DATABASE MODELS
AND
RETRIEVAL LANGUAGES

PROEFSCHRIFT

VIEREN VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE
TECHNISCHE WETENSCHAPPEN AAN DE TECHNISCHE
HOGEESCHOOL EINDHOVEN, OP GEZAG VAN DE RECTOR
MAGNIFICUS, PROF. DR. S. T. M. ACKERMANS, VOOR
EEN COMMISSIE AANGEWEEZEN DOOR HET COLLEGE
VAN DEKANEN IN HET OPENBAAR TE VERDEEDIGEN 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. Kruiseman Areta
Aan mijn ouders
CONTENTS

General introduction and summary 1

Part I. Database models

0. Preliminaries 1
1. Two conceptual database models 4
  1.1. Type 1 models 4
  1.2. Type 2 models 6
2. Sequential storage structures 11

Part II. Some retrieval languages for databases

3. Grammars 15
4. Conceptual languages 20
  4.0. Introduction and summary 20
  4.1. CL-bases 20
  4.2. Queries 26
  4.3. Examples of queries 34
5. A class of programming languages 37
  5.0. Introduction and summary 37
  5.1. The common syntax 38
  5.2. Commentary on the syntax 40
  5.3. A comparison with PASCAL and with the CBTU proposal 50
6. Translating conceptual languages into programming languages 54
  6.0. Introduction and summary 54
  6.1. Translating types 55
  6.2. Two auxiliary functions for determiners 55
  6.3. Translating queries into programs 57
  6.4. An example 62
  6.5. Some improvements 65
  6.6. Translation rules concerning connector indices 72
7. The structure of queries in English 80
   7.0. Introduction and summary 80
   7.1. The general syntax 80
   7.2. Commentary on the syntax 85
   7.3. Intermediate forms 92
8. Translating fragments of English into conceptual languages 94
   8.0. Introduction and summary 94
   8.1. On the form of the translation 94
   8.2. Translating the general rules 99
   8.3. Some examples 105
   8.4. Translating the intermediate forms 107

Appendix. A nontrivial example 110
   A1. A nontrivial type 1 model 110
   A2. A nontrivial type 2 model 120
   A3. A nontrivial fragment of English 123
   A4. Translating the nontrivial fragment of English 128

References 134

Subject index 136

Samenvatting 139

Curriculum vitae 141
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 grammar. 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). For considerable 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. ML, CL, and PL stand for natural, conceptual, and programming language, respectively:

\[
\begin{align*}
&\text{ML} \rightarrow \text{CL} \rightarrow \text{PL} \\
&7 \rightarrow 8 \rightarrow 4 \rightarrow 6 \rightarrow 5
\end{align*}
\]
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:

\[ \text{NL}_1 \rightarrow \text{CL} \rightarrow \text{PL} \]

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

\[ \text{NL}_1 \rightarrow \text{PL}_1 \rightarrow \text{PL}_2 \rightarrow \text{PL}_3 \rightarrow \text{PL}_4 \]

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

The interdependence of the chapters is as follows:

\[ 6 \]
\[ 7 \]
\[ 4 \]
\[ 5 \]
\[ 2 \]
\[ 1 \]
\[ 0 \]

Since the well-known supplier-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 $\square$ is used to indicate the end of an example, that "iff" stands for "if and only if", and that the symbols $\sqsubset$ and $\sqsupset$ 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 relation if R is a set of ordered pairs. If R is a relation then:

\[ \text{dom}(R) = \{ x \mid (x, y) \in R \}, \text{ called the domain of } R; \]

\[ \text{rng}(R) = \{ y \mid (x, y) \in R \}, \text{ called the range of } R; \]

\[ R^{-1} = \{ (y, x) \mid (x, y) \in R \}, \text{ called the inverse of } R. \]

F is a function if 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 \(x \in \text{dom}(F)\) then we denote the unique \(y\) for which \((x, y) \in F\) by \(F(x)\), as usual, or sometimes by \(F_x\). By the pre-image of \(y\) under \(F\) we mean the set \(\{ x \in \text{dom}(F) \mid F(x) = y \}\). We note that \(\emptyset\), the empty set, is also a function and that \(\text{dom}(\emptyset) = \text{rng}(\emptyset) = \emptyset\).

If \(f\) and \(g\) are functions then:

\[ (g \circ f)(x) = g(f(x)) \mid x \in \text{dom}(f) \text{ and } f(x) \in \text{dom}(g), \]

called the composition of \(g\) after \(f\).

If \(f\) is a function and \(A\) is a set then:

\[ f \upharpoonright A = \{ (x, y) \mid x \in A \}, \text{ i.e., } f \text{ restricted to } A. \]

If \(T\) is a set of functions and \(A\) is a set then:

\[ T \upharpoonright A = \{ f \upharpoonright A \mid f \in T \}, \text{ i.e., } T \text{ projected on } A. \]

If \(A\) is a set then:

\(f\) is a function over \(A \iff f\) is a function and \(\text{dom}(f) = A;\)

\(f\) is a function into \(A \iff f\) is a function and \(\text{rng}(f) \subseteq A;\)

\(f\) is a function onto \(A \iff f\) is a function and \(\text{rng}(f) = A.\)
If A and B are sets then:

\[ A \rightarrow B \overset{\Delta}{=} \{ f \mid f \text{ is a function and } \text{dom}(f) = A \text{ and } \text{rng}(f) \subseteq B \} \],
known as the set of all functions from A into B.

\( F \) is an injection if and only if \( F \) is a function and
\[ \forall x \in \text{dom}(F) : \forall x' \in \text{dom}(F) : \text{if } F(x) = F(x') \text{ then } x = x' \]. An injection is also called a one-to-one function.

If \( n \in \mathbb{N} \) then:
\( F \) is an \( n \)-tuple if \( F \) 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: \( <x> \) denotes the 1-tuple \( F \) defined by \( F(0) = x \). \( <x,y> \) denotes the 2-tuple \( G \) defined by \( G(0) = x \) and \( G(1) = y \), etc.

\( F \) is a sequence if \( \exists n \in \mathbb{N} : F \) 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 \( f \circ g \), the concatenation of \( f \) and \( g \), as follows:
when \( n \) is the length of \( f \) and \( m \) is the length of \( g \) then \( f \circ g \) is the function over \( \{ k \in \mathbb{N} \mid k < n+m \} \) defined by
\[ f \circ g(k) = \begin{cases} f(k) & \text{if } 0 \leq k < n, \\ g(k-n) & \text{if } n \leq k < n+m. \end{cases} \]

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

\[ \text{Gconc}(\emptyset) = \emptyset, \]
\[ \text{Gconc}(\gamma \circ <q>) = \text{Gconc}(\gamma) \circ q \] where \( q \) is a sequence and \( \gamma \) is a sequence of sequences.

If \( A \) is a set then:

\[ A^* \overset{\Delta}{=} \{ f \mid f \text{ is a sequence and } \text{rng}(f) \subseteq A \}; \]
\[ A^\infty \overset{\Delta}{=} A^* - \{ \emptyset \}; \]
\( F \) is an enumeration of \( A \) if \( F \) is a sequence and \( \text{rng}(F) = A \) and \( F \) is one-to-one. We note that if there is an enumeration of \( A \) then \( A \) is a \( \mathbb{N} \)-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:

\[ \bigcup W \overset{\text{def}}{=} \{ x \mid \text{Each: } x \in A \}, \]

called the generalized union of $W$.

$P$ is a set function if $P$ is a function and $\text{VXdom}(P): P(x)$ is a set\(^{(1)}\).

If $F$ is a set function then:

\[ \Pi F \overset{\text{def}}{=} \{ f \mid f \text{ is a function over } \text{dom}(F) \text{ and } \forall x \in \text{dom}(F): f(x) \in F(x) \}, \]

called the generalized product of $F$.

$P$ is a function-valued function if $P$ is a function and $\text{VXdom}(P): P(x)$ is a function.

If $F$ and $G$ are function-valued functions then:

\[ G \circ F \overset{\text{def}}{=} \{ (x; G(x) \circ F(x)) \mid x \in \text{dom}(F) \cap \text{dom}(G) \}, \]

called the generalized composition of $G$ after $F$. In other words, $G \circ F$ is the function over $\text{dom}(F) \cap \text{dom}(G)$ defined by $G \circ F(x) = G(x) \circ F(x)$ for every $x \in \text{dom}(F) \cap \text{dom}(G)$.

\(^{(1)}\) 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 table over $A$ \( \iff \) $T$ is a set of functions over $A$.

Example 1.1: Figure 1.1(a) shows a table $T_1$ over $A_1 = \{DPT, NAME, NR, SAL, SEX\}$ and figure 1.1(b) shows a table $T_2$ over $A_2 = \{DNR, NAME, MAN\}$. The table $T_2$, for instance, consists of the two functions $t = \{(DNR, 5), (MAN, 7), (NAME, planning)\}$ and $t' = \{(MAN, 9), (DNR, 7), (NAME, production)\}$. Indeed $\text{dom}(t) = \text{dom}(t') = A_2$, as required by the definition. Furthermore, $t(DNR) = 5$, $t(MAN) = 7$, and so forth. (We note that MAN stands for "manager").

\[
\begin{array}{|c|c|c|c|c|}
\hline
NR & NAME & SAL & SEX & DPT \\
\hline
0 & Smith & 1200 & \checkmark & 7 \\
7 & Jones & 1300 & \checkmark & 5 \\
9 & Brown & 1300 & \checkmark & 7 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|c|}
\hline
DNR & NAME & MAN \\
\hline
5 & planning & 7 \\
7 & production & 9 \\
\hline
\end{array}
\]

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 $v_1$ over, say, $(\text{EMP}, \text{DEP})$ defined by $v_1(\text{EMP}) = T_1$ and $v_1(\text{DEP}) = T_2$, thus distinguishing the employee table from the department table. If $g_1$ is the function over $(\text{DEP}, \text{EMP})$ defined by $g_1(\text{EMP}) = A_1$ and $g_1(\text{DEP}) = A_2$, then $v_1$ is what we call a $DB$ snapshot over $g_1$. We define this notion for arbitrary set functions $g$:

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

$v$ is a DB snapshot over $g \iff v$ is a function over $\text{dom}(g)$ and $\forall z \in \text{dom}(g) \cdot v(z)$ is a table over $g(z)$. 

4
If \( v \) is a DB snapshot over a set function \( g \) then \( \langle g; v \rangle \) is called a type 1 snapshot.

In our example, \( v_1 \) represents the state of affairs of the company at one particular moment. The state of affairs at another moment will be represented by another function \( v_2 \); \( v_2 \) must also be a function over \([\text{DEP}, \text{EMPL}]\) such that \( v_2(\text{DEP}) \) is a table over \( A_2 \) and \( v_2(\text{EMPL}) \) is a table over \( A_1 \). In other words, \( v_2 \) must also be a DB snapshot over \( g_1 \). Indeed, each possible state of affairs of the company can be represented by a DB snapshot over \( g_1 \). On the other hand, not every DB snapshot over \( g_1 \) — 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 rules of that company of course — is an example of what we call a DB universe over \( g_1 \). Our definition of this notion is rather general:

D1.3: If \( g \) is a set function then:

\[ U \text{ is a DB universe over } g \iff U \text{ is a set of DB snapshots over } g. \]

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: \( \langle g; U \rangle \) is a type 1 model \( \iff g \) is a set function and

\[ U \text{ is a DB universe over } g. \]

If \( \langle g; U \rangle \) is a type 1 model then \( g \) is called the conceptual skeleton of \( \langle g; U \rangle \). By a table index of \( \langle g; U \rangle \) we mean an element of \( \text{dom}(g) \). If \( E \) is a table index of \( \langle g; U \rangle \) then \( g(E) \) is called the heading of \( E \) in \( \langle g; U \rangle \) and each element of \( g(E) \) is called an attribute of \( E \) in \( \langle g; U \rangle \).

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 \( B \subseteq A \) and \( T \) is a table over \( A \), then:

\[ B \text{ is u.i. for } T \iff \forall \langle t, t' \rangle \in T: \forall \langle x \rangle = t^B \iff \langle x \rangle = t'^B \text{ then } t = t'. \]
D1.6: If \( q:U \) is a type 1 model and \( x \in \text{dom}(q) \) then:
\[ k \text{ is a key for } x \text{ in } q:U \quad \Rightarrow \quad \exists B : k \subseteq q(E) \quad \text{and} \quad \forall \text{v:} U, B \text{ is a.k.r. for } v(x). \]

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. Type 2 models

Example 7.2: Let \( U_1 \) be a DB universe over the set function \( q_1 \) (introduced just before D1.2) such that

(1) \( \{ \text{DNR} \} \) is a key for \( \text{DEP} \) in \( q_1:U_1 \) (i.e., at each "moment" every department has a unique department number), and

(2) for every \( v \in U_1 \), every \( \text{DPT-value} \) in the table \( v(\text{EMPL}) \) also appears as a DNR-value in \( v(\text{DEP}) \) (i.e., at each moment every employee belongs to an "actual" department).

Then this induces for every \( v \) in \( U_1 \) a function \( F_1(v) \) from \( v(\text{EMPL}) \) into \( v(\text{DEP}) \), assigning to each employee tuple in "state" \( v \) the tuple of his department. We deliberately use the function notation \( F_1(v) \) because we will consider \( F_1 \) itself as a function too, a (function-valued) function over \( U_1 \). \( F_1 \) is an example of what we call a DB function, in this case a DB function within \( q_1:U_1 \) for the ordered pair \( (\text{EMPL}, \text{DEP}) \).

D1.7: If \( q:U \) is a type 1 model and \( (M;D) \subseteq \text{dom}(q) \) then:
\[ F \text{ is a DB function within } q:U \text{ for } (M;D) \quad \Rightarrow \quad F \text{ is a function over } U \text{ and } \forall v(U) : F(v) \in v(M) \Rightarrow v(D). \]

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 7.3: Assume that

(1) \( \{ \text{NR} \} \) is a key for \( \text{EMPL} \) in \( q_1:U_1 \) (from example 1.2) and
(2) for every \( v \in U \) the "MAN-column" \( \{ t(MAN) \mid t \in v(\text{DEF}) \} \) in \( v(\text{DEF}) \)

is a subset of the "NR-column" \( \{ t(NR) \mid t \in v(\text{EMPL}) \} \) in \( v(\text{EMPL}) \).

then there is a DB function \( F_2 \) within \( <g_1;U> \) for \( (\text{DEF},\text{EMPL}) \), namely the one for which \( F_2(v) \) assigns to each department tuple the tuple of its manager in "state" \( v \in U \). But then there also is a DB function \( F_3 \)

within \( <g_1;U> \) for \( (\text{EMPL},\text{EMPL}) \), namely the one for which \( F_3(v) \) assigns to each employee tuple in "state" \( v \) the tuple of the manager of his department. \( F_3 \) is an example of a "reflexive" DB function.

Other examples of DB functions can be found in the appendix. Our DB functions \( F_1 \) and \( F_2 \) 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, \( (M;D) \in \text{dom}(g) \times \text{dom}(\vec{g}), a \in g(M) \), \( a' \in g(D) \), and

\((C_1)\) \( a' \) is a key for \( D \) in \( <g;U> \) and

\((C_2)\) \( \{ t(a) \mid t \in v(M) \} \subseteq \{ t'(a') \mid t' \in v(D) \} \) for every \( v \in U \)

then the function \( F \) over \( U \) defined by

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

is a DB function within \( <g;U> \) for \( (M;D) \).

The proof is almost trivial: According to B.7 we still have to check that \( F(v) \in v(M) + v(D) \); well, \( F(v) \) is a function because of

\((C_1)\), \( \text{dom}(F(v)) = v(M) \) because of \( (C_2) \), and \( \text{rng}(F(v)) \subseteq v(D) \) is trivial.

If both \((C_1)\) and \((C_2)\) hold then \( a \) is sometimes called a foreign key, see for example [Da 81] or [Fe 84].

The DB functions \( F_3 \) in example 1.3 and \( \text{Emp}(\text{REL-ADM}) \) and \( \text{Emp}(\text{REL-ADM}) \) 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, \( (M,D,D') \in \text{dom}(g), \)

\( F \) is a DB function within \( <g;U> \) for \( (M,D) \), and

\( G \) is a DB function within \( <g;U> \) for \( (D,D') \)

then \( G \circ F \) is a DB function within \( <g;U> \) for \( (M,D') \).
Again, the proof is simple: Clearly, \( G \circ F \) is a function (see chapter 9), and \( \text{dom}(F) \cap \text{dom}(G) = \emptyset \). Furthermore, \( F(v) \in v(D) \) and \( G(v) \in v(D') \) for every \( v \in U \); thus \( G \circ F(v) = G(v) \circ F(v) \epsilon v(D) \rightarrow v(D') \). 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 \( \langle q; U \rangle \) on employees and departments:

\[ F3 = F2 \circ F1. \]

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 \( x \in \text{dom}(D) \) 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 \( x \in \text{dom}(D) \) 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: \( \langle q; h; g \rangle \) is a type 2 skeleton if \( g \) is a set function and \( h \) is a function into \( \text{dom}(g) \times \text{dom}(g) \) and \( \text{dom}(g) \cap \text{dom}(h) = \emptyset \).

D1.9: \( \langle q; h; U; l \rangle \) is a type 2 model if \( \langle q; h; g \rangle \) is a type 2 skeleton and \( U \) is a DB universe over \( g \) and \( l \) is a function over \( \text{dom}(h) \) and \( W \circ \text{dom}(h) : l(C) \) is a DB function within \( \langle q; U \rangle \) for \( h(C) \).
If \( \langle g; h \rangle \) is a type 2 skeleton, \( C \in \text{dom}(h) \), and \( h(C) = (M,D) \) then we call \( M \) the source index of \( C \) in \( \langle g; h \rangle \) and \( D \) the target index of \( C \) in \( \langle g; h \rangle \). Each element of \( \text{dom}(h) \) is called a connector index under \( \langle g; h \rangle \).

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 \( h \) be the function over \( \langle \text{DEPOF}, \text{MANAGEROF} \rangle \) defined by \( h(\text{DEPOF}) = (\text{EMPL}; \text{DEP}) \) and \( h(\text{MANAGEROF}) = (\text{DSP}; \text{EMPL}) \); then \( \langle g; h \rangle \) is a type 2 skeleton (where \( g \) is the set function introduced just before 3.1.2). In figure 1.2(a), the typing function \( h \) is depicted. The complete type 2 skeleton is depicted in figure 1.2(b).

![Diagram](image)

(a) Picture of \( h \). (b) Picture of \( \langle g; h \rangle \).

The connector index \( \text{DEPOF} \) is intended to refer to the DB function \( F_1 \) and \( \text{MANAGEROF} \) is intended to refer to \( F_2 \). The interpretation function \( I_1 \) over \( \langle \text{DEPOF}, \text{MANAGEROF} \rangle \) defined by \( I_1(\text{DEPOF}) = F_1 \) and \( I_1(\text{MANAGEROF}) = F_2 \) formally represents that interpretation. Now \( \langle g; h; U; I_1 \rangle \) is an example of a type 2 model.

If \( F \) is a DB function (within a type 1 model \( \langle g; U \rangle \)) for the ordered pair \( (M,D) \) and \( v \in U \) then \( F(v) \) is what we call a connector for \( (M,D) \) wrt. \( (M,D) \) wrt. \( v \). 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 connector for $(M,D)$ wrt. $v$ if $f \in v(M) \cup v(D)$.

With this terminology it is easy to formulate what the essential ingredients of a "snapshot" of a type 2 model $\langle g,h;U;V \rangle$ 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 $\langle g,h \rangle$ these ingredients constitute a type 2 snapshot:

D1.11: $\langle g,h;v;w \rangle$ is a type 2 snapshot if

- $\langle g,h \rangle$ is a type 2 skeleton and
- $v$ is a DB snapshot over $g$ and
- $w$ is a function over $\text{dom}(h)$ 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 \( E \in \text{dom}(v) \). More precisely (and without the noise in the previous sentence):

D2.1: If \( v \) is a set function then:

\[ \mu \] is an arrangement of \( v \) if \( \mu \) is a function over \( \text{dom}(v) \) and
\[ \forall E \in \text{dom}(v): \mu(E) \] is a one-to-one function onto \( v(E) \).

For "positions" (i.e., elements of \( \text{dom}(\mu(E)) \) for any \( E \in \text{dom}(v) \)) we may think of relative or absolute addresses, or so-called "database key values" (in which cases \( \text{dom}(\mu(E)) \) and \( \text{dom}(\mu(E')) \) 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:

\[ \mu \] is a sequential arrangement of \( v \) if \( \mu \)
\[ \nu \] is a function over \( \text{dom}(v) \) and
\[ \forall E \in \text{dom}(v): \nu(E) \] is an enumeration of \( v(E) \).

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

D2.3: If \( \mu \) is a function-valued function then:

\[ \tau \] is an ordering function for \( \mu \)
\[ \tau \] is a function over \( \text{dom}(\mu) \) and
\[ \forall E \in \text{dom}(\mu): \tau(E) \] is an enumeration of \( \text{dom}(\mu(E)) \).
We note that if \( \nu \) is an arrangement of a VB snapshot \( \nu \) and \( \tau \) is an ordering function for \( \nu \) then, after all, the generalized composition of \( \nu \) after \( \tau \), i.e., the function \( \{ (x y) (y) z (y) \mid \xi \in \text{dom}(\nu) \} \), constitutes a sequential arrangement of \( \nu \).

If \( f \) is a connector for a pair \( (M;D) \) w.r.t. a VB snapshot \( \nu \) and \( \mu \) is an arrangement of \( \nu \) then we have the situation as depicted in figure 2.1. (We recall from chapter 0 that we may write \( \nu_x \) instead of \( \nu (x) \) w.hich will be convenient here — and from D2.1 that the function \( \nu_D \) is one-to-one and onto \( \nu (D) \).)

A location link for \( (M;D) \) based on \( f \) and \( \mu \) determines for every "location" \( p \in \text{dom}(\mu_D) \) an enumeration of the locations of those tuples in \( \nu (M) \) that are mapped to \( \nu_D (p) \in \nu (D) \) by the function \( f \):

D2.4: If \( \nu \) is a set function and \( (M;D) \in \text{dom}(\nu) \times \text{dom}(\nu) \) and \( f \) is a connector for \( (M;D) \) w.r.t. \( \nu \) and \( \nu \) is an arrangement of \( \nu \) then:

1. \( f \) is a location link for \( (M;D) \) based on \( f \) and \( \nu \).

\[ f \circ \text{dom}(\nu_D) : \{ p \in \text{dom}(\nu_D) \mid f(\nu_D (p)) = \nu_D (p) \}. \]

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

In addition to an arrangement and an ordering function, a sequential storage structure for a type 2 snapshot \( \langle q; b; v; w \rangle \) will also
contain for every connector index $C \in \text{dom}(h)$ 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 $\langle q; h; v; w \rangle$ is a type 2 snapshot then:

- $\langle u; r; x \rangle$ is a sequential storage structure for $\langle q; h; v; w \rangle$
- $u$ is an arrangement of $v$ and
- $r$ is an ordering function for $u$ and
- $x$ is a function over $\text{dom}(h)$ and

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

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

- $r$ accounts for the distinction between tuples and their "locations" in a stored database;
- $r$ delivers, indirectly via $u$, for each table index $E$ an enumeration of the set $v(E)$;
- $K$, also indirectly via $u$, delivers, for each connector index $C \in \text{dom}(h)$, 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 $\langle q; h; u; I \rangle$ 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: \( <V; N; P; S> \) is a quasi-cfg if \( V \) is a set and \( N \subseteq V \) and \( P \subseteq N \times V^* \) and \( S \in N \).

If \( G \) is a quasi-cfg, say \( G = <V; N; P; S> \), then the following additional terminology and notations will be used:

(a1) by \( \text{Voc}(G) \) we mean \( V \), called the vocabulary of \( G \);
(a2) by \( \text{Nv}(G) \) we mean \( N \), called the nonterminal vocabulary of \( G \);
(a3) by \( \text{Tv}(G) \) we mean \( V - N \), called the terminal vocabulary of \( G \);
(a4) by \( \text{R}(G) \) we mean \( P \), called the rule set of \( G \);
(a5) by \( S(G) \) we mean \( S \), called the start symbol of \( G \);

(b1) \( A \) is a symbol of \( G \) iff \( A \in \text{Voc}(G) \);
(b2) \( A \) is a nonterminal of \( G \) iff \( A \in \text{Nv}(G) \);
(b3) \( A \) is a terminal of \( G \) iff \( A \in \text{Tv}(G) \);
(b4) \( A \) is a production rule of \( G \) iff \( A \in \text{R}(G) \).

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 cfg if \( G \) is a quasi-cfg and \( \text{Voc}(G) \) and \( \text{R}(G) \) 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 \( (a \vDash \beta') \) is written as \( a \vDash \beta' \) (when \( \beta' \) denotes the juxtaposition of the components of the sequence \( \beta \) and, for instance, \( a \vDash q\|\alpha \) stands for the set \( \{ a \vDash q, a \vDash \alpha \} \).

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

\[
\begin{align*}
<\text{int.}> & : = <\text{pos.int.}>|0<\text{pos.int.}> & (1) \\
<\text{pos.int.}> & : = <\text{nz.digit}> |<\text{pos.int.}><\text{digit}> & (2) \\
<\text{digit}> & : = 0|<\text{nz.digit}> & (3) \\
<\text{nz.digit}> & : = 1|2|3|4|5|6|7|8|9 & (5)
\end{align*}
\]

The suggestive names for the nonterminals of this cgy 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-cgy is, in its general form, not a "finitary" concept, because the vocabulary or the rule set can be infinite. However, many quasi-cgy'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 [W 78] for the definition of ALGOL 68 (see also [W 65]), affix grammars, see [Kn 71], or attribute grammars, introduced in [Kn 69] (see [He 84] for a definition devoid of implementation aspects). For an overview and other references we refer the reader to [ML 76] and [BR 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, metatokens 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; ∅) and not simply as A.

D3.3: If V is a set then:

(a) \( \mathcal{L}(V) \) is the smallest set Y such that
\[
\forall v \in V: \forall y \in Y: (A; y) \in Y;
\]
(b) T is a tree over V if \( T \in \mathcal{L}(V) \).

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 \( \mathcal{R}(T) \). We see that if T is a tree over V then \( \mathcal{R}(T) \in V \).

