Behavior Research Methods & Instrumentation
1975, Vol. 7 (6), 557-562

Computer translation with paired grammars

T. R. G. GREEN
MRC Social and Applied Psychology Unit, Sheffield University, Sheffield S10, 2TN, England

In certain types of experiment, the subject controls an on-line computer by giving commands in a simple source language—possibly a subset of English or of a high level computer language. The commands must then be decoded before they can be obeyed. One method is to write an ad hoc program for the specific purpose. An alternative is to write a general purpose translator to decode the source language into a more primitive target language. A suitable translator is described, driven principally by "paired" context-free grammars of the source and target languages but also able to accommodate context-sensitive rules. Using the translator has several advantages. It is obviously much easier to write an ad hoc recognizer for a very primitive language than for a subset of English. Also, for small languages it is very easy to write and check grammars; minor modifications are a trivial job, and the finished product is unlikely to contain hidden bugs. An example of the method is given.

This paper describes the construction of a syntax oriented translator and its use in on-line computer controlled experiments. A syntax oriented translator is a computer program capable of accepting sentences in one phrase-structure language and translating them into another, and it has the distinguishing feature that it is general purpose: the grammars of the two languages are supplied to it as data before starting to translate. Such translators are not by any means new, but they have been little used in experimental psychology; partly because experiments to which they would be appropriate have not been performed until recently, partly because published descriptions (e.g., Ingerman, 1966; Foster, 1970; Gries, 1971) have described techniques more appropriate to computer science in terms also more appropriate to computer science.

The technique described here has properties which make it particularly suitable for applications in experimental psychology. It is simple in conception and easy to implement; it is compact enough to fit into laboratory computers; the grammars are presented to it in a form which is both familiar and easy to work with; and it is powerful enough to allow context-sensitive features to be included in the languages. These advantages are obtained by accepting compensating disadvantages. Relative to methods used in computer science, this technique is slow in operation; and it is not universal—the target language, into which it translates, must resemble the source language in certain ways. Although these would be crippling disadvantages for, say, a compiler for programming languages, they are not serious in psychological applications, and the techniques have been used with success in a number of studies.

The function of such a program in the laboratory is to set up quickly on-line experiments in which the subject controls the computer by giving commands in some language. One such situation is where the experimenter wishes to study strategies in problem solving; the computer displays the current state of of the problem environment and the subject tells it what move to make next. If the environment is at all complex, it is easiest for the subject to give instructions in a language based on English. These can then be translated into a language with much less syntactic structure—possibly a pseudomachine code. The program controlling the experiment still must to able to comprehend the target language, but it is much easier to write a program to comprehend a very simple language in an ad hoc way than one to comprehend a language with considerable syntactic structure. The translator described has been used for this purpose in the study of job-shop scheduling (Fitter, 1974). An alternative situation is one in which the experimenter wishes to compare two artificial languages. If each source language is translated into the same target language, only one ad hoc program need be constructed, a considerable saving. The translator was originally devised for comparisons of different types of programming language (Sime, Green, & Guest, 1973, 1974).

In brief, instead of controlling an experiment by a program containing an ad hoc recognizer to comprehend a fairly complex language, which is difficult to write, it can be controlled by a program containing an ad hoc recognizer for a very simple language. Messages are fed to it by the translator as it decodes the subject's inputs. As the translator can be relied upon to spot faulty inputs by the subject, the ad hoc recognizer is further simplified because it does not have to detect errors. It is, therefore, much easier both to set up a new experiment and to modify a detail. At the same time, because it is easier to check out a grammar than a complicated program, reliability is increased, and there is less chance of an unsuspected bug ruining an experimental run.

Note that the input to the translator need not be typed words. Experience suggests that it is best to avoid teleprinter input since typing skills vary rather widely. We have instead reinvented lexigraphy, and use a system in which the subject touches a contact by the side of each word (Fitter & Daly, 1975).

