[From: *Overcoming the language barrier*, 3-6 May 1977, vol.1 (München: Verlag Dokumentation, 1977)]

'PROTRAN' - A GENERALIZED TRANSLATION TOOL FOR NATURAL AND ALGORITHMIC LANGUAGES

I. D. K. KELLY

Calikarn Ltd., London

Abstract

Considerations of the problems inherent in mechanically translating between natural languages have given rise to a software tool, PROTRAN, which runs on IBM 360/370, and has been used successfully to translate between a wide range of computer languages. PROTRAN is a generalized double-string syntax analyzer-synthesizer to which are given, in a language-description language, two descriptions: that of the language to be translated from, and that of the language to be translated into. These descriptions are called "correlators".

The system includes (i) the ability to parse input text according to such descriptions, (ii) the ability to transform conditionally parse trees into other, non-isomorphic, trees, (iii) the ability to manipulate text: in particular, to generate text from transformed parse-trees, (iv) the ability to both update and look-up tables (dictionaries), and (v) the ability to re-analyze partially parsed strings.

The paper describes the general operation of PROTRAN, and illustrates specific techniques both by reference to computer-language translation, and to an Esperanto-English project.

Possible future developments are discussed.

1. Background

1.1 The problem

In the mid 1960s a group of people in a particular commercial environment had need of a general tool to help them transport computer programs between a range of machines which had differing levels of high-level language support. No control could be exercised over the programs - they were, in fact, customer's programs - and they came in a variety of high-level algorithmic languages peculiar to certain machines. It was necessary to transform these programs to run on different machines, and hence to transform the dialects in which they were written, and sometimes even to translate them into quite different languages.

Two members of this group, Dr. Raphael Mankin and the present author, already had strong interest both in compiler construction and also in non-computational linguistics, and we began to view the problem of translating from, say, the IBM/360 dialect of Cobol into the ICL 1900 dialect as merely a weakened form of the problem of translating natural languages: English into French, for example. The syntax and idiom of the two languages might differ markedly, but it is the semantic mapping that is required.

1.2 A Solution

A preliminary survey of the techniques then available showed considerable weaknesses in all of them, and we decided to construct a tool with the best features of the work of Alpiar *[1]*, Tosh *[2],* and the Compiler-Compiler of Brooker and Morris *[3]* without their limitations, and accordingly we set about building a program to run on the IBM/360, which we called PROTRAN.

It was essential to the operation of PROTRAN that we should be able to describe any practical language transformation for high-level computer languages in an easy-to-understand fashion. We also wished to be relieved of the chore of coding

afresh for each translation the syntax analysis routines, table handling routines, and the many other standard functions on which such a transformation relies.

PROTRAN is conceptually a very simple program which firstly reads the description of the syntax of a language, then the description of a transformation to be applied to that syntax. These two descriptions are written in a language-description language: we call them 'correlators'. When these correlators have been read, PROTRAN re-configures itself to recognize and analyze texts in the first of these languages, and transform them into the second.

2. Description

2.1 Action of PROTRAN

PROTRAN executes in three main repeated sections. The first section reads the text to be translated into an input buffer. The second section analyzes the syntax of the contents of the buffer according to the rules specified in the input correlator, using a top-down parser. When this buffer has been successfully analyzed the syntax tree thus formed is passed over to the third phase which scans that tree from the root, filling an output buffer. The output correlator drives this third phase, which amongst other things determines when and if to write the output buffer. We call this phase the SOUTAX, because it is the converse of SYNTAX.

