PHLIQA 1, a question-answering system for data-base consultation in natural English

PHLIQA Project Group

II. The artificial languages and translation operations

In part I of this article \[1\] a description was given of the basic structure of PHLIQA 1. It was shown that a question put to it in natural English is not directly translated by the system, but subjected to three successive translation operations: from natural English to EFL (English-oriented Formal Language), from EFL to WML (World-Model Language), and from WML to DBL (Data-Base Language). The value of the expression obtained in DBL is then computed with the aid of the data contained in the database. During the translation operation English $\rightarrow$ EFL the question is parsed and its logical structure is established in the form of an expression in EFL. The translation operations from EFL via WML to DBL establish the relationship between words in natural English and primitive terms in the data base. This is done by translating the primitive terms of EFL, which correspond to natural English words, into expressions in DBL that have the same meaning.

In this second part we shall examine the nature of the artificial languages and deal in more detail with the translation operations \[2\]. It should first be noted that between EFL and WML and between WML and DBL the PHLIQA I system contains other, intermediate languages. These serve to divide up the translation process into small steps, which is important for clarity and ease of approach to the system in the design stage. It would be going too far to deal with these other languages here, though we shall touch upon them in passing.

In the section that now follows we shall discuss the rules of grammar and the semantic rules of the artificial languages. First of all we show how the expressions of these 'construction languages' — the constructions — are built up. We then discuss the assignment of 'types' to the constructions. The types make it possible to judge whether the constructions are 'well-formed'. Next we deal with the introduction of variables as primitive terms. The last part of the section is devoted to the semantic rules, which make it possible to assign a value to any construction in the language concerned, once the values of the primitive terms are given.

In the final section we consider the principal mechanisms of the translation operations and illustrate them with some examples. The first translation operation, English $\rightarrow$ EFL, is obviously different in character from the two other translation operations owing to the fact that English is a natural language, whereas in the other cases both the languages are construction languages.

The artificial languages

Constructions

The expressions of the artificial languages used in PHLIQA 1 are the constructions referred to above. An example of a construction is given in fig. 1. It is a branching diagram that resembles a tree (upside down). Compared with ordinary sentences, i.e. rows of letters and characters, constructions have the advantage that their syntactic structure is completely visible. This facilitates their treatment in a computer. The syntactic structure of a construction corresponds to its semantic structure, as will be explained below.


The construction in fig. 1a consists of two primitive terms, i.e. the language constants COMPUTERS and DUTCH, and what we have called a branching. A branching consists of a branching category (in this case "selection") and a number of selectors associated with this branching category ("head" and "mod" (= modifier), in the case of "selection"). The branching category determines the type of branching involved. The selectors point to what is termed a subconstruction. The names of the selectors are chosen so as to characterize the roles played by the subconstructions within the total construction. In fig. 1a the subconstructions are the primitive terms COMPUTERS and DUTCH. The subconstructions may have branchings themselves. An example is given in fig. 1b. To make complicated branching diagrams clearer in this article, we shall occasionally substitute green triangles containing a piece of text in natural English for certain subconstructions (fig. 1c). For simplicity the expression 'branching with the branching category a' will be abbreviated to 'a-branching'. A construction with an a-branching at the top will be referred to as an 'a-construction'. The construction in fig. 1b is thus an existential-quantification-construction containing a selection-construction as subconstruction.

Each branching has a corresponding semantic rule (further particulars are given in the last part of this section) that lays down how the value of a construction that is built up with the aid of that branching from subconstructions follows from the values of the subconstructions. In this way it is known what the value of every construction is, provided that the values of the primitive terms are given. The case in fig. 1a can again be taken as an example. We take the value of the language constant COMPUTERS to be The Set Of All Computers In The World and the value of the language constant DUTCH to be a function that yields the value TRUE when applied to an object in the Netherlands and FALSE for all other objects. The semantic rule for the selection-branching is: take from the set 'under' the selector "head" the subset consisting of the elements with the property that application of the function 'under' the selector "mod" yields the value TRUE. This subset is the value of the selection-construction. In our example that value is The Set Of All Computers In The Netherlands.

The second branching in fig. 1b is of the category "existential quantification". This category has the selectors "for some" and "holds". The subconstruction to which the selector "for some" points must relate to a set, and the subconstruction under the selector "holds" must relate to a predicate. The value of the existential-quantification-construction is TRUE if there is an element in the set for which the predicate is true, otherwise it is FALSE. The value of the complete construction in fig. 1b is thus TRUE if there are expensive computers in the Netherlands.

In addition to the branching categories "existential quantification" and "selection", which we have discussed at some length, we shall briefly mention here another four, which will be used later in examples (fig. 1d).

---

![Fig. 1.](image-url)

**Fig. 1.** a) Example of a construction. This represents the logical structure of the phrase: "Dutch computers". The construction contains one branching, with two selectors, and two primitive terms -- located at the end points. The 'branching category' is shown above the horizontal line of the branching. The selectors are indicated beside the vertical lines. b) Example of a construction with two branchings. This represents the logical structure of the sentence "Some Dutch computers are expensive". The existential-quantification-branching represents a well-known concept from logic: the existential quantifier. The construction in (a) occurs here as a subconstruction. c) An abbreviated form of the construction in (b). The triangle with the text in natural English replaces the selection-branching, which is a subconstruction of the complete construction. d) The branching categories "universal quantification", "application", "pair" and "equal". These are used in the examples of translations in EFL, WML and DBL (figs 9, 10, 11).
The first is the branching category "universal quantification". This has two selectors, "for all" and "holds". The selector "for all" indicates — by virtue of the type rules, which we shall presently discuss — a subconstruction that has a set as its value; the selector "holds" points to a subconstruction that corresponds to a predicate. The value of the construction is true only if the predicate under "holds" is true for all elements of the set that is the value of the subconstruction under "for all".