OVERVIEW

The technique described could be called paired-grammar translation. It is based on an ordinary context-free phrase-structure parsing system, and we have found it simplest to work with a top-down left-to-right parser, because that is very easy to program and does not unduly constrain the grammar. The notation used for the grammars is Backus-Naur form, in which every rule is a set of alternative ways in which a nonterminal symbol can be rewritten into other nonterminals or into terminal symbols. The essence of top-down parsing is as follows. Consider the first alternative of the rule defining a sentence: if it mentions, as its first symbol, another nonterminal, consider the first alternative way of defining that, and so on down to a rule whose first alternative starts with a nonterminal symbol. If that matches the first input symbol, go on, otherwise back up. A precise account is given below.

The parser does nothing more than discover the structure of sentences in the input, or "source," language (except to report when a string is not grammatical). To translate into an output, or "target" language it is necessary, if full universality is wanted, to use a technique in which the translation depends not only on the wording of the input sentence but also on its structure. That is possible but elaborate. Instead, a much simpler technique is proposed, in which the target grammar is paired with the source grammar in such a way that every nonterminal symbol in the source grammar is associated with the same nonterminal symbol in the target which, by definition, is its translation.

Informally, this is equivalent to translating, e.g., from English to French, by assuming that if an English sentence consists of a noun phrase and a verb phrase, then its translation consists of the translation of the noun phrase plus the translation of the verb phrase (possibly not in the same order, and possibly with a few extra terminal symbols thrown in). For natural language, that assumption is obviously wrong. For example, if "I go" is to become "je vais" while "you go" becomes "vous allez," we have to pass on a message about the subject of the verb in order to translate the verb correctly. In other words, the translation of "go" is context-sensitive. However, this translator is not being put forward as a method for translating English into French, but from, at very best, restricted English to a machine language, and in these conditions experience to date suggests that such problems are few. When they do arise, a device is included to allow context-sensitive rules to be expressed.

Simplicity being the keynote of this method, context-sensitivity is handled not by elegant notations and powerful mechanisms but by special-purpose subroutines written as needed. Although this may sound a little unsatisfactory at first, it works out well when the languages are not heavily context-sensitive. A standard indication or "trigger symbol" is inserted into the output string at the point where a given subroutine should be applied. These symbols actually form part of the target grammar. All that need then be done is to sweep the output string and call the indicated subroutines. In any one group of applications, it is likely that such subroutines will rarely need altering from one study to the next, so they do not interfere with the stated aim of versatility. Examples of their use are provided below.

Finally, with the programming medium I shall assume that the language used has facilities for list processing, recursion, and the representation of strings. It is not difficult to provide such a language (Green & Guest, 1974), but if one is not available FORTRAN would be adequate.

REPRESENTATION OF CONTEXT–FREE GRAMMARS

Each rule of the grammar specifies all the ways in which one nonterminal symbol can be rewritten. Here is part of a grammar in a commonly used notation: nonterminal symbols are in capitals, alternatives are separated by a solidus, and the meta-symbol NONE stands for the empty string, meaning that that symbol may be omitted. For a while translation from English to French will be used for the sake of familiarity, but the technique is not intened to cope with large subsets of natural language.

S → NP VP
NP → the ADJ N
ADJ → black/white/NONE
N → cat/dog
VP → etc.

The grammar must then be stored in the computer in some form. A convenient way to do so, though by no means the only way, is to put each rule into one element of a one-dimensional array and to dispense with the left sides of the rules. Nonterminal symbols can then be replaced by the index number, in the array, of the rule that defines the symbol, e.g., all references to the subject can be replaced by 1, all references to NP by 2, etc. However, if the index is stored in its raw form we shall deprive ourselves of the possibility of using integers in the language, so it is stored as a negative number and made positive when needed. To distinguish between alternatives, the rule is stored as a list of sublists, each sublist being one alternative. The previous rules are, therefore, represented as follows, using parentheses to show list structure:

```
1: ((-2 -5))
2: ((the -3 -4))
3: ((black) (white) (NONE))
4: ((cat) (dog))
5: etc.
```

The algorithm has to be provided with grammars of both the source and the target language. These grammars must be properly paired off so that the same nonterminal symbols are defined in the same order, and within each rule corresponding alternatives are given in the same order. If the grammar above is the source, the target might include these rules:

```
S → NP VP
NP → le N ADJ
ADJ → noir/blanc/NONE
N → chat/chien
VP → etc.
```

Notice that definitions in the target grammar may change the order of constructs (e.g., to put adjectives after their noun) but may not mention a nonterminal symbol unless it is mentioned in the corresponding alternative in the source grammar. It is sensible to write an input routine to accept grammars in the human notation and turn them into internal representation, at the same time checking that they are properly paired.