If V is a set then \( F_T \) is the function over \( \mathcal{L}(V) \) that assigns to each tree T over V the sequence consisting of its "leaf labels";

(2) In the context of VMLs, 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 \(<a>\), 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 \(q(k), k \in \text{dom}(q)\). Formally \(F_{\mathcal{V}}\) is defined,
recursively, by:
\[
F_{\mathcal{V}}(a;\emptyset) = <a>.
\]
\[
F_{\mathcal{V}}(a;q) = \text{Gconc}(F_{\mathcal{V}} + q) \text{ if } q \neq \emptyset.
\]
If \(T\) is a tree over \(V\) as well as over \(V'\) then \(F_{\mathcal{V}}(T) = F_{\mathcal{V}'}(T)\). 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 \(\mathcal{R}l\,(q)\). Formally:
\[
D.3.4: \text{If } V \text{ is a set and } q \in \text{lot}(V)^* \text{ then:} \quad \mathcal{R}l\,(q) \triangleq \{(x;\mathcal{R}l(q(x))) \mid x \in \text{dom}(q)\}.
\]

Note that if \(q \in \text{lot}(V)^*\) then \(\mathcal{R}l\,(q) \subseteq V^*\).

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
[BM 82].

\[
D.3.5: \text{If } G \text{ is a quasi-cfg then:} \quad \text{(a) } D\mathcal{R}l(G) \text{ is the smallest set } Y \text{ such that:}
\]
\[
(a) \ (a;\emptyset) \in Y \text{ for every terminal } a \text{ of } G, \text{ and}
\]
\[
(b) \ (a;q) \in Y \text{ for every nonterminal } a \text{ of } G \text{ and}
\]
\[
every q \in Y^{*} \text{ for which } (a;\mathcal{R}l(q)) \in \mathcal{R}l(G);
\]
\[
\text{(b) } T \text{ is a derivation tree based on } G \overset{D}{\vdash} T \subseteq D\mathcal{R}l(G).
\]

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

\[
D.3.2: \text{If } G \text{ is a quasi-cfg then:}
\]
\[
(1) \text{ if } a \text{ is a terminal of } G \text{ then } (a;\emptyset) \text{ is a derivation tree}
\]
\[
\text{based on } G;
\]
(2) if \( (A:z) \) is a production rule of \( G \) and \( q \) is a (nonempty) sequence of derivation trees based on \( G \) for which
\[
R \in (q) = r \quad \text{then} \quad [A;q] \quad \text{is a derivation tree based on} \ G; \\
(3) DTT(G) \supseteq \text{Lot}(\text{Voc}(G)).
\]

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

\[
\begin{align*}
\text{D2.6: IF} \ G \text{ is a quasi-cfg and} \ A \in \text{Voc}(G) \text{ then:} \\
\text{\hspace{1cm}(a) } DDT(G,A) & \supseteq \{ T \mid T \in DTT(G) \text{ and } R(T) = A \} \\
\text{\hspace{1cm}(b) } Fad(G,A) & \equiv \{ FA(T) \mid T \in DTT(G) \text{ and } R(T) = A \} \\
\text{\hspace{1cm}(c) } DL(G) & \equiv DDT(G,St(G)) \\
\text{\hspace{1cm}(d) } L(G) & \equiv Fad(G,St(G)).
\end{align*}
\]

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.

D2.7: IF \( G \) is a quasi-cfg then:
\( G \) is unambiguous \( \iff \forall T \in DL(G) \forall T' \in DL(G) \left( FA(T) = FA(T') \implies T = T' \right) \). In other words, \( G \) is unambiguous iff \( FA \) restricted to \( DL(G) \) 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 fits 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: \(<T;B>\) is a CL-basis if \(T\) is a set and \(B\) is a 7-tuple such that \(N_0, N_1,\) and \(N_2\) are set functions over \(T\), \(N_3\) and \(N_4\) are set functions over \(T \times T\), and \(N_5\) and \(N_6\) are set functions over \((T \times T) \times T\).

If \(B\) is a CL-basis, say \(B = <T;B>\), and \(t, t',\) and \(s\) are elements of \(T\) then the following notations will often be used:
$\text{Typ}_B$ will denote the set $T$.

$\text{Plh}_B(\sigma)$ will denote the set $H_0(\sigma)$.

$\text{Con}_B(\sigma)$ will denote the set $H_1(\sigma)$.

$\text{Int}_B(\sigma)$ will denote the set $H_2(\sigma)$.

$\text{Unop}_B(t, \sigma)$ will denote the set $H_3(\{t; \sigma\})$.

$\text{Arg}_B(t, \sigma)$ will denote the set $H_4(\{t; \sigma\})$.

$\text{Binop}_B(t, t', \sigma)$ will denote the set $H_5(\{t, t'; \sigma\})$, and

$\text{Det}_B(t, t', \sigma)$ will denote the set $H_6(\{t, t'; \sigma\})$.

With respect to a CL-basis $B$ we will call

$\text{Typ}_B$ the set of types of $B$.

$\text{Plh}_B(\sigma)$ its set of placeholders of type $\sigma$.

$\text{Con}_B(\sigma)$ its set of constants of type $\sigma$.

$\text{Int}_B(\sigma)$ its set of intensional constants of type $\sigma$.

$\text{Unop}_B(t, \sigma)$ its set of unary operation symbols with operand type $t$ and result type $\sigma$.

$\text{Arg}_B(t, \sigma)$ its set of arguments with operator type $t$ and result type $\sigma$.

$\text{Binop}_B(t, t', \sigma)$ its set of binary operation symbols with first operand type $t$, second operand type $t'$, and result type $\sigma$, and

$\text{Det}_B(t, t', \sigma)$ its set of determiners with domain type $t$, range type $t'$, and result type $\sigma$.

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

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

21
- \text{Con}_B(e) \text{ will be the set of individual constants and}
\text{Con}_B(t) \text{ will be the set of proposition symbols.}

- There are no intensional constants: \text{Int}_B(e) = \text{Int}_B(t) = \emptyset.

- \text{Unop}_B(e,e) \text{ will be the set of unary function symbols,}
\text{Unop}_B(e,t) \text{ will be the set of unary predicate symbols,}
\text{Unop}_B(t,t) = \{=\}, \text{ and } \text{Unop}_B(t,e) = \emptyset.

- There are no arguments: \text{Arg}_B(\tau,\sigma) = \emptyset \text{ for every } \tau \text{ and } \sigma \text{ in } \text{Typ}_B.

- \text{Binop}_B(e,e,e) \text{ will be the set of binary function symbols,}
\text{Binop}_B(e,e,t) \text{ will be the set of binary predicate symbols (often containing } = \text{, the equality symbol),}
\text{Binop}_B(t,t,t) = \{\land, \lor, \rightarrow, \equiv\}, \text{ and }
\text{Binop}_B(t,\tau,\sigma) = \emptyset \text{ in the other five cases.}

- \text{Det}_B(e,t,t) = \{\forall, \exists\} \text{ and } \text{Det}_B(\tau,\tau',\sigma) = \emptyset \text{ 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 \text{Phi}(e),
\text{Con}(e), \text{Unop}(e,e), \text{ and } \text{Binop}(e,e,e), \text{ where } e \in \{e,t\}.

Aside: A popular choice for \text{Phi}(e) is the set \{x1, x2, 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

\[<P;e> ::= X <\text{pos} \cdot \text{int}>\]

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,

\[<P;e> ::= X \gamma y z <P;e>^*\]

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 \mathcal{B} is often defined as the language
generated by a small unambiguous CFG \mathcal{G}_0, i.e., \text{Typ}_B = \{G_0\}. In
example 4.2 the set of types will be defined in this way. As another
example of an infinite (1) set of types, we consider the set of types
of Montague's language of intensional logic ([No 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:

\[
<Ty> ::= t | e | s <Ty> | ( <Ty> <Ty> )
\]

In example 4.1 a finite set of types was used.

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

\[
\begin{align*}
\mathcal{P}_B(t) & = \{0, 1\}, \quad \text{i.e., the set of "truth values";}\\
\mathcal{P}_B(\text{int}) & = \mathbb{Z}, \quad \text{i.e., the set of integers;}\\
\mathcal{P}_B(\text{so} \langle \tau \rangle) & = \mathcal{P}(\mathcal{P}_B(\tau)), \quad \text{i.e., the power set of the set denoted by } \tau;\\
\mathcal{P}_B(\text{fc} \langle \tau, \sigma \rangle) & = \mathcal{P}_B(\tau) \times \mathcal{P}_B(\sigma), \quad \text{i.e., the set of all functions from } \mathcal{P}_B(\tau) \text{ into } \mathcal{P}_B(\sigma);\\
\mathcal{P}_B(\boldsymbol{p} \langle \tau, \sigma \rangle) & = \mathcal{P}_B(\tau) \times \mathcal{P}_B(\sigma), \quad \text{i.e., the cartesian product of } \mathcal{P}_B(\tau) \text{ and } \mathcal{P}_B(\sigma).
\end{align*}
\]

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 \(\&\), \(\forall\), \(\exists\), and \(\bot\) will be typical elements of \(\text{Binop}_B(\langle \tau, \tau \rangle, \tau)\) for those CL-bases \(B\) in which they occur at all. The only useful unary operation symbol with operand type \(\tau\) and result type \(\tau\) is \(\neg\), the negation symbol. We will use the symbols \(\bot\) (for "false") and \(\top\) (for "true") as constants of type \(\top\).
The equality symbol \(=\) can be put in Binop\(_{g,t}\) for all or, if desirable, only some types \(g\). Also the symbol \(\neq\) can be included.

The well-known symbol \(\notin\) typically belongs to Binop\(_{g,so[0]}\). 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 Binop\(_{so[0],so[0],so[0]}\). The symbol \(\varepsilon\) (for the empty set) can be put in Con\(_{g,so[0]}\) for any type \(g\).

We will use \(sgp\) as a unary operation symbol with operand type \(g\) and result type \(so[0]\), denoting singleton formation.

It is useful to have a symbol in Binop\(_{g,so[0]}\) denoting the formation of ordered pairs. We will use the symbol \(\langle\rangle\) for that purpose.

An interesting candidate for Con\(_{g,\text{int}}\) 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 \(\text{int}\) is supposed to be denoting, is represented by exactly one constant of type \(\text{int}\).) Both Unop\(_{g,\text{int},\text{int}}\) and Binop\(_{g,\text{int},\text{int},\text{int}}\) could contain the symbols \(\times\) and \(\div\). Other typical elements of the latter set are \(\ast\) and \(\cdot\) (for integer division). Typical elements of Binop\(_{g,\text{int},\text{int},\text{t}}\) are (besides \(\ast\)) the "relational" symbols \(<\), \(\leq\), \(=\), and \(\geq\).

We recall that the type \(\text{fc}[\tau\langle\rangle]\) can be thought of as denoting the set of all functions from the set denoted by \(\tau\) into the set denoted by \(\langle\rangle\). Therefore it is useful to have a symbol in Binop\(_{g,\text{fc}[\tau\langle\rangle],\tau,\text{t}}\) denoting function application. We will use the symbol \(\theta\) for that purpose. For our database applications it is also useful to have a symbol in Binop\(_{g,\text{fc}[\tau\langle\rangle],so[\tau]}\) denoting the formation of the pre-image - see chapter 9 - of a "\(\tau\)-object" under a function (expression) of type \(\text{fc}[\tau\langle\rangle]\). For that purpose we will use \(\text{inv}\).

In order to express the queries that are relevant to a database based on a type 2 skeleton \(\langle g,h,\tau\rangle\), 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 \(i\) 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 \(so[i]\).

As a consequence, both \(so[i]\) and \(i\) should be types of \(B\).
(b) Every connector index \( C \) in \( \text{dom}(h) \) should be an intensional constant of type \( h(C) \). (Thus, if \( N \) denotes the source index of \( C \) and \( D \) its target index, i.e., if \( h(C) = (h;D) \), then the type of the intensional constant \( C \) can be written as \( \text{fc}[\text{ty}^B] \).) As a consequence, \( h(C) \) should be a type of \( B \).

(c) For every table index \( E \) in \( \text{dom}(g) \), each attribute of \( B \) 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:

1. \( B \) is a CL-basis fit for \( g:h \).
2. \( B \) is a CL-basis and
   - \( \forall E \in \text{dom}(g) \quad (E \subseteq \text{ty}^B \land E \in \text{Int}_B(\text{ty}^B)) \) and
   - \( \forall C \in \text{dom}(h) \quad h(C) \subseteq \text{ty}^B \land C \in \text{Int}_B(h(C)) \) and
   - \( \forall E \in \text{dom}(g) \quad g(E) = \text{Arg}_B(E; \sigma) \land \sigma \subseteq \text{ty}^B \) and
   - \( \forall \tau, \sigma, \tau' \in \text{ty}^B \quad (\text{if } \sigma \neq \sigma' \text{ then } \text{Arg}_B(\tau; \sigma) \cap \text{Arg}_B(\tau; \sigma') = \emptyset) \).

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 4.3:** We will check what it means for a CL-basis to be fit for \( g:h \), the type 2 skeleton presented in example 4.4. We recall that \( \text{dom}(g) = \{\text{DEP,EMPL}\} \), \( \text{dom}(h) = \{\text{DEP,MAN,MANAGER,EMPLOYEE}\} \), \( g(\text{DEP}) = \{\text{NBR,MAN,NAME}\} \), \( g(\text{EMPL}) = \{\text{DEP,NAME,HR,SSN,SEX}\} \), \( h(\text{DEP}) = \{\text{EMPL,DEP}\} \), and \( h(\text{MAN,MANAGER,EMPLOYEE}) = \{\text{DEP,EMPL}\} \); see also figure 1.2(b).

By requirements (a) and (b), \( \text{ty}^B \) must contain \( \text{DEP,EMPL,so[DEP],so[EMPL,fc[EMPL,DEP]], \text{and fc[DEP,EMPL]}} \) as elements. One of the candidates for \( \text{ty}^B \) is, for instance, the language generated by the grammar with start symbol \( \text{type} \) and the following 8 production rules:

\[
\text{type} \rightarrow \text{DEP}\text{EMPL}
\ |
\text{int}\text{str}
\ |
\text{so[type]}
\ |
\text{fc[type]}
\]

25
(Here the type \textbf{str} is intended to denote the set of all sequences of characters and the "genus type" \textbf{g} is intended to denote a set with exactly two elements, say \{0,1\}. Furthermore, we like \textbf{Con}_B[^g] to be the set \{\phi, \delta\}.

There must be at least 4 intensional constants in \textbf{B} (again, by (a) and (b)): \textbf{Int}_B[^\text{so}[^\text{DEP}]] must contain \text{DEP} , \textbf{Int}_B[^\text{so}[^\text{EMPL}]] must contain \text{EMPL} , \textbf{Int}_B[^\text{fc}[^\text{EMPL},^\text{DEP}]] must contain \text{DEPOF} , and \textbf{Int}_B[^\text{fc}[^\text{EMPL},^\text{DEP}]] must contain \text{MANAGEROF}.

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

\begin{align*}
\text{Arg}_B[^\text{EMPL},^\text{int}] & = (\text{NR}, \text{SAL}, \text{DPT}) & \quad \text{Arg}_B[^\text{DEP},^\text{int}] & = (\text{DN}, \text{MAN}) \\
\text{Arg}_B[^\text{EMPL},^\text{str}] & = (\text{NAME}) & \quad \text{Arg}_B[^\text{DEP},^\text{str}] & = (\text{NAME}) \\
\text{Arg}_B[^\text{EMPL},^\text{g}] & = (\text{SEX})
\end{align*}

Examples of the use of intensional constants and arguments within queries will be given in section 4.2.

4.2. Queries

We start this section with a recursive definition of a relation \text{Frew}(\text{B}) , for each CL-basis \text{B} . If \langle a; c \rangle \in \text{Frew}(\text{B}) then we say that \text{a} is a well-formed expression of type \text{c} over \text{B} . 4.4.1 contains an alternative description of this notion. Further explanation is given after 4.4.1.

D4.3: If \text{B} is a CL-basis then:

\text{Frew}(\text{B}) is the smallest set \text{Y} such that for every \text{t}, \text{t}' , and \text{c} in \text{Ty}_{\text{B}}:

\begin{enumerate}
\item if \text{a} \in \text{Pl}_{\text{B}}[^\text{c}] then \langle a; c \rangle \in \text{Y},
\item if \text{a} \in \text{Con}_B[^\text{c}] then \langle a; c \rangle \in \text{Y},
\item if \text{a} \in \text{Int}_B[^\text{c}] then \langle a; c \rangle \in \text{Y},
\item if \text{a} \in \text{Union}_B[^\text{c}][^\text{t},^t'] then \langle a; c \rangle \in \text{Y},
\item if \text{a} \in \text{Bin}_B[^\text{c}][^\text{t},^t',^c], \langle b; t \rangle \in \text{Y}, and \langle y; t' \rangle \in \text{Y} then \langle \langle b; a \rangle; c \rangle \in \text{Y},
\item if \text{a} \in \text{Arg}_B[^\text{t},^\text{c}] and \langle b; t \rangle \in \text{Y} then \langle \langle b; a \rangle; c \rangle \in \text{Y}.
\end{enumerate}
(7) if $\alpha \in \text{Prt}_B(\tau)$, $(\beta;\gamma) \in Y$, and $(\delta;\epsilon) \in Y$ then 
\[\delta \leftrightarrow \epsilon;\gamma \in Y.\]

(8) if $\alpha \in \text{Det}_B(\tau,\tau',\delta)$, $\beta \in \text{Prt}_B(\tau)$, $(\gamma;\delta) \in Y$, 
$(\gamma;\epsilon) \in Y$, and $(\delta;\epsilon') \in Y$ then $(\alpha;\gamma;\Lambda;\alpha;\delta;\epsilon) \in Y$, and 

(9) if $\beta \in \text{Prt}_B(\tau)$, $(\gamma;\alpha;\delta;\epsilon) \in Y$, $(\gamma;\epsilon) \in Y$, and $(\delta;\epsilon) \in Y$ 
then $(\alpha;\gamma;\Lambda;\alpha;\delta;\epsilon) \in Y$.

The set of all well-formed expressions of type $\sigma$ (over $B$) is 
denoted by $W_B(\sigma)$:

$$W_B(\sigma) \equiv \left\{ \alpha \mid (\alpha;\sigma) \in \text{Pne}(B) \right\}.$$ 

The idea of avoiding a "concurrent" recursive definition of all 
the sets $W_B(\sigma)$, for $\sigma \in \text{Type}_B$, 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 $\sigma$ is obtained by saying that nothing is in $W_B(\sigma)$, for 
any $\sigma \in \text{Type}_B$, except as required by (1)-(9) in the following lemmas.

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

1. if $\alpha \in \text{Prt}_B(\tau)$ then $\alpha \in W_B(\sigma)$;
2. if $\alpha \in \text{Con}_B(\tau)$ then $\alpha \in W_B(\sigma)$;
3. if $\alpha \in \text{Int}_B(\tau)$ then $\forall \beta \in W_B(\tau)$;
4. if $\alpha \in \text{Unop}_B(\tau,\sigma)$ and $\beta \in W_B(\tau)$ then $\alpha \beta \in W_B(\sigma)$;
5. if $\alpha \in \text{Binop}_B(\tau,\tau',\sigma)$, $\beta \in W_B(\tau)$, and $\gamma \in W_B(\tau')$ then 
$\beta \gamma \alpha \in W_B(\sigma)$;
6. if $\alpha \in \text{Acy}_B(\tau,\sigma)$ and $\beta \in W_B(\tau)$ then $(\beta;\alpha) \in W_B(\sigma)$;
7. if $\alpha \in \text{Prt}_B(\tau)$, $\beta \in W_B(\tau)$, and $\gamma \in W_B(\sigma)$ then 
\[\beta \leftrightarrow \gamma; \alpha \in W_B(\sigma);
8. if $\tau \in \text{Type}_B$, $\text{Sol}_B(\tau) \in \text{Type}_B$, $\alpha \in \text{Det}_B(\tau,\tau',\sigma)$, $\beta \in \text{Prt}_B(\tau)$, 
$\gamma \in W_B(\text{Sol}_B(\tau))$, $\varphi \in W_B(\tau)$, and $\delta \in W_B(\tau')$ then 
$\alpha \beta \gamma \Lambda \varphi \delta \in W_B(\sigma)$. 

27
(9) If \( i \in \text{Typ}_B \), \( s_0[i] \in \text{Typ}_B \), \( \beta \in \text{Lin}_B(i) \), \( \gamma \in \text{We}_B(s_0[i]) \), \( \eta \in \text{We}_B(i) \), and \( \delta \in \text{We}_B(\alpha) \) then \( \alpha \in \gamma \times \eta \times \beta \times \delta \in \text{We}_B(\alpha) \).

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 \( c \) is an intensional constant then \( \forall c \) can be read as "the current value of the variable \( c \)". The main difference between a constant and an intensional constant is that the "value" of an intensional constant does not 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 \( \Theta \) 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: For application, some "primitive" types are introduced for such functions, for instance, the types \( \text{DEF} \) and \( \text{EMPL} \) in example 4.2. Furthermore, if \( r_\alpha \) is such a "function type" then \( \text{Arg}(r_\alpha, c) \) must be the set of all arguments for which \( c \) is the result type of application of a function (expression) of type \( r_\alpha \) to that argument. Typical examples of such "function types" will be the table indices in a database system (hence the requirement \( \text{dom}(g) \subseteq \text{Typ}_B \) in 54.2).
ad (7): This clause accounts for an abbreviation facility: \( a \neg b \) may be read as "\( y \) where \( a = b \).

ad (8): Familiar examples of determiners are the symbols \( \vee \) and \( \exists \) in \( \text{Det}(r,t,t) \) as well as \( \Sigma \) (for general addition) and \( \Pi \) (for general multiplication) in, for instance, \( \text{Det}(r,\text{int},\text{int}) \).

\( \forall \in \text{A} \neg \exists \) is usually written as \( \forall (\exists \in \text{A} \neg \exists \in \text{A}) \),

\( \exists \in \text{A} \neg \exists \) usually as \( \exists (\exists \in \text{A} \neg \exists \in \text{A}) \),

\( \Sigma \in \text{A} \neg \exists \) sometimes as \( \Sigma \delta \), and

\( \Pi \in \text{A} \neg \exists \) as \( \Pi \delta \).

For later purpose, we will add the symbol \( \exists \) as a useful (though superficial) element of \( \text{Det}(r,t,t) \). The expression \( \exists \in \text{A} \neg \exists \) will be equivalent to the expression \( \neg \exists \in \text{A} \neg \exists \). (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 \( \text{Det}(r,t,t) \) we introduce the determiners \( \langle b|n \rangle \) where \( b \in \{\overline{u}, \overline{a}, \overline{b}, \overline{y}\} \) and \( n \in \text{Con}(\text{int}) \). The expression \( \langle b|n \rangle \in \text{A} \neg \exists \) should be read as "there are exactly \( n \) elements \( b \) in \( y \) for which \( a \) and \( b \) hold". If \( b \) is replaced by \( \overline{u}, \overline{a}, \overline{b}, \) or \( \overline{y} \), respectively, then "exactly" should be replaced by "less than", "at least", "at most", or "more than", respectively. In other words, the expression \( \langle b|n \rangle \in \text{A} \neg \exists \) is equivalent to the expression \( \exists (\text{L}(\text{A} \neg \exists ; b|n)) \).

We also want to treat the common way of set formation, usually expressed by means of \( \langle \ldots | \ldots \rangle \), as a determiner. For that purpose we will use the symbol \( \forall \) (for \( \forall \text{ext} \)) and \( \exists \) can be placed in \( \text{Det}(r,t',\text{set}[t']) \). In traditional notation, our expression \( \forall \in \text{A} \neg \exists \) would read as \( \langle b \mid \in \text{A} \neg \exists \rangle \) or, rather, as \( \{a \mid \exists \in \text{A} \neg \exists \; (a \neq 0)\} \) where \( a \) is any "fresh" placeholder of type \( r' \), i.e., \( a \neq 0 \) and \( a \) does not occur in any of the expressions \( y, q, \) or \( d \).

---

(1) It would be more general to allow \( n \in \text{We}(\text{int}) \), 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 \( n \in \text{Con}(\text{int}) \cup \text{Plh}(\text{int}) \), or just \( n \in \text{Plh}(\text{int}) \). In that case we would have to adapt D4.6, case (8).
The symbol $\cup$ (denoting generalized union) can be treated as an element of $\operatorname{Det}(\tau, \mathbb{S}0[1', \mathbb{S}0[1'])$. We note that $\mathbb{S}E_{\mathbb{S}0}1$ is equivalent to $\cup \mathbb{S}E_{\mathbb{S}0}[1'], \mathbb{S}0[1']) 1$.

Also the symbol $\lambda$ (for functional abstraction) could be treated as a determiner. However, treating $\lambda$ as an element of $\operatorname{Det}(\tau, \mathbb{S}0[1', \mathbb{S}0[1'])$, as might be expected, creates a problem. The type $\mathbb{S}E_{\mathbb{S}0}[1']$ represents the set of all "total" functions over $\mathbb{S}0(\tau)$, i.e., the set denoted by $\tau$, while $\lambda \mathbb{S}E_{\mathbb{S}0}[1']$ denotes a "partial" function over the set denoted by $\tau$. There are two reasons for this "partiality": (1) $\gamma$ denotes, in general, just a subset of the set denoted by $\tau$, and (2) $\gamma$ acts as a further restriction on this set. We are inclined to treat $\lambda$ as an element of $\operatorname{Det}(\tau, \mathbb{S}0[1', \mathbb{S}0[1'])$, in accordance with the usual treatment of functions in set theory. As a consequence, function application (denoted by the symbol $\Theta$) does not work for $\lambda$-expressions since $\Theta$ only works for expressions of type $\mathbb{S}E_{\mathbb{S}0}[1']$. This is not a serious consequence, however, because $\lambda$-expressions in our query languages are only meant to construct functions, without actually having to apply these functions. Moreover, the expression $(\lambda \mathbb{S}E_{\mathbb{S}0}[1'] \Theta a)$ would be equivalent to $[\Theta a]_\gamma$ and hence, for our purposes, superfluous.

We follow with some general remarks concerning determiners.

For each $a$ in $\operatorname{Det}(\tau, \mathbb{S}0[1'], \mathbb{S}0[1'])$, $\mathbb{S}E_a$ might contain an expression $e_a$ such that for every $\beta$ in $\mathbb{S}E(\beta)$ and every $\delta$ in $\mathbb{S}E(\beta')$, the expression $(\lambda \mathbb{S}E_{\mathbb{S}0}[1'] \Theta a)$ is equivalent to $e_a$ or, in terms of axiom systems, such that $(\lambda \mathbb{S}E_{\mathbb{S}0}[1'] \Theta a)$ can be chosen as an axiom. Loosely speaking, the expression $e_a$, which is independent of $\beta$ and $\delta$, describes the "result" of the determiner $a$ when applied to the empty set. For each determiner $a$ in the first column of the table below, $e_a$ is given in the second column.
Table 4.1.

