle in two: For a derivation tree β with a root label of the form $\langle PP;\tau;\sigma;\sigma'\rangle$ we have to define both $\mathcal{M}(x,x',\beta)$ and $\mathcal{M}(\beta,y,x')$, for all $y \in Plh(\tau)$, $x \in Plh(\sigma)$ and $x' \in Plh(\sigma')$. Thus, both

$Plh(\sigma) \times Plh(\sigma') \times \{\beta\} \subseteq dom(\mathcal{M})$ and
$\{\beta\} \times Plh(\tau) \times Plh(\sigma') \subseteq dom(\mathcal{M})$.

In continuation of section 8.1, we require that

$\mathcal{M}(x,x',\beta) \in Restr_B(x')$ and $FP_B(\mathcal{M}(x,x',\beta)) \subseteq \{x,x'\}$, and
$\mathcal{M}(\beta,y,x') \in We_B(\mathbf{t})$ and $FP_B(\mathcal{M}(\beta,y,x')) \subseteq \{y,x'\}$

if B is the CL-basis for the conceptual language we translate to. The translation rules for the rule forms S180 and S181 are:

S180: $\mathcal{M}(x,x',\alpha\beta) = \mathcal{M}(x,x',\beta)$
$\mathcal{M}(x,x',\alpha\beta$ **by** $\gamma) = (\mathcal{M}(x,x',\beta) \wedge \mathcal{M}(y,\gamma)\mathcal{M}(\beta,y,x'))$

S181: $\mathcal{M}(y,x,x',\alpha\beta) = (\mathcal{M}(x,x',\beta) \wedge \mathcal{M}(\beta,y,x'))$

The reason for splitting up now follows from the translation rules for S180.

S182 - S185: No translation rules are needed here.

S186 - S192: We recall that for a derivation tree β with a root label of the form $<PP;\tau;\sigma;\sigma'>$, both $M(x,x',\beta)$ and $M(\beta,y,x')$ have to be defined; here $y \in Plh(\tau)$, $x \in Plh(\sigma)$, and $x' \in Plh(\sigma')$. Our first translation rule is of the form

$M(x,x',\beta) = x' \in ({}^{\vee}C\ \mathbf{inv}\ x)$

where C is the "appropriate" connector index:

S186	**ADM-P**	S188	**PT-SP**	S190	**PT-TM**	S192	**MP-M**
S187	**PT-P**	S189	**PT-TM**	S191	**MP-M**		

The other translation rule is of the form:

$M(\beta,y,x') = ((x.a) = (y.b))$

where a and b are the following attributes:

	S186	S187	S188	S189	S190	S191	S192
a	**SNR**	**SNR**	**ASNR**	**SNR**	**PNR**	**PNR**	**SNR**
b	**ENR**	**ENR**	**ENR**	**ENR**	**PNR**	**PNR**	**ENR**

S193: $M(x,x',\alpha\ \mathbf{released}) = x' \in {}^{\vee}\mathbf{REL} \wedge ((x'.\mathbf{PNR}) = (x.\mathbf{PNR}))$

S194: $M(x,\alpha\ \mathbf{in}\ \beta) = ((x.\mathbf{CTY}) = M(\beta))$

S195: $M(x,\alpha\ \mathbf{in}\ \beta) = ((x.\mathbf{CTY}) = M(\beta))$

S196 - S197: No translation rules are needed (cf. S194 and S195).

S198: $M(x,\alpha\beta\ \mathbf{duty}) = ((x.\mathbf{ONO}) = M(\beta))$

S199: $M(\mathbf{on}) = \mathbf{1}$

S200: $M(\mathbf{off}) = \mathbf{0}$

REFERENCES

[AA 78] M.A. Arbib and S. Alagić: The design of well-structured and correct programs.
Springer-Verlag, New York, 1978.

[Ba 82] H. Balsters: A formal definition of derivation trees in systems of natural deduction.
Memorandum 82-20, Dep. of Math., Eindhoven University of Technology, 1982.

[BE 76] F.L. Bauer and J. Eickel (eds.): Compiler construction.
Springer-Verlag, New York, 1976.

[BM 77] J.L. Bell and M. Machover: A course in mathematical logic.
North-Holland, Amsterdam, 1977.

[CC 81] CODASYL Programming Language Committee:
COBOL Journal of Development, 1981.

[Da 81] C.J. Date: An introduction to database systems.
Addison-Wesley, Reading (Mass.), 1981.

[Dk 68] E.W. Dijkstra: Go to statement considered harmful.
Comm. of the ACM 11 (1968), pp. 147-148.

[Ea 70] J. Earley: An efficient context-free parsing algorithm.
Comm. of the ACM 13 (1970), pp. 94-102.

[He 84] C. Hemerik: Formal definitions of programming languages as a basis for compiler construction.
Doctoral thesis, Eindhoven University of Technology, 1984.