CONTEXT-FREE PARSING AND TRANSLATION

The message input by the subject is parsed against the grammar of the source language. While doing so, a translation is prepared, using the grammar of the target language. This section presents a semiformal specification of the algorithm preceded by a verbal account.

Although the parsing algorithm is well known, it has been described first to make it clear how the translation is performed. In what follows, identifiers and step numbers refer to the specification below. It is assumed that when the process is recursively reentered, current values of all identifiers are preserved on a stack to be restored when the recursion is complete.

The input string is held in String, and is initially matched against the first alternative of some rule, whichever is specified by Ruleno. The rule is held in Arule (Step 1), and the current alternative is held in Aoption. If the match against the first alternative fails, the second is tried, and so on (Step 2). The members of Aoption are matched one by one, with the current member to be matched held in Asymb (Step 3). The next word of the input is indicated by Stringpointer, which points to somewhere in String. Each time a new alternative is tried, Stringpointer is reset to the start of String (Step 2). Performance of the actual match depends on the value of Asymb. If it is a terminal symbol, it is compared with the next word of the input; if they are identical, Stringpointer is advanced (Step 6). If Asymb is a negative number, then it is a nonterminal symbol by convention. For that to match, we have to get a match for the rule it points to (i.e., if it refers to a noun phrase the input must actually be a noun phrase). The process is re-entered recursively to perform a match against the rule whose number is in Asymb (Step 7). That rule will use up a variable amount of the input string, so one of the results of the recursion gives the value to which Stringpointer should be reset (also Step 7). The last possibility for Asymb is the symbol NONE, which is always taken as a match, without using up any input (Step 4). If the end of an alternative is reached successfully, then the whole rule matches (Step 3), and the process completes with one result indicating a success and another result giving the final value of Stringpointer (Step 9).

An alternative fails when a symbol fails to match in Step 6 or 7) or when the input runs out while the alternative still contains symbols other than "NONE" (in Step 5). If all alternatives fail, the whole rule fails (Step 10). The whole parsing attempt is taken to have failed if no match is found against the first rule, which defines the forms of possible sentences, or if there is some input left over—as though the subject had put in "The cat sat on the mat mat mat." The algorithm does not, as specified, perform the final check that no input is left over, and the user therefore has to check that it terminated with results indicating not only a success but also that the unused portion of the input string is zero.

Given the parser, the translator needs little extra mechanism. While considering an alternative Aoption from the source grammar, the corresponding alternative from the target grammar is located and held in Boption (Steps 1 and 2). If Aoption is entirely composed of terinal symbols, then Boption is the translation required, without further ado, and if the match is successful, then Boption is supplied as the translation result (Step 9). If Aoption contains nonterminal symbols, e.g., for a noun phrase, then the translation must depend on the particular noun phrase in the input. So the same nonterminal symbol is put into the target grammar, and is rewritten during parsing. The nonterminal symbol causes a recursion, which by definition completes with the translation as one result, and that translation is then inserted into Boption. In the algorithm, Asymb is the nonterminal symbol, and its translation is put into Bsymb (Step 7). Every occurrence of the nonterminal Asymb in the target string Boption is then replaced with Bsymb (Step 8). The replacement of every occurrence is the simplest technique, but not necessarily the best. It can be clumsy, as the final example shows.

ALGORITHM FOR CONTEXT-FREE TRANSLATION

The data required for this algorithm are the arrays Sourcegrammar and Targetgrammar, which are never altered, plus an input string held in String which is to be parsed with respect to the rule whose index number is held (negated) in Ruleno. The first call will attempt to parse the entire input with respect to the first rule; recursive entries may then occur.