For some determiners $\delta$ in table 4.1, the third column contains a binary operation symbol $d_\delta$ of which that determiner is a generalization in the following sense: if $\alpha \in \text{Det}(\tau, \tau', \sigma)$ then $d_\alpha \in \text{Binop}(\sigma, \tau', \sigma)$ and for every $\beta$ in Plh$(\tau)$, every $\gamma$ in $\text{We}(\sigma|\tau'|\delta|\tau')$, every $\psi$ in $\text{We}(\tau)$, every $\delta$ in $\text{We}(\tau')$, and every $\delta'$ in $\text{Plh}(\tau') - \{\delta\}$ that does not occur in any of the expressions $\gamma$, $\sigma$, or $\delta$, the expression

$$\forall \epsilon \forall A \forall \delta: \epsilon \in \{\alpha \in \text{Binop}(\sigma|\tau'|\delta|\tau'); \alpha \notin \delta \text{ or } \alpha = \delta\}$$

can be chosen as an axiom, an axiom that reduces quantification over $\gamma$ 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 $(\alpha \in \text{Binop}(\sigma|\tau'|\delta|\tau')) \wedge (\delta \notin \text{Binop}(\sigma|\tau'|\delta|\tau'))$ together "define" $\delta$ for those $\gamma$ 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 $\hat{\delta}$ is an element of $\text{Binop}(\sigma|\tau'|\delta|\tau')$. Denoting union with a singleton, in other words, $(\hat{\delta} \cup \delta)$ is equivalent to $(\hat{\delta} \cup \text{sgn}\ 0)$.

We finally note that it turns out to be convenient to use $\alpha \in \text{Binop}(\sigma|\tau'|\delta|\tau')$ as shorthand for $\alpha \in \text{Binop}(\sigma|\tau'|\delta|\tau')$, a special case which occurs very frequently in practice. This could be sanctioned by adding another clause to D4.3:

$(\delta')$ if $\alpha \in \text{Det}_\beta(\tau, \tau', \sigma)$, $\beta \in \text{Plh}_\beta(\tau)$, $(\gamma; \sigma|\tau') \subseteq Y$, and $(\delta; \tau') \subseteq Y$ then $(\alpha; \text{Binop}(\sigma|\tau'|\delta|\tau') \subseteq Y$. 

31
ad (9): The expression \( \Phi \exists (a \in \mathbb{E}) \) should be read as "\( \Phi \) where \( \mathbb{E} \) is such that \( (a \in \mathbb{E}) \mathbb{A} \) 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 \( \mathbb{E} \) such that \( (a \in \mathbb{E}) \mathbb{A} \) holds. (Therefore, clause (8) and clause (9) will have different translations in chapter 6, where \( \Phi \exists (a \in \mathbb{E}) \) 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), \( \mathbb{E} \) will denote a table element and \( \Phi \) 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 \( \mathbb{E} \) such that \( (a \in \mathbb{E}) \mathbb{A} \) holds.

ad (7) and (9): Although \( [a \in \mathbb{E}] \gamma \) is equivalent to \( \Phi \exists (a \in \mathbb{E}) \gamma \), the former expression will have a completely different (and in fact a more "efficient") translation in chapter 6.

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

\[
W_B = \{ \gamma \mid \exists \mathbb{E} \in \mathbb{E} : a \in W_B (\gamma) \}.
\]

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-CF G: the set of types shall be \( \mathcal{T}(G) \). In practice, a CF G suffices, like in example 4.2.

Next, let \( P (G) \) be the rule set consisting of all production rules of one of the following 10 forms (where \( t, t', \) and \( s \) can vary over \( \mathcal{T}(G) \)):
(0) \text{<Query> ::= <E>}

(1) \text{<E> ::= <F>}

(2) \text{<C>}

(3) \text{\forall E_1, E_2}

(4) \text{<t1,t2> <E_1>}

(5) \text{\{<E_1,t2>-<E_1,t2'><E_1,t2'>\}}

(6) \text{\{<E_1> + <A_1,t3>\}

(7) \text{\{<F> - <E_1>t2>\}<E_1,t2>}

(8) \text{<t1,t2'> <F> + \text{<E_1>sol[t] >> A <E_1> <t2'>}}

(9) \text{A <E_1>t2> \text{<E_1>sol[t] >> A <E_1> <t2'>}}

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 \(<C_0>\), \(<C_1>\), \(<C_2>\), \(<C_3>\), \(<C_4>\), \(<C_5>\), \(<C_6>\), \(<C_7>\), \(<C_8>\), \(<C_9>\), and \(<C_{10}>\) (for any \(t, t',\) and \(c\) in \(I(G_0)\), the intended set of types). We require that the nonterminals \(<Query>\) and \(<E_1>\), for all \(c\) in \(I(G_0)\), 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 \((E')\) can be incorporated by adding the following rule form to the other 10 rule forms:

\[(E') \text{<E> ::= <C_2>t1,t2'}<F>t1 > \text{<E_1>t2'}\)

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 \(\langle I(G_1); 8 \rangle\) where \(8\) is the 7-tuple defined as follows (see D4.1 and D3.6): For all \(t, t',\) and \(c\) in \(I(G_0)\)

\[\begin{align*}
\eta_0(c) &= \text{Fad}(G_1,<C_0>) , \\
\eta_1(c) &= \text{Fad}(G_1,<C_1>) , \\
\eta_2(c) &= \text{Fad}(G_1,<C_2>) , \\
\eta_3(\langle t,t1 <t1,t2 >\rangle) &= \text{Fad}(G_1,<C_3>) , \\
\eta_4(\langle t,t1,t2 >\rangle) &= \text{Fad}(G_1,<C_4>) , \\
\eta_5(\langle t,t1,t2 >\rangle) &= \text{Fad}(G_1,<C_5,t1,t2>t2') , \\
\eta_6(\langle t,t1,t2 >\rangle) &= \text{Fad}(G_1,<C_6,t1,t2>t2') .
\end{align*}\]

33
Finally, let $B$ be $\langle G, \langle e \rangle \rangle$; then $WE_B\langle j \rangle$ is just $FP_B\langle j \rangle$ for every $j$ in $\text{Typ}_B\langle e \rangle$ (thus in $L\langle G \rangle$), and $WE_B\langle e \rangle$ is just $L\langle G \rangle$.

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

**D4.6:** If $B$ is a CL-basic then:

1. $FP_B\langle e \rangle = \{e\}$,
2. $FP_B\langle \emptyset \rangle = \emptyset$,
3. $FP_B\langle e \cdot e \rangle = FP_B\langle e \rangle$,
4. $FP_B\langle \langle e \rangle \rangle = FP_B\langle e \rangle$,
5. $FP_B\langle e_n \rangle = FP_B\langle e \rangle \cup FP_B\langle n \rangle$,
6. $FP_B\langle e_n \rangle = FP_B\langle e \rangle$,
7. $FP_B\langle e \cdot e \rangle = FP_B\langle e \rangle \cup \{e\}$,
8. $FP_B\langle e_n \rangle = FP_B\langle e \rangle \cup \{e_n\}$,
9. $FP_B\langle e_n \rangle = FP_B\langle e \rangle \cup \{e_n\}$.

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 $\langle$.

**D4.7:** If $B$ is a CL-basis and $a \in \text{Typ}_B$ then:

(a) $\text{Clq}_B\langle e \rangle = \{a \in WE_B\langle e \rangle \mid FP_B\langle e \rangle = \emptyset\}$

(b) $\text{Clq}_B\langle e \rangle = \{a \mid B \in \text{Typ}_B, a \in \text{Clq}_B\langle e \rangle\}$

### 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

34
skeleton \langle q; h \rangle as well as some concrete examples of queries with
respect to the sample type 2 skeleton \langle q_1; h_1 \rangle (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, \langle q; h \rangle will be an arbitrary
type 2 skeleton, \( E \) will be a table index (i.e., \( E \in \text{dom}(q) \)), \( a \) and \( b \)
will be attributes of \( E \) (i.e., \( (a,b) \in q(E) \)), and \( B \) will be a CL-basis
fit for \langle q; h \rangle. From D4.2 we now conclude that \( E \) and \( \text{so}(s) \) are types of
\( B \), that \( E \) is an intensional constant of type \( \text{so}(s) \), and that there is
one type \( a_0 \) and one type \( a_1 \) such that \( a \in \text{Arg}_B(E,a_0) \) and \( b \in \text{Arg}_B(E,a_1) \).
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 \( \exists x \in \text{EMPL} \cdot \langle(x,a) \rangle \) which is built up from clauses (3),
(1), (6), (8'), and finally (6). In particular, if \( a = \text{NR} \) and \( E = \text{EMPL} \)
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 \( a_0 \),
i.e., \( w \in \text{Cle}_B(a_0) \). This request is expressed by
\( \exists x \in \text{EMPL} \cdot \langle(x,a) \rangle = \langle(x,a) \rangle \cdot x \), a closed query that is built up by using
the clauses (3), (1), (6), (8'), again (1), (8), and (6). In particular,
if \( E = \text{EMPL} \), \( a = \text{SEX} \), \( w = q \), and \( x = q \) 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: \( \exists x \in \text{EMPL} \cdot \langle(x,a) \rangle \neq \langle(x,a) \rangle \cdot x \).

If \( \langle q; h \rangle \) 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 \((1)\))
\( E \)-tuple for which the \( a \)-value is \( w \). This can be expressed by the query
\( \exists x \in \text{EMPL} \cdot \langle(x,a) \rangle = \langle(x,a) \rangle \cdot x \) 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 \( E = \text{DEP} \), \( a = \text{DNR} \), \( w = 5 \), and
\( b = \text{NAME} \).

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 \( \Theta \) and the
other in combination with \( \text{inv} \). We explain these uses below.

Let \( C \) be a connector index under \( \langle q; h \rangle \), i.e., \( C \in \text{dom}(h) \). By
D1.8, \( h(C) \) will be a pair of table indices, say the pair \( (\text{H}, \text{E}) \). From
If \( \delta \) is an expression of type \( N \) then \( (\gamma \circ \delta) \) is an expression of type \( E \) which denotes the result of application of the \( DEP \) function denoted by \( \gamma \) to the \( N \)-tuple denoted by \( \delta \). For instance, \( (\text{DEP} \circ \delta) \) denotes the department tuple of the employee table denoted by \( \delta \), and \( (\text{MANAGEROF} \circ (\text{DEP} \circ \delta)) \) denotes the manager tuple of that department tuple.

If \( \gamma \) is an expression of type \( E \) then \( (\gamma \text{ inv } \gamma) \) is an expression of type \( \text{SALE} \) which denotes the pre-image of the \( E \)-tuple denoted by \( \gamma \) under the \( DEP \) function denoted by \( \gamma \). For instance, \( (\text{DEP} \text{ inv } \gamma) \) denotes the set of all employee tuples belonging to the department tuple denoted by \( \gamma \).

We finally illustrate the core-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) \( \exists e \in \text{EMPL}: ((e, \text{SAL}) > ((\text{MANAGEROF} \circ (\text{DEP} \circ e)), \text{SAL})) \)

(b) \( \exists d \in \text{DEP}: (n + ((\text{MANAGEROF} \circ d), \text{SAL})) \exists e ((\text{DEP} \text{ inv } d) : ((e, \text{SAL}) > n)) \)

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 23 rule forms is presented. Together with \texttt{<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 \texttt{<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

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

(14) \textit{<type>} & ::= \emptyset \\
(15) & \quad | \textit{\mathbb{Z}} \\
(16) & \quad | \textit{file of} \textit{<type>} \\
(17) & \quad | \textit{product} \textit{<field.decl.part>} \textit{end} \\
(18) \textit{<field.decl.part>} & ::= \textit{<field.decl>} \\
(19) & \quad | \textit{<field.decl.part>} , \textit{<field.decl>} \\
(20) \textit{<field.decl.>} & ::= \textit{<field.id.>} : \textit{<type>} \\
(21) \textit{<var.decl.>} & ::= \textit{<var.id.},T,T;T} \\
(22) \textit{<assign.stat.>} & ::= \textit{<var.},T := \textit{<expr.},T} \\
(23) \textit{<var.},T & ::= \textit{<var.id.},T} \\
(24) & \quad | \textit{<var.},T ; \textit{<field.id.}} \\
(25) \textit{<expr.},T & ::= \textit{<var.},T} \\
(26) & \quad | \textit{<const.},T} \\
(27) & \quad | \textit{un.opr.},T,T,T;T<T};<expr.},T} \\
(28) & \quad | \textit{<expr.},T,T,T,T;T<T};<expr.},T} \\
(29) \textit{<cond.>} & ::= \textit{<expr.},\mathbb{B}}
\end{align*}
(30) \texttt{<const.,B>} ::= \texttt{true} | \texttt{false}
(31) \texttt{<un.opr.,B,B>} ::= \texttt{not}
(32) \texttt{<un.opr.,Z,Z>} ::= \texttt{-}
(33) \texttt{<bin.opr.,B,B,B>} ::= \texttt{and/or} =
(34) \texttt{<bin.opr.,Z,Z,Z>} ::= +|-|\#|\texttt{div|mod}
(35) \texttt{<bin.opr.,Z,Z,B>} ::= <\texttt{|x|\texttt{=}x}>

(36) \texttt{<stand.proc.stat.>} ::= \texttt{return(<expr.,T>)}
(37) \quad \texttt{| mess(<const.,Z>)} \\
(38) \quad \texttt{| file gener.ps.} \\
(39) \quad \texttt{| file insp.ps.} \\
(40) \quad \texttt{| link insp.ps.} \\

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

(45) \texttt{<var.decl.>} ::= \texttt{<var.id.,P> : <P>}
(46) \texttt{<assign.stat.>} ::= \texttt{<var.id.,P> := <fv.,T>}
(47) \texttt{<file insp.ps.>} ::= \texttt{fdp(<fv.,T>;} <\texttt{<var.id.,P>})
(48) \quad \texttt{| fdc(<fv.,T>;} <\texttt{avd.part};<Tv>)
(49) \quad \texttt{| fff(<fv.,T>)}
(50) \quad \texttt{| fnf(<fv.,T>)}
(51) \quad \texttt{| read(<fv.,T>;} <\texttt{<var.r>})
(52) \texttt{<avd.part.;T>} ::= \texttt{<avd.;T>}
(53) \quad \texttt{| <avd.part;> <avd.;T>}
(54) \texttt{<avd.;T>} ::= \texttt{<field id.;> ; <expr.,T>}

(55) \texttt{<link insp.ps.>} ::= \texttt{fte(<var.id.,li T to T>)}
(56) \quad \texttt{| fft(<var.id.,li T to T>)}
(57) \quad \texttt{| fnf(<var.id.,li T to T>)}
5.2. Commentary on the syntax

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

\texttt{ad (00): A program is a block followed by a period.}

\texttt{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.}

\texttt{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.}

\texttt{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).}

\texttt{ad (10): The statement \texttt{if \ a \ then \ v \ fi} is equivalent to the statement \texttt{if \ a \ then \ \texttt{\&\&} \ v \ fi} and, hence, superfluous. Nevertheless, it is a convenient abbreviation.}

\texttt{ad (14): Instead of \texttt{\&\&}, the type identifier \texttt{Boolean} is commonly used.}

\texttt{ad (15): Instead of \texttt{\&\&}, the type identifier \texttt{Integer} is commonly used.}

\texttt{ad (16): We call \texttt{a} the base type of the type \texttt{File of a}.}

\texttt{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 \{((13),(19),(53))\}).

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

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 5. In specific applications, however, various extra types (e.g., a "string type" \(S\)) 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 \(<\text{t}w\text{e}>\) when the following production rules are added to the syntax of section 5.1:

\[<\text{t}w\text{e}>::=<\text{t}y\text{p}e>\]
\[|\text{P}\]
\[|\text{I}\ <\text{t}y\text{p}e>\ \text{to}\ <\text{t}y\text{p}e>\]

The position type \(\text{P}\) 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 \(<\text{t}y\text{p}e>\).) 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 (25), (35), (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 \( \xi \) 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 \( \xi(\sigma) \). For example, one of the production rules obtained from the rule form in line (27) is

\[
\text{<expr.B> ::= <un.opr.,/Z,B><expr.Z>}
\]

using the function \( \xi \) for which \( \xi(T) = B \) and \( \xi(T_1) = Z \).

A reasonable (though superfluous) example of a unary operator with operand type \( Z \) and result type \( B \) is the operator even where the Boolean expression \( \text{even}\ \emptyset \) is equivalent to the expression \((\emptyset \mod 2 = 0)\).

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

\[
\text{<var.decl.> ::= <var.id.;B> : B}
\]

\[
\text{<var.decl.> ::= <var.id.;file of Z> : file of Z}
\]

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

Note that line (43) contains another production rule for \( \text{<var.decl.>} \).

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.
(24), (27), (26), (54): In each of these cases several nonterminals are possible on the left and, moreover, per possible nonterminal there are several production rules.

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

(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'.

(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.

(26): In line (30) we specify the constants of type B. The details of a specification of constants of type 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.; Z> ::= 0 | <pos.int.>
<const.; Z> ::= <digit> | <const.; 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

$$\text{const.} S ::=} \text{\langle char.list\rangle}$$

$$\text{char.list} ::=} \text{\langle char.\rangle}$$

$$| \text{\langle char.\rangle} \text{\langle char.\rangle}$$

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 "gender type" $G$ with two constants, one for "male" and one for "female":

$$\text{const.} G ::=} M | F$$

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:

$$\text{bin. opr.} S . S . S ::=} A$$

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($c$), the value of the expression $c$ 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): \textit{fv,T} is introduced only as an abbreviation, in behalf of
the rule forms to come. \textit{fv,φ} is the nonterminal for file
variables with base type \textit{φ}.

ad (42): At any time at most one component per file \textit{R} (then called the
"current" component of \textit{R}) is "directly accessible". The
standard Boolean expression \textit{in(n)} holds when indeed some
component of \textit{R} is directly accessible, and \textit{in(n)} does not
hold when no component of \textit{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
\textbf{rewrite(n)} and \textbf{write(n;φ)}. For both statements the post-
assertion is that \textit{in(n)} does not hold.

If \textit{λ} denotes concatenation of sequences with components
of type \textit{T}, \textbf{<>} denotes the empty sequence, and \textbf{<i>}
denotes the one-place sequence (i.e., 1-tuple) with the value of the
expression \textit{i} as its component, then the statement

\textbf{rewrite(n)} is equivalent to \textit{R} := \textbf{<>}
and
\textbf{write(n;φ)} is equivalent to \textit{R} := \textit{R} & \textbf{<>}.

ad ((45),(47),(48),(49),(50),(51)): As already suggested by the expla-
nation of the file generation procedures, the value of a file
variable \textit{R} can best be conceived of as a sequence. Such a
sequence, say \textit{s}, will be considered as the composition of an
enumeration \textit{e} of the "positions" (or "addresses" or "locations")
occupied by the components of \textit{R}, and an "occupation
function" \textit{b} assigning to each occupied position the \textit{R}-compo-
nent concerned. In a formula: \textit{e} = \textit{b} + \textit{e} where \text{rng}(\text{e}) = \text{dom}(\text{b}).

And in a figure:

45
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 $<u;r;x>$ is a sequential storage structure for a type 2 snapshot $<y;r';x'>$ and $R$ is the database file corresponding to some table index $E \in \text{dom}(g)$ then $b$ will be $u(E)$ 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 $IP$, where $IP$ 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 $\text{in}(R)$ holds when and only when some component of $R$ is directly accessible, when $\text{in}(R)$ 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 $\text{dom}(b)$.

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 $IP$ which can be used, by (46), to “remember” the current position of a file $R$. By means of (47) an “old position” can be reassigned to the currency variable of $R$, for the statement $\text{fdp}(p;z)$ is equivalent to the assignment statement $R \downarrow := y$, a “fual” of (46).

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

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

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

(a) for \( T \) a product type \( \upsilon \) must be chosen;
(b) within each attribute/value declaration (see (54)), a field identifier \( a \) of \( \upsilon \) must be mentioned and, moreover, for \( T \), the result type of \( a \) within \( \upsilon \) must be chosen;
(c) within the attribute/value declaration part \( \beta \) of the procedure statement \( \text{fdc}(R;\beta) \), each field identifier may be declared at most once.

Because of these requirements, \( \beta \) represents a function \( \beta \), consisting of attribute/value pairs. In other words, \( \beta \) represents a "tuple fragment" \( \beta \).

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

\textbf{ad (49):} There is no special pre-condition for the statement \( \text{fff}(s) \). The post-assertion is that \( \text{in}(s) \) holds iff \( 0 \in \text{dom}(e) \), and that if \( \text{in}(s) \) 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 \( \text{in}(s) \) holds and \( R \downarrow \) will point to the first component of \( R \); otherwise \( \text{in}(s) \) does not hold. Moemonics: \text{fff} stands for "find first of file".
ad (50): The pre-condition for \texttt{fnd}(a) is that \texttt{in}(a) holds. In that case, \texttt{H \downarrow} will have a value \(p \in \text{dom}(a)\). We recall that \(\text{dom}(a) = \text{rng}(e)\). Let \(j \in e^{-1}(p)\), i.e., the serial number of \(p\) under the enumeration \(e\). The corresponding post-assertion is that \texttt{in}(a) holds \(i f j+1 \in \text{dom}(e)\), and that if \texttt{in}(a) holds then \(\texttt{H \downarrow} will have the value \(e(j+1)\). In other words, if there was a next component then \texttt{in}(a) holds afterwards and \(\texttt{H \downarrow}\) will then point to that next component; otherwise \texttt{in}(a) does not hold afterwards. Memonics: \texttt{fnd} stands for “find next of file”.

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

ad ((53), (56), (57)): The effect of the link inspection procedures will be explained in terms of a sequential storage structure \(<\mu; r; n>\) for a type 2 snapshot \(<\mu; h; y; n>\), cf. D2.5. In the explanation below, \(C\) will be a connector index, \(N\) will be the source index of \(C\), and \(D\) will be the target index of \(C\); in other words, \(C \in \text{dom}(h)\) and \(h(C) = (N; D)\). Furthermore, \(\nu_C\) will denote the location connector for \((M; D)\) based on \(\nu_C\) and \(\nu\), i.e., the function \((\nu_D^{-1} \circ \nu_C) \circ \nu\) from \(\text{dom}(\nu_C)\) into \(\text{dom}(\nu_D)\), cf. figure 2.1.

The connector index \(C\) will be treated as a variable of type \(\lambda \texttt{X to D}\), where \(\lambda \texttt{X to D}\) 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, \texttt{fll} will operate on the current target position, while \texttt{fnl} and \texttt{fte} will operate on the current source position, also by definition.

Memonics: \texttt{fte} stands for “find target element”, \texttt{fll} for “find first in link”, and \texttt{fnf} for “find next in link”.

48
ad (55): The precondition for \( \text{fte}(c) \) is that \( \text{in}(u) \) holds. In that case, \( M \uparrow \) will have a value \( m \in \text{dom}(M') \). The corresponding post-assertion is that \( \text{in}(u) \) holds and that \( D \uparrow \) will have the value \( \rho_C(m) \). In other words, afterwards \( D \uparrow \) 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 case, 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 precondition for \( \text{ffl}(c) \) is that \( \text{in}(u) \) holds. In that case, \( D \uparrow \) will have a value \( d \in \text{dom}(M) \). The corresponding post-assertion is that \( \text{in}(u) \) holds iff \( D \in \text{dom}(K_C(0)) \), and that if \( \text{in}(u) \) holds then \( M \uparrow \) 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 \( \text{in}(u) \) holds and \( M \uparrow \) will point to the first M-component in the enumeration \( K_C(0) \) of the M-components corresponding to the D-component that was current beforehand; otherwise \( \text{in}(u) \) does not hold.

ad (57): The precondition for \( \text{fm}(c) \) is that \( \text{in}(u) \) holds. In that case, \( M \uparrow \) will have a value \( m \in \text{dom}(M') \). Let \( a \) be the enumeration \( K_C(c(m)) \) and \( j \) be \( a^{-1}(m) \). The corresponding post-assertion is that \( \text{in}(u) \) holds iff \( j+1 \in \text{dom}(a) \), and that if \( \text{in}(u) \) 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 \( \text{in}(u) \) holds afterwards and \( M \uparrow \) will then point to that next component; otherwise \( \text{in}(u) \) does not hold afterwards.
5.3. A comparison with PASCAL and with the DRAC 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 DRAC proposal, cf. [CC 61]), 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 (16): If α is an expression of type W, 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 as "ad (36)."

ad (42): in(α) corresponds to not eof(α).

ad ((43),(44)): Our file generation procedures rwrite 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, [AW 73].
ad (49): \texttt{fff(n)} corresponds to \texttt{reset(n)}.

ad (50): \texttt{fnf(n)} corresponds to \texttt{get(n)}.

ad (51): Our \texttt{read(x; y)} is equivalent to PASCAL's \texttt{x := R \uparrow} whereas PASCAL's \texttt{read(x; y)} is equivalent to our statement list \texttt{read(x; y); fnf(n)}. 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 \texttt{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 continue with a comparison with the DBTG proposal, using COBOL as the so-called "host language".

ad (61): 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 (68): An assignment statement can be simulated by a \texttt{COMPUTE} statement in COBOL.

ad (109), (110), (111), (112): We have to add a (procedure) name to each "block statement" and to each statement list occurring in a \texttt{while} - or \texttt{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 \texttt{STOP RUN}. Furthermore, \texttt{while c do pn od} becomes \texttt{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.)

51
ad (36): return(a) corresponds to the statement DISPLAY a if the (grammatical) construction of the type of a 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 files. 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 [9k 68]. (Such a statement obscures the structure of a program.)

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

ad (48), (52), (53), (54): Conversion to DPSS 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) $C_1(\text{first}(a)) = C_2(a)$

\text{FIND ANY R.}

52) $C_2(a) = C_3(a)$

53) $C_2(\text{first}(\gamma)) = C_2(\delta)$

$C_3(\gamma)$

54) $C_3(\text{first}(\delta)) = \text{MOVE } \delta \text{ TO } \delta$.

ad (51): Our read($s$; $x$) 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 $i$ M to D corresponds more or less to a DPSS 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 $q : B^{q}$. (The purpose of the program $Z(q)$ is to compute the answer to the query $q$.) The translation function $Z$ over $B^{q}$ 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) $V(R,a)$ to a well-formed expression $a$ over $B$ and a variable $R$ of $P$. The post-assertion of $V(R,a)$ is that $(R = a)$ holds (provided that all presuppositions of $a$ - see chapter 4 - are fulfilled). In other words, after execution of $V(R,a)$ the variable $R$ will have the value of $a$ in the "current state" (provided that the value of $a$ 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 $F$, 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 $F$ is usually defined recursively as well.

Natural choices for the types $t$ and $\textbf{int}$ are $F(t) = B_t$ and $F(\textbf{int}) = Z$. For other types $\alpha$ of $B$ there is usually some freedom in the choice of $F(\alpha)$. In this thesis, we let $F(\text{str}$-$\lceil\cdot\rceil)$ be the type file of $F(t)$, 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 $F(\text{func}$-$\lceil\cdot\rceil)$ to be $\textbf{lik}$ $F(t)$ to $F(t')$, a type "in the wider sense".