The second is the branching category "application". To this belong the selectors "fun" (= function) and "arg" (= argument). The value of the application-construction is the result of applying the function that is the value of the subconstruction under "fun" to the value of the subconstruction under "arg".

The third is the branching category "pair", with the selectors "1" and "2". These refer to two subconstructions, which thus become the first and the second member of a pair.

The fourth branching category is the category "equal", with the selectors "arg 1" and "arg 2". The value of the equal-construction is true if the values of the subconstructions under "arg 1" and "arg 2" are equal, and otherwise false.

The grammar

To define a construction language — a set of constructions — we need a grammar of that language, that is to say a specification of branchings and primitive terms, and rules telling us how constructions are built up from the branchings and the primitive terms.

To begin with, we must therefore have a set of branchings, each consisting of a branching category and the associated selectors. There must also be a set of primitive terms. The branchings can be regarded as a generalization of the language constructions like those that occur in logical languages, for example in proposition logic and predicate logic [3]. All artificial languages of PHLIQA 1 have the same branchings. As already mentioned in part I of this article, these languages differ in their sets of primitive terms.

The rules for forming a construction consist of two parts. The first part states that the primitive terms themselves must be counted among the constructions. The second part establishes that a construction is formed by taking a branching and connecting an arbitrary group of constructions C1, . . . , Cn to its group of selectors sel1, . . . , seln (fig. 2).

The subconstructions C1, . . . , Cn of the newly formed construction may be primitive terms or constructions that in their turn also contain branchings. When they are written out in full, however, all branching diagrams end with primitive terms.

Any arbitrary combination of branchings with primitive terms at the ends is a construction, according to the rules. This would imply that a construction language might be built that could also contain meaningless constructions. To exclude such expressions, the grammar must in addition comprise rules indicating which constructions belong to the language and which do not. In the case of PHLIQA 1 this has been done by introducing type rules, which assign a type to every construction. The system accepts the constructions with a 'permissible type' as belonging to the language concerned and the others as not belonging to it. An example of a construction with an unacceptable type is given in fig. 3.

Types

To ensure that all grammatically correct expressions of a construction language also have a meaning, we can extend the grammar with the rules mentioned earlier for the types [4].

The type rules are introduced as follows. First of all, a type language is defined, in much the same way as a construction language. The expressions of the type language are the types. Next, every expression of the construction language is assigned a type by means of

\[ a (sel_1 : C_1, sel_2 : C_2, \ldots , sel_n : C_n). \]

Fig. 2. The composition of a construction. The constructions C1, C2, . . . , Cn are connected to the branching with branching category a, to which the selectors sel1, sel2, . . . , seln belong. The rules for forming the construction are part of the grammar of the appropriate construction language. The construction C obtained can also be written:

\[ a (sel_1 : C_1, sel_2 : C_2, \ldots , sel_n : C_n). \]

Fig. 3. Example of a meaningless construction. The selector "head" should refer to a subconstruction that has the value of a set, whereas the selector "mod" should refer to a subconstruction that has the value of a predicate. The primitive term COMPUTERS cannot refer to a set and to a predicate at the same time. The construction is therefore meaningless.


a ‘type function’ \( (T) \). There is a special type, empty, reserved for all constructions regarded as ungrammatical.

Only constructions of a type not equal to empty are regarded as belonging to the language. The construction language thus obtained is referred to as a ‘typed construction language’.

A function that has a type of the form \( (\beta \rightarrow \text{truthvalue}) \), where \( \beta \) is an arbitrary type, is called a predicate. The type \( (\beta \rightarrow \text{truthvalue}) \) is therefore called ‘predicate type’. Such a predicate indicates a property that objects of the type \( \beta \) can have. If the function, applied to a particular object, yields the value \text{true}, the object has that property, otherwise it does not.

The type function \( T \) establishes a relation between the construction language and the type language by assigning a type to each construction. This is done by first of all establishing what \( T \) does with the primitive terms of the associated construction language. For the language constant \text{HOLLAND}, for example, we have:

\[
T(\text{HOLLAND}) = \text{country}.
\]

Then, for each branching it is established how the type of a construction that consists of that branching, with associated subconstructions, follows from the types of those subconstructions. For unwanted combinations of subconstructions the type of the construction is empty. In this way the system can compute the type of every construction.

To illustrate how the type function \( T \) works for constructions with branchings, we return to the construction in fig. 1a. Here the set type \{\text{machine}\} has been assigned to the language constant \text{COMPUTERS} and the type \( (\text{machine} \rightarrow \text{truthvalue}) \) has been assigned to the language constants \text{EXPENSIVE} and \text{DUTCH}. The application of \( T \) to a selection-branching implies that a check is made first of all to see whether the type of the subconstruction under the selector “head” is a set type, i.e. of the form \{\( \alpha \)\}, and whether the type of the subconstruction under the selector “mod” is a predicate type, in other words of the form \( (\beta \rightarrow \text{truthvalue}) \). Next a check is made to see whether the element type \( \alpha \) of the set ‘is contained in’ the argument type \( \beta \) of the predicate. (The relation ‘is contained in’ between
types will not be discussed further in this article; it is sufficient here to know that \( \alpha = \beta \).

In this case the selection-construction receives the same type as the subconstruction under the selector "head", here \{machine\}; in the other cases the construction receives the type \textit{empty}. In the branching category "existential quantification" \( T \) imposes the same kind of requirement on the agreement between the types of the subconstructions under the selectors "holds" and "for some". If this requirement is satisfied, the type truthvalue is assigned to the existential-quantification-construction. The construction in fig. 1 is again shown in fig. 4, but now with the types included. It will be evident from the foregoing that the application of \( T \) in the case of fig. 3 will lead to the type empty being assigned to the meaningless expression.