(1) Set Ruleno = –Ruleno [the index numbers were made negative for convenience]; set Arule = Sourcegrammar (Ruleno); set Brule = Targetgrammar (Ruleno).

(2) [Try a new alternative.] If all the alternatives of Arule have been tried go to Step 10. Otherwise, set Aoption = next alternative of Arule, set Boption = next alternative of Brule, and set Stringpointer to start of String.

(3) If Aoption has no members left unmatched, then go to Step 9 [the match is now complete.] Otherwise set Asymb = next member of Aoption.

(4) If Asymb = "NONE" then go to Step 3.

(5) If Stringpointer has reached the end of String, go to Step 2. [The input string has run out before the end of the match; another alternative must be tried.] Otherwise look at Asymb. If Asymb is a negative number go to Step 7, and if it is not, go to Step 6.

(6)[Asymb is not a negative number, and is therefore a terminal symbol.] If Asymb is identical with the next input symbol, to which Stringpointer points, then advance Stringpointer and go to Step 3. Otherwise go to Step 2. [Match failed.]

(7) [Asymb is a negative number, and therefore a rule index.] Recursively re-enter the algorithm to parse against the rule Asymb points to. If the result is *false* [no match was found], go to Step 2. Otherwisde set Stringpointer = the unused portion of String; set Bsymb = the translation of Asymb and go to Step 8.

(8) [Rewrite Boption.] Examine every member of Boption and replace with Bsymb all those that are the same as Asymb. Go to Step 3.

(9) [Success.] Exit with three results: *true*, to indicate success; the final value of Stringpointer; and Boption, the translation. [If context-sensitive elements are to be used, Boption should first be swept. This procedure returns a result. If *false*, the match has failed so go to Step 2].

(10) [The whole rule fails.] Exit with the result *false*.

Illustration

Suppose that the input string "The black dog . . ." is to be parsed and translated, using the English-to-French fragments given above. The key point is the successive values of Aoption and Boption, and to illustrate these I have used indenting to indicate recursion.

Aoption = NP VP	Boption = NP VP
Aoption = the ADJ N	Boption = le N ADJ
"the" matches	
Aoption = black	Boption = noir
"black" matches	
"ADJ" matches, rewrite Boption	Boption = le N noir
Aoption = cat	Boption = chat
"cat" fails	
Aoption = dog	Boption = chien
"dog" matches	
"N" matches, rewrite Boption	Boption = le chien noir
"NP" matches, rewrite Boption	Boption = le chien noir VP

CONTEXT-SENSITIVE ELEMENTS

The need for context-sensitive rules has been mentioned in the OVERVIEW. One way to handle them would be to include a notation to state that certain rules of grammar could only be used in certain contexts, but that is not so easy to implement. Instead "trigger symbols" are recommended. These are symbols included in the target grammar, each of which points to a particular subroutine and says, in effect, when it occurs in the translation string, "apply that subroutine at this point." Whenever a constituent has been successively parsed and translated (Step 9 in the algorithm) the translation string is swept, searching for triggers, and the appropriate subroutines are called.

The subroutines and their triggers are defined by the user, as additions to the context-free system already described. The subroutines need to have access to the translation string and to any private registers that may be necessary, and in principle their power is unlimited. In practice it is useful to adopt a standard interface in which the translation string is supplied as an input parameter, and two results are returned: the translation string again, because it may have been modified; and a Boolean value, so that if a routine wants to reject a parsing it can do so by giving the result *false*. The test in Step 9 treats such failures as syntactic failures which cause a new alternative to be sought.

Trigger symbols have a variety of uses that might be classified as follows according to the complexity of the context-sensitive behavior. First, there are counter symbols that are simply replaced by a unique number or symbol throughout any one constituent. A typical use of counter symbols is illustrated in the final example. Secondly, triggers are useful when the source language includes commands like "goto label"; trigger symbols inserted with the goto commands and with the labels allow them to be linked up properly, and do the bookkeeping necessary to ensure that all labels are set. Private registers are used for this purpose, to maintain lists of labels that have been mentioned. Finally, triggers may result in the replacement of quite large amounts of the translation string in some cases. The author has, for example, set up an experimental