One advantage of this technique is that no output action is taken until the syntax of the input has been fully recognized, so that we do not have the problem of undoing actions embedded into a parse, when that parse has to back-track (a frequent failing with syntax-driven compilers). In general we do not want to write one-track grammars (or bottom-up parsers), even when they exist, as these are usually very difficult to alter, and do not represent intuitive understanding. For example, the common stem "the quick" in the phrases "the quick brown fox" and "the quick and the dead" serves two totally different functions: a grammar which never had to back-track over it

would be unwieldy and expressed in terms of categories quite different from the English speaker's traditional 'noun' and 'adjective'. A second advantage of PROTRAN's technique is that there is no enforced isomorphism between the syntax analysis tree of the input, when it has finally been determined, and that of the output. Primitives exist within PROTRAN to transform nodes of such analysis trees by re-ordering or omitting branches, moving nodes to other levels, and changing their connectivity. Moreover, both the syntax scan and the 'soutax' scan are conditional, and the rules encoded in the correlators which express them may depend on the contents of tables, counters and switches which themselves are manipulable in an algorithmic language.

2.2 Notation

Good notations for expressing the syntax of a language are hard to come by; though, interestingly enough, the earliest example of formalism was also the most ambitious: the complete syntax of the Sanskrit language - a natural language - compiled by Paṇini in the third or fourth century B.C., using the Devanagari alphabet:

अधारो ऽधिकरणम्
अधिशीङ्स्थासां कर्म

No real advance was made on this notation until the BNF of 1958, rather more suited to the Roman alphabet:

<a> : : = <b> | c <d> e | <f> <a>

We began by expressing syntax in an equivalent notation to this, but again altered to suit the alphabet - this time that of the punched card computer input:

```
LOAD A
      B, C'C'  D C'E',
      F A  ;
```

which represents the same fragment of syntax as the BNF above.

It is found in practice to be advisable to delimit the terminal strings, but leave the non-terminals (the category class names) undelimited, as these are referred to much more often. The main features of the syntax notation can be gleaned from the fragment given: 'LOAD' to mark the left-hand-side of a definition; a terminating semicolon; alternatives separated by a comma rather than a vertical bar; and character strings for terminal symbols of the form: C'text'.

2.3 Notation - Soutax

Consider the following grammar:

BNF	PROTRAN
<sigma> : : = <alpha><beta><gamma>	LOAD SIGMA ALPHA BETA GAMMA ;
<alpha> : : = AA | A	LOAD ALPHA C'AA' , C'A' ;
<beta> : : = B <beta> | B	LOAD BETA C'B' BETA , C'B' ;
<gamma> : : = C	LOAD GAMMA C'C' ;

which allows sentences of the form $A^{2,1}B^{n}C$, taking SIGMA to be the root of the language; and suppose that we require to transform texts in this small language in the following way:

input	output
ABC	CAWAZBZ1ONE1
ABBC	CAWAYBYZBZ2ONE2
AABBBC	CAAWAAYBYYBYZBZ3TWO3
...	...

that is, move the C to the front; duplicate the A's separated by a W; surround each B with a Y, except the last which is to be encased in Z's; and finally, produce a count in words of the number of A's, surrounded by a numeric count of the number of B's. The soutax for this is:

```
LOAD   SAMEKH
       &GAMMA &ALPHA 'W' &ALPHA
       PUSHCLEAR POPGLOB COUNTER
       (BETA) BETH PUSHGLOB COUNTER PUTNUM
       (ALPHA) ALEPH PUTNUM APOP ;
```

```
          LOAD BETH
                COUNT  'YBY  (BETA) BETH ,
                COUNT  'ZBZ'  ;
          LOAD COUNT
                PUSHGLOB COUNTER AUGM POPGLOB COUNTER ;
          LOAD ALEPH
                'TWO' ,  'ONE' ;
```

and it may be understood as follows:

SAMEKH is used to process the syntax rule SIGMA - it will become clear later how this correspondence is set up. The soutax rule begins with &GAMMA which means "copy from the input buffer all that text which was analyzed as an example of 'GAMMA' ". Then &ALPHA 'W' &ALPHA twice copies that part of the text analyzed as being of class ALPHA, putting the explicitly stated character string 'W' between the two copies. The structure PUSHCLEAR POPGLOB COUNTER firstly pushes onto a last-in-first-out notional accumulator stack a zero cell, and then pops that into the variable COUNTER, which would have to be declared elsewhere: we call such arithmetic variables 'globals' as they may be referred to from all trees (rules). Then the structure (BETA) BETH which we read as "point to BETA, call BETH", uses the soutax rule BETH to determine the next output, and passes it the actual analysis performed by BETA as a parameter. In fact this 'point' determines three things for the subsequent recursive call to SOUTAX (the 'BETH'):

- (i) the input buffer pointer is set to the start of the text analyzed as BETA (the original is restored on exit),
- (ii) only those non-terminals referred to in BETA may be referred to in BETH: a new 'root' is set, and
- (iii) the same number of commas is skipped in invoking rule BETH as had to be skipped in rule BETA to get the successful scan achieved.