### Variables

The expressive scope of the typed construction languages has been enlarged by admitting \textit{variables} as primitive terms in addition to the kinds of language constants already mentioned. The PHLIQA 1 system is therefore also capable of representing functions that do not already occur as primitive terms. It does this by means of a branching category called "abstraction". This has two selectors, "var" (= variable) and "descr" (= descriptor). The subconstruction under "var" must be a variable. The subconstruction under "descr" may be any arbitrary construction in which the associated variable may occur. The value of an abstraction-construction is a function. With each value that the variable \( x \) can assume, this function associates the value of the subconstruction — for the same \( x \) — under "descr".

An example is the mathematical function \( f: x \rightarrow x + 5 \), where the variable \( x \) has a value range consisting of the set of integers. This function is given by the construction in fig. 5.

Like the other language constants, variables have a type. Thus, the variable \( x \) in fig. 5 has the type integer, which indicates that the value range comprises all integers. The subconstruction for \( x + 5 \) also has the type integer. From these two pieces of information the type function \( T \) assigns to the construction in fig. 5 the type \((\text{integer} \rightarrow \text{integer})\), a second example of a function type.

### Semantics

In the above discussion of the branchings we touched briefly on the associated semantic rules. The complete set of semantic rules is called the semantics of the related construction language. The semantic rules assign to each construction of the language a particular value, given an assignment of values to the primitive terms. Such an assignment of values to the primitive terms is called an \textit{interpretation} of the language.

In a typed construction language the interpretation proceeds in two phases. The first yields an \textit{interpretation} of the type language, referred to as the \textit{type interpretation}; on the basis of this, the second phase provides the \textit{interpretation} of the construction language itself, called the \textit{language interpretation}.

A type interpretation is an assignment of a \textit{value domain} to each atomic type. A value domain is a set of objects from the subject domain that are characterized by the associated type. The value domain of the atomic type \textit{country}, for example, could be the set of all countries. The value domain chosen in the PHLIQA 1 system is the set of the countries that are members of the E.E.C. Some atomic types, called formal types, have the \textit{same} value domain for each interpretation; for example, the type truthvalue always has the set \{true, false\} as its value domain. The non-formal atomic types are called \textit{referential} types, since they refer to "real" objects from the subject domain.

The semantic rules of the type language indicate how the value domain of a compound type follows from the value domains of the atomic types contained in it. Two examples will help to make this clear. The value domain of the compound type \{country\}, our first example, is defined by these rules as the set whose elements are the subsets of the value domain of the atomic type \textit{country}. The value domain of the com-
pound type \((\text{machine} \rightarrow \text{truthvalue})\), in a similar way to that of the first example, is the set of all (predicate) functions that are applicable to all elements in the value domain of the type machine and which, when applied, yield an element in the value domain of the type truthvalue. In this way the value domains of all compound types can be derived from the value domains of the atomic types.

A language interpretation is an assignment of a value to each primitive term of the construction language. This value assignment must be in accordance with the type of the primitive term concerned and the type interpretation; this means that the assigned value of a primitive term \(C\) should be an element of the value domain that has been assigned to the type of \(C\):

\[
\text{Value} (C) \in \text{Value domain} (T(C)),
\]

where \(T\) is the type function. If, for example, the language constant HOLLAND has the type country, and the type interpretation assigns to country the value domain consisting of all countries, then the value of HOLLAND must be one of these countries. Some primitive terms, called formal constants, have the same value for each interpretation. An example is the constant 5, whose value is always the number 5. The non-formal constants are called referential constants.

Every branching of the construction language has a semantic rule, which specifies how the value of the construction consisting of that branching together with the subconstructions present is derived from the values of those subconstructions. The foregoing sentence implies that the semantic structure of a construction corresponds to the syntactic structure — which is determined by the branchings. The semantic rules for a few examples have already been touched upon in our discussion of the branchings in the first part of this section.

The semantic rules and the definition of the type function for the branchings are matched in such a way that for each construction \(t\) of the language we can write:

\[
\text{Value} (t) \in \text{Value domain} (T(t)).
\]

To understand this properly it is important to make a sharp distinction between the interpretation of a language and the semantic rules of that language. The interpretation does not belong to the language itself, but establishes the relation between the primitive terms of the language and what they denote. The semantic rules of the branchings, on the other hand, do form part of the language definition. They are mathematical rules that indicate, by operations on sets or on separate elements, how the value of a construction follows from the values of the subconstructions and ultimately from the values of the primitive terms.

A semantic rule that belongs to a particular branching operates in fact as a function that maps the values of the subconstructions under the selectors of the branching on to the value of the total construction. The way in which the types in the grammar and in the semantics of the construction language function guarantees that the semantic rules will in fact be applicable to all possible values of the subconstructions (for all possible interpretations of the language). The relation mentioned above thus ensures that the total construction has a well-defined meaning.

As an example, let us look once again at fig. 4. The language constant COMPUTERS in this construction has the type \(\{\text{machine}\}\) and therefore receives as its value for each interpretation a subset of the value domain of machine. The language constant DUTCH has the type \((\text{machine} \rightarrow \text{truthvalue})\) and its value is thus a function that is applicable, again for each interpretation, to all elements of the value domain of machine. It follows from the two foregoing sentences, to put it rather informally, that DUTCH is applicable to all elements of the set COMPUTERS, for every interpretation. This conclusion implies that the selection construction with DUTCH and COMPUTERS as subconstructions will therefore have a value for every interpretation. The type rules for the selection-branching thus guarantee that the selection-construction built up from DUTCH and COMPUTERS will have a well-defined meaning.

The language DBL is the only construction language of PHLIQA I for which an interpretation is given directly, that is to say by the data base. For example, the data base assigns to the language constant COUNTRIESDBL the set of records of the record type "COUNTRY".