version of the data base used by Anderson and Bower (1973), using the translator to accept sentences in a very limited subject of English and to translate them into web structures used in the data base. In this case, the data base contains assertions about objects, so it becomes possible to refer to objects either directly by name or else by a sufficient description. In other words, provided the data base contains the assertion that Shakespeare is the author of *Hamlet*, the sentences "Shakespeare slept here" and "The author of *Hamlet* slept here" should produce the same translation. To have them do so, the target grammar was arranged so that any constituent of the input parsed as a noun phrase headed by "the" produced a translation containing a trigger; the trigger called a subroutine to search the data base for an object whose properties matched those asserted by the noun phrase; and, if a suitable object was found, its name was substituted for the noun phrase's translation—while if no such object was found, the sentence was failed.

AN EXAMPLE

To give a moderately realistic example (unlike the translation of English to French), consider the problem of translating a string of commands, some of them conditional, out of a language using nested conditionals and into a language using jumps to labels. That is to say, we want to go from something like this:

if juicy *then* boil;
fry;
if tall *then* chop *else* peel; (etc.)

to a language looking something like this:

```
     test juicy; jump-on-false L1; boil;
L1:  fry;
     test tall; jump-on-false L2; chop; jump L3;
L2:  peel;
L3:  (etc.)
```

The latter type of language is easily interpreted by a program to control a piece of experimental equipment, the more so as it can be guaranteed that any text in the target language is grammatically correct, since any errors in the source will have been detected by the translator. In a practical application, the target language has another useful feature, i.e., jumps are always forward and never backwards.

Before much can be done, it is necessary to observe that every conditional in the source language generates at least one label in the target, and all these labels will have to be distinct. To get distinct labels, it is necessary to use counter symbols. We, therefore, introduce a symbol, e.g., @1 and a register COUNT. The symbol @1 is put into the list of trigger symbols and is arranged to point to a subroutine which, when triggered, will replace every occurrence of @1 in the translation string by the symbol L*n*—where *n* is the current value of the register COUNT, which is then incremented. We shall also need a second symbol e.g., @2, pointing to a routine to do just the same but replacing @2, not @1.

With the help of the counter symbols, we can translate the simple conditional "*if* juicy *then* boil" in two stages. The context-free translation obtained from the target grammar is "test juicy; jump-on-false @1; boil; @1:," and then the counter symbols are replaced by L1 (or L2, or L3, etc., depending upon how many labels have been used before). The *if-then-else* construction needs two labels.

This example has been chosen because it illustrates, besides the use of counter symbols, most of the tactical niceties that can arise. In the section on the representation of grammars, mention has been made of the possibility of using the grammar-reading routine to perform a mechanical check on the grammars, to ensure that the target grammar mentions no nonterminals that are not mentioned by the source grammar in the corresponding alternative. Of course, several other checks are useful. Here, then, is an appropriate pair of grammars, followed by remarks on their construction.

Source

```
PROGRAM → STATEMENT; MORETEXT
MORETEXT → STATEMENT; MORETEXT / NONE
STATEMENT → if PRED then STATEMENT else STATEMENT2
            /
            if PRED then STATEMENT
            /
            ACT
STATEMENT 2 → STATEMENT
PRED → short / tall / juicy / • • •
ACT → boil / fry / chop / peel / • • •
```

Target

```
PROGRAM → STATEMENT; MORETEXT
MORETEXT → STATEMENT; MORETEXT / NONE
STATEMENT → test PRED; jump-on-false @1; STATEMENT;
            jump @2; @1: STATEMENT2; @2:
            /
            test PRED; jump-on-false @1; STATEMENT; @1:
            /
            ACT
STATEMENT 2 → STATEMENT
PRED → short / tall / juicy / • • •
ACT → boil / fry / chop / peel / • • •
```

Remarks

The first pair of rules says that a program consists of one statement plus any number more, including zero. It would also be possible to express the second rule without using NONE, as

```
MORETEXT → STATEMENT; MORETEXT/STATEMENT;
```

but that form would be less efficient—the last statement would be parsed twice, once as the beginning of the first alternative, which fails because there is no more text, and then again as the second alternative.