[HMS 73] J. Hintikka, J. Moravcsik, and P. Suppes (eds.):
Approaches to natural language.
Reidel, Dordrecht, 1973.

[HW 73] C.A.R. Hoare and N. Wirth: An axiomatic definition of the programming language PASCAL.
Acta Informatica 2 (1973), pp. 335-355.

[Kn 68] D.E. Knuth: Semantics of context-free languages.
Math. Systems Th. 2 (1968), pp. 127-145 and Math. Systems Th. 5 (1971), p. 95.

[Ko 71] C.H.A. Koster: Affix grammars.
In [Pe 71], pp. 95-109.

[MLB 76] M. Marcotty, H.F. Ledgard, and G.V. Bochmann:
A sampler of formal definitions.
Computing Surveys 8 (1976), pp. 191-276.

[Mo 73] R. Montague: The proper treatment of quantification in ordinary English.
In [HMS 73], pp. 221-242; reprinted in [Th 74], pp. 247-270.

[Pe 71] J.E.L. Peck (ed.): ALGOL 68 implementation.
North-Holland, Amsterdam, 1971.

[Re 84] F. Remmen: Fundamentals of databases.
Prentice-Hall, New York, 1984.

[Sc 83] R.J.H. Scha: Logical foundations for question answering.
Doctoral thesis, University of Groningen, 1983.

[Sh 67] J.R. Shoenfield: Mathematical logic.
Addison-Wesley, Reading (Mass.), 1967.

[SS 77] D.C.P. Smith and J.M. Smith:
Database abstractions: aggregation and generalization.
ACM Trans. on Database Systems 2 (1977), pp. 105-133.

[Th 74] R.H. Thomason (ed.): Formal philosophy.
Yale University Press, London, 1974.

[Ul 80] J.D. Ullman: Principles of database systems.
Computer Science Press, Potomac, 1980.

[vW 65] A. van Wijngaarden: Orthogonal design and description of a formal language.
Math. Centrum Report MR 76, Amsterdam, 1965.

[vW 75] A. van Wijngaarden: Revised report on the algorithmic language ALGOL 68.
Acta Informatica 5 (1975), pp. 1-236.

SUBJECT INDEX

SAMENVATTING

Dit proefschrift bestaat uit twee delen. In het eerste deel worden diverse *wiskundige modellen* voor gegevensbanken gedefinieerd. Het tweede deel is syntactisch van aard en gaat over *vraagtalen* voor gegevensbanken. De semantiek van deze vraagtalen berust op de modellen uit deel I.

In hoofdstuk 1 worden twee conceptuele modellen voor gegevensbanken geïntroduceerd. Een type 1 model komt min of meer overeen met het relationele model waarin dan wel alle statische "integrity constraints" zijn verwerkt. Een type 2 model is een type 1 model uitgebreid met (namen voor) *database-functies*. Het begrip database-functie speelt een centrale rol; zo is het belangrijkste verschil op conceptueel niveau tussen een geïntegreerde gegevensbank en een "losse" verzameling bestanden de aanwezigheid van database-functies.

In hoofdstuk 2 wordt een wiskundig (opslag)model gedefinieerd waarmee de semantiek van programma's werkend op "direct-sequentieel" toegankelijke gegevensbanken kan worden beschreven.

In het tweede deel - de hoofdstukken 3 t/m 8 - worden drie soorten vraagtalen (retrieval languages) beschouwd, te weten programmeertalen, conceptuele (of "logische") talen en fragmenten van een natuurlijke taal (in dit geval het Engels).

In hoofdstuk 3 worden ten behoeve van de volgende hoofdstukken enige basisbegrippen gedefinieerd betreffende formele grammatica's (met name kontekstvrije grammatica's en "two-level grammars").

In hoofdstuk 4 wordt een klasse van conceptuele vraagtalen gedefinieerd die zijn gebaseerd op de type 2 modellen uit hoofdstuk 1. Deze klasse bevat zowel talen in de stijl van "relational calculus" als talen in de stijl van "relational algebra" (maar ook "mengvormen").

In hoofdstuk 5 wordt een grammatica gegeven voor een klasse van "PASCAL-achtige" programmeertalen die ook een kleine maar krachtige verzameling "database statements" bevatten. De semantiek van deze statements wordt uitgelegd in termen van het model geïntroduceerd in

hoofdstuk 2. Aan het eind van hoofdstuk 5 wordt een vergelijking getrokken met twee bestaande programmeertalen bestemd voor gegevensverwerking.

In hoofdstuk 6 worden vertalingen gegeven van de niet-procedurele vraagtalen uit hoofdstuk 4 naar de procedurele vraagtalen uit hoofdstuk 5. Er wordt bij de vertalingen ook rekening gehouden met het specifieke database probleem van currency conflicten.