The languages EFL and WML can only be interpreted in an indirect manner, that is to say by means of the translation operations of EFL via WML to DBL. The translation rules establish the relation between the values of the primitive terms in EFL and WML, as the case may be, and the values of constructions in DBL. We shall return to this point in the next section, where we deal with the translation operations.

There is one respect in which EFL differs from the other artificial languages: the primitive terms of EFL can have more than one value. An interpretation of EFL assigns to each primitive term a set of values, and not a unique value. The reason for this is that EFL terms correspond to English words, and a word in natural English can have more than one meaning. The
term COMPUTERSEFL, which corresponds to the English word "computers", is an obvious example. The PHLIQA 1 system assigns two values to COMPUTERSEFL, the one in the meaning of complete computer configurations (including peripherals) and the other in the meaning of central processing units (= CPUs).

In the next section we shall deal with the principal aspects of the various translation operations in PHLIQA 1. The consequences of the multiple meaning of primitive terms of EFL will be discussed there in more detail.

The translation operations

From English into EFL

The operation of translating from natural English into EFL is performed by a "parser" program. This program parses the question, that is to say introduces a syntactic analysis of the question, and with its aid synthesizes the corresponding EFL construction, the first representation of the meaning of the question.

Table I. An example of information from the lexicon used by the "parser" program in the PHLIQA 1 system in translation from English into EFL. The system needs this information to parse the question "Does each computer have a CPU?". The right-hand column contains the EFL language constants that correspond to the referential aspects of certain English words (nouns and verbs in this case). The syntactic categories for each of the English words and a few other syntactic features are shown under the heading "syntactic information".

<table>
<thead>
<tr>
<th>English</th>
<th>Syntactic information</th>
<th>EFL primitive term</th>
</tr>
</thead>
<tbody>
<tr>
<td>does</td>
<td>aux[*]</td>
<td>(&quot;Stem&quot;: do)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(&quot;Verb form&quot;: sing 3)</td>
</tr>
<tr>
<td>each</td>
<td>det[*]</td>
<td>(&quot;Quant&quot;: universal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(&quot;Exp&quot;: number&quot;: singular)</td>
</tr>
<tr>
<td>computer</td>
<td>noun</td>
<td>COMPUTERSEFL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(&quot;Number&quot;: singular)</td>
</tr>
<tr>
<td>have</td>
<td>verb</td>
<td>HAVESFL</td>
</tr>
<tr>
<td>a</td>
<td>det</td>
<td>(&quot;Quant&quot;: existential)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(&quot;Exp&quot;: number&quot;: singular)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(&quot;Number&quot;: singular)</td>
</tr>
<tr>
<td>CPU</td>
<td>noun</td>
<td>CPUSEFL</td>
</tr>
</tbody>
</table>

[+] aux = auxiliary, det = determiner, Quant = quantification, Exp = expected.

The "parser" program is based on an English grammar consisting of a lexicon and a set of reduction rules. With the aid of the lexicon a syntactic category and a set of syntactic features are assigned to each word. In the case of referential words, such as nouns and verbs, the lexicon also gives the corresponding EFL term. The reduction rules describe how a string of successive parts of a sentence can be reduced to one compound part when the syntactic categories and the syntactic features of these sentence elements satisfy certain conditions. Associated with every reduction rule is a composition rule, which can compose the EFL construction corresponding to the new sentence element from the EFL constructions that belong to the constituent sentence elements.

The reduction process begins at the original string of words with their syntactic categories, syntactic features and EFL terms; the process attempts to reduce this string of words to a single sentence element of the syntactic category 'sentence'. The result of this reduction process can be represented in the form of a tree diagram, displaying the syntactic structure of the sentence (compare the example given below in fig. 6a and 6b). With each completed reduction step the appropriate composition rule forms the EFL construction that represents the EFL translation of the sentence element produced by the reduction. Once the string has been reduced to 'sentence' the result of the composition rules carried out is the EFL translation of the complete sentence. A string of words that cannot be reduced to 'sentence' is treated by the system as ungrammatical.

A sentence is said to be syntactically ambiguous if different syntactical structures can be assigned to it. An example of such a sentence is "What companies have a computer with a CPU that costs more than 100 000 dollars?" Here the clause "that costs ..." could refer to "a CPU" or to "a computer with a CPU".

In general, and in this example as well, different syntactic structures lead to different EFL constructions. The "parser" program has the task of finding all possible syntactic structures of the question and of making the corresponding representations of their meanings.

We shall now illustrate the translation operation English \( \rightarrow \) EFL by a highly simplified example. The simplification relates mainly to the grammar used here. The English sentence to be translated reads: "Does each computer have a CPU?".

We shall not go into the algorithmic aspect of the matter; we will, however, use the example to show how the "parser" program applies the rules.

Table I shows the part of the lexicon relevant to this question. We see here that the word "does" has the syntactic category 'auxiliary'. Also, this word has two syntactic features. These have the form of a pair consisting of an attribute followed by a value:

"Stem" : do
"Verb form" : sing 3,

which indicates that the stem "do" belongs to "does" and that the verb form is the third person singular. "Does" is a function word, which has no separate EFL terms corresponding to it.
The reduction rules we use in this example are:

1. \( \text{determiner} + \text{noun} \rightarrow \text{noun phrase} \) (R1)
2. \( \text{verb} + \text{noun phrase} \rightarrow \text{verb phrase} \) (R2)
3. \( \text{auxiliary} + \text{noun phrase} + \text{verb phrase} \rightarrow \text{sentence} \) (R3)

where a 'determiner' is an article, a demonstrative pronoun, etc.

A reduction rule indicates that if in a sentence we have a succession of elements with syntactic categories like those to the left of the arrow (e.g. 'determiner' and 'noun' in rule R1), these successive elements can be reduced to a single sentence element with the syntactic category to the right of the arrow ('noun phrase' in R1). To each reduction rule certain conditions are attached which the syntactic features of the sentence elements must satisfy, and also an instruction for determining the syntactic features of the newly formed sentence element. Not until the conditions for the syntactic features have been fulfilled can the reduction be carried out.