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

$$F(\text{DEF}) = \text{product DMR : Z;},$$
$$\text{MAN : Z;},$$
$$\text{NAME : S}$$

end

A good choice for the "genus type" $\text{G}$ of example 4.2 is $F(\text{G}) = \text{G}$; 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 $I$ which assigns a statement list $I(\alpha, R)$ to a determiner $\alpha$ and a variable $R$. More precisely, if $\varphi \in \text{Det}_B(\tau, \tau', \alpha)$ then $R$ is a variable of type $F(\alpha)$. The purpose of $I(\alpha, R)$ is to initialize the variable $R$ to the value of the expression $\alpha \text{EVAR}^{\tau'}$. We recall from section 4.2 that for a determiner $\alpha$ there is an expression
such that \( (a \cdot b \cdot c + \cdot d) \) is an axiom schema, whereas \( c_d \) is independent of \( r \) and \( s \). For each determiner \( a \) in the first column of the table below, \( c_d \) is given in the second column and the initialization \( I(a,r) \) is given in the fourth column.

In clause (8) we also use a function \( \bar{a} \) which assigns a statement list \( \bar{a}(a,r,s) \) to each determiner \( a \) in combination with a variable \( r \) and an expression \( s \) of the programming language. More precisely, if \( a \in \text{Det}_\alpha(r,\tau',u) \) then \( r \) is of type \( F(\alpha) \) and \( s \) is of type \( F(\tau') \). If \( a \) denotes a generalization - in the sense of section 4.2 - of some binary operation symbol \( d_0 \), then the purpose of \( \bar{a}(a,r,s) \) is to assign to the variable \( r \) the result of applying the binary operation denoted by \( d_0 \) 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 \( (\exists a) \) we recall from section 4.2 that \( (\exists a) \in \text{Det}_\alpha(r,s) \) is equivalent to \( (\exists \alpha \in \alpha_a) \cdot 1(\exists a) \). In translations concerning those determiners, we need an extra variable of type \( \mathbb{Z} \), in order to compute the value of \( \exists \alpha \in \alpha_a \cdot 1 \). In the table below, this variable is denoted by \( k \). A better translation for determiners of the form \( (\exists a) \) will be given in section 6.5.

<table>
<thead>
<tr>
<th>( a )</th>
<th>( c_d )</th>
<th>( d_0 )</th>
<th>( I(a,r) )</th>
<th>( \bar{a}(a,r,s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Sigma )</td>
<td>0</td>
<td>+</td>
<td>( r := 0 )</td>
<td>( k := (r + s) )</td>
</tr>
<tr>
<td>( \Pi )</td>
<td>1</td>
<td>( \times )</td>
<td>( r := 1 )</td>
<td>( r := (r \cdot s) )</td>
</tr>
<tr>
<td>( \forall )</td>
<td>( \land )</td>
<td>( \lor )</td>
<td>( r := \text{false} )</td>
<td>( r := (r \text{ or } s) )</td>
</tr>
<tr>
<td>( \forall )</td>
<td>( \forall )</td>
<td>( \forall )</td>
<td>( r := \text{true} )</td>
<td>( r := (r \text{ and } s) )</td>
</tr>
<tr>
<td>$ ( (\exists a) )</td>
<td>( (\exists a) )</td>
<td>( (\exists a) )</td>
<td>( \text{rewrite}(s) )</td>
<td>( \text{write}(k) )</td>
</tr>
<tr>
<td>( (\exists a) )</td>
<td>( (\exists a) )</td>
<td>( (\exists a) )</td>
<td>( k := 0; r := (k \cdot n) )</td>
<td>( \text{if } s \text{ then } k := (k + 1); r := (k \cdot n) )</td>
</tr>
</tbody>
</table>
6.3. Translating queries into programs

A translation function \( T \) from the conceptual language generated by a CN-basis \( B \) into a programming language \( F \) is defined in clause (0) below. \( T \) is defined in terms of a function \( V \) which assigns a statement list \( V(R,a) \) to each well-formed expression \( a \) over \( B \) in combination with a variable \( R \) of \( F \). More precisely, if \( a \) is of type \( \sigma \) then \( R \) is of type \( F(\sigma) \). The definition of \( V \) 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 \( V(R,a) \) will terminate if all "set-valued" subexpressions of \( a \) (i.e., all subexpressions with a type of the form \( \mathbb{S}(\cdot) \)) denote finite sets. In that case, the variable \( R \) will have the value of the expression \( a \) in the "current state", provided that the value of \( a \) 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 \( a \) is not defined then an appropriate message will be returned; see for instance the translation of clause (9). It can be proved that if \( V(R,a) \) 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 \( F(\sigma) \) if the type of \( a \) is \( \sigma \).

The straightforward 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) \( \text{\#(a)} \quad \text{begin} \quad \text{return (r)} \; \text{end.} \)

(1) \( \psi(\text{a}, \text{c}) \quad = \text{r} : = \text{a} \)

(2) \( \psi(a, a) \quad = \text{a}_0, \text{R} \)

(3) \( \psi(\text{a}, \text{v}, \text{a}) \quad = \text{R} : = \text{a} \)

(4) \( \psi(\text{a}, \text{v}, \text{a}) \quad = \text{begin} \quad \text{r} : = \text{a} \)

(5) \( \psi(\text{a}, \text{v}, \text{a}) \quad = \text{begin} \quad \text{r} : = \text{a} \)

(6) \( \psi(\text{a}, \text{v}, \text{a}) \quad = \text{begin} \quad \text{r} : = \text{a} \)

(7) \( \psi(\text{a}, \text{v}, \text{a}) \quad = \text{begin} \quad \text{r} : = \text{a} \)

(8) \( \psi(\text{a}, \text{v}, \text{a}) \quad = \text{begin} \quad \text{r} : = \text{a} \)

\[
\begin{align*}
\text{begin} & \delta : F(\varepsilon), \Sigma : F(\varepsilon) \;
\delta : F(\varepsilon), \varepsilon : F(\varepsilon) \;
\delta : F(\varepsilon), \varepsilon : F(\varepsilon) \;
\delta : F(\varepsilon), \varepsilon : F(\varepsilon) \;
\end{align*}
\]

\[
\begin{align*}
\psi(\text{a}, \text{v}, \text{a}) \quad = \text{begin} \quad \text{r} : = \text{a} \)

(9) \( \psi(\text{a}, \text{v}, \text{a}) \quad = \text{begin} \quad \text{r} : = \text{a} \)

\[
\begin{align*}
\psi(\text{a}, \text{v}, \text{a}) \quad = \text{begin} \quad \text{r} : = \text{a} \)
\end{align*}
\]
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 \( a, b, \) and \( c \), respectively.) As a consequence, a placeholder of type \( \tau \) 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, \( S_0(a, b) \), \( S_1(a, b, c) \), and \( S_2(a, b, c, d) \) 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 \( S_0(a, b) \) is that \( (a = c) \) holds, for \( S_1(a, b, c) \) it is that \( (a = b) \) holds, and for \( S_2(a, b, c, d) \) the intended post-assertion is that \( (a = (b + c)) \) holds. In other words, \( S_0(a, b) \) is equivalent to "\( R := a \)". \( S_1(a, b, c) \) is equivalent to "\( R := a \)" and \( S_2(a, b, c, d) \) is equivalent to "\( R := (b + c) \)". 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): $S_2(a,R)$ needs to be specified for each constant $a$ separately; it has to be a statement (list) equivalent to "$R := a$". Examples are:

$S_2(0,R) = \text{rewrite}(n)$

$S_2(1,R) = R := \text{false}$

$S_2(\top,R) = R := \text{true}$

$S_2(a,R) = R := a$ for every $a \in \text{Const}$

ad (4): $S_2(a,R,E)$ needs to be specified for each unary operation symbol $a$ separately; it has to be a statement (list) equivalent to "$R := aE". Examples are:

$S_2(\text{neg},R,E) = \text{rewrite}(n); \text{write}(n;h)$

$S_2(\neg,R,E) = R := \text{not} \; E$

$S_2(\neg,R,H) = R := \neg H$

$S_2(\neg,R,H) = R := \neg H$

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

$S_2(\text{neg},R,H,W') = R := \text{not}(E := W')$

$S_2(\text{if},R,H,W') = \text{if } E \text{ then } R := W' \text{ else } R := \text{true fi}$

$S_2(\text{if},R,H,W') = \text{if } (H = 0) \text{ then mess(0); halt}$

else $R := (n \text{ div } W')$

fi

Hence, if $\div$ is the binary operation symbol $\div$ 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 possible, cf. ad (9).

Furthermore, if α is Λ, ν, ←φ, or Θ then there is a "corresponding" symbol α' in the programming language (and, or, * and # respectively), and Σₐ(α, θ, θ') will simply be Σₐ(θ, θ'). For each α ∈ {+, −, , ≤, ≥, =} 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(τ), the type of θ.

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

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

\[ V(\delta', 0); 0(\alpha, \delta, \delta'); \]  \hspace{1cm} (8').

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

2) If α is of the form (3αn) then an extra auxiliary variable of type Σ must be declared in line (8.1); see also section 6.2 and section 6.5.

3) If the expression \texttt{init}(α) denotes the set of file components up to (but excluding) the current component of θ whenever \texttt{in}(α) holds, and \texttt{init}(α) denotes the set of all file components of θ when \texttt{not in}(α) holds, then

\[
\text{if } n = \text{α} \in \text{init}(\alpha) \text{ then } c
\]

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

ad (9): There are two important invariants of this \texttt{while}-loop:

(a) \texttt{all init}(α)\texttt{1c}

(b) \texttt{if } n \texttt{ then } c

After termination of the \texttt{while}-statement, (α or \texttt{not in}(α)) holds. If \texttt{δ} holds then by (b) \texttt{θ} holds for the current value of \texttt{δ} (for which (α \texttt{∈ } γ) also holds); then \texttt{δ} is computed for this value of \texttt{δ}. If \texttt{δ} does not hold then \texttt{not in}(α) holds. In that
case, init(n) denotes the value of N; this value will be the value of \( y \), thanks to the preceding statement list \( \forall (n, x) \).

Hence, if \( S \) does not hold then \( \neg \exists \xi : \sigma \) holds by (a) and, consequently, the presupposition \( \exists \xi : \sigma \) (of the source expression \( \sigma \in \exists \xi : \alpha \)) 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 \( \neg \exists \xi : \sigma \), i.e., the juxtaposition of the symbol \( \neg \), the symbol \( \exists \), the text \( \xi \), the symbol \( : \), the text \( \sigma \) stands for \( (\cdot) \). the symbol \( \in \) the text \( \exists \) stands for, etc. This can be accomplished by replacing mess(1) in line (9.9) by return(\( \neg \exists \xi : \sigma \)).

Similarly, mess(0), in one of the examples of clause (5), could be replaced by return(\( \lambda \gamma = 0 \)).

6.4. An example

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

\[ s \in \forall \text{EMPL} \wedge ((\text{DPT}) \neq 5) : e? \]

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, \( s \in \forall \text{EMPL} \wedge ((\text{DPT}) \neq 5) : e \) is abbreviated by \( s_1 \).

\((\text{DPT}) \neq 5\) by \( s_1 \) and \((\text{DPT})\) by \( s_1 \). The type of \( a_1 \) is \( s_0 \{ \text{EMPL} \} \) and from section 6.1 it follows that

\[ \text{F}(\text{EMPL}) = \text{product DPT} : \mathbb{Z}, \]
\[ \text{NAME} : S, \]
\[ \text{NR} : \mathbb{Z}, \]
\[ \text{SAL} : \mathbb{Z}, \]
\[ \text{SEX} : \mathbb{G} \]
\[ \text{end} \]

62
$F(\text{EMPL})$ will not be written out again hereafter. As auxiliary variables we will successively choose $X_0$, $X_1$, $X_2$, etc.

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

\[
\begin{align*}
Z(n1)? & \quad (\text{?}) \quad \begin{align*}
\text{begin} & \quad X_0 \!: F(\text{so[EMPL]}); \\
& \quad V(X_0,n1); \quad \text{return}(X_0) \\
\text{end}
\end{align*}
\end{align*}
\]

where $F(\text{so[EMPL]}) = \text{file of } F(\text{EMPL})$

and $V(X_0,n1) = V(X_0,\$e \in ^{\text{EMPL}} \land q1 \!: e) \quad (\text{?})$

\[
\begin{align*}
\text{begin} & \quad e : F(\text{EMPL}), \ X_2 : F(\text{so[EMPL]}), \ X_2 : \mathbf{E}, \ X_3 : F(\text{EMPL}); \\
& \quad I(\$,X_0); \quad V(X_1,^{\text{EMPL}}); \quad fff(X_1); \\
& \quad \text{while } \text{in}(X_1) \\
& \quad \quad \text{do } \text{read}(X_1;e); \\
& \quad \quad \quad V(X_2,n1); \quad \text{if } X_2 \quad \text{then } V(X_3,e); \quad \text{C(\$,X_0,X_3); fi;} \\
& \quad \quad \quad \text{fnf}(X_1) \\
& \quad \quad \text{od}
\end{align*}
\]

\text{end}

From section 6.2 we conclude that $I(\$,X_0) = \text{write}(X_0)$ and $\text{C(\$,X_0,X_3)} = \text{write}(X_0;X_3)$. Furthermore,

\[
\begin{align*}
V(X_1,^{\text{EMPL}}) \quad (\text{?}) \quad X_1 := \text{EMPL} \\
V(X_3,e) \quad (\text{?}) \quad X_3 := e \\
V(X_2,n1) & \quad = \quad X_2.(s_1 \neq 5)) \\
\begin{align*}
(\text{?}) \text{ begin } & \quad X_4 : F(\text{int}), \ X_5 : F(\text{int}); \\
& \quad V(X_4,s_1); \quad V(X_5,5); \\
& \quad S_2(9,X_2,X_4,X_5)
\end{align*}
\end{align*}
\]

\text{end}

where $S_2(9,X_2,X_4,X_5) = X_2 := \text{not}(X_4 = X_5)$

and $V(X_5,5) \quad (\text{?}) \quad S_0(5,X_5) \\
& \quad = \quad X_5 := 5$

and $V(X_4,s_1) = V(X_4,\{e, \text{DPT}\}) \quad (\text{?})$

63
begin \(X6 : F(\text{EMPL})\); \(V : X6.e\); \(X4 := X6.DPT\) end

Finally, \(V(X6.e)\) (1) \(X6 := e\)

The complete program (in which, however, \(F(\text{EMPL})\) 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:

\[
\begin{align*}
\forall (s e \in \text{EMPL} \land (e.DPT) \neq 5) : \emptyset = \\
\begin{align*}
\text{begin } X0 : \text{file of } F(\text{EMPL}); \\
\text{begin } e : F(\text{EMPL}), X1 : \text{file of } F(\text{EMPL}), X2 : \text{IB}, X3 : F(\text{EMPL}); \\
\text{begin } X1 := e; X1 := e; fff(X1); \\
\text{while } \in(X1) \\
\text{do read(X1e);} \\
\text{begin } X4 : Z, X5 : Z, X6 := X6.DPT \\
\text{begin } X6 : F(\text{EMPL}); \\
\text{end;} \\
X5 := 5; \\
X2 := \text{not}(X4 = X5) \\
\text{end;} \\
\text{if } X2 \text{ then } X3 := e; \text{write}(XO; X3) \text{ fl;} \\
fnn(X1) \\
\text{od} \\
\text{end;} \\
\text{return(XD)} \\
\text{end.}
\end{align*}
\]
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 \( V \):

\[
Z'(x) = \begin{cases} \text{if } x \in \mathcal{S} & \text{then } F(z) ; V'(z, x) ; \text{return}(x) \end{cases}
\]

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;
- an expression that arises from (2), is simple iff \( a \) is of type \( t \) or \( \alpha \) is of type \( \text{int} \);
- an expression that arises from (4), is simple iff \( \beta \) is simple and \( \alpha \in \{-, -, +\} \);
- an expression that arises from (5), is simple iff \( \delta \) and \( \gamma \) are simple and
  - either \( \alpha \in \{+, \times, x, +, -\} \)
  - or \( \alpha \in \{=, =\} \) and \( \delta \) is of type \( \text{int} \);
- an expression that arises from (6), is simple iff \( \delta \) is a so-called 13E-expression, i.e., an expression that is built up by using only clauses (1), (2), and (6);
- an expression that arises from (7), (8), or (9) is not simple.

We leave it to the reader to prove that all 13E-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) \( W(a) = a \)

(2) \( W(false) = false \)
\( W(true) = true \)
\( W(a) = a \) for every \( a \in \text{Con(int)} \)

(4) \( W(\neg b) = \neg W(b) \)
\( W(\neg y) = \neg W(y) \)
\( W(\neg z) = W(\neg) \)

(5) \( W(\forall x : \gamma) = \forall \{ W(x) = W(\gamma) \} \)
\( W(\exists x: \gamma) = \exists \{ W(x) : W(\gamma) \} \)
\( \) if \( \alpha \) is \( \forall, \exists, \leftarrow, \rightarrow, \neg, \land, \lor, \leq, \geq, <, >, \) or \( = \), and \( \) \( \alpha' \) is \( \land, \lor, =, \neg, \leftarrow, \rightarrow, \leq, \geq, <, >, \) or \( = \), respectively

(6) \( W(\langle \cdot, a \rangle) = W(b) \cdot a \)

An example of a simple expression is the expression \( (e \cdot \text{DPT} \neq 5) \) which we used in section 6.4. By the rules above,

\( W(\langle e \cdot \text{DPT} \neq 5 \rangle) = \neg W(\langle e \cdot \text{DPT} = 5 \rangle) \)

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

By means of the function \( W \), 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 \( W' \). The clauses (5), (8), and (9) will be treated later. In the following survey, the definition of our new translation function \( W' \) – in terms of the not yet completely defined function \( V' \) – will also be repeated.

(0) \( W'(e) = \) begin \( k : F(e) ; V'(R, a) ; \) return(a) end.

(1) \( V'(R, a) = R \leftarrow a \)

(2) \( V'(R, a) = S_0(a, R) \)

(3) \( V'(R, a) = R \leftarrow a \)
(4) \( U'(R, a B) \) = \( S_1(a, R, W(S)) \) if \( S \) is simple

\[ U'(R, a B) = \begin{align*}
& \text{begin } \gamma : F(r) ; \\
& \quad U'(R, B) ; \\
& \quad S_1(a, R, R)
\end{align*} \] if \( S \) is not simple

(6) \( U'(R, \{a, \gamma\}) = R \mapsto W(S) \cdot a \) if \( \{a, \gamma\} \) is simple

\[ \begin{align*}
U'(R, \{a, \gamma\}) &= \begin{align*}
& \text{begin } \gamma : F(r) ; \\
& \quad U'(R, a B) ; \\
& \quad R \mapsto M(a)
\end{align*}
\] if \( \{a, \gamma\} \) is not simple

(7) \( U'(R, [a \cdot r] \gamma) = \begin{align*}
& \text{begin } \gamma : F(r) ; \\
& \quad U'(a, B) ; \\
& \quad U'(R, \gamma) \end{align*} \)

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 \( \alpha \) in (5):

(C11) both \( S \) and \( \gamma \) are simple,

(C10) \( S \) is simple and \( \gamma \) is not simple,

(C01) \( S \) is not simple and \( \gamma \) is simple,

(C00) both \( S \) and \( \gamma \) are not simple.

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

(a) \( \alpha \in \{A, V, +\} \) and \( \beta \in \{A, V, \cdot\} \).

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

(5) \( U'(R, [a \cdot r]) = S_2(a, R, W(S), \gamma) \)

\[ \begin{align*}
U'(R, [a \cdot r]) &= \begin{align*}
& \text{begin } \gamma : F(r) ; \\
& \quad U'(a, B) ; \\
& \quad S_2(a, R, W(S), \gamma)
\end{align*} \ \text{end}
\]

\[ \begin{align*}
U'(R, [a \cdot r]) &= \begin{align*}
& \text{begin } \gamma : F(r) ; \\
& \quad U'(a, B) ; \\
& \quad S_2(a, R, W(S), \gamma)
\end{align*} \ \text{end}
\]
As an example of the difference between \( V \) and \( V' \), we note that
\[ V'(X2, ((e, OPT) \neq 5)) \]
is the single statement
\[ X2 := \text{not}(e, OPT = 5) \]
whereas
\[ V(X2, ((e, OPT) = 5)) \]
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 \( e \) is simple then
\[ V'(R, a) \]
will be \( R \leftarrow W(a) \).

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

\[ V'(R, (\alpha \land \gamma)) = V'(R, \beta) \]
if \( R \) then \( V'(R, \gamma) \)
else \( R \) := true

\[ V'(R, (\beta \lor \gamma)) = V'(R, \alpha) \]
if \( R \neq \beta \)
else \( V'(R, \gamma) \)

\[ V'(R, (\beta \rightarrow \gamma)) = V'(R, \alpha) \]
if \( R \neq \beta \)
else \( V'(R, \gamma) \; R := \text{not} \ R \)

Thanks to these rules, the computation of the value of \( \gamma \) 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:

\[ V'(R, (\beta \land \gamma)) = V'(R, \alpha) \]
if \( R \) then \( V'(R, \beta) \)
else \( V'(R, \gamma) \)

\[ V'(R, (\beta \lor \gamma)) = V'(R, \alpha) \]
if \( R \neq \beta \)
else \( V'(R, \gamma) \)

\[ V'(R, (\beta \rightarrow \gamma)) = V'(R, \alpha) \]
if \( R \neq \beta \)
else \( V'(R, \gamma) \; R := \text{not} \ R \)

Here the computation of the value of \( \beta \) 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 \( \gamma \) (respectively \( \beta \)) is not fulfilled then the first (respectively the second) set of alternatives for \( \land \), \( \lor \), 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 \( \land \) respectively \( \lor \) 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 \( \beta \) is expected to be much more laborious than the computation of \( \gamma \), and (on "semantic" grounds) the "situation" \( \gamma \) seems to be unlikely when dealing with \( \land \) (or rather likely when dealing with \( \lor \) or \( \rightarrow \)).
\( (\gamma \land \delta) \) respectively \( (\gamma \lor \delta) \). The original proposals, on the other hand, preserve the "symmetry" (or commutativity) of \( \land \) and of \( \lor \).

One of the reasons that we paid extra attention to improvements on the translation of the symbols \( \land \), \( \lor \), and \( \rightarrow \) is that those symbols occur rather frequently in practice, notably the symbol \( \land \).

Also for the clauses (a') and (b') 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, \( V'(R, a \delta \varphi \gamma \psi \delta) \) will not be expressed in terms of \( V \) but in terms of \( V' \), an improvement in itself.

(b) Furthermore, for some determiners the condition \( \text{in}(\delta) \) in line (8.3) can be replaced by a stronger one. For the determiners \( \pi \), \( \exists \), and \( \forall \) a stronger condition is given in the table below. (In (d), the determiners of the form \( \exists \delta \alpha \) will be treated separately.)

<table>
<thead>
<tr>
<th>a stronger condition wrt. a</th>
</tr>
</thead>
</table>
| \( \pi \) | \( \text{in}(\alpha) \) and \( \neg (a = 0) \) \( \text{in}(\alpha) \) and \( \neg (a = 0) \)  
| \( \exists \) | \( \text{in}(\alpha) \) and \( \neg \exists \delta \) \( \text{in}(\alpha) \) and \( \neg \exists \delta \)  
| \( \forall \) | \( \text{in}(\alpha) \) and \( \exists \delta \) \( \text{in}(\alpha) \) and \( \exists \delta \)  

(c) If \( a \) is the determiner \( \forall \) and the stronger condition wrt. \( a \) is used, then \( 0(\alpha, R, R') \). i.e., the assignment statement \( R := (a \land R') \), is equivalent to \( R := R' \). Consequently, the fragment "\( V'(R', \delta) \); \( \text{in}(\alpha, R, R') \)" occurring in line (8.5) and in line (8.5'), can be simplified to "\( V'(R, \delta) \)". Moreover, the variable \( R' \) and its declaration in line (8.1) can be dispensed with.

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

(d) We recall from section 6.2 that in the translation concerning a determiner of the form \( \exists \delta \alpha \) an extra variable of type \( F: \text{int} \) will be used. If \( k \) denotes that variable then the condition \( \text{in}(\alpha) \) in line (8.3) can be replaced by the stronger condition \( \text{in}(\alpha) \) and \( (k \leq \eta) \) because, informally speaking, the definite value for the Boolean variable \( R \) is known as soon as \( (k \geq \eta) \) holds. (If \( \delta \) is the symbol \( \phi \) or \( \forall \) then even \( \text{in}(\alpha) \) and \( (k \leq \eta) \) would do.) Furthermore, it is sufficient to perform the assignment statement \( R := (k \alpha \eta) \) only once, namely, after the \textbf{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:

\[
\begin{align*}
V'(R, (\exists n) \beta \in \gamma) &= \\
\begin{cases}
\text{begin } \beta \colon F(n), \beta \colon F(\exists n), \beta \colon \mathbb{N}, \beta' \colon \mathbb{N}, k \colon \mathbb{Z}; \\
b \colon 0 \downarrow V'(R, y); f \in \mathbb{N}; \\
\text{while } (\text{in}(a) \text{ and } (k \leq n)) \\
\text{do read } (\text{in}(a)); \\
V'(R, y); f \in \mathbb{N}; \\
\text{if } b \text{ then } V'(R', y) \uparrow \\
\quad \text{if } b' \text{ then } k := (k + 1) \downarrow \\
\quad \text{fi}; \\
f \in \mathbb{N}; \\
\text{od}; \\
R := (k \in \beta)
\end{cases}
\end{align*}
\]

The expression \((a \circ \exists \in \text{init}(a) \land (a \circ \delta) \cdot \exists)\) is an invariant of the
while-loop. In this connection, we recall that \((\exists n) \beta \in \gamma) \circ \gamma) \text{ is equi-

The main advantage of the translation given above over the translation associated with the latter expression
is the stronger "while-condition", namely, \((\text{in}(a) \text{ and } (k \leq n)) \text{ versus } \text{in}(a)\).

We note that the extra variable \(R\) of type \(\mathbb{N}\) can be saved by
using \(R'\) instead.

(e) The next improvement applies if \(a\) or \(\delta\) is simple:
- if \(a\) in clause (8) is simple then the fragment "\(V'(R, \alpha) \circ \beta\) if \(\beta\) then in
original proposal (and "\(V'(R, \alpha) \circ \beta\) if \(\beta\) then" in our proposal
for \((\exists n) \beta \in \gamma) \circ \gamma) \text{ can be replaced by "if } \beta(s) \text{ then }\), and the declaration of \(\beta\) can be omitted.
- if \(\delta\) in clause (8), or in clause (8'), is simple and \(a \neq (\forall, \beta)\) then
the fragment "\(V'(n, \delta) \circ \beta(s, R, H')\)" in the original proposal - see
line (8.5) and line (8.5') - can be replaced by "\(\beta(s, R, H')\)" and
the fragment "\(V'(n, \delta) \circ \beta(s, R, H')\) if \(\beta\) then" in our latest proposal for \((\exists n) \beta \in \gamma) \circ \gamma)
can be replaced by "if \(\beta(s) \text{ then }\). In both cases, the declaration of
\(H\) can be omitted.
For \( a \in (Y,3) \) we already introduced a better proposal in (c), namely, to replace \( V(Y',3) ; \delta (C,R,H') \) by \( V'(R,5) \). This proposal even applies if \( \delta \) 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 (c.DEFT = 5) then write (XNge) 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 \( \gamma \) is a 136-expression then \( \gamma \) is a simple expression and \( \delta (Y) \) is a variable. Therefore, it is feasible to choose for \( H \) the variable \( \delta (Y) \) itself. In this case, the declaration of \( H \) and the fragment \( V(Y,\gamma) \) can be omitted, and the other (four) occurrences of \( H \) must then be replaced by \( \delta (Y) \).