In hoofdstuk 7 wordt met behulp van een grammatica de algemene structuur van in het Engels geformuleerde vragen en opdrachten beschreven. Per toepassing moet deze grammatica worden uitgebreid met productieregels voor de woorden en uitdrukkingen die specifiek zijn voor die toepassing.

In hoofdstuk 8 wordt een vertaling gegeven van de structuren uit hoofdstuk 7 naar die uit hoofdstuk 4. Bovendien worden er in paragraaf 8.1 voorschriften gegeven voor de vorm van de vertaling van de per toepassing toe te voegen specifieke productieregels.

In het volgende schema wordt nog eens kort weergegeven in welke hoofdstukken de diverse talen en vertalingen worden behandeld; NT, CT en PT staan hierin achtereenvolgens voor natuurlijke taal, conceptuele taal en programmeertaal.

$$\begin{tikzcd}
NT \arrow[rr] & & CT \arrow[rr] & & PT \\
7 & 8 & 4 & 6 & 5
\end{tikzcd}$$

De appendix dient ter illustratie. Aan de hand van een (gefingeerde) ziekenhuis-organisatie worden een type 1 model en een type 2 model met een redelijke mate van complexiteit gedefinieerd. Bovendien wordt er een voor deze ziekenhuistoepassing geschikte uitbreiding van de grammatica uit hoofdstuk 7 gegeven. Tenslotte wordt dit Engelse fragment vertaald naar een vraagtaal op conceptueel niveau. De vertaling is in overeenstemming met de richtlijnen uit hoofdstuk 8.

CURRICULUM VITAE

De schrijver van dit proefschrift werd op 5 oktober 1953 in Groningen geboren. Vanaf 1966 bezocht hij de Rijks HBS te Groningen, waar hij in 1971 het diploma HBS-B behaalde. Daarna studeerde hij Wis- en Natuurkunde aan de Rijks Universiteit Groningen. In januari 1979 slaagde hij met lof voor het doctoraal examen Wiskunde met hoofdvak logica en grondslagen van de wiskunde. Sinds 5 februari 1979 is hij verbonden aan de onderafdeling der Wiskunde en Informatica van de Technische Hogeschool Eindhoven, waar hij werkzaam is bij Prof.dr. W. Peremans. Hij verricht er onderzoek op het gebied van databases, in samenwerking met drs. F. Remmen.

STELLINGEN

behorende bij het proefschrift

DATABASE MODELS AND RETRIEVAL LANGUAGES

van

E.O. DE BROCK

Eindhoven, 16 maart 1984

STELLINGEN

1. De stelling van Sagiv, Delobel, Parker en Fagin dat voor Boole-afhankelijkheden logische consequentie en database consequentie equivalent zijn, is onjuist.

   Y. Sagiv, C. Delobel, D.S. Parker en R. Fagin:
   An equivalence between relational dependencies and a fragment of propositional logic.
   Journal of the ACM 28 (1980), 435-453.

   E.O. de Brock: On complete proof systems, with an application to positive and Boolean dependencies in database relations.
   Memorandum, onderafd. WSK/I, TH Eindhoven, 1982.

2. Verzamelingenleer is het middel bij uitstek voor het definiëren van database-modellen.

   Deel I van dit proefschrift.

3. Het automatisch vertalen van een opdracht in natuurlijke taal naar een computerprogramma dat die opdracht uitvoert, kan op een overzichtelijke en wiskundig goed onderbouwde manier gebeuren.

   Deel II van dit proefschrift.

4. Met behulp van "two-level grammars" kunnen grote delen van natuurlijke talen op een natuurlijke wijze worden beschreven. De lapmiddelen van de transformationele grammatica zijn derhalve overbodig.

   Hoofdstuk 7 van dit proefschrift.

5. Currency-conflicten kunnen op een systematische manier worden opgelost.

   Hoofdstukken 5 en 6 van dit proefschrift.

6. Het in de database-literatuur voorkomende begrip "eerste normaalvorm" (en daarmee ook elke daarop gebaseerde normaalvorm) is niet goed gedefinieerd.

7. Het is opmerkelijk dat vrijwel alle toonaangevende leerboeken op het gebied van databases nog altijd beginnen met een hoofdstuk over "physical organization".

8. Het classificeren van objectsystemen in de praktijk (zoals bijvoorbeeld persoonsregistratiesystemen, archiefsystemen, voorraadsystemen en stuklijstsystemen) is niet de aangewezen werkwijze om te komen tot een fundamentele theorie van gegevensmodellen.

9. Voor te veel "optimizers" is de naam "optimizer" veel te optimistisch.

10. De lege verzameling speelt vaak een grotere rol dan haar omvang doet vermoeden.

11. De elfde stelling is bij uitstek geschikt als schertsstelling.