When parsing begins, the sentence elements are the individual words. In our example the syntactic categories of these are:

auxiliary, determiner, noun, verb, determiner, noun. (1)

In this sequence 'determiner' occurs twice followed by 'noun'. Rule R1 is applicable to this, provided the conditions for the syntactic features have been satisfied. In this case the condition is that the value of the attribute “Exp(ected) number” of the determiner shall be equal to the value of the attribute “Number” of the noun. In the example the value of both these attributes is singular, and therefore the condition is fulfilled. If the sentence to be parsed contained 'each computers' instead of 'each computer', the reduction rule R1 would not be applicable and the sentence would finally be treated as ungrammatical.
The reduction rule R1 can be applied to (1) twice. The successive sentence elements determiner + noun are joined to form a single sentence element of the category 'noun phrase'. Both noun phrases are given the syntactic feature "Number": singular, originating from their nouns; the first also receives the feature "Quant": universal, and the second the feature "Quant": existential, originating from their determiners.

In this way (1) has been reduced to:

auxiliary, noun phrase, verb, noun phrase.

(2)

Here we see the sequence verb + noun phrase, to which the reduction rule R2 is applicable. In what follows we shall take no further account of the syntactic features.

Rule R2 now reduces the string of sentence elements (2) to:

auxiliary, noun phrase, verb phrase.

(3)

Finally, application of rule R3 leads to:

sentence.

(4)

The sentence "Does each computer have a CPU?" is thus treated by this grammar as grammatically correct. The reduction process has assigned to the sentence a form of a tree diagram (fig. 6a) that can be represented in the form of a tree diagram (fig. 6b). The branchings of the tree correspond to the reduction rules applied.

As noted, each reduction rule has a composition rule associated with it, which composes the EFL construction corresponding to the newly formed sentence element. This composition begins with the EFL terms associated with the referential words. In our example these are the language constants COMPUTERS_{EFL} and CPU_{EFL}, which both have the type of a set, and the language constant HAVE_{EFL}, which has the type of a two-place predicate. The word 'two-place' indicates that the predicate in this case is applicable to a pair of arguments (which corresponds to the fact that the verb "to have" has a subject and a direct object).

Let us now look first at the part of the sentence "... have a CPU". The reduction rules R1 and R2 assign to this the syntactic category 'verb phrase'. Application of the composition rules leads to the EFL construction given in fig. 7. The primitive term \( \Omega \) contained in this relates to the subject of "... have a CPU", which has not yet been named.

The reduction rule R1 has led here to the noun phrase "a CPU" (fig. 6). The primitive term corresponding to the English word CPU is CPU_{EFL} (Table I). The composition rule associated with R1 merely passes on this primitive term, so that CPU_{EFL} is also the EFL construction that corresponds to the newly formed noun phrase.

In the next step, R2 leads to a verb phrase. The composition rule associated with R2 constructs an existential-quantification-branching and places CPU_{EFL} as a subconstruction under the selector "for some" of this branching. The existential-quantification-branching is chosen because the syntactic feature "Quant": existential belongs to the noun phrase. This feature of the noun phrase of the sentence comes from the determiner "a" which occurs in it (see Table I).

Under the selector "holds" of the existential-quantification-branching a function is constructed by means of an abstraction-branching, with a variable (\( x \)) and an application-branching. In the application-branching the two-place predicate HAVE_{EFL} is applied to the argument pair \( \Omega \) and \( x \). The argument \( x \) corresponds, via the abstraction-branching and the existential-quantification-branching, to CPU_{EFL} (which itself corresponds to the direct object "a CPU"). The argument \( \Omega \) stands for the as yet unspecified variable that corresponds to the subject. The construction shown in fig. 7a may be paraphrased as follows: 'The set of CPUs contains at least one (in the figure: "existential quantification") element \( x \) for which it can be said ("abstraction") that the expression "\( \Omega \) has \( x \)" is true of an unknown element \( \Omega \). The expression "\( \Omega \) has \( x \)" is shown in more detail in fig. 7b as 'the result of the application of the two-place predicate HAVE_{EFL} to the pair ("pair") of elements (\( \Omega, x \)).'
Finally, as soon as the reduction rule R3 becomes operative, the remaining parts of the sentence — the auxiliary verb "Does" and the noun phrase "each computer" — also become involved in the process. The associated composition rule, for reasons similar to those applied in the case copusEFL, places the subconstruction computerEFL under a universal-quantification-branching (fig. 8). The primitive term θ from fig. 7, as yet unknown, can now be filled in (in the figure it is the variable y). The construction in fig. 8 is the expression in EFL of the whole of the original question. We can express this construction in words as follows: 'For each computer y it holds that there is at least one CPU x such that "y has x" is true'.

The EFL construction in fig. 8 has the type truth-value. Our example thus obeys the general rule that Yes/No questions lead to EFL constructions of this type. WH questions ("what", "which", "who", etc.) lead to EFL constructions that have the type string. This is because WH questions are interpreted as questions about a description of objects, not as questions about the objects themselves. What that description turns out to be depends on the kind of object asked about and on the information present. If the question is about companies, the description consists of a list of names and addresses (a string of symbols); if the question is about CPUs, the description is a list of models. The system decides on this during one of the translation operations following EFL.

For the remainder of this article it is important to bear in mind that a complete EFL construction — a construction that corresponds to the complete question — always has a formal type: truthvalue, string or integer (the type integer in the case of a WH question about a quantity).