The second rule can*not* be expressed in the form

MORETEXT → MORETEXT STATEMENT; /NONE

which would cause an infinite recursion in the parser. This can be checked mechanically by the grammar-reading routine; no definition may start with the symbol being defined. (There is always a way around this restriction.)

In the third rule, the first alternative mentions two statements. They must be distinguished somehow, say by calling one STATEMENT2; otherwise they will automatically be given the same translation. Remember that every occurrence of a particular nonterminal symbol in the translation string is replaced by the first translation found (see section on context-free parsing and translation.) The grammar-reading routine can give a warning if the same nonterminal is used twice in one alternative.

In the third rule, the order of the first two alternatives is critical. When one alternative is the same as the beginning of another, as here, the second corresponds exactly with the beginning of the first, the longer one must be tried first. This can be checked mechanically by the grammar-reading routine.

The grammars have been devised to allow nested conditionals to be used. Suppose that following the example input at the start of this section the next statement was

if short *then*
  if pink *then* stew; . . .

(Because the aim is just to illustrate a simple nested construction, the syntax defined is too weak for practical use, in fact it is actually ambiguous in some cases, but never mind) if the parsing is followed through, it will be observed that while parsing the outer conditional, "*if* short *then* . . .," a recursive entry is made to parse the inner conditional. This recursive entry is the first entry to reach completion and so the translation string is swept for counter symbols to be replaced, e.g., by L4 at that particular time, giving "test pink, jump-on-false L4, stew, L4:." The parsing of the outer conditional is then completed and the counter symbols are replaced, this time by L5. The entire translation is, therefore:

```
    test short; jump-on-false L5;
        test pink; jump-on-false L4; stew;
    L4:
L5: . . .
```

(Indenting for clarity is not part of the translation.)

CONCLUSIONS

The paired-grammar translator is intended for use in situations where the subject is to convey information to the computer by using a simple control language. Writing the program is an investment that will prove worthwhile if many different languages are eventually to be used, because it is much easier to write and test new grammars than to write and test one-off special-purpose programs to handle each language, particularly because with the translator checks can be included to detect most types of finger trouble in the grammars. The progamming investment required has been brought to a minimum by excluding all unnecessary frills from the algorithm, as described above. The user may or may not come to feel, with experience, that bells and whistles should be fitted.

The technique is not intended to cope with full natural language, high-speed compilation, highly context-sensitive langages, or cases where the source and target are structurally dissimilar. The area that it does cope with, however, is, in our experience, the one that is desirable in on-line experiments requiring linguistic control. Within that area, the technique has proved sufficiently versatile, to take two extremes, to translate from ALGOL conditionals to a simple interpreter code, and to translate from a subset of English to the web structures used by Anderson and Bower (1973).

REFERENCES

Anderson, J. R., & Bower, G. H. *Human associative memory*. Washington, D.C.: Winston, 1973.

Fitter, M. J. An extensible system for controlling a scheduling task. Memo No. 51, MRC Social and Applied Psychology Unit, University of Sheffield. Sheffield, England, 1974.

Fitter, M. J., & Daly, A. C. An extensible touchboard for on-line experiments. Quarterly Journal of Experimental Psychology, 1975, 27, 673-676.

Foster, J. M. *Automatic syntactic analysis*. London, MacDonald/Elsevier, 1970.

Green, T. R. G., & Guest, D. J. An easily-implemented language for controlling complex experiments. International Journal of Man-Machine Studies, 1974, 6, 335-359.

Gries, D. *Compiler construction for digital computers*. New York, Wiley, 1971.

Ingerman, P. Z. *A syntax oriented translator*. London: Academic Press, 1966.

Sime, M. E., Green, T. R. G., & Guest, D. J. Psychological evaluation of two conditional constructions used in computer languages. International Journal of Man-Machine Studies, 1973, 5, 105-113.

Sime, M. E., Green, T. R. G., & Guest, D. J. Scope marking in in computer conditionals—a psychological evaluation. Memo No. 48, MRC Social and Applied Psychology Unit, University of Sheffield, England, 1974.

(Received for publication June 9, 1975;
accepted for publication August 10, 1975.)