We note that this improvement is very effective since the omitted statement list \( V(H,\gamma) \) 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 \( \delta (Y) \) can play a role in the program fragment that arises from line (3.5), (8'), (5.5), or (9.5), respectively. In particular, the currency variable \( \delta (Y) \) 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 file variable concerned is \( \delta (Y) \),
- by a link inspection procedure of the form fte(c), namely, if \( \delta (Y) \) happens to be the target index of \( C \), or
- by fln(c) or fnl(c) if \( \delta (Y) \) happens to be the source index of \( C \).

On account of the statement \( \text{fln}(\delta(Y)) \) in the sixth line of the translation scheme, the original value of \( \delta (Y) \) has to be "remembered" when such a currency problem is menacing. We can do this by introducing an auxiliary variable \( \delta ' \) of type \( \mathbf{P} \) and enclosing the fifth line of the translation scheme by the "dual" statements \( \delta ' \leftarrow \delta (Y) \) and \( \text{fdp}(\delta (Y) \leftarrow \delta ' ) \), respectively. For clause (8), for instance, this results in the following translation scheme:
\( V'(R, o \in \text{Var} : o) = \)

\[
\begin{align*}
\text{begin } & \text{S1: } F(c), \text{S2: } F, \text{S3: } F(c'), \text{S4: } fff(W(y)); \\
& \text{\{a, R\}, } fdp(W(y); S') \text{ end}
\end{align*}
\]

\end{quote}

In our example used in section 6.4, \( \text{\textit{EMPL}} \) is a \textit{L{\oe} expression}, so the fore-mentioned improvement applies; it does not introduce any currency problems. When we use a 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:

\[
\begin{align*}
\text{\textquote{\textit{EMPL}} \wedge (x.e.DPT \neq 5): e} = \\
\text{begin } & \text{XO: file of } F(\text{\textit{EMPL}}); \\
& \text{begin e: } F(\text{\textit{EMPL}}); \\
& \text{\textit{REWRITE}(XO); } fff(\text{\textit{EMPL}}); \\
& \text{while } in(\text{\textit{EMPL}}) \\
& \text{\textit{DO READ}(\text{\textit{EMPL}}); e} \\
& \text{\textit{IF NOT}(x.e.DPT = 5) \text{ THEN WRITE (XO}; e) \text{ FI}; } \\
& \text{\textit{FNF}(\text{\textit{EMPL}}); } \\
& \text{\textit{AD}} \\
& \text{\textit{RETURN}(XO)} \\
& \text{\textit{END}}.
\end{align*}
\]

6.6. Translation rules concerning connector indices

In database applications, the binary operation symbols \( \oplus \) and \( \text{\textit{INV}} \) are useful in combination with connector indices, see section 4.3. The translation of these special combinations is given below.

With the rule forms
\[ \text{set of all terminal strings of the nonterminal}<\text{AUX;}\sigma>\text{ is a subset of the set of all well-formed expressions of type}\ \sigma.\ \text{In the context of a CL-basis}\ B,\ \text{this special subset will be denoted by}\ \text{Spdb}_B(\sigma).\]

In the remainder of this section, \(\phi;\lambda\) will be a type 2 skeleton, \(\theta\) will be a CL-basis fit for \(\phi;\lambda\) (see D4.2), \(C\) will be a connector index under \(\phi;\lambda\), and \(h(C)\) will be \((N;D)\). From D4.2 we conclude that \(C\) is an intensional constant of type \text{fc}[\text{fp}]\ within the CL-basis\ B.

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

\[ U'(R, (\text{fc} \ [\theta \ | \ \psi])) = A_R(\psi) \ [\text{fc}(c) ; \ \text{read}(\lambda);\gamma]\]

where \(A_R(\psi)\) is a statement list to be defined later on. The intended post-assertion of \(A_R(\psi)\) is that \(\text{in}(\psi)\) holds and that the value of the "position variable" \(M \downarrow\) will be the position of the \(\text{M-component}\) described by \(\psi\). With this post-assertion of \(A_R(\psi)\), the post-assertion of the fragment "\(A_R(\psi) ; \ \text{fc}(c)\)" implies that the value of \(\text{D} \downarrow\) will be the position of the "\(\text{D-target}\)" of the \(\text{M-component}\) described by \(\psi\); cf. section 5.2, ad (55). Finally, it follows from section 5.2, ad (51), that the post-assertion of \(U'(R, (\text{fc} \ [\theta \ | \ \psi]))\) implies that \(R = (\text{fc} \ [\theta \ | \ \psi])\) holds, as it should be.

Another special case of interest for database applications is the case that the well-formed expression \(\gamma\) in clause (8), (8'), or (9) in section 4.2 is of the form \((\text{fc inv} \ \psi')\) where \(\psi' \in \text{Spdb}_B(0)\). 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:

\((8)\) \[ U'(R, \text{as} \in (\text{fc inv} \ \psi') \land \ \gamma ; \ \zeta) - \]

73
begin $E: F(x), \delta; P, B, F(\delta')$;
$F(x, R); A_0(x); ff\ell(c)$;
while in($h$)
    do read($x, h$); $\delta' \leftarrow x, h$;
    $v' \leftarrow B, x$; if $= then$ $v' \leftarrow B', x, R'; G(x, R, R') fi;
    fdp($\delta' v'$); fnl($c$)
end

In no currency problem can arise then the statements $\delta' \leftarrow x, h$ and $fdp(\delta' v')$ can be omitted, just as the declaration of $\delta'$. The translation schemes for the other clauses are similar.

We note that the post-assertion of $A_0(x)$ is that in($x$) holds and that the value of $B'$ will be the position of the D-component described by $\delta'$. For an explanation of the statements ff\ell($c$) 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 $A_0$ is a function over Spd($g, h$) for each $E \in$ dom($g$). The definition will be by recursion on the rule forms (RP1) and (RP2) at the beginning of this section; the parameters $\tau$ and $c$ in these rule forms will vary over dom($g$).

For each $\tau \in$ dom($g$), $\sigma \in$ dom($g$), $c \in$ Spd($g, h$), and $h \in$ dom($h$) such that $h(X) = fc[I, t, o]$ we define:

(RP2) $A_0(c, \delta, \phi) = A_0(g, \delta, \phi) ff\ell(c)$

For $c \in$ dom($g$) and $\delta \in$ Fld($g$), $A_0(c)$ can be defined as

(RP1) $A_0(c, \delta) = fdc(c, \delta, \delta, a)$

for any key $a$ for $c$. (We note that we used (46), (52), and (54) of section 5.1.1. Depending on the context in which $A_0(\delta)$ is used, other choices might be better. For instance, if it is used in a context in which there is a position variable $\delta'$ of which the value happens to be the position of the D-component described by $\delta$ then the choice

(RP1) $A_0(\delta) = fdp(\delta, \delta')$

would do. For example, in the translation schemes for clause (8) given in this section and at the end of the previous section, $v'(h, x)$ and

74
\( V'(H', e) \) occur in such a context. In a context in which the value of \( e \) is already the position of the e-component described by \( H' \), a situation that occurs frequently in practice, \( a_v(\mathcal{S}) \) can even be omitted.

As an illustration of the forementioned translation rules, the two closed queries presented at the end of section 4.3 will be trans- lated. 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) \( \exists e \) EMPL: \((\langle e, \text{SAL} \rangle > \langle \text{MANAGEROF} @ (\langle \text{DEPOF} @ e \rangle), \text{SAL} \rangle) \)?

(b) \( \exists d \) DEPO: \( n = \langle \text{MANAGEROF} @ d, \text{SAL} \rangle \) \& \( \exists e \) DEPO: inv \( d \):
\( (\langle e, \text{SAL} \rangle > n) \)?

In the following translation of (a), \( \langle \text{DEPOF} @ e \rangle \) is abbreviated by \( v_1 \), \( \langle \text{MANAGEROF} @ v_1 \rangle \) by \( v_1, \langle \text{SAL} @ v_1 \rangle \) by \( v_1, \langle e, \text{SAL} \rangle \) by \( v_2, \langle v_2 > v_1 \rangle \) by \( v_1, \text{and} \exists e \) EMPL: \( v_1 \) by \( a_1 \).

(a) \( Z'(a_1) = \begin{align} & \text{begin} \ X_0: v_1; \ V'(X_0,a_1); \ \text{return}(X_0) \ \text{end} \end{align} \)

where \( V'(X_0,a_1) = V'(X_0, \exists e \) EMPL: \( v_1)(\theta') \)

\begin{align} & \text{begin} \ e: \ F(\text{EMPL}), \ pe: \ \mathbb{I}^p; \ F(X_0); \ F(\text{EMPL}) ; \end{align} \\
& \text{while \ (in}(\text{EMPL}) \ \text{and \ not \ X_0) \ \text{do} \ \text{read}(\text{EMPL}) \ pe; \ \text{pe} := \ \text{EMPL} + 1; \ V'(X_0, v_1) ; \end{align} \\
& \text{fdp}(\text{EMPL}; pe); \ \text{fnf}(\text{EMPL}) \ \text{od} \end{align}

since \( \exists e \) EMPL is a 136-expression and the use of \( \exists e \) EMPL will introduce a currency problem (see below). Furthermore, we applied improvement (c) mentioned in section 6.5 for clause \( (\theta') \).

\( F(\text{EMPL}) \) is written out in section 6.4. By section 6.2,

\( I(3, X_0) = X_0 := \text{false} \)

Next,
\[ U'(X_0, \xi_1) = U'(X_0, [\xi_2 > \xi_1]) \quad (\xi_1) \]

begin \( X_1 : X_0, U'(X_1, \xi_1) ; S_2 > X_0, U'(\xi_2), X_1 \) end

by the \([a]-version of case (C10) for clause \((5)\), see section 6.5, ad \((5)\),

\[ S_2 > X_0, U'(\xi_2), X_1 \rightarrow X_0 := U'(\xi_2) > X_1 \]

\[ = X_0 := (e, \text{SAL} > X_1) \]

Furthermore,

\[ U'(X_1, \xi_1) = U'(X_1, [\xi_1, \text{SAL}]) \quad (\xi_1) \]

begin \( X_2 : F(\text{EMPL}) ; U'(X_2, \xi_1), X_1 := X_2, \text{SAL} \) end

where

\[ U'(X_2, \xi_1) = U'(X_2, \langle \text{MANAGEROF @ \xi_1} \rangle) = \]

\[ A_{\text{DEP}}(\xi_1) ; \text{fte(MANAGEROF)} ; \text{read} (\text{EMPL}; X_2) \]

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

\[ A_{\text{DEP}}(\xi_1) = A_{\text{DEP}}(\langle \text{DEPFOF @ e} \rangle), A_{\text{EMPL}}(e) ; \text{fte(DEPFOF)} \]

and by the first alternative for \((RF1)\),

\[ A_{\text{EMPL}}(e) = \text{rde} (\text{EMPL}; \text{NR}: e, \text{NR}) \]

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 \( A_{\text{EMPL}}(e) \) in the translation result of \( A_{\text{DEP}}(\xi_1) \) - 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 \( F(\text{EMPL})\)):

\[ Z' : \text{SEE} E:\langle (e, \text{SAL}) > (\langle \text{MANAGEROF @ \langle \text{DEPFOF @ e} \rangle, \text{SAL}\rangle) \rangle \]
begin X0 := IB;
begin e := f(EMPL); pe := IB;
X0 := false; fff(EMPL);
while (in(EMPL) and not X0)
do read (EMPL; e); pe :=EMPL ↓;
begin X1 := IB;
begin X2 := f(EMPL);
fdc(EMPL; NR := e.WR);
fte(DEP0F);
fte(MANAGER0F);
read (EMPL; X2);
X1 := X2.SAL
end;
end;
X0 := (e.SAL > X1)
end;
fdp(EMPL; pe); fmf(EMPL)
end.

In the following translation of (b) - the other closed query -
((e.SAL) > n) is abbreviated by δ3, bec(’DEP0F inv d) by δ3 by γ3, 
(’MANAGER0F @ d) by δ4, (δ4.SAL) by δ3, [n = δ3]γ3 by δ2, and
δδ’DEP: δ2 by δ2.

(b) Z’(a2?) = begin X0 := IB; V’(X0,a2); return(X0) end.

where

V’(X0,a2) = V’(X0,δδ’DEP: δ2) (8’)

begin d := f(DEP);
T(a,X0); fff(DEP);
while (in(DEP) and not X0)
do read (DEP;d);
V’(X0,d); fmf(DEP)
end
since \( \text{DEP} \) is a 136-expression and the use of \( W(\text{DEP}) \) will not introduce a currency problem (see below). We also applied improvement (c) mentioned in section 6.5 for clause (B').

\[ F(\text{DEP}) \] is written out in section 6.1. We recall that
\[ f(\exists X_0) = X_0 := \text{false} \]

Furthermore,
\[ V'(X_0, 52) = V'(X_0, (n \leftarrow 83), 33) \]
\[ \begin{array}{l}
\text{begin } n := Z; V'(n, 83); V'(X_0, 83) \end{array} \]

First, we elaborate \( V'(n, 83) \):
\[ V'(n, 83) = V'(n, (84, \text{SAL})) \]
\[ \begin{array}{l}
\text{begin } X_1 := f(\text{EMPL}); V'(X_1, 84); n := X_1, \text{SAL} \end{array} \]

since \( (84, \text{SAL}) \) is not simple; cf. section 6.3.
\[ V'(X_1, 84) = V'(X_1, (\text{MANAGEROF } @ d)) \]
\[ \text{and } A_{\text{DEP}}(d) := \text{file(_MANAGEROF); read(EMPL; X_1)} \]

by the first alternative for (RFL). The second alternative for (RFL) does not apply here but, fortunately, the third one does (since the variable \( \text{DEP} \) still "points to" the current value of the variable \( d \)).

Next, we elaborate \( V'(X_0, 83) \):
\[ V'(X_0, 83) = V'(X_0, \exists E(\text{DEPOF } \text{inv } d); 63) \]
\[ \begin{array}{l}
\text{begin } e := f(\text{EMPL});
\text{if } E(X_0); A_{\text{DEP}}(d); f(\text{DEPOF});
\text{while } (\text{in}(\text{EMPL}) \text{ and not } X_0)
\text{do read(EMPL); e;}
V'(X_0, 63); f(\text{DEPOF})
o d
\end{array} \]

by the \( (B') \)-variant of the translation scheme for \( \text{inv} \) mentioned in this section, improvement (c) mentioned in section 6.5, and the fact that \( V'(X_0, 83) \) will not introduce a currency problem.
Finally, by case (C11) for clause (5) and the definitions of $S_2$ and $\mathcal{N}$,

$$\mathcal{N}(XO, BO) = \mathcal{N}(XO, ((e.SAL) > n))$$

$$S_2 \rightarrow NO, \mathcal{N}(e.SAL) > n)$$

$$XO := (e.SAL > n)$$

The resulting program is as follows:

\[
\begin{align*}
2: (\exists d \in \text{DEP}; [n &\leftarrow ((\forall \text{MANAGEROF} @ d).SAL)] \\
\exists e \in \text{DEP}; \text{inv} d; ((e.SAL) > n)] =
\end{align*}
\]

begin $XO \leftarrow false$

begin d $\in$ DEP;

$XO := false; fff(DEP)$

while (in(DEP) and not XO)

do read (DEP);$

begin n $\leftarrow Z$;

begin XI $\leftarrow EML$;

\begin{align*}
&fde(DEP; DNR; d.DNR); \\
&fde(MANAGEROF); \\
&read (EML; XI); \\
&n := XI.SAL
\end{align*}

end$

begin e $\leftarrow EML$;

\begin{align*}
&XO := false; \\
&fde(DEP; DNR; d.DNR); \\
&fff(DEP);$ \\
&while (in(EML) and not XO) \\
&do read (EML);$

\begin{align*}
&XO := (e.SAL > n); \\
&fff(DEP)
\end{align*}

od

od

end$

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 9. These translations are independent of any particular application. For 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 examples 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 t, t', s, and c' will vary over the set of
types relevant to the (database) application concerned, just as in
chapter 4.

The parameters u and u' indicate the feature of number (of noun
phrases and verb phrases for instance). They will vary over the
(application-independent) set {ni,pl}, where ni 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 retriev-
al). 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
<EN;ni>; <EN;pl>; <EN;i>; <ET;ni>; <ET;i>; <EN;pf>; <EN;cpf>; <EN;pg>
<EN;cpg>; or <EN;cpa>. 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 <RIP> (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:1>()
2   | Is it true that <ET> ?
3  <ET> ::= <SN:1>()<VP:1>()
4  <NP:1>() ::= <CT:1>()<CN:1>()
5  <NP:1>() ::= <FN:1>()
6  <CN:1>() ::= <SN:1>()
7     | <SN:1>()<BC:1>()
8  <SN:1>() ::= <BS:1>()
9     | <CS:1>()<SN:1>()
10    | <SN:1>()<EN:1>()
11  <BC:1>() ::= <SC:1>()
12     | <CC:1>()
13  <CC:1>() ::= <SC:1>()<SA:1>()<SC:1>()
14     | <SC:1>()<CC:1>()
15  <SC:1>() ::= <SP:1>()<VP:1>()
16     | whose <TN:1>()<VP:1>()
17  <VP:1>() ::= <IV:1>()
18     | <CJ:1>()
19  <CJ:1>() ::= <CV:1>()<A:1><IV:1>()
20     | <TV:1>()<CJ:1>()
21  <IV:1>() ::= <ET:1>()<AT:1>()<AL:1>()<AD:1>()
22     | <ET:1>()<AT:1>()<ET:1>()<AT:1>()<AL:1>()<AD:1>()
23     | <CJ:1>()<GJ:1>()
24  <TV:1>() ::= <CV:1>()<TV:1>()
25  <TV:1>() ::= <CP:1>()<CN:1>()
26  <ET:1>() ::= <ET:1>()<ET:1>()<CP:1>()
27  <AL:1>() ::= <AV:1>()
28     | <AV:1>()<AL:1>()
29  <AD:1>() ::= <ND:1>() times
30     | <CO:1> once
31  <EX:1>() ::= <RE:1>()
32     | <CF:1>()<SN:1>()
33  <EX:1>() ::= <CP:1>()<SN:1>()<CN:1>()
34  <CT:1>() ::= <EB:1>()
35     | <TV:1>()
36  <EB:1>() ::= <CP:1>()<EB:1>()
37  <EB:1>() ::= the <TN:1>()<DB:1>()
$\langle \text{FL} 0 0 \rangle \tau : 1^* \rangle$ := $<\text{FN} 0 1^* \rangle \texttt{?} <\text{FN} 0 1^* \rangle \Downarrow \text{ppr}$

$\langle \text{FN} 0 0 \rangle$ := $<\text{FU} 0 1^* \rangle$

$\langle \text{FQ} 0 \rangle$ := $<\text{RE} 0 \rangle$

$\langle \text{SR} 0 \rangle$ := $<\text{CE} 0 \rangle$

the $\langle \text{GQ} 0 \rangle <\text{FN} 0 1^* \rangle$ of all $<\text{CN} 0 \rangle$

the number of $<\text{CN} 0 \rangle$

$\langle \text{FN} 0 \rangle$ := $<\text{FQ} 0 \rangle$

$\langle \text{FQ} 0 \rangle$ := $<\text{IA} 0 <\text{CN} 0 \rangle$

$\langle \text{SD} 0 \rangle$ := $<\text{SD} 0 \rangle$

$\langle \text{SD} 0 \rangle$ := $<\text{ND} 0 \rangle$

$\langle \text{CD} 0 \rangle$ := $<\text{IA} 0$

$\langle \text{ND} 0 \rangle$ := $<\text{CO} <\text{CA} 0 \rangle$

$\langle \text{CA} 0 \rangle$ := one

$\langle \text{CA} 0 \rangle$ := zero

$\langle \text{CA} 0 \rangle$ := two

$\langle \text{CA} 0 \rangle$ := three

$\langle \text{DD} 0 \rangle$ := at least

$\langle \text{DD} 0 \rangle$ := at most

$\langle \text{DD} 0 \rangle$ := exactly

$\langle \text{DD} 0 \rangle$ := more than

$\langle \text{DD} 0 \rangle$ := less than

$\langle \text{GD} 0 \rangle \texttt{?} \text{int} \rangle$ := total

$\langle \text{GD} 0 \rangle \texttt{?} \text{int} \rangle$ := minimal

$\langle \text{GD} 0 \rangle \texttt{?} \text{int} \rangle$ := maximal

$\langle \text{GD} 0 \rangle \texttt{?} \text{rot} \rangle$ := average

$\langle \text{GD} 0 \rangle$ := some

$\langle \text{GD} 0 \rangle$ := no

$\langle \text{SD} 0 \rangle$ := every|each

$\langle \text{SD} 0 \rangle$ := all

$\langle \text{IA} 0 \rangle$ := a|an

$\langle \text{DD} 0 \rangle$ := every|each

$\langle \text{DD} 0 \rangle$ := all|the

$\langle \text{A} 0 \rangle$ := and|and

$\langle \text{OA} 0 \rangle$ := is

$\langle \text{DD} 0 \rangle$ := are

$\langle \text{PPR} 0 \rangle$ := of|for|in

$\langle \text{RP} 0 \rangle$ := who|that

$\langle \text{RP} 0 \rangle$ := himself

$\langle \text{RP} 0 \rangle$ := herself

$\langle \text{RP} 0 \rangle$ := itself

$\langle \text{RP} 0 \rangle$ := themselves
<table>
<thead>
<tr>
<th>Item</th>
<th>Occurs on the Left Hand Side In</th>
<th>Occurs on the Right Hand Side In</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>79</td>
<td>13, 19, 20</td>
</tr>
<tr>
<td>AB2</td>
<td>29, 10</td>
<td>11, 13</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>25, 20</td>
<td>11, 12, 20</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td></td>
<td>17, 28</td>
</tr>
<tr>
<td>AB1</td>
<td>52, 61, 64</td>
<td>50</td>
</tr>
<tr>
<td>AB2</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>78, 79</td>
<td>23, 24, 25</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td>57, 59, 67, 69</td>
<td>58, 66</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>13, 14</td>
<td>13, 14</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td>34, 35, 36</td>
<td>32, 33, 36</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>79, 79</td>
<td>18, 20</td>
</tr>
<tr>
<td>AB1, AB3</td>
<td>6, 7</td>
<td>4, 25, 33, 65, 66, 69</td>
</tr>
<tr>
<td>AB2</td>
<td>61, 63, 64, 65</td>
<td>62, 65</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td>75, 76</td>
<td>33</td>
</tr>
<tr>
<td>AB2</td>
<td></td>
<td>9, 21</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td></td>
<td>50, 61</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td>26</td>
<td>23</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td></td>
<td>30, 31</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td>21, 22, 23</td>
<td>1</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>22, 37</td>
<td>36, 37, 45</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td>38, 39</td>
<td>35, 39</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>40, 41</td>
<td>40</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>68, 69</td>
<td>65</td>
</tr>
<tr>
<td>AB2</td>
<td>16, 37, 38, 39, 43</td>
<td>32, 33, 36</td>
</tr>
<tr>
<td>AB2</td>
<td>66, 67</td>
<td>39, 40</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>74</td>
<td>40, 49</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>21, 22, 23, 24, 25</td>
<td>15, 19, 19, 20</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>22, 24</td>
<td>29, 26</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>24, 25</td>
<td>3, 22</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>75, 76</td>
<td>53</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>87</td>
<td>45</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>47, 48</td>
<td>5, 48</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>80</td>
<td>37, 40</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>49</td>
<td>24</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>42, 43</td>
<td>24, 47</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>11, 12</td>
<td>7</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>72, 73</td>
<td>51</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>92, 93</td>
<td>35, 36</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>81</td>
<td>3</td>
</tr>
<tr>
<td>AB1, AB2</td>
<td>1, 12</td>
<td>11, 13, 13, 14</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td>16, 17</td>
<td>31, 42</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td>44, 45, 46</td>
<td>31, 42</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td>9, 19, 30</td>
<td>6, 7, 9, 10</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>AB2, AB3</td>
<td>17, 18</td>
<td>3, 15, 16</td>
</tr>
</tbody>
</table>
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 81], Ch.11, 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

\[<\text{SN;}u;0> ::= <\text{SN;}u;0> \mid <\text{SN;}u;0><\text{PO;}0>\]
\[<\text{SN;}u;0> ::= <\text{SN;}u;0> \mid <\text{DJ;}0><\text{SN;}u;0>\]

ad ((11),(12),(13),(14),(17)): 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),(6)): 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 (8) 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 $o$ in rule form (16)) is equal to the type of the simple clause and whose result type ($r$ 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 {{(11),(18),(19),(20),(77)}}: Here a similar construction as in
{(11),(12),(13),(14)} is used: "NC" 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
genral 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 ($o$ in these rule
forms), and the type of the adverbial list is equal to the
underlying type of the existential (in)transitive verb phrase
($r$ 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 9. (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), (28), (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 9), 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 time or a complement 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 (3) 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 1 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 *or*, 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 simple 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 \{(38), (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 \texttt{pair}.\texttt{pair}.

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 \{(47)\}: 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 that can be used in popular (application-dependent) appositive adjectival phrases such as

with 6 employee(s)

of type DBF in our employees-and-departments application.
(see, e.g., figure 1.2). Such a determiner \( c \) 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 \(<\text{CA}> := <\text{int}>\) or \(<\text{CA}> := <\text{pos}\text{.int}>\).

ad \((57)\) and following: There 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 adversarial phrases of time. For instance, in some applications March 16, 1904 should be translated to 19040316, in others to 19040316, 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 "JY-Phrases" of time, i.e., phrases that can be used both as
adjectival and as adverbial phrases of time.

In the rules below, "FT" 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 \(<\text{FT;}\text{date}>\) := \(<\text{BE;}\text{date}>\) 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 "JY-Phrases" we refer the reader to
the appendix.