From EFL into WML and from WML into DBL

The translation operations from EFL, via intermediate languages, into DBL need only operate on the referential language constants and on the variables. This is connected with the fact that all the artificial languages in the PHLIQA I system, as mentioned earlier, have the same branchings and also the same formal language constants. In the translation operation English→EFL the system has already taken all the decisions concerning the logical aspects of meaning. The other translation operations are concerned with the referential aspects of meaning, in particular with the question of the relations that the EFL language constants, such as computerEFL, copusEFL and haveEFL in fig. 8, have with the language constants of WML and DBL.

Every operation involved in the translation from a source language into a target language — the first source language is EFL, the final target language is DBL — is carried out by a referential convertor [5]. The number of referential convertors in the system thus depends directly on the number of intermediate languages introduced. The application of such a convertor to a construction τ of a source language implies that the language constants and the variables of τ are replaced by constructions and variables of the target language, respectively.

A referential convertor operates with two sets of transformations: constant transformations, which transform the constants of the source language into constructions of the target language and variable transformations, which transform the variables of the source language into variables of the target language. Both sets of transformations have to fulfill certain type requirements. These requirements follow from a third set of transformations, called type transformations.

The type transformations show how the types of the appropriate source language are related to the types of the target language. They have the form

\[ \alpha_i = \phi_j \]

where \( \alpha_i \) is an atomic type of the source language and \( \phi_j \) is an atomic or compound type of the target language.

The constant transformations have the form

\[ C_f = e_j \]

where \( C_f \) is a language constant of the source language and \( e_j \) is a construction of the target language. A constant transformation works 'in accordance with' the type transformation. This means that the type of the construction \( e_j \) 'is contained in' the transformed type of the associated language constant \( C_f \). (As stated, the relation 'is contained in' between types will not be dealt with in this article; it is sufficient here to know that a type \( \alpha \) is certainly 'contained in' a type \( \beta \) if \( \alpha = \beta \).) The transformed type of \( C_f \) is the type that occurs when the atomic types \( \alpha_i \) found in the type of \( C_f \) are replaced by the types \( \phi_j \) that correspond to them in accordance with the type transformations.

The variable transformations have the form

\[ x_k = y_k \]

where \( x_k \) is a variable of the source language and \( y_k \) is a variable of the target language. The type of this

---

variable of the target language is the transformed (as described above) type of the associated variable $x_k$.

It can be shown that the application of a referential convertor to a syntactically correct construction of a source language always leads to a syntactically correct construction of the target language. (Syntactically correct’ means that the type of the construction is unequal to empty.) The system thus does not produce any surprises in this respect. The proof, which we shall not give here, is based on the type requirements for the constant transformations and the variable transformations and also on certain conditions that the type function $T$, operating between construction language and type language, fulfils for the various branchings.

**The semantic aspects of the referential convertors**

In addition to the syntactic aspects discussed, the translation operations also have semantic aspects, of course. In our system we distinguish between two classes of referential convertors with regard to semantics. The first consists of the synonymic referential convertors, or *synonymy convertors*, where the value of the constructions is preserved during the translation operation, and the second consists of the identifying referential convertors, or *identification convertors*, where there is no equality but a one-to-one relation between the values.

In the case of the synonymy convertors the constant transformations $C_I \Rightarrow e_I$ mentioned above are value-preserving. More precisely, a constant transformation is only correct for those pairs of interpretations of the source language and target language where $C_I$ and $e_I$ have the same value.

In designing a system like PHLIQA 1 it is necessary to determine whether each constant $C_I$ of the source language has the same meaning as the corresponding construction $e_I$ of the target language. This is done by establishing whether $C_I$ and $e_I$ do in fact have the same value for every possible state of the subject domain. In PHLIQA 1 this meant that the meaning of English words had to be brought into relation with the meaning of the primitive terms contained in the database, since EFL is the first source language and DBL the final target language. The correctness of the established relations is easy to check in this case for the designers and in principle also for others — because these relations are established by simple rules that transform only the *primitive terms* of a source language and not the larger constructions.

It can be shown that every complete construction of the source language (i.e. the representation of the meaning of a complete question) is transformed by the convertor into a target-language construction that has the same value — this only applies of course to interpretations of the source language and the target language for which the constant transformations are value-preserving. The fact that the synonymy convertors, and also the other convertors not yet dealt with, possess this feature gives the system its reliability. The term ‘reliability’ means here that correct constructions are correctly translated as soon as primitive terms are correctly translated.

There are certain kinds of semantic relationships between the artificial languages of PHLIQA 1 for which the synonymy convertors are not sufficient. For this reason *identification convertors* were introduced. These are required for situations where it is possible to ‘speak’ about objects of a particular type in the source language, whereas this is not possible in the target language. This does not imply that the constructions of the source language are untranslatable in such situations: the questioner is not interested in the objects themselves as much as in their *properties* that the system is capable of presenting in language form. In many cases such *properties* can be ‘spoken’ about in the target language, thus establishing a semantic relationship with the source language.

The operation and the usefulness of the identification convertors for semantic relationships of this kind will be illustrated by the following example. WML contains the atomic type `cpu`, which relates to the type of individual, physical CPUs; on the other hand DBL does not contain the type `cpu`, nor a type synonymous
with it. Language constants in WML that relate to CPUs, whose type thus contains the atomic type cpu, cannot therefore be translated into DBL with the synonymy convertors. This is because no DBL constructions can be made with a value equal to that of the WML language constants referred to. Nevertheless, questions about CPUs in general can be answered, because the information about a CPU is to be found in the record of the computer configuration of which this CPU is a part, or in the record of the model of the CPU (of record type “CPU MODEL”, see fig. 6, part I) that is linked with the record of the computer configuration (with the record type “CONFIGURATION”). Identification transformations can now translate all language constants relating to CPUs into constructions about configurations, with the addition of a mark that distinguishes them from constructions that are ‘really’ about configurations. In this way CPUs are thus indirectly identified, by means of the computer configuration to which they belong.