This notion of 'point' is fundamental to PROTRAN, and at every level it is what matches the syntax to the soutax rules: it is what matched SAMEKH to SIGMA in some higher-level rules not

quoted here. To fix a clearer idea of what 'point' means look at the structure (ALPHA) ALEPH . This invokes the first alternative of ALEPH only if in the actual text being parsed the first alternative of ALPHA was the one recognized, and so on.

2.3 Other Features

This superficial introduction has not shown the full range of constructs available for manipulating syntax trees and their associated texts, but amongst the other features of PROTRAN are:

- the selection of a soutax rule or alternative as the result of dynamic computation
- conditional syntax (and soutax); that is, syntax rules which are 'tailored' dynamically as the result of past actions of the parser or synthesizer
- the stacking of output buffers, and the passing them back to be re-analyzed by SYNTAX, using a possibly different set of rules
- table handling routines of great generality
- error recovery from syntax failures which might be due either to a failure of the input text to match the language defined, or of the supplied rules to match the language intended.

3. Usage and Development of PROTRAN

3.1 Already Active

PROTRAN has been used in actual commercial projects with a high degree of success. The first pair of correlators to be completed translated from Algol to PL/I: the Bourrough's dialect of the former, and the IBM dialect of the latter. Rather more than a million Algol statements, representing a particular customer's research and development systems, were successfully converted.

Cobol-to-Cobol correlators have been coded, making strong use of the conditional syntax features mentioned above. The Cobol

recognized is extended to cover both slack use of the language and also parameter cards which can set any combination of the following dialects:

input	output
Honeywell 200	Honeywell 6000
IBM/360 ANSI	IBM/360/370 ANSI
ICL 1900	ICL 1900
IBM/360 'F'	ICL 2900
	DEC PDP/11
	DEC PDP/10

together with a lot of other options (default clearing of working-storage, layout options, use of working-storage rather than buffer areas for files, etc.). The Cobol language produced is always very strict and carefully laid out. These correlators too have been used to translate over 700,000 Cobol input records to contract. The quality of the translation is such that after translating a complicated payroll suite from ICL 1900 dialect into IBM 370 (ANSI) dialect, only one statement in every 2000 required manual alteration to achieve successful running in the new environment.

A Coral 66 compiler for the IBM 370 has been written and installed. These correlators to translate from an Algol-like block-structured language into and assembler code were written and tested to acceptance in well under a man-year.

3.2 The Future

Apart from extensions to existing correlators for computer languages that are now being worked on, there are two main avenues of development that we are following simultaneously. These are: enhancements to PROTRAN itself, and the development of natural language correlators. Two main enhancements are planned: the embedding of correlators into a higher-level language, and the use of 'zig-zag' parse techniques which are a mixture of top-down and bottom-up, to improve analysis speed. Neither of these is strictly necessary: they merely

ease the production of correlators, and speed their operation, without adding any fundamentally new powers.

The development of natural language correlators is being researched, though it must be admitted that insufficient effort has so far been spent on the Esperanto-to-English project to judge finally how long this is going to take to yield practical linguistic results (the first stage of which we define as being the acceptable translation of business texts, particularly letters). There has already been a lot of spin-off into PROTRAN itself as we tend to require facilities for the natural-language project before there is a call for them from the algorithmic language correlator pairs being developed.