SG01 \(<\text{JY;}\text{date}>\) := \(<\text{FT}>\text{<BE;}\text{date}>\)
SG02 |between \(<\text{BE;}\text{date}>\) and \(<\text{BE;}\text{date}>\)
SG03 |in the period \(<\text{BE;}\text{date}>\) to \(<\text{BE;}\text{date}>\)
SG04 |in the period \(<\text{month;}\text{day}>\) to \(<\text{day}>\)\<\text{year}>
SG05 |in \(<\text{month}>\text{<year}>\)
SG06 |in \(<\text{year}>\)
SG07 \(<\text{CE;}\text{date}>\) := \(<\text{month}>\text{<day}>\)\<\text{year}>
SG08 |\(<\text{D2}>\text{<D2}>\text{<D2}>\)
SG09 \(<\text{year}>\) := \(<\text{D2}>\text{<D2}>\)
SG10 \(<\text{day}>\) := \(<\text{D2}>\)
SG11 |\(<\text{digit}>\)
SG12 \(<\text{D2}>\) := \(<\text{digit}>\text{<digit}>\)
SG13 \(<\text{FT}>\) := \text{on}
SG14 |\text{before}
SG15 |\text{after}
SG16 \(<\text{month}>\) := \text{January}
SG17 |\text{February}
SG18 |\text{March}
SG19 |\text{April}
SG20 |\text{May}
SG21 |\text{June}
SG22 |\text{July}
SG23 |\text{August}
SG24 |\text{September}
SG25 |\text{October}
SG26 |\text{November}
SG27 |\text{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 [Kn 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-csg 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 a is a derivation tree based on G (see D3.5) and the root label of a is:

- <RQ> then M(a) ∈ cl(B), i.e., the translation of a is a closed query (see D4.7).
- \( \langle ST \rangle \) then \( M(a) \in \text{Cls}_B(t) \), i.e., the translation of \( a \) is a closed expression of type \( t \).

If the root label of \( a \) is of the form:
- \( \langle SN; o \rangle \) or \( \langle SE; o \rangle \) then \( M(a) \in \text{Cls}_B(o) \);
- \( \langle PU; i, o \rangle \) then \( M(a) \in \text{Cls}_B(\text{fc}[\text{ct}]) \);
- \( \langle AS; i, o \rangle \) then \( M(a) \in \text{Arq}_B(o, o') \);
- \( \langle CE; i \rangle \) then \( M(a) \in \text{Con}_B(o) \).

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

\[
\begin{align*}
\text{Ro}_B & \triangleq \{ \langle z, <e, s, r > \rangle \} \\
\text{Dro}_B & \triangleq \{ \langle \#s \rangle \mid a \in \text{Ro} \text{ and } \eta \in \text{Con}_B(\text{int}) \}. 
\end{align*}
\]

We require of \( B \) that \( \text{Ro}_B \subseteq \text{Bino}_B(\text{int}, \text{int}, t) \) and \( \text{Dro}_B \subseteq \text{Det}_B(t, t, t) \) for each \( t \in \text{Typ}_B \).

Further conditions on the translation of a derivation tree \( a \) are:

If the root label of \( a \) is of the form:
- \( \langle GD; o, o' \rangle \) then \( M(a) \in \text{Det}_B(t, o, o') \);
- \( \langle GD; i \rangle \) then \( M(a) \in \text{Det}_B(t, t, t) \);
- \( \langle CA; i \rangle \) then \( M(a) \in \text{Con}_B(\text{int}) \);
- \( \langle CD \rangle \) then \( M(a) \in \text{Ro}_B \);
- \( \langle NO; i \rangle \) then \( M(a) \in \text{Dro}_B \);
- \( \langle AD \rangle \) then \( M(a) \in \text{Dro}_B \);
- \( \langle NO; i \rangle \) then \( M(a) \in \text{Det}_B(t, t, t) \);
- \( \langle CD \rangle \) then \( M(a) \in \text{Det}_B(t, t, t) \).

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 \( a \) is of the form
\( \langle GP; i, o \rangle \), \( \langle PE; i, o \rangle \), \( \langle FL; i, o \rangle \), or \( \langle FN; o \rangle \)
then \(a\) must be translated in combination with a placeholder of type \(\sigma\): If \(x \in \text{Plh}_B(\sigma)\) then we require that \(M(x, a) \in \text{We}_B(\sigma')\) and \(\text{Fr}_B(M(x, a)) \subseteq \{x\}\), i.e., \(M(x, a)\) is a well-formed expression of type \(\sigma'\) in which only the placeholder \(x\) might be free.

If the root label of \(a\) is of the form
\(<\text{AL}>\), \(<\text{V}>\), \(<\text{CG}>\), \(<\text{CD}>\), \(<\text{D} >\), \(<\text{D}2>\), \(<\text{D}3>\), \(<\text{D}4>\), \(<\text{D}5>\), \(<\text{D}6>\), \(<\text{RD}>\), \(<\text{VR}<\text{D} >\), \(<\text{VR}<\text{D}2>\), \(<\text{VR}<\text{D}3>\), \(<\text{VR}<\text{D}4>\), \(<\text{VR}<\text{D}5>\), \(<\text{VR}<\text{D}6>\) and \(x \in \text{Plh}_B(\sigma)\)
then \(M(x, a) \in \text{We}_B(\tau)\) and \(\text{Fr}_B(M(x, a)) \subseteq \{x\}\).

If the root label of \(a\) is of the form
\(<\text{BN}>\), \(<\text{CN}>\), \(<\text{SN}>\), \(<\text{SN}>\)
then \(M(x, a) \in \text{We}_B(\tau)\) and \(\text{Fr}_B(M(x, a)) \subseteq \{x\}\). More specifically, \(M(x, a)\) should be a terminal string of the auxiliary nonterminal \(<\text{RE}>x\) having the following production rules (in combination with the grammar presented after D4.5 in section 4.2):

\[
\text{RE} > x : = (x \in \text{So}(\gamma);) \quad (\text{PB1})
\]

\[
(\text{RE} > x) \quad \tau \quad (\text{PR2})
\]

The set of all terminal strings of the nonterminal \(<\text{RE}>x\) will be denoted by \(\text{Re}_B(x)\). Thus, \(\text{Re}_B(x)\) is a subset of \(\text{We}_B(\tau)\) and the fore-mentioned requirement of \(M(x, a)\) is that \(M(x, a) \in \text{Re}_B(x)\). In other words, \(M(x, a)\) is a conjunction of well-formed expressions of type \(\tau\) such that the first conjunct is of the form \((x \in \gamma\); for some \(\gamma \in \text{We}_B(\text{So}(\gamma))\).

If the root label of the derivation tree \(a\) is of the form \(<\text{E} >\)
then \(a\) must be translated in combination with a placeholder \(x\) in \(\text{Plh}_B(\sigma)\) and a placeholder \(x'\) in \(\text{Plh}_B(\sigma')\). We stipulate that \(M(x, x', a) \in \text{Re}_B(x')\) and that \(\text{Fr}_B(M(x, x', a)) \subseteq \{x, x'\}\).

If the root label of \(a\) is of the form \(<\text{F} >\)
then \(a\) must be translated in combination with three placeholders, one from \(\text{Plh}_B(\sigma)\), one from \(\text{Plh}_B(\sigma')\), and one from \(\text{Plh}_B(\sigma'')\); for \(x \in \text{Plh}_B(\sigma)\), \(x' \in \text{Plh}_B(\sigma')\), and \(x'' \in \text{Plh}_B(\sigma'')\) we stipulate that \(M(x, x', x'', a) \in \text{Re}_B(x'')\) and that \(\text{Fr}_B(M(x, x', x'', a)) \subseteq \{x, x', x''\}\).
We note that each \( \psi \in \text{Restr}_B(x) \), 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 \( \psi \) is built up by means of (PR1) only, then the outer parentheses should be omitted. If \( \psi \) 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 \( \{x \in Y\} \), should be omitted. We will ignore this (harmless) difference from now on.

If the root label of \( a \) is of the form \(<\text{NP}':a'>\), \(<\text{PR}:a'>\), \(<\text{PS}:a'>\), or \(<\text{QS}:a'>\) then \( a \) must be translated in combination with a placeholder \( x' \in \text{Plh}_B(\varepsilon) \).

If the root label of \( a \) is of the form \(<\text{NP}':a'>\) then we stipulate that \( M(x,a) \) followed by any well-formed expression of type \( t \) together constitute a well-formed expression of type \( t \) again. More precisely: \( \text{Vec}_{\text{We}_B(t)}'; M(x,a) \in \varepsilon \in \text{We}_B(t) \), where \( M(x,a) \in \varepsilon \) denotes the concatenation of the sequences (of "characters") \( M(x,a) \) and \( \varepsilon \) (cf. chapter 3). Moreover, we stipulate that \( \text{FP}_B(M(x,a) \in \varepsilon \in \text{FP}_B(\varepsilon) - (x) \).

Similarly, if the root label of \( a \) is of the form \(<\text{PR}:a'>\), \(<\text{PS}:a'>\), or \(<\text{QS}:a'>\) then we stipulate that \( M(x,a) \) followed by any well-formed expression of any type \( t \) together constitute a well-formed expression of type \( t \) again. More precisely: \( \text{Vec}_{\text{Typ}_B}; \text{Vec}_{\text{We}_B(t)}'; M(x,a) \in \varepsilon \in \text{We}_B(t) \). Furthermore, we stipulate that \( \text{FP}_B(M(x,a) \in \varepsilon \in \text{FP}_B(\varepsilon) - (x) \).

If the root label of \( a \) is of the form \(<a>, <Gc:\nu>, <Gd:\nu>, <E\nu>, <F\nu>, <R\nu>, \text{<np}\nu>, \text{<ps}\nu>, \text{<qs}\nu>\), or \(<\text{RP}\) then no translation rule is needed for \( a \). 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 \( x \in \text{Plh}_B(\varepsilon) \), \( x' \in \text{Plh}_B(\varepsilon') \), \( x'' \in \text{Plh}_B(\varepsilon'') \), and \( a \) is a derivation tree with a root label of the form:
\[ \text{if } (\text{AD}) \text{ then } \text{M}(a) \in \text{Dom}_B; \]
\[ \text{if } (\text{AR};0;i') \text{ then } \text{M}(a) \in \text{Arg}_B(0,i'); \]
\[ \text{if } (\text{DR};0) \text{ then } \text{M}(a) \in \text{Doc}_B(t,t,t); \]
\[ \text{if } (\text{CA};u) \text{ then } \text{M}(a) \in \text{Con}_B(\text{int}); \]
\[ \text{if } (\text{CE};0) \text{ then } \text{M}(a) \in \text{Con}_B(0); \]
\[ \text{if } (\text{CO}) \text{ then } \text{M}(a) \in \text{Roi;} \]
\[ \text{if } (\text{CD};u) \text{ then } \text{M}(a) \in \text{Dec}_B(t,t,t); \]
\[ \text{if } (\text{EX};u) \text{ then } \text{M}(a) \in \text{Cle}_B(0); \]
\[ \text{if } (\text{FU};i,j') \text{ then } \text{M}(a) \in \text{Cle}_B(f_{\text{int}}(i,j')); \]
\[ \text{if } (\text{GB};0,0) \text{ then } \text{M}(a) \in \text{Dec}_B(t,0,0'); \]
\[ \text{if } (\text{NG};u) \text{ then } \text{M}(a) \in \text{Dom}_B; \]
\[ \text{if } (\text{OB};u) \text{ then } \text{M}(a) \in \text{Dec}_B(t,t,t); \]
\[ \text{if } (\text{EQ}) \text{ then } \text{M}(a) \in \text{Cle}_B; \]
\[ \text{if } (\text{SE};u) \text{ then } \text{M}(a) \in \text{Cle}_B(0); \]
\[ \text{if } (\text{ST}) \text{ then } \text{M}(a) \in \text{Cle}_B(t); \]
\[ \text{if } (\text{CP};0;i') \text{, } (\text{PQ};0;i') \text{, or } (\text{PN};0;i') \text{ then } \text{M}(x,a) \in \text{Wen}_B(0') \text{ and } \text{FP}_B(M(x,a)) \subseteq \{x\}; \]
\[ \text{if } (\text{AL};u) \text{, } (\text{AV};u) \text{, } (\text{CC};u); \text{, } (\text{CJ};u); \text{, } (\text{CM};u); \text{, } (\text{CV};u); \text{, } (\text{PF};u); \]
\[ \text{if } (\text{SN};u) \text{, } (\text{SC};u); \text{, } (\text{SM};u); \text{, or } (\text{VP};u); \text{ then } \text{M}(x,a) \in \text{Wen}_B(t) \text{ and } \text{FP}_B(M(x,a)) \subseteq \{x\}; \]
\[ \text{if } (\text{SN};u); \text{, } (\text{CN};u): \text{, or } (\text{SN};u): \text{ then } \text{M}(x,a) \in \text{Ras}_B(x) \text{ and } \text{FP}_B(M(x,a)) \subseteq \{x\}; \]
\[ \text{if } (\text{EI};u); \text{ then } \text{M}(x,x',a) \in \text{Ras}_B(x') \text{ and } \text{FP}_B(M(x,x',a)) \subseteq \{x,x'\}; \]
\[ \text{if } (\text{ET};u); \text{ then } \text{M}(x,x',x'',a) \in \text{Ras}_B(x'') \text{ and } \text{FP}_B(M(x,x',x'',a)) \subseteq \{x,x',x''\}; \]
\[ \text{if } (\text{NP};u) \text{ then } \text{M}(x,a) \in \text{Wen}_B(t) \text{ and } \text{FP}_B(M(x,a) \in \text{FP}_B(s) \subseteq \{x\} \text{ for each } d \in \text{Wen}_B(t); \]
\[ \text{if } (\text{PE};u) \text{, } (\text{PN};u) \text{, or } (\text{QG};u) \text{ then } \text{M}(x,a) \text{ and } d \in \text{Wen}_B(t) \text{ and } \text{FP}_B(M(x,a) \in \text{FP}_B(s) \subseteq \{x\} \text{ for each } t \in \text{Typ}_B \text{ and each } d \in \text{Wen}_B(t). \]
8.2. Translating the general rules

In the translation rules given below, \( x \) will be an arbitrary placeholder in \( \text{Plh}_b(G) \) and \( x' \) an arbitrary placeholder in \( \text{Plh}_b(G') \).

A placeholder \( y \) of type \( \tau \) 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 \( \tau' \) 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, \( \delta \) will always denote the derivation tree with root label \( \langle \text{AD} \rangle \) and \( \gamma \) always the derivation tree with a root label of the form \( \langle \text{AL:}\tau \rangle \).

In all translation rules below, \( a, b, \gamma, \) and \( \delta \) denote derivation trees. In (3), for instance, \( a \) and \( b \) denote derivation trees with a root label of the form \( \langle \text{NP1:}\tau \rangle \) and \( \langle \text{VP1:}\tau \rangle \), respectively. As it stands, the left hand side does not contain a derivation tree. For the sake of simplicity, we wrote \( M(a) \), where we should write \( M((\text{NP1:}\langle\text{AD}\rangle:\langle\text{AL:}\tau\rangle)) \), according to L1.2(2) and the notation for 2-tuples introduced in chapter C. A similar remark holds for the other translation rules. Notably, a seemingly circular translation such as in (5) should be read as \( M((\text{NP1:}\langle\text{AD}\rangle:\langle\text{AL:}\tau\rangle)) = M(a) \).
1. \text{M(Give} a, b) = M(a) 
2. \text{M(If it is true that} a)? = M(a) 
3. \text{M(a)} = M(a) 
4. \text{M(x, a)} = M(x, a) 
5. \text{M(x, a)} = M(x, a) 
6. \text{M(x, a)} = M(x, a) 
7. \text{M(x, a)} = \{M(x, a) \land M(x, a)\} 
8. \text{M(x, a)} = M(x, a) 
9. \text{M(x, a)} = \{M(x, a) \land M(x, a)\} 
10. \text{M(x, a)} = \{M(x, a) \land M(x, a)\} 
11. \text{M(x, a)} = M(x, a) 
12. \text{M(x, a)} = M(x, a) 
13. \text{M(x, a)? y} = \{M(x, a) \land M(x, y)\} 
14. \text{M(x, a)? y} = \{M(x, a) \land M(x, y)\} 
15. \text{M(x, a)} = M(x, a) 
16. \text{M(x, whose} a)? = \{y \leftarrow M(x, a)\} M(y, b) 
17. \text{M(x, a)} = M(x, a) 
18. \text{M(x, a)} = M(x, a) 
19. \text{M(x, a)? y} = \{M(x, a) \land M(x, y)\} 
20. \text{M(x, a)? y} = \{M(x, a) \land M(x, y)\} 
21. \text{M(x, a)? y} = \{M(x, a) \land M(x, y)\} 
22. \text{M(x, a)? y} = \{M(x, a) \land M(x, y)\} 
23. \text{M(x, a)? y} = \{M(x, a) \land M(x, y)\} 
24. \text{M(x, a)? y} = \{M(x, a) \land M(x, y)\} 
25. \text{M(x, a)? y} = \{M(x, a) \land M(x, y)\} 
26. \text{M(x, a)? y} = \{M(x, a) \land M(x, y)\} 
27. \text{M(x, a)} = M(x, a) 
28. \text{M(x, a)} = M(x, a) 
29. \text{M(a) times} = M(a) 
30. \text{M(a once} = \{3 \times M(a)\} 
31. \text{M(a) = M(a)} 
32. \text{M(a)} = M(a) 
33. \text{M(a)} = M(a) 

100
<table>
<thead>
<tr>
<th>Line</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>$M(x,a) = M(x,a)$</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>$M(x,a)$</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>$M(x,a)$</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>$M(x,a^\beta) = {y \leftarrow M(x,b) \mid M(y,a)}$</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>$M(x,\text{the } a^\beta) = M(x,a)$</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>$M(x,a^\gamma b) = {M(x,a) \mid M(x,\gamma)}$</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>$M(x,a^\gamma b) = {M(x,a) \mid M(x,\gamma)}$</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>$M(x,a) = (x \in M(a))$</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>$M(x,a) = {x \leftarrow M(a)}$</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>$M(x,a)$</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>$M(a)$</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>$M(\text{the } a^\beta \text{ of all } \gamma) = M(a) \cup {M(x,a) \cup M(x,\gamma)}$</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>$M(\text{the number of } a) = \sum M(y,a) \cdot 1$</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>$M(x,a)$</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>$M(x,a^\beta) = M(y,b) \cup {x \leftarrow M(y,a)}$</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>$M(x,a^\beta) = M(x,a)$</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>$M(a)$</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>$M(a)$</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>$M(a)$</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>$M(a)$</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>$M(a)$</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>$M(\beta) = {\exists M(a) \mid M(\beta)}$</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>$M(a)$</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>$M(\text{one}) = 1$</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>$M(\text{zero}) = 0$</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>$M(\text{two}) = 2$</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>$M(\text{three}) = 3$</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>$M(\text{at least}) = \geq$</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>$M(\text{at most}) = \leq$</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>$M(\text{exactly}) = = \geq$</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>$M(\text{more than}) = \rangle$</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>$M(\text{less than}) = \langle$</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>$M(\text{total}) = \Sigma$</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>$M(\text{minimal}) = \text{MIN}$</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>$M(\text{maximal}) = \text{MAX}$</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>$M(\text{average}) = \text{AVE}$</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>$M(\text{some}) = \exists$</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>$M(\text{no}) = \bar{\exists}$</td>
<td></td>
</tr>
</tbody>
</table>
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 <3C;1,S>, we have to prove that \( M(x, \text{whose } a_1) \in \mathcal{W}_B^+ \) and \( \mathcal{P}_B^+(M(x, \text{whose } a_1)) \notin \{x\} \). We can use the induction hypotheses for \( M(x, a_1) \) and \( M(y, b) \), the subexpressions occurring on the right hand side of the translation rule. (We recall from the beginning of this section that \( y \in \mathcal{P}L_B^+(\tau^*) \).)

The root label of \( \tau \) is <UP;3,A,1>, so the IH for \( M(y, b) \) is:

(a) \( M(y, b) \in \mathcal{W}_B^+ \) and

(b) \( \mathcal{P}_B^+(M(y, b)) \notin \{y\} \).

The root label of \( \tau \) is <UP;3,A,1>, so the IH for \( M(y, b) \) is:

(c) \( M(y, b) \in \mathcal{W}_B^+ \) and

(d) \( \mathcal{P}_B^+(M(y, b)) \notin \{y\} \).

From (a), (c), translation rule (16), and L4.1(7) we conclude that \( M(x, \text{whose } a_1) \in \mathcal{W}_B^+ \). Furthermore,

\[
\begin{align*}
\mathcal{P}_B^+(M(x, \text{whose } a_1)) &= \mathcal{P}_B^+(y \leftarrow M(x, a_1)M(y, b)) \\
&= \mathcal{P}_B^+(M(x, a_1)) \cup \{y\} - \{y\} \\
&= \{x\} \cup \{y\} - \{y\} \\
&= \{x\}
\end{align*}
\]

(48): According to the condition associated with a derivation tree with a root label of the form <3N;0>, we have to prove for each \( \tau^* \in \mathcal{P}_B^+ \) and each \( d \in \mathcal{W}_B^+(\tau^*) \) that \( M(x, a_1) \& d \in \mathcal{W}_B^+ \) and \( \mathcal{P}_B^+(M(x, a_1) \& d) \notin \mathcal{P}_B^+(d) \). We can use the induction hypotheses for \( M(y, b) \) and \( M(y, a_1) \), where \( y \in \mathcal{P}L_B^+(\tau) \) and the root.
labels of $a$ and $b$ are $\langle \text{FR}; r; o \rangle$ and $\langle \text{PV}; r \rangle$, respectively. The IH for $M(y,a)$ is:

(a) $M(y,a) \in W_B(\epsilon)$ and

(b) $FP_B(M(y,a)) \subseteq \{y\}$.

The IH for $M(y,b)$ is that for every $r' \in \text{Type}_B$ and every $e \in W_B(r')$:

(c) $M(y,b) \cdot e \in W_B(r')$ and

(d) $FP_B(M(y,b) \cdot e) \subseteq FP_B(e) \cdot \{y\}$.

The proof, for any $r' \in \text{Type}_B$ and any $d \in W_B(r')$, now runs as follows. Since $x \in \text{Fl}_B(\epsilon)$, it follows from (a) and L4.1(7) that $[x \mapsto M(y,a)]d \in W_B(r')$. Hence, by translation rule (46) and the induction hypothesis (c), $M(x,a) \cdot d = M(y,b)[x \mapsto M(y,a)]d \in W_B(r')$, q.e.d. Furthermore,

$$FP_B(M(x,a) \cdot d)$$

$=$ $FP_B(M(y,b)[x \mapsto M(y,a)]d)$

$\subseteq FP_B([x \mapsto M(y,a)]d) \cdot \{y\}$

$=$ $FP_B([x \mapsto M(y,a)]d) \cdot \{y\}$

$=$ $FP_B(M(y,a)) \cup [FP_B(d) - \{x\}] - \{y\}$

$=$ $[FP_B(d) - \{x\}] - \{y\}$

$\subseteq FP_B(d) - \{x\}$

by translation rule (46)

by (d)

by D4.6,

clause (7)

by (c), see below

using (b)

Above, $(\cdot)$ 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

103
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:

$$
\begin{array}{c}
\text{SP} \\
\downarrow \\
\text{PT-SP} \\
\downarrow \\
\text{PT-P} \\
\end{array}
$$

In our hospital database, has treated is a terminal string of the nonterminal $<\text{ET}, g, \text{SP}, P, \text{PT}>$. 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, $H(x, x', x'' \text{has treated})$ should express that

$x''$ is a treatment of $x'$ by $x$.

A correct choice would be

$$
H(x, x', x'' \text{has treated}) = x'' \in C \left( ^{\prime \prime} \text{PT-P inv } x' \right) \land \left( ^{\prime \prime} \text{PT-SP } x'' \right) = x
$$

In general, let $<g,h>$ be a type 2 skeleton, $\{D, D', M\} \subseteq \text{dom}(g)$, and $[C, C'] \subseteq \text{dom}(h)$ such that $h(C) = (M; D)$ and $h(C') = (M; D')$. In a figure:

Furthermore, let $B$ be a $\text{CH}$-basis fit for $<g,h>$, $d \in \text{Plh}_B(D)$, $d' \in \text{Plh}_B(D')$, and $m \in \text{Plh}_B(M)$. For a derivation tree $a$ with root label $<\text{ET}, u; D; D'; M>$, $H(d, d', m, a)$ is equal to

$$
n \in \left( ^{\prime}\text{C' inv } d' \right) \land \left( ^{\prime}\text{C' rel } a \right) = a
$$

or to an equivalent expression in $\text{Rest}_B(m)$, provided that $C$ and $C'$ are the connector indices that $a$ refers to. Various examples of existential transitive verb phrases can be found in the appendix.
8.3. Some examples

With the production rules

\[ <\text{NN}_1, s_1, s_2> \rightarrow \text{specialist} \]
\[ <\text{NN}_1, s_1> \rightarrow \text{patient} \]
\[ <\text{ET}_1, s_1, s_2> \rightarrow \text{has treated} \]
\[ <\text{CA}_1, p_l> \rightarrow \text{50} \]

the request near the end of the previous section can be completely analyzed. In the following translation, \text{specialist} is abbreviated by \text{a2}, \text{patient} by \text{a4}, the noun phrase \text{no specialist} by \text{a2}, \text{has treated} by \text{a3}, the noun phrase \text{some patient} by \text{a3}, the adverbial of degree more than 50 times by \text{a3}, and the comparative \text{more than} by \text{a4}. Furthermore, the concatenation of \text{a3}, \text{a3}, and \text{a3} is abbreviated by \text{a2}. We also indicate the translation rules used.

\[ M(\text{Is it true that a2 a2?}) \]
\[ = M(\text{a2 a2}) \]
\[ = M(\text{no} M(\text{a2,a3}), M(\text{a3,a3})) \]
\[ = \exists \ M(\text{a2,a3}) : M(\text{a3,a3}) : \tau \]
\[ = \exists \ M(\text{a2,a3}) : \exists \ M(\text{a3,a3}) : (3 > M(\text{a3,a3})) M(\text{a3,a3}) : \tau \]

With the translation rules

\[ M(\text{specialist}) = s \in \mathcal{VP} \]
\[ M(\text{patient}) = p \in \mathcal{VP} \]
\[ M(\text{has treated}) = t \in (\mathcal{VP} \text{ inv p}) \land ((t.\text{SNR}) = (t.\text{ENR})) \]
\[ M(\text{50}) = 50 \]

the final translation is as follows:

\[ M(\text{Is it true that no specialist has treated some patient more than 50 times?}) = \]
\[ \exists s \in \mathcal{SP} : \exists p \in \mathcal{VP} : (3 > 50) t \in (\mathcal{VP} \text{ inv p}) \land ((t.\text{SNR}) = (t.\text{ENR})) : \tau \]

As a second example, we translate the request

Give name, number, and department number of the manager of each department.
concerning our employees–departments database. In chapter 7, this request was already partly analyzed. With the production rules

\[ \text{<mn;ai;dep> ::= department} \]
\[ \text{<tu;dep;empl> ::= manager} \]
\[ \text{<ar;empl;ste> ::= name} \]
\[ \text{<ar;empl;int> ::= number} \]

|department number

this analysis can be completed. In the following translation, \text{name} is abbreviated by \text{o4}, \text{number} by \text{o5}, \text{department number} by \text{o6}, \text{manager} by \text{o3}, \text{department} by \text{o1}, and the \text{manager of} by \text{o2}. Furthermore, \text{a4} stands for \text{o4}, and \text{a5} of \text{a2} stands for \text{o5}. \text{a2} stands for \text{o2}, \text{a4} for \text{o4}, \text{a1} for \text{o1} each \text{v1}.

\[ M(\text{Give } a0) \]
\[= M(0) \square \]
\[= $M(d, v1); M(d, o1) \square \]
\[= $M(d, v1); \{e \leftarrow M(d, o2)\} M(e, o2) \square \]
\[= $M(d, v1); \{e \leftarrow (M(a3) \oplus d)\} M(e, o2) \square \]

where

\[ M(e, o2) \]
\[= \{M(e, a4) \}; M(a, o4) \}
\[= \{M(e, a5) \}; (M(e, o5) \oplus M(e, o5)) \}
\[= \{M(e, a6) \}; (e, M(a5)) \}; (e, M(a5)) \}

With the translation rules:

\[ M(\text{department}) = d \in \text{DEP} \]
\[ M(\text{manager}) = \text{MANAGEROF} \]
\[ M(\text{name}) = \text{Name} \]
\[ M(\text{number}) = \text{NR} \]
\[ M(\text{department number}) = \text{DPT} \]

the final translation is as follows:
\$ \text{deep: } [e \rightarrow (\text{NAMEOF}\ @ d) \{(e, \text{NAME}); \{(e, \text{NR}); \{(e, \text{OPT})\})\}$

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 551005).