In these identification transformations the value of the translated primitive terms is not preserved. There is, however, an ‘isomorphism’ between the values of these terms and the values of their translations in the target language.

Isomorphism — a mathematical concept — refers to similarity in form, a structural identity, of two sets; it amounts to stating that the relations between the elements of the one set agree with the relations between the corresponding elements of the other set, although the elements themselves do not have to be equal to each other.

On the subject of isomorphism two further remarks should be made. Firstly, it can be shown that if isomorphism exists between the values of the primitive terms and their translations it will also exist between the values of complete constructions and their translations. Secondly, for the formal types, e.g. truthvalue, integer, string, isomorphism amounts to identity. As we have seen in the section discussing the translation operation English→EFL, the complete constructions in EFL always receive a formal type. In the subsequent translation operations the complete constructions preserve their formal type. From the foregoing it follows that, like a synonymy convertor, an identification convertor is value-preserving for complete constructions.

The instantiation convertor

In the discussion on semantics at the end of the section dealing with the artificial languages we pointed out that primitive terms of EFL are sometimes ambiguous (can have more than one value). In translation operations from EFL into the next, unambiguous construction language this has the consequence that each primitive term is translated into a set of expressions in that language, called instantiations of the EFL term. For dealing with such ambiguity the system possesses a special convertor, called the instantiation convertor. This generates all possible combinations of instantiations of the different primitive terms in the original EFL construction, and then finds which combinations are well-formed in regard to type. Only these well-formed instantiations are passed by the instantiation convertor to the next convertor.

Instead of a single expression the instantiation convertor thus yields a set of expressions, and in this respect it differs fundamentally from the convertors already discussed. The set can be empty in which case the question is meaningless (see fig. 4, part I).

During the instantiation the types of the EFL terms are limited to those of WML, to which the actual — limited — subject domain corresponds. For example, CPUs have a price in that subject domain whereas cities do not. In EFL, however, both “expensive CPUs” and “expensive cities” are correct. During the instantiation the system concludes that the combination “expensive CPUs” is permissible but not “expensive cities”.

The formal convertors

The convertors so far described constitute in principle a powerful mechanism for all translation operations from EFL into DBL, ensuring both syntactic and semantic correctness. This does not necessarily mean, however, that these convertors process a submitted question in the simplest way. Owing to their ‘locality’ (they operate on only one primitive term at a time) the referential convertors often generate unnecessarily awkward constructions. To simplify such expressions the system uses formal convertors. These consist of a set of formal transformations that are not local but operate on larger parts of an expression — within the same language. A formal transformation reduces a construction to a simpler one that is logically equivalent to it, that is to say, it has the same value as the original construction for all interpretations of the language.

The application of a formal transformation can usefully be compared with the simplification of an algebraic expression, e.g. $y = \sqrt{x} \Rightarrow x$, since the constructions with which the PHLIQA 1 system works have much in common with algebraic expressions. The formal transformations may occasionally have an advantageous side-effect in the system, since they may transform constructions with an untranslatable primitive term into a logically equivalent construction (in the same language) in which this term does not occur. This has further increased the translation capacities of PHLIQA 1, since it makes translation possible in cases where the identification transformations and the synonymy transformations alone are unable to produce a translation.
The evaluator

After a question has been translated into DBL by the convertors discussed above, the resultant representation of its meaning in DBL is passed to the evaluator (see fig. 3, part I). This computes the value of the construction and produces it as the answer to the question. The value of a construction has so far been defined in mathematical terms: an interpretation assigns values to the language constants, and the semantic rules of the branchings show how the value of the construction follows from the values of the language constants contained in it.

To calculate the value of a construction in practice the evaluation program must make use of actual representations of values. In PHLIQA 1 these representations are again expressions of a construction language, called the value language. The primitive terms of this value language are unique names of records and formal constants (the truth values true and false, numbers and strings of symbols such as P-H-I-L-I-P-S). The value language contains only a few branchings, one of which permits a set to be represented by enumerating its elements.

Assigning values to the language constants of a construction, which is what the evaluator does with the aid of the data base, amounts in the terminology of the value language to applying a synonymy convertor. A transformation from this synonymy convertor replaces a language constant (of DBL) by an expression in the value language that represents its value unambiguously. Thus, for example, the language constant CONFIGURATIONSDBL is replaced by a list of names of records with the record type “CONFIGURATION”.

In the evaluator the semantic rules of the branchings take the form of formal transformations that convert a branching whose subconstructions are expressions in
the value language into a new expression in the value language. This ultimately results in an expression in the value language that represents the value of the complete DBL construction.

The evaluator can thus be regarded as a synonym convertor followed by a formal convertor. In fact there

An example of the translation operations $EFL \rightarrow WML$ and $WML \rightarrow DBL$

Having described the formal means available for the translation operations, we shall conclude by giving a rather simplified example to show how the system can use these mechanisms in the translation process.

Fig. 10. Translation of the question “Does each computer in Eindhoven have a CPU made by Philips?” in the World-Model Language (WML). The complete construction is shown above the dashed line, and the subconstructions for which the triangles are used as an abbreviated notation are shown below. This branching diagram, obtained from the construction in fig. 9 by the translation operation $EFL \rightarrow WML$, is also a representation of the meaning, but now in WML, of the original question. An asterisk in front of a primitive term indicates that it has to be translated into DBL by means of an identification transformation. Synonymy transformations operate on the other primitive terms.

is a third component, called the answer formulator, which presents the expression obtained in the value language in a readable form to the questioner. The answer formulator presents the value TRUE as “YES” in an external format and the value FALSE as “NO”, and gives lists of items such as names and addresses in clearly tabulated form.

The example is the question “Does each computer in Eindhoven have a CPU made by Philips?”.