Esperanto was chosen initially because of its regularity, but none-the-less it raises complex problems. We are investigating the use of a multi-dimensional vector-space to represent meaning, in order to determine the English equivalent of a single Esperanto word. The basis of this vector-space is an abridgement of Roget's Thesaurus: the *n*'th element of the vector representing the degree of relevance of Roget's *n*'th paragraph *[4],* The search for an equivalent then becomes the search for a 'close' point within this space (not necessarily using the Euclidean metric). On top of this must be superposed the knowledge of the real world which is used to understand language, and disambiguate texts. We have been strongly attracted by the Preference Semantics of Wilks *[5],* and we are investigating means of associating the dictionary/thesaurus more closely with the parsing rules themselves, as this approach seems to require: breaking down the distinction between syntax and semantics.

Acknowledgements

Particular thanks are due to Dr. Raphael Mankin for both the means of expression and the basic ideas contained herein. I must also mention the encouragement and assistance of Mr. John Cale, with much gratitude.

References

1. Alpiar, R.; *Double Syntax Oriented Processing* Comp J, 14 1 (Feb 1971), pp. 25-37.

2. Tosh, Wayne; *Syntactic Translation* Janua Linguarum, series minor 27, Mouton, the Hague, 1965.

3. Brooker, R., et al.; *The Compiler Compiler* Annual Review in Automatic Programming, Vol 3, Pergamon, 1963.

4. Roget, P. M.; *Thesaurus of English Words and Phrases* Penguin, Harmondsworth, 1953.

5. Wilks, Yorick; *An Intelligent Analyzer and Understander of English* Comm ACM, 18 5 (May 1975), pp. 264-274.

Appendix

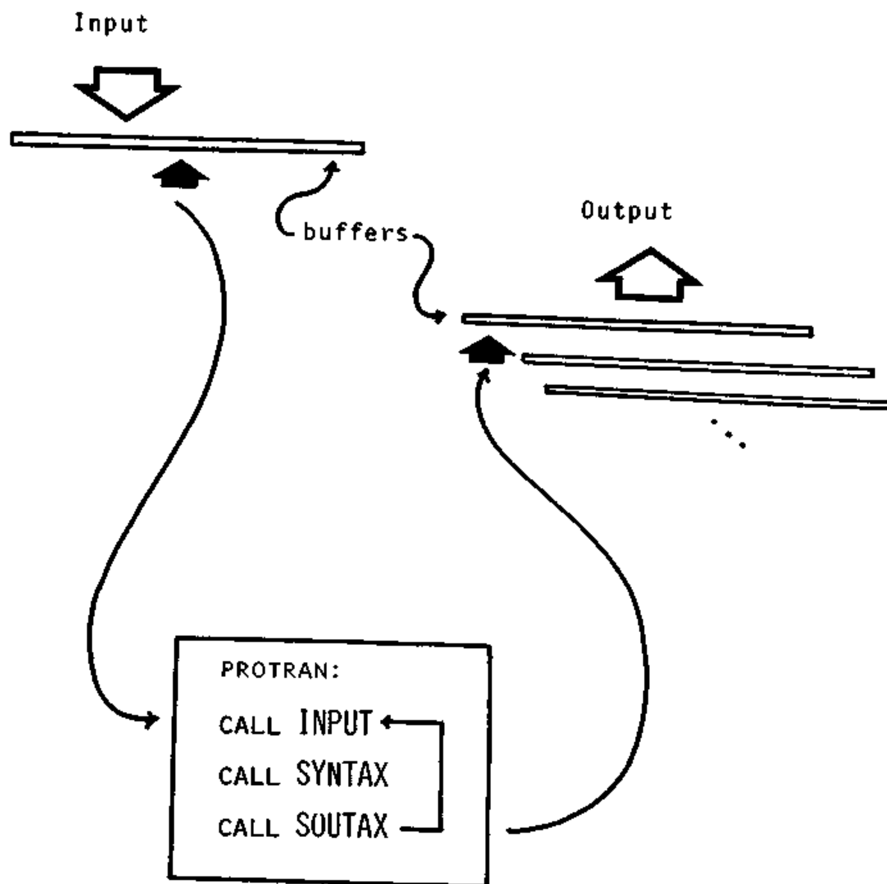
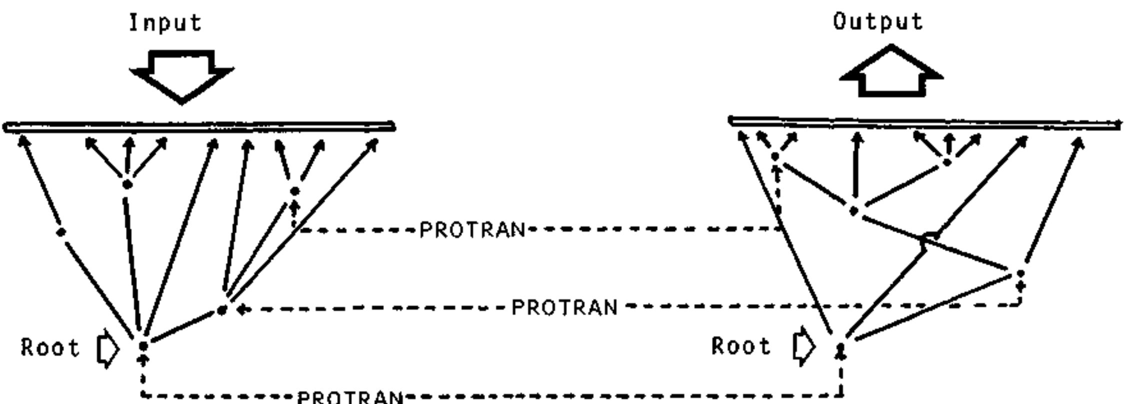