If \(a\) is a derivation tree with root label <\text{Yjdate}> then \(a\) must be translated in combination with a placeholder \(x\) of type \text{int}. We then require that \(M(x, a) \in \text{WE}_y(t)\) and \(FP_y(M(y, a)) \in \{x\}\).

If the root label of \(a\) is <\text{digit}> then \(M(a)\) will be the digit concerned or, formally, \(M(a) = \text{Fa}_a\), i.e., the frontier of the tree \(a\) (see chapter 3).

If the root label of \(a\) is <\text{month}>, <\text{D2}>, <\text{day}>, or <\text{year}> then \(M(a)\) will be a sequence (or "string") of 2 digits.

If the root label of \(a\) is <\text{Mdate}> then \(M(a)\) will be a string of 6 digits and, hence, an element of \text{Co}_y(\text{int}).

If the root label of \(a\) is <\text{Njdate}> then we require that \(M(a) \in \text{Co}_y(\text{int})\).

Finally, if the root label of \(a\) is <\text{PT}> then we require that \(M(a) \in \text{Binop}_y(\text{int}, \text{int}, \text{t})\).

We point out that no "new" (or "fresh") placeholders are needed in the translation rules below.
As an illustration, we translate the JV-phrase in the period January 4 to 20, 1986.

By SG04,

\[ M(x, \text{in the period January 4 to 20, 1986}) \]

\[
\left( \left( H(1986) \leq M(\text{January}) \leq 4 \right) \land (x \leq M(4)) \right) \land \left( x \leq M(\text{January}) \land M(\text{February}) \right)
\]

where \( M(1986) \leq M(\text{January}) \leq M(\text{February}) \)

and \( M(\text{January}) = 01 \)
and \( M(4) \wedge M(4) = 04 \)
and \( M(20) \wedge M(20) \wedge M(20) = 20 \)

Thus,

\[
M(x \text{ in the period January 4 to 20, 1986}) = \\
((960104 \leq x) \wedge (x \leq 960120))
\]

As a second example, we translate in January 1986:

\[
M(x \text{ in January 1986})
\]

\[
\equiv ((x \div 100) = M(1986)M(\text{January}))
\]

\[
= ((x \div 100) = 9601)
\]
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 case 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 I model

We start with the definition of a conceptual skeleton Ghs0 and then give the intuitive meaning of the table indices and attributes of Ghs0. The definition of a DB universe over Ghs0 will follow later on.

The set function Ghs0 is defined by

\[ \text{dom}(Ghs0) = \{ F, EXP, NRS, SP, DPT, RM, ADM, REL, TN, PT, H, TP \} \] and

\[ Ghs0(F) = \{ FNR, SN, AD, CTY, HET, BIL, BPF, SX \} \]
\[ Ghs0(EXP) = \{ ENS, SN, AD, CTY, SAL \} \]
\[ Ghs0(NRS) = \{ NNR, DNR, SVNR \} \]
\[ Ghs0(SP) = \{ ENR, DSR, LOC, NR, OND \} \]
\[ Ghs0(DPT) = \{ DNR, DSR, MNR, ENR \} \]
\[ Ghs0(RM) = \{ FNR, DNR, NR \} \]
\[ Ghs0(ADM) = \{ FAM, FITE, SNR, SNR, ARCE, RSN \} \]
\[ Ghs0(REL) = \{ FNR, FITE, RITE, INFAM \} \]
Gesp(TM) = \{TCD, TNR, TSNR, TAR\}
Gesp(PT) = \{PWR, TCD, NT, SNR, ASNR, DTE, LENGTH, TNR\}
Gesp(NR) = \{MCD, MNR, MNR2, DCD\}
Gesp(MP) = \{SNR, MCD, DTE, SNR, LENGTH, PD, AMT\} .

In other words, Gesp 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 medicines 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 [§ 77] - or as differentiation.)

The relevant features for a patient are his (or her) patient number (PWR), name (NM), address (AD), city (CTY), date of birth (BRT), blood group (BLC), rhesus factor (RFR), and sex (SEX).

Employee number (ENO), 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 Gesp(NRS) and Gesp(SF) contain an attribute DEN (for department number) and Gesp(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 (CDM), 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 (SVSR). 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 (SNR).
For a nursing room, its room number (SNR), 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 (DOTE) 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 (RN), the employee number of the specialist responsible
for that admittance (SNR), and an indication whether or not that
admittance is finished (FNSN, which stands for "archive"). Additional
features for releases are the date of release (DOTE) and the invoice
amount (INVAM).

The relevant features for a treatment are code (TC), name (TNK),
sort (TSRT), and tariff (TAR).

Relevant features for an application of a treatment are treatment
code (TC), 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 assistant (EHR, respec-
tively ASNR), treatment date (DTE), treatment room (TRN), and length
of treatment (LENGTH), in minutes.

The relevant features for a medicine are code (MC), name (MN),
sort (MSRT), and danger code (DGCD).

The relevant features for a medicine prescription are number of
the receiving patient (PNR), medicine code (MC), 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 SF tuple and over an EMP tuple. In order to
relate each SF tuple to the corresponding EMP tuple, Gexp(SF) contains
the attribute EHR. Similarly, Gexp(EMP) contains the attribute NNR.
Also the data of each release will be spread over two tuples. The
attributes PNR and DOTE in Gexp(REL) will be used to relate each REL
tuple to the corresponding ADM tuple.
Further on, we will define a DB universe over Ghap 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: Admissions last at least one night.
R12: The treating specialist and the assistant 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:

- \( \{\text{PNR}\} \) for \( P \)  
- \( \{\text{DNR}\} \) and \( \{\text{DSN}\} \) for \( DPT \)  
- \( \{\text{PNR},\text{INOTE}\} \) for \( ADM \)
- \( \{\text{ENR}\} \) for \( EMP \)  
- \( \{\text{TCD}\} \) and \( \{\text{TMN}\} \) for \( TH \)  
- \( \{\text{PNR},\text{INOTE}\} \) for \( REL \)
- \( \{\text{NNR}\} \) for \( NRS \)  
- \( \{\text{MCD}\} \) and \( \{\text{MNRR}\} \) for \( M \)  
- \( \{\text{PNR},\text{TCD},\text{NT}\} \) for \( FT \)
- \( \{\text{ENR}\} \) for \( SB \)  
- \( \{\text{PNR},\text{MCD},\text{DTS}\} \) for \( NP \)
- \( \{\text{NNR}\} \) for \( SM \)

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 NF table.

R20: The DPR table must contain all department numbers mentioned in the ADM table, the SP table, or the NM table.

R21: The IP table must contain all specialist numbers mentioned in the DPR table, the ADM table, the PT table (twice), or the NF table.

R22: The EM table must contain all room numbers mentioned in the ADM table.

R23: The Th table must contain all treatment codes mentioned in the PT table.

R24: The H table must contain all medicine codes mentioned in the NF table.

R25: The EMP table must contain all nurse numbers mentioned in the NM 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 Genp.

If \( n \in \mathbb{N} \) and \( m \in \mathbb{N} \) then:

\[
[m..n] \triangleq \{ k \in \mathbb{N} \mid m \leq k \land k \leq n \} ;
\]

\[
[m..] \triangleq \{ k \in \mathbb{N} \mid m \leq k \} .
\]

If \( A \) is a set, \( T \) is a table over \( A \), and \( a \in A \) then:

\[
Cv(a,T) \triangleq \{ t(a) \mid t \in T \} .
\]

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 \texttt{PROD} ... \texttt{END} as in

\texttt{PROD STRT : STREET, HNR: [1..] END}

which stands for \( \Pi(F_0) \) where \( F_0 \) is the set function \( \{(STRT;STREET), (HNR: [1..])\} \).

In our DB universe, \( STRT \) stands for "street" and \( HNR \) for "house number". The set \( STREET \) 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 $\mathcal{U}_{\text{Hep}}$ over $\mathcal{H}_{\text{ep}}$. By Di.4, $<\mathcal{H}_{\text{ep}},\mathcal{U}_{\text{Hep}}>$ is an example of a type I model.

For referential purposes, the formal counterparts of the fore-mentioned requirements (from the tenth one on) will be indicated by a corresponding number.
\[ T_{USP} = \{ t \in PRD | t(LOC) \neq t(BNR) \}; \]

\[ T_{USL} = \{ t \in PRSL | t(INDE) = t(INDS) \}; \]

\[ T_{USPT} = \{ t \in PRPT | t(SNR) \neq t(ASN) \}; \]

\[ E_{PS} = \{ T \in PPRP | (PRR) \text{ is u.i. for } T \}; \]

\[ E_{PMP} = \{ T \in PPRP | (PRR) \text{ is u.i. for } T \text{ and } \forall t \subseteq t \text{ such that } t(BNR) \leq 999 \text{ then } t(SAL) \leq \frac{4 \times (t'(SAL) \# T)}{t'(SAL) \# T} \}; \]

\[ N_{PS} = \{ T \subseteq PPRP | (PRN) \text{ is u.i. for } T \text{ and } C \subseteq C \text{ for } \forall t \subseteq t \text{ such that } t(BNR) = t'(BNR) \text{ then } t(DNR) = t'(DNR) \}; \]

\[ S_{PS} = \{ T \subseteq T_{USP} | (PRN) \text{ is u.i. for } T \text{ and } C \subseteq C \text{ for } \forall t \subseteq t \text{ such that } t(BNR) = t'(BNR) \text{ then } t(DNR) = t'(DNR) \}; \]

\[ E_{PDP} = \{ T \subseteq PDMP | (PRR) \text{ is u.i. for } T \text{ and } (PRM) \text{ is u.i. for } T \}; \]

\[ E_{PRP} = \{ T \subseteq PPRP | (PRR) \text{ is u.i. for } T \}; \]

\[ D_{PS} = \{ T \subseteq PPRP | (PRN) \text{ is u.i. for } T \text{ and } (PRM) \text{ is u.i. for } T \}; \]

\[ R_{PS} = \{ T \subseteq PRP | (PRR) \text{ is u.i. for } T \}; \]

\[ D_{PS} = \{ T \subseteq PPRP | (PRN) \text{ is u.i. for } T \text{ and } (PRM) \text{ is u.i. for } T \}; \]

\[ R_{PS} = \{ T \subseteq PRP | (PRR) \text{ is u.i. for } T \}; \]

\[ T_{TP} = \{ T \subseteq PRP | (PRM) \text{ is u.i. for } T \text{ and } (PRM) \text{ is u.i. for } T \}; \]

\[ E_{PMP} = \{ T \subseteq PRP | (PRR) \text{ is u.i. for } T \text{ and } (PRM) \text{ is u.i. for } T \}; \]

\[ T_{TP} = \{ T \subseteq PRP | (PRM) \text{ is u.i. for } T \text{ and } (PRM) \text{ is u.i. for } T \}; \]

\[ M_{PS} = \{ T \subseteq PRP | (PRM) \text{ is u.i. for } T \text{ and } (PRM) \text{ is u.i. for } T \}; \]

\[ M_{PS} = \{ T \subseteq PRP | (PRM) \text{ is u.i. for } T \text{ and } (PRM) \text{ is u.i. for } T \}; \]
**PSHS = PDEC**

\[ U_{sp} = \{ v \mid v \in \text{PSHS and} \]

\[ \text{CV}(\text{PNR}, v(P)) \geq \text{CV}(\text{PNR}, v(\text{ADM})) \cup \text{CV}(\text{PNR}, v(DPT)) \cup \text{CV}(\text{PNR}, v(\text{MH})) \] and \[(P219)\]

\[ \text{CV}(\text{DSN}, v(DPT)) \geq \text{CV}(\text{DSN}, v(\text{NRS})) \cup \text{CV}(\text{DSN}, v(\text{SP})) \cup \text{CV}(\text{DSN}, v(\text{RS})) \] and \[(P220)\]

\[ \text{CV}(\text{ENR}, v(\text{SF})) \geq \text{CV}(\text{ENR}, v(DPT)) \cup \text{CV}(\text{SNR}, v(\text{ADM})) \cup \text{CV}(\text{SNR}, v(\text{PT})) \cup \] \[(P221)\]

\[ \text{CV}(\text{SNR}, v(\text{PT})) \cup \text{CV}(\text{SNR}, v(\text{NP})) \] and \[(P222)\]

\[ \text{CV}(\text{SNR}, v(\text{EM})) \geq \text{CV}(\text{SNR}, v(\text{ADM})) \] and \[(P223)\]

\[ \text{CV}(\text{NNR}, v(\text{NP})) \geq \text{CV}(\text{NNR}, v(\text{NP})) \] and \[(P224)\]

\[ \text{CV}(\text{NNR}, v(\text{EM})) \geq \text{CV}(\text{NNR}, v(\text{NP})) \] and \[(P225)\]

\[ \text{CV}(\text{NNR}, v(\text{SF})) = (t(\text{EM}) \mid t \in \text{v(ENM)} \text{ and } t(\text{NMR}) \leq 999) \] and \[(P226)\]

\[ v(\text{REL}) \searrow \{ \text{PNR}, \text{IND THE} \} = \{ t \uparrow \{ \text{PNR}, \text{IND THE} \} \mid t \in \text{v(ADM)} \text{ and } t(\text{ARCH}) = 1 \} \] and \[(P227)\]

\[ v(DPT) \searrow \{ \text{DSN}, \text{NRS} \} \neq \{ \text{DSN}, \text{NRS} \} \] and \[(P228)\]

\[ \text{CV}(\text{ENR}, v(DPT)) = (t(\text{NRR}) \mid t \in v(\text{NRS}) \text{ and } t(\text{NRR}) = t(\text{SNR})) \] and \[(P229)\]

\[ v(DPT) \searrow \{ \text{DSN}, \text{NRS} \} \geq \{ t \uparrow \{ \text{DSN}, \text{NRS} \} \mid t \in v(\text{SF}) \text{ and } t(\text{SNR}) \in \text{CV}(\text{SNR}, v(DPT)) \} \] and \[(P230)\]

\[ (\forall t(\text{ADM}) \colon \forall t'(\text{REL}) : \text{if } t(\text{ADM}) = t'(\text{PNR}) \text{ and } t(\text{ARCH}) = 0 \]

\[ \text{then } t'(\text{NOTE}) \leq t(\text{IND THE}) \] and \[(P231)\]

\[ \forall t(\text{PT}) : \exists t'(\text{ADM}) : t(\text{PNR}) = t'(\text{PNR}) \text{ and } t'(\text{IND THE}) \leq t(\text{DTE}) \] and \[(P232)\]

\[ \text{if } t'(\text{ARCH}) = 1 \]

\[ \text{then } \exists r(\text{REL}) : (r \uparrow \{ \text{PNR}, \text{IND THE} \} = t' \uparrow \{ \text{PNR}, \text{IND THE} \} \]

\[ \text{and } t(\text{DTE}) < r(\text{NOTE}))]. \]
A.2. A nontrivial type 2 model

We will "extend" the type 1 model $\langle\text{Infsp},\text{Usps}\rangle$ to a type 2 model with 21 distinguished DB functions. The function $\text{Usps}$ introduces 21 connector indices - see section 1.2 - together with their source index and their target index:

\begin{align*}
\text{Usps}(\text{NRS-EMD}) &= (\text{NRS};\text{EMP}) & \text{Usps}(\text{DPT-NRS}) &= (\text{DPT};\text{NRS}) \\
\text{Usps}(\text{NRS-DPT}) &= (\text{NRS};\text{DPT}) & \text{Usps}(\text{DPT-SP}) &= (\text{DPT};\text{SP}) \\
\text{Usps}(\text{NRS-NRS}) &= (\text{NRS};\text{NRS}) & \text{Usps}(\text{NRS-OMP}) &= (\text{NRS};\text{DPT}) \\
\text{Usps}(\text{SP-EMP}) &= (\text{SP};\text{EMP}) & \text{Usps}(\text{MP-SP}) &= (\text{MP};\text{SP}) \\
\text{Usps}(\text{SP-DPT}) &= (\text{SP};\text{DPT}) & \text{Usps}(\text{MP-SP}) &= (\text{MP};\text{SP}) \\
\text{Usps}(\text{SP-SP}) &= (\text{SP};\text{SP}) & \text{Usps}(\text{MP-NP}) &= (\text{MP};\text{NP}) \\
\text{Usps}(\text{ADM-P}) &= (\text{ADM};\text{P}) & \text{Usps}(\text{PT-P}) &= (\text{PT};\text{P}) \\
\text{Usps}(\text{ADM-SP}) &= (\text{ADM};\text{SP}) & \text{Usps}(\text{PT-TM}) &= (\text{PT};\text{TM}) \\
\text{Usps}(\text{ADM-RE}) &= (\text{ADM};\text{RE}) & \text{Usps}(\text{PT-SP}) &= (\text{PT};\text{SP}) \\
\text{Usps}(\text{REL-ADM}) &= (\text{REL};\text{ADM}) & \text{Usps}(\text{PT-AST}) &= (\text{PT};\text{AST}) \\
\text{Usps}(\text{PT-ADM}) &= (\text{PT};\text{ADM}) \\
\end{align*}

Summarized informally: $\text{Usps}(a; b) = (a; b)$, except for $\text{PT-AST}$.

According to 3.1.3 we still have to define an "interpretation function" $\text{Infsp}$ over $\text{dom}(\text{Usps})$. This function tells which DB function each connector index stands for. For 19 of the 21 connector indices $C$, the DB function $\text{Infsp}(C)$ is an instance of the "standard" situation described after example 1.3. For $\text{REL-ADM}$, however, the "foreign key" consists of two attributes and for $\text{PT-ADM}$ there is not even a "foreign key". Nevertheless, the relation $\text{Infsp}(\text{PT-ADM})(v)$, which is defined below, happens to be a function over $v(\text{PT})$. This can be proved by using the definition of $\text{Usps}$, more specifically, by using the formal requirements (FR12), (FR16), (FR17), and (FR31).

For any type 1 model $\langle q; U \rangle$, $V \subseteq U$, $N \subseteq \text{dom}(q)$, $a' \in g(N)$, $D \subseteq \text{dom}(g)$, and $a' \in g(D)$, we define the following special variant of an equi-join (cf. [UL 80]):

\[
\text{Speq}(V, N, a, D, a') \quad D \subseteq \{(t, t') \in v(M) \times v(D) \mid t(a) = t'(a')\}.
\]
The definition of Ihsp now runs as follows: For every v ∈ Ihsp we define

\[
\begin{align*}
\text{Ihsp}(\text{NRs-EMP})(v) &= \text{Spec}(v, \text{NRs}, \text{SNr}, \text{EMP}, \text{ENr}) \\
\text{Ihsp}(\text{NRs-DPT})(v) &= \text{Spec}(v, \text{NRs}, \text{DNR}, \text{DPT}, \text{ENr}) \\
\text{Ihsp}(\text{NRs-NRs})(v) &= \text{Spec}(v, \text{NRs}, \text{SNr}, \text{NRs}, \text{ENr}) \\
\text{Ihsp}(\text{SP-EMP})(v) &= \text{Spec}(v, \text{SP}, \text{ENr}, \text{EMP}, \text{ENr}) \\
\text{Ihsp}(\text{SP-DPT})(v) &= \text{Spec}(v, \text{SP}, \text{SNr}, \text{DPT}, \text{ENr}) \\
\text{Ihsp}(\text{SP-SR})(v) &= \text{Spec}(v, \text{SR}, \text{LOC}, \text{SR}, \text{ENr}) \\
\text{Ihsp}(\text{ADM-P})(v) &= \text{Spec}(v, \text{ADM}, \text{Pnr}, \text{Pnr}) \\
\text{Ihsp}(\text{ADM-SP})(v) &= \text{Spec}(v, \text{ADM}, \text{SNr}, \text{SP}, \text{ENr}) \\
\text{Ihsp}(\text{ADM-PN})(v) &= \text{Spec}(v, \text{ADM}, \text{ENr}, \text{PN}, \text{ENr}) \\
\text{Ihsp}(\text{REL-ADM})(v) &= \\
\{(t,t') \in v(\text{REL}) \times v(\text{ADM}) \mid t \uparrow \{\text{Pnr}, \text{INr}\} = t' \uparrow \{\text{Pnr}, \text{INr}\}\}
\end{align*}
\]

\[
\begin{align*}
\text{Ihsp}(\text{DPT-NRs})(v) &= \text{Spec}(v, \text{DPT}, \text{SNr}, \text{SNr}, \text{SNr}) \\
\text{Ihsp}(\text{DPT-SP})(v) &= \text{Spec}(v, \text{DPT}, \text{SNr}, \text{SP}, \text{ENr}) \\
\text{Ihsp}(\text{SN-DPT})(v) &= \text{Spec}(v, \text{SN}, \text{SNr}, \text{DPT}, \text{ENr}) \\
\text{Ihsp}(\text{MP-P})(v) &= \text{Spec}(v, \text{MP}, \text{Pnr}, \text{Pnr}) \\
\text{Ihsp}(\text{MP-SP})(v) &= \text{Spec}(v, \text{MP}, \text{SNr}, \text{SP}, \text{ENr}) \\
\text{Ihsp}(\text{MP-M})(v) &= \text{Spec}(v, \text{MP}, \text{Mcd}, \text{M}, \text{Mcd}) \\
\text{Ihsp}(\text{PT-P})(v) &= \text{Spec}(v, \text{PT}, \text{Pnr}, \text{P}, \text{Pnr}) \\
\text{Ihsp}(\text{PT-TM})(v) &= \text{Spec}(v, \text{PT}, \text{Tcd}, \text{TM}, \text{Tcd}) \\
\text{Ihsp}(\text{PT-SP})(v) &= \text{Spec}(v, \text{PT}, \text{SNr}, \text{SP}, \text{ENr}) \\
\text{Ihsp}(\text{PT-ASp})(v) &= \text{Spec}(v, \text{PT}, \text{ASr}, \text{SP}, \text{ENr}) \\
\text{Ihsp}(\text{PT-ADM})(v) &= \\
\{(t,t') \in v(\text{PT}) \times v(\text{ADM}) \mid t(\text{Pnr}) = t'(\text{Pnr}) \text{ and} \\
t'(\text{INr}) \leq t(\text{DTE}) \text{ and} \\
\text{if } t'(\text{APr}) = 1 \text{ then} \\
\exists x(v(\text{SNr}): \\
( x \uparrow \{\text{Pnr}, \text{INr}\} = t' \uparrow \{\text{Pnr}, \text{INr}\} \text{ and} \\
t(\text{DTE}) < x(\text{NOTE}))\}
\end{align*}
\]
It follows from the definition of \( \text{Hisp} \) and \( \text{Hisp} \) that \(<\text{Hisp};\text{Hisp};\text{Hisp} ; \text{Hisp} > \) is an example of a type 2 model.

The functions \( \text{Hisp}(\text{NRS} \rightarrow \text{SP}) (v) \), \( \text{Hisp}(\text{SP} \rightarrow \text{REI}) (v) \), and \( \text{Hisp}(\text{REL} \rightarrow \text{ADM}) (v) \) are one-to-one since the "foreign keys" \{\text{INR} \}, \{\text{ENR} \}, and \{\text{PSR};\text{INDT} \} are keys too (for the source indices \text{NRS} \), \text{SP} \, and \text{REL} \, respectively). We note that the concept of generalization applies here; see [55, 77].

We further note that the function \( \text{Hisp}(\text{DPT} \rightarrow \text{NRS}) (v) \) is also one-to-one. This is a consequence of the formal requirement (FR28). (Another consequence of (FR28) is that \{\text{NKR} \} is also a key for \text{DPT} \) in \(<\text{Hisp};\text{Hisp} > \) even though this was not explicitly required.)

It is easily checked that \( \text{Hisp}(\text{PT} \rightarrow \text{ADM}) (v) \) is the composition of \( \text{Hisp}(\text{ADM} \rightarrow \text{PT}) (v) \) after \( \text{Hisp}(\text{PT} \rightarrow \text{ADM}) (v) \). Furthermore, the composition of \( \text{Hisp}(\text{NRS} \rightarrow \text{DPT}) (v) \) after \( \text{Hisp}(\text{DPT} \rightarrow \text{NRS}) (v) \) is the identity function on \( v(\text{DPT}) \); this follows from (FR28) again.

Finally, let \( f_v \) be the function \( \text{Hisp}(\text{DPT} \rightarrow \text{SP}) (v) \); then \( f_v = f_v \circ \text{Hisp}(\text{SP} \rightarrow \text{DPT}) (v) = f_v \), because of (FR30).