Fig. 9 shows the EFL construction for this question. Apart from the additions “in Eindhoven” and “made by Philips”, which are displayed in an abbreviated notation as triangles, this expression has the same structure as that in fig. 8, the example of the translation operation
English → EFL. As in our treatment of the “parser” program, the procedure shown is simplified.

The principal mechanisms for translating in this example are:
— The instantiation convertor. The primitive terms COMPUTER\textsubscript{EFL} and HAVE\textsubscript{EFL} are ambiguous, since the word “computers” can refer either to CPUs or to complete computer configurations. The meaning of the verb “to have” can include “to possess” and “to have as a part”. Only one of the possible combinations of the instantiations in our example leads to a construction of a permissible type, and this is the combination where COMPUTER\textsubscript{EFL} is instantiated to the primitive term CONFIGURATIONS\textsubscript{WML}, and where HAVE\textsubscript{EFL} is instantiated to the primitive term HAVE\textsubscript{AS-PART} (a term of an intermediate language between EFL and WML).
— A synonymy convertor. The language constant HAVE\textsubscript{AS-PART}, which has the type of a two-place predicate, does not correspond to one WML constant, but is translated into a subconstruction, called \textit{CWML} here, in which this predicate is written out for various possible combinations of argument types. In \textit{CWML} we find, for example, the language constant COUNTRY\textsubscript{OFWML} with the type \textit{country}, a function that for each city gives the country where it is situated, and we also find the language constant CONF\textsubscript{OFWML} with the type \textit{configuration}, a function that for each CPU gives the configuration to which it belongs.
— A formal convertor. This reduces, for instance, the construction \textit{CWML} written in full to the part that is relevant in this context, i.e. the function CONF\textsubscript{OFWML}.

The operation of these three kinds of convertors leads to the WML construction shown in fig. 10.

\begin{center}
\textbf{Fig. 11.} Translation of the question “Does each computer in Eindhoven have a CPU made by Philips?” in the Data-Base Language (DBL). The complete construction is shown above the dashed line, and the subconstructions for which the triangles are used as an abbreviated notation are shown below. This branching diagram, obtained from the construction in fig. 10 by the translation operation WML→DBL, is the third representation of the meaning of the original question. The system can assign to this translation the value \textit{true} or \textit{false}, on the basis of the data contained in the data base. The answer “\textit{yes}” or “\textit{no}” then appears on the screen.
\end{center}
The application of the translation operation \( WML \rightarrow DBL \) to the construction in fig. 10 leads to the construction given in fig. 11. The most striking difference compared with fig. 10 is the absence of the existential quantification for CPUs. This is achieved by successively using an identification convertor, which reformulates primitive terms relating to CPUs in terms of configurations, and a formal convertor, which simplifies the expression. Fig. 11 incidentally shows only one of the many DBL constructions that are possible for the given question and are logically equivalent. Just which construction is generated depends on the formal transformations available.

In the DBL construction shown in fig. 11 the primitive term CONFIGURATIONS\(_{DBL}\) stands for the set of records with the record type "CONFIGURATION" (the computer configurations). The primitive term SITE-OF\(_{DBL}\) is a function that, when applied ("application" in the figure) to a given record of the record type "CONFIGURATION", gives a record of the record type "SITE" (of the establishments where the configuration is used). The function CITY NAME-OF\(_{DBL}\) gives for a given record of the record type "SITE" the name of the city where the establishment is situated. The evaluator carries out these function applications and investigates whether the city thus found for each configuration (the variable \( x \) in the figure) is "equal" to the string E-I-N-D-H-O-V-E-N, which is found under the equal-branching as a primitive term (that has the type string). (A primitive term of the type string is a formal language constant, and its value is therefore the same for each interpretation. The value is the string itself.)

Only records of the record type "CONFIGURATION" that fulfill this requirement are included in the value of the selection-branching (the configurations in Eindhoven). The part of the construction on the right in fig. 11 is treated similarly by the system.

In this way the evaluator computes the value of the construction. Depending on the data, the value is TRUE or FALSE. The answer formulator displays this result on the screen as "YES" or "NO".

Summary. The experimental computer program PHLIQA 1 enables a computer of medium capacity (e.g. the Philips P1400) to answer certain questions put to it in natural English. The answers are taken from a data base, organized in accordance with the CODASYL standards, which contains fictitious data about European computer installations and their users. The question, typed in by means of a keyboard with a visual display, undergoes a series of translation operations. The first, English \( \rightarrow \) EFL (English-oriented Formal Language), determines the logical relations between the various parts of the sentence. The second, EFL \( \rightarrow \) WML (World-Model Language), reformulates the question in terms of the subject domain and checks whether the question makes sense in that domain. The third translation operation, WML \( \rightarrow \) DBL (Data-Base Language), provides the link between the contents of the question and the chosen data base. The computer calculates the value of the expression in DBL and produces an answer on the display screen ("YES"; "NO"; a number; a list of names and addresses). If ambiguity exists the computer can systematically trace different meanings; if the questions are unanswerable the computer indicates at what language level the answering procedure breaks down and why. The organization of the data base and the syntax of the artificial languages used are described. All the artificial languages have the same grammar; they differ, however, in their primitive terms. The language expressions are referred to as constructions; the extent to which they are "well-formed" is determined by means of type rules. The syntactic structure of the constructions corresponds to their semantic structure. For translating into EFL the computer uses a lexicon, reduction rules and semantic composition rules. To translate from EFL into WML and DBL the computer uses referential and formal transformations. The various parsing and translating mechanisms are explained with a number of examples. An example illustrates how the whole sequence of translation operations provides, in the form of a three-stage analysis followed by a value calculation, an answer to a particular question. The article is in two parts. Part I gives a general introduction to the organization and external behaviour of the system, and part II deals more with technical particulars, the syntax and the semantics of the artificial languages and the mechanisms of the translation operations.