Figure 1.



PROTRAN :-

1. READS ABSTRACT SYNTAX RULES
2. PARSES INPUT ACCORDINGLY
3. TRANSFORMS THE PARSE TREE
4. PRODUCES OUTPUT

Figure 2.

Worked Example

In translating from COBOL to PL/I the following problem arises:

A Cobol qualified name consists of a series of 'words' separated by 'IN' or 'OF'.
A Cobol 'word' consists of a hyphenated sequence of identifiers.
The order of the names is from minor to major - most particular to most inclusive.

A PL/I qualified name consists of a series of 'words' separated by full stops (points).
A PL/I 'word' consists of a 'broken' sequence of identifiers.
The order of the names is from major to minor - most inclusive down to most particular.

Thus in translating a Cobol name to its corresponding PL/I name it would seem reasonable to:

(i) change all hyphens to 'break' characters (underline)
(ii) reverse the order of the 'words'
and (iii) separate them by '.' rather than by ' IN' or 'OF' .

The following sample rules to achieve this fragment of translation assume that all multiple spaces have been edited to single spaces, and that ID is a given (built-in) rule with the same format as the Algol *<identifier>*; and under these simplifying assumptions we may code the syntax:

```
LOAD    COBNAME
        COBWORD  COFIN    COBNAME ,
        COBWORD   ;

LOAD    COBWORD
        ID       C'-'     COBTAIL ,
        ID       ;
```

```
LOAD   COBTAIL
       ID          COBTAIL ,
       NN          COBTAIL ,
       C'-'        COBTAIL ,
       ID ,        NN        ;
LOAD   COFIN
       C' OF' , C' IN'    ;
```

NN is a built-in rule which recognizes any string of decimal digits. The corresponding soutax to the above would be:

```
LOAD    PLINAME
        (COBNAME) PLINAME    &'.'
        (COBWORD) PLIWORD    ,
        (COBWORD) PLIWORD    ;
LOAD   PLIWORD
       &ID      '_'         (COBTAIL) PLITAIL ,
       &ID      ;
LOAD    PLITAIL
       &ID      (COBTAIL)  PLITAIL ,
       &NN      (COBTAIL)  PLITAIL ,
       '_'      (COBTAIL)  PLITAIL ,
       &ID ,     &NN       ;
```

Then the use of the combination

```
(COBNAME) PLINAME
```

will perform the required particle of translation.