The following pictures summarize much of the foregoing discussion. The first picture indicates the "direction" of the 21 distinguished \text{DB} \) functions: the four one-to-one functions are indicated by a "dashed" arrow (-----). The equation

\[ \text{Hisp}(\text{PT} \rightarrow \text{ADM}) (v) = \text{Hisp}(\text{ADM} \rightarrow \text{PT}) (v) \circ \text{Hisp}(\text{PT} \rightarrow \text{ADM}) (v) \]

is indicated by the symbol \( \oplus \) 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.
s154: <V:HM> := of <NH:NP>
s155:  by <NH:NP>
s156:  to <NP:P>
s157:  <V:VP> := of <NH:NP>
s158:  by <NH:NP>
s159:  assisted by <NH:SP>
s160:  during <NH:ACEN>
s161:  <V:ACEN> := of <NH:NP>
s162:  by <NH:NP>
s163:  to <NH:NP>
s164:  to <NH:NP>
s165:  <V:REL> := of <NH:NP>
s166:  <V:MOD> := <V:DATE>
s167:  <V:MOD> := <V:DATE>
s168:  <V:EXT> := <V:DATE>
s169:  <V:MOD> := <V:DATE>
s170:  <V:ACEN> := for <NH:ACEN>
s171:  to <NH:NP>
s172:  to <NH:NP>
s173:  <V:DATE>
s174:  <V:REL> := <V:DATE>
s175:  <V:PP> := in treatment room <NH:ACEN>
s176:  during <NH:ACEN>
s177:  <V:DATE>
s178:  <V:MOD> := for <NH:INT> days
s179:  to <NP:P>
s180:  <V:ACEN> := by <NH:NP>
s181:  <V:ACEN> := has
s182:  has
s183:  have
s184:  admitted
s185:  treated
s186:  assisted
s187:  performed
s188:  used
s189:  prescribed
s190:  released
s191:  in <NH:CTYP>
s192:  in <NH:CTYP>

s193:  lived
s194:  live
s195:  duty
s196:  on
s197:  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.

$S01$: \( H(x, \alpha) = H(x, \alpha) \).

$S02$ - $S13$: In each of these cases, the root label of the derivation tree is of the form \(<\text{NRS}; \sigma; c>\). The translation of such a derivation tree \( \alpha \) in combination with a placeholder \( x \) of type \( c \) is as follows:

\[ H(x, \alpha) = x \in \mathcal{V}_c \]

$S14$: \( H(x, \text{female}) = ((x, \text{ SEX}) = \varnothing) \)

$S15$: \( H(x, \text{male}) = ((x, \text{ SEX}) = \sigma) \)

$S16$: \( H(x, \text{past}) = ((x, \text{ ARCH}) = 1) \)

$S17$: \( H(x, \text{current}) = ((x, \text{ ARCH}) = 0) \)

$S18$: \( H(x, \text{dangerous}) = ((x, \text{ MDCD}) \geq 16) \)

$S19$ - $S33$: In each of these cases, the root label of the derivation tree is of the form \(<\text{FU}; \sigma; \sigma'>\). The translation of such a derivation tree \( \alpha \) is of the form

\[ H(\alpha) = \mathcal{V}_c \]

where \( C \) is a connector index with source index \( \sigma \) and target index \( \sigma' \), i.e., \( C \in \text{dom}(\text{NRS}) \) and \( \text{NRS}(C) = (\sigma, \sigma') \). For each of the 15 cases, the corresponding connector index is given below.

$S19$ NRS-DFT | $S23$ DFT-NRS | $S27$ MP-P | $S30$ PT-P
$S20$ NRS-NRS | $S24$ DFT-SP | $S28$ MP-SP | $S31$ PT-SP
$S21$ SP-DFT | $S25$ 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 \( \alpha \) with a root label of the form \(<\text{FU}; \sigma; \sigma'>\) will result in an attribute of the table index \( \sigma \), i.e., \( H(\alpha) \in \text{NRS}(\sigma) \). For each of the 58 cases, \( H(\alpha) \) is given below.
N(x,a) = x

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

\[ \mu(a) = Fx(a) \]

Thus, \( N(a) \) will be the frontier of the tree \( a \) (see Chapter 3), i.e., the original text.

\( \mu(\text{<char>}) = Fx(a) \)

\( N(\text{<char>}) = Fx(a) \)

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.

For each of the 4 cases, the translation is

\[ N(\text{<char>}) = Fx(a) \]

\[ N(\text{<char>}) = \Sigma \gamma \in \gamma N(\gamma,\text{NB}) \]

S100 - S112: Here, each derivation tree \( a \) with a root label of the form \( \text{<expr>} \) has two direct subtrees, namely, a derivation tree \( \delta \) with root label \( \text{<expr>} \) and a derivation tree \( \gamma \) with a root label of the form \( \text{<char>} \). The translation of \( a \) in combination with a placeholder \( x \) of type \( c \) is
\[ M(x, a) = \sigma M(x, b) \land ((x, b) \in M(y)) \]

where \( b \), an attribute of the table index \( a \), is given below:

\[
\begin{align*}
S100: & \text{ PNR} & S102: & \text{ ENR} & S105: & \text{ RNR} & S106: & \text{TCD} & S111: & \text{ MCD} \\
S101: & \text{ PNR} & S103: & \text{ RNR} & S106: & \text{ DNR} & S109: & \text{TCD} & S112: & \text{ NAM} \\
S104: & \text{ ENR} & S107: & \text{ DNM} & S110: & \text{TIN} \\
\end{align*}
\]

S113: \( M(x, \text{the } a \text{ of } b \text{ on } y) = M(y, b) \sigma x \in (\text{‘ADV-P inv } y) \land ((x, \text{INDTE}) = M(y)) \)

S114: \( M(x, \text{the } a \text{ of } b \text{ on } y) = \sigma M(x, a) \land ((x, \text{RDATE}) = M(y)) \land ((x, \text{PAR}) = M(y, b)(y, \text{PNR})) \)

S115 - S119: Here, each derivation tree \( D \) with a root label of the form \( \langle \text{PD}, \theta \rangle \) has one direct subtree, namely, a derivation tree \( S \) with a root label of the form \( \langle \text{CM}, \theta' \rangle \). The translation of \( \alpha \) in combination with a placeholder \( x \) of type \( \sigma \) is

\[ M(x, a) = (x, b) \in M(S) \]

where \( b \), an attribute of the table index \( a \), is given below:

\[
\begin{align*}
S115: & \text{ PLS} & S116: & \text{ WLM} & S117: & \text{ CTY} & S118: & \text{ WLM} & S119: & \text{ CTY} & S120: & \text{ DNR} \\
S121: & \text{ DNR} & S122: & \text{ WLM} & S123: & \text{ DNR} & S124: & \text{ WLM} & S125: & \text{ KSN} & S126: & \text{ PNR} \\
S127: & \text{ SNR} & S128: & \text{ RNR} & S129: & \text{ ASNR} & S130: & \text{ TCD} & S131: & \text{ PNR} & S132: & \text{ SNR} \\
S133: & \text{ ASNR} & S134: & \text{ TRM} & S139: & \text{ PNR} & S136: & \text{ MSRT} & S137: & \text{ NCM} & S138: & \text{ MCD} \\
\end{align*}
\]

S144: \( M(x, \text{of a units a day}) = (((x, \text{AMT}) \times (x, \text{FD})) \in M(a)) \)

S145: \( M(x, \text{of a units}) = (((x, \text{AMT}) \times (x, \text{FD})) \times (x, \text{LENGTH})) = M(a) \)

S146: \( M(x, \text{with a beds}) = (M(a) = \sum y \in (\text{‘RM-DPT inv } x)) : (y, \text{NB}) \)

S147 - S149: Here, the translation is of the form

\[ M(x, \text{with } a b) = M(a) y \in (\text{‘inv } y) : M(y, b) \]

where \( C \) is the "appropriate" connector index:

\[
\begin{align*}
S147: & \text{ IRS-DPT} & S148: & \text{ SP-DPT} & S149: & \text{ RH-DPT} \\
\end{align*}
\]
Here, each derivation tree $\alpha$ with a root label of the form $\langle b, \emptyset \rangle$ has one direct subtree, namely, a derivation tree $\beta$ with a root label of the form $\langle b, \emptyset \rangle$. A correct (but not "optimal") translation of $\alpha$ in combination with a placeholder $x$ of type $\sigma$ is

$$M(x, \sigma) = M(y, \beta)(\langle y, \emptyset \rangle = y)$$

where $C$ is a connector index with source index $\alpha$ and target index $\sigma'$. For each of the 14 cases, the corresponding connector index is given below.

<table>
<thead>
<tr>
<th>$S_{150}$</th>
<th>$S_{154}$</th>
<th>$S_{157}$</th>
<th>$S_{161}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NRS-DPT</td>
<td>MP-M</td>
<td>PT-P</td>
<td>ADM-P</td>
</tr>
<tr>
<td>$S_{151}$</td>
<td>$S_{155}$</td>
<td>$S_{158}$</td>
<td>$S_{162}$</td>
</tr>
<tr>
<td>NRS-NRS</td>
<td>MP-SP</td>
<td>PT-SP</td>
<td>ADM-SP</td>
</tr>
<tr>
<td>$S_{152}$</td>
<td>$S_{156}$</td>
<td>$S_{159}$</td>
<td>$S_{163}$</td>
</tr>
<tr>
<td>SP-DPT</td>
<td>MP-P</td>
<td>PT-AST</td>
<td>ADM-RM</td>
</tr>
<tr>
<td>$S_{153}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RN-DPT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For 13 of the 14 rules - not for $S_{160}$ - $\langle (y, \emptyset) = y \rangle$ can be replaced by $\langle (x, \emptyset) = (y, \emptyset) \rangle$ where $a$ and $a'$ are the following attributes (of the table indices $\sigma$ and $\sigma'$, respectively):

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$\sigma'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNR</td>
<td>DNR</td>
</tr>
<tr>
<td>SYNR</td>
<td>SNR</td>
</tr>
<tr>
<td>DOVR</td>
<td>DNR</td>
</tr>
<tr>
<td>DNR</td>
<td>DNR</td>
</tr>
<tr>
<td>MCO</td>
<td>MCD</td>
</tr>
<tr>
<td>SNR</td>
<td>ENR</td>
</tr>
<tr>
<td>PNR</td>
<td>PNR</td>
</tr>
</tbody>
</table>

$S_{164}$: $M(x, \text{to } b) = M(y, b)(\langle (y, \emptyset), \text{DNR} \rangle = \langle y, \text{DNR} \rangle)$

$S_{165}$: $M(x, \text{of } b) = M(y, b)(\langle x, \text{PNR} \rangle = \langle y, \text{PNR} \rangle)$

$S_{166}$ - $S_{169}$: Here, the translation is of the form

$$M(x, \sigma) = \langle y, (x, b) \rangle M(y, \alpha)$$

where $b$ is the "appropriate" attribute:

$S_{166}$ INDTÉ

$S_{167}$ ROTE

$S_{168}$ DÉTÉ

$S_{169}$ DÉTÉ
§170 - §179: We note that each of these adverbial phrases is also an appositive adjectival phrase. This follows from §125, §163, §164, §166, §167, §134, §160, §168, §141, and §156, respectively. The translation of each adverbial phrase will be the same as the translation of the corresponding appositive adjectival phrase. For instance, by §141:

\[ H(x, \text{for 3 days}) = (x \cdot \text{LENGTH}) = H(3) \]

§180 - §191: These rule forms are of a more general nature than the other production rules. §180, in combination with §182 and §183, 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.) §181, in combination with §184 and §185, 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 §180, we had to split the translation of a past participle in two: For a derivation tree $\bar{s}$ with a root label of the form $\langle PP, VP, S \rangle$ we have to define both $N(x, x', \bar{s})$ and $N(y, x', \bar{s})$, for all $y \neq \text{Ph}(r)$, $x \in \text{Ph}(r)$ and $x' \in \text{Ph}(r')$. Thus, both

\[ \text{Ph}(r) \times \text{Ph}(r') \times \{ \bar{s} \} \subseteq \text{dom}(H) \quad \text{and} \quad \{ \bar{s} \} \times \text{Ph}(r) \times \text{Ph}(r') \subseteq \text{dom}(H). \]

In continuation of section 6.1, we require that

\[ N(x, x', \bar{s}) \in \text{Rest} \quad \text{and} \quad PP_{\text{B}}(H(x, x', \bar{s})) \subseteq (x, x'), \quad \text{and} \]

\[ N(y, x', \bar{s}) \in \text{We}_{\text{B}}(\bar{s}) \quad \text{and} \quad PP_{\text{B}}(H(y, x', \bar{s})) \subseteq (y, x'). \]

If $\text{B}$ is the CL-basis for the conceptual language we translate to, the translation rules for the rule forms §180 and §181 are:

\[ \begin{align*}
\text{§180: } N(x, x', \text{aS by } \gamma) &= H(x, x', \text{aS by } \gamma) \\
N(x, x', \text{aS by } \gamma) &= (H(x, x', \text{aS by } \gamma) \land H(\gamma, y, x')) \\
\text{§181: } N(y, x, x', \text{aS}) &= (H(x, x', \text{aS}) \land H(y, x'))
\end{align*} \]

The reason for splitting up now follows from the translation rules for §180.
S182 - S185: No translation rules are needed here.

S186 - S192: We recall that for a derivation tree $\delta$ with a root label of the form $<\text{root label}>$, both $\mathcal{M}(x,x',\delta)$ and $\mathcal{M}(\delta, y, x')$ have to be defined; here $y \in \Phi_{\text{th}}(\cdot)$, $x \in \Phi_{\text{th}}(\cdot)$, and $x' \in \Phi_{\text{th}}(\cdot)$. Our first translation rule is of the form

$$\mathcal{M}(x,x',\delta) = x' \in \langle c \text{ inv } x \rangle$$

where $c$ is the "appropriate" connector index:

<table>
<thead>
<tr>
<th>S186</th>
<th>S187</th>
<th>S188</th>
<th>S189</th>
<th>S190</th>
<th>S191</th>
<th>S192</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>SNR</td>
<td>SNR</td>
<td>ASNR</td>
<td>SNR</td>
<td>PNR</td>
<td>SNR</td>
</tr>
<tr>
<td>b</td>
<td>ENR</td>
<td>ENR</td>
<td>ENR</td>
<td>ENR</td>
<td>ENR</td>
<td>ENR</td>
</tr>
</tbody>
</table>

S193: $\mathcal{M}(x,x',a \text{ released}) = x' \in \langle \text{REL} \land (x'.\text{PNR}) = (x.\text{PNR}) \rangle$

S194: $\mathcal{M}(x,a \text{ in } \delta) = (x.\text{CTY}) = \mathcal{N}(\delta)$

S195: $\mathcal{M}(x,a \text{ in } \delta) = (x.\text{CTY}) = \mathcal{N}(\delta)$

S196 - S197: No translation rules are needed (cf. S194 and S195).

S198: $\mathcal{M}(x,a \text{ duty}) = (x.\text{Duty}) = \mathcal{N}(\delta)$

S199: $\mathcal{N}(\text{om}) = 1$

S200: $\mathcal{N}(\text{off}) = 0$
REFERENCES

[AA 76] K.A. Archib and S. Alagić: The design of well-structured and correct programs.


[CC 81] CODASYL Programming Language Committee:

[Da 81] C.J. Date: An introduction to database systems.
Addison-Wesley, Reading (Mass.), 1981.

[Di 68] E.W. Dijkstra: Go to statement considered harmful.
Comm. of the ACM 11 (1968), pp. 147-149.


[He 94] G. Hemerik: Formal definitions of programming languages as a basis for compiler construction.

[HME 73] J. Hintikka, J. Moraweski, and P. Suppes (eds.):
Approaches to natural language.

[HW 73] C.A.R. Hoare and N. Wirth: An axiomatic definition of the programming language PASCAL.

[Kr 68] D. Knuth: Semantics of context-free languages.
In [Pe 71], pp. 95-109.

[MLB 76] M. Marcus, N.F. Ledgard, and G.V. Bochmann:
A sampler of formal definitions.

[Ko 73] R. Montague: The proper treatment of quantification in
ordinary English.
In [AMS 73], pp. 221-242; reprinted in [Th 74], pp. 247-270.


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.


[VW 65] A. van Wijngaarden: Orthogonal design and description of a
formal language.

[VW 75] A. van Wijngaarden: Revised report on the algorithmic
language ALGOL 68.
<table>
<thead>
<tr>
<th>SUBJECT INDEX</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>abbreviation facility</td>
<td>29</td>
</tr>
<tr>
<td>ambiguous</td>
<td>19</td>
</tr>
<tr>
<td>argument</td>
<td>21</td>
</tr>
<tr>
<td>arrangement</td>
<td>11</td>
</tr>
<tr>
<td>- sequential -</td>
<td>11</td>
</tr>
<tr>
<td>- attribute</td>
<td>5</td>
</tr>
<tr>
<td>Backus-Naur Form</td>
<td>16</td>
</tr>
<tr>
<td>BNF</td>
<td>16</td>
</tr>
<tr>
<td>cfg</td>
<td>15</td>
</tr>
<tr>
<td>CL-basis</td>
<td>20</td>
</tr>
<tr>
<td>- fit for</td>
<td>75</td>
</tr>
<tr>
<td>closed</td>
<td></td>
</tr>
<tr>
<td>- expression</td>
<td>34</td>
</tr>
<tr>
<td>- query</td>
<td>34</td>
</tr>
<tr>
<td>composition</td>
<td>1</td>
</tr>
<tr>
<td>- generalized -</td>
<td>3</td>
</tr>
<tr>
<td>concatenation</td>
<td>2</td>
</tr>
<tr>
<td>- generalized -</td>
<td>2</td>
</tr>
<tr>
<td>conceptual</td>
<td></td>
</tr>
<tr>
<td>- model</td>
<td>5</td>
</tr>
<tr>
<td>- skeleton</td>
<td>5</td>
</tr>
<tr>
<td>connector</td>
<td>10</td>
</tr>
<tr>
<td>- index</td>
<td>9</td>
</tr>
<tr>
<td>location</td>
<td>12</td>
</tr>
<tr>
<td>constant</td>
<td>21</td>
</tr>
<tr>
<td>- intensional -</td>
<td>21</td>
</tr>
<tr>
<td>context-free grammar</td>
<td>15</td>
</tr>
<tr>
<td>currency problem</td>
<td>46, 71</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>DB function</td>
<td>6</td>
</tr>
<tr>
<td>- reflexive -</td>
<td>6</td>
</tr>
<tr>
<td>DB snapshot</td>
<td>4</td>
</tr>
<tr>
<td>DB universe</td>
<td>5</td>
</tr>
<tr>
<td>derivation tree</td>
<td>18</td>
</tr>
<tr>
<td>determiner</td>
<td>21</td>
</tr>
<tr>
<td>direct access</td>
<td></td>
</tr>
<tr>
<td>- by contents</td>
<td>47</td>
</tr>
<tr>
<td>- by position</td>
<td>47</td>
</tr>
<tr>
<td>disambiguated</td>
<td></td>
</tr>
<tr>
<td>- language</td>
<td>19</td>
</tr>
<tr>
<td>- request</td>
<td>94</td>
</tr>
<tr>
<td>domain</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>enumeration</td>
<td>2</td>
</tr>
<tr>
<td>equi-join</td>
<td>120</td>
</tr>
<tr>
<td>expression</td>
<td></td>
</tr>
<tr>
<td>- closed</td>
<td>34</td>
</tr>
<tr>
<td>- well-formed</td>
<td>26</td>
</tr>
<tr>
<td>136-expression</td>
<td>65</td>
</tr>
<tr>
<td>foreign key</td>
<td>7</td>
</tr>
<tr>
<td>free placeholder</td>
<td>34</td>
</tr>
<tr>
<td>frontier</td>
<td>18</td>
</tr>
<tr>
<td>function</td>
<td>1</td>
</tr>
<tr>
<td>- into</td>
<td>1</td>
</tr>
<tr>
<td>- onto</td>
<td>1</td>
</tr>
<tr>
<td>- over</td>
<td>1</td>
</tr>
<tr>
<td>function-valued -</td>
<td>3</td>
</tr>
<tr>
<td>- interpretation -</td>
<td>8</td>
</tr>
<tr>
<td>ordering</td>
<td>11</td>
</tr>
<tr>
<td>set</td>
<td>3</td>
</tr>
<tr>
<td>typing</td>
<td>8</td>
</tr>
<tr>
<td>Term</td>
<td>Frequency</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>generalized</td>
<td>2</td>
</tr>
<tr>
<td>composition</td>
<td>2</td>
</tr>
<tr>
<td>concatenation</td>
<td>2</td>
</tr>
<tr>
<td>product</td>
<td>1</td>
</tr>
<tr>
<td>union</td>
<td>1</td>
</tr>
<tr>
<td>one-to-one</td>
<td>2</td>
</tr>
<tr>
<td>operation symbol</td>
<td>1</td>
</tr>
<tr>
<td>binary -</td>
<td>1</td>
</tr>
<tr>
<td>unary -</td>
<td>1</td>
</tr>
<tr>
<td>ordering function</td>
<td>1</td>
</tr>
<tr>
<td>placeholder</td>
<td>1</td>
</tr>
<tr>
<td>free -</td>
<td>1</td>
</tr>
<tr>
<td>post-assertion</td>
<td>1</td>
</tr>
<tr>
<td>precondition</td>
<td>1</td>
</tr>
<tr>
<td>pre-image</td>
<td>1</td>
</tr>
<tr>
<td>presupposition</td>
<td>1</td>
</tr>
<tr>
<td>production rule</td>
<td>1</td>
</tr>
<tr>
<td>projection</td>
<td>1</td>
</tr>
<tr>
<td>program</td>
<td>1</td>
</tr>
<tr>
<td>quasi-cfg</td>
<td>1</td>
</tr>
<tr>
<td>query</td>
<td>1</td>
</tr>
<tr>
<td>closed -</td>
<td>1</td>
</tr>
<tr>
<td>well-formed -</td>
<td>1</td>
</tr>
<tr>
<td>range</td>
<td>1</td>
</tr>
<tr>
<td>relation</td>
<td>1</td>
</tr>
<tr>
<td>request</td>
<td>1</td>
</tr>
<tr>
<td>disambiguated -</td>
<td>1</td>
</tr>
<tr>
<td>lexicon</td>
<td>1</td>
</tr>
<tr>
<td>restriction</td>
<td>1</td>
</tr>
<tr>
<td>root label</td>
<td>1</td>
</tr>
<tr>
<td>rule</td>
<td>1</td>
</tr>
<tr>
<td>lexical -</td>
<td>1</td>
</tr>
<tr>
<td>production -</td>
<td>1</td>
</tr>
<tr>
<td>form</td>
<td>1</td>
</tr>
<tr>
<td>set</td>
<td>1</td>
</tr>
<tr>
<td>sequence</td>
<td>1</td>
</tr>
<tr>
<td>empty -</td>
<td>1</td>
</tr>
<tr>
<td>sequential</td>
<td>1</td>
</tr>
<tr>
<td>arrangement</td>
<td>1</td>
</tr>
<tr>
<td>storage structure</td>
<td>1</td>
</tr>
<tr>
<td>Term</td>
<td>Frequency</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>set function</td>
<td>3</td>
</tr>
<tr>
<td>skeleton</td>
<td>5</td>
</tr>
<tr>
<td>conceptual</td>
<td>8</td>
</tr>
<tr>
<td>type 2</td>
<td>10</td>
</tr>
<tr>
<td>snapshot</td>
<td>4</td>
</tr>
<tr>
<td>DB</td>
<td>4</td>
</tr>
<tr>
<td>type 1</td>
<td>5</td>
</tr>
<tr>
<td>type 2</td>
<td>10</td>
</tr>
<tr>
<td>source index</td>
<td>9</td>
</tr>
<tr>
<td>symbol</td>
<td>15</td>
</tr>
<tr>
<td>start</td>
<td>15</td>
</tr>
<tr>
<td>table</td>
<td>4</td>
</tr>
<tr>
<td>index</td>
<td>5</td>
</tr>
<tr>
<td>target index</td>
<td>9</td>
</tr>
<tr>
<td>terminal</td>
<td>15</td>
</tr>
<tr>
<td>vocabulary</td>
<td>15</td>
</tr>
<tr>
<td>nonterminal</td>
<td>15</td>
</tr>
<tr>
<td>terminal</td>
<td>15</td>
</tr>
<tr>
<td>well-formed</td>
<td>26</td>
</tr>
<tr>
<td>expression</td>
<td>26</td>
</tr>
<tr>
<td>query</td>
<td>32</td>
</tr>
<tr>
<td>tuple</td>
<td>1</td>
</tr>
<tr>
<td>n-tuple</td>
<td>2</td>
</tr>
<tr>
<td>type</td>
<td>21, 41</td>
</tr>
<tr>
<td>in the wider sense</td>
<td>41</td>
</tr>
<tr>
<td>type 1</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>5</td>
</tr>
<tr>
<td>snapshot</td>
<td>5</td>
</tr>
<tr>
<td>type 2</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>8</td>
</tr>
<tr>
<td>skeleton</td>
<td>8</td>
</tr>
<tr>
<td>snapshot</td>
<td>10</td>
</tr>
<tr>
<td>typing function</td>
<td>8</td>
</tr>
<tr>
<td>u.i.</td>
<td>5</td>
</tr>
<tr>
<td>unambiguous</td>
<td>19</td>
</tr>
<tr>
<td>uniquely identifying</td>
<td>5</td>
</tr>
</tbody>
</table>
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 vraagtales voor gegevensbanken. De semantiek van deze vraagtales 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, en is het belangrijkste verschil op conceptueel niveau tussen een geïntegreerde gegevensbank en een "class" verzameling bestaande de aanwezigheid van database-functies.

In hoofdstuk 2 wordt een wiskundig (opslag)model gedefinieerd waarmee de semantiek van programma's wordt op "direct-sequentieel" toegankelijke gegevensbanken kan worden beschreven.

In het tweede deel - de hoofdstukken 3 t/m 8 - worden drie soorten vraagtales (retrieval languages) beschouwd, te weten prograamertalen, 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 vraagtales 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-zichtige" prograamertalen die ook een kleine maar krachtige verzameling "database statements" bevat. 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 gevenenverwerking.

In hoofdstuk 6 worden vertalingen gegeven van de niet-procedurele vraagtaalen uit hoofdstuk 4 naar de procedurele vraagtaalen 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.

\[
\begin{array}{cccccc}
NT & \rightarrow & CT & \rightarrow & PT \\
7 & 8 & 4 & 5 & 5 \\
\end{array}
\]

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

STELLINGEN

behorende bij het proefschrift

DATABASE MODELS AND RETRIEVAL LANGUAGES

van

E.O. DE BROCQ

Eindhoven, 18 maart 1984
STELLINGEN

1. De stelling van Sagiv, Delobel, Parkin en Fagin dat voor Boole-
afhankelijkheden logische consequentie en database consequentie
equivalent zijn, is onjuist.

Z. Sagiv, C. Delobel, D.S. Parkin en R. Fagin: Is the equivalence between relational dependencies and a fragment
of propositional logic.

E.D. de Groc: On complete proof systems with an application to
positive and Boolean dependencies in database relations.

2. Verzamelingenleer is het middel bij uitstek voor het definieren
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 over-
zichtelijke en wiskundig goed onderbouwde manier gebeuren.

Deel II van dit proefschrift.

4. Het gebruik van "two-level grammars" kunnen grote delen van natuuri-
lijke talen op een natuurlijke wijze worden beschreven. De lat-
middelen van de transformationele grammatica zijn derhalve over-
bodig.

Hoofdstuk 7 van dit proefschrift.

5. Current-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 personenregistratiesystemen, archiefsystemen, voorraadssystemen 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.