An algorithm for inverting certain translators of context-free languages

D. T. Goodwin

Department of Computer Science, University of Keele, Keele, Staffs, ST5 5BG

This paper considers translators between related pairs of context-free languages. These translators are defined as grammars for input to a syntax-directed translation system on the lines of Metcalfe (1964) and Reeves (1967). By applying a simple algorithm to each of the production rules of such a grammar it is shown how to derive the inverse translator, under given conditions.

(Received February 1972, Revised March 1974)

Introduction

After the author had implemented a syntax-directed translation system following Metcalfe (1964) and Reeves (1967), he considered whether it was possible to write inverse translators in the same system, i.e. translators which took the former output as input and produced the former input as output. This was found to be possible in certain cases, and this led to the development of an algorithm for automatically converting a translator into its inverse. It is this work that is reported in this paper, which may be read without prior knowledge of the Metcalfe-Reeves system.

To write a translator from a CF (Context-free) language L to a language S' the user begins by writing a BNF-like grammar for L. To every rule of this grammar he adds material *in situ* which indicates the desired corresponding output in language S'. The result is a 'translator' from L to S', or more precisely, from L to a subset S of S'. When the system has been loaded with this translator it accepts a string of L-text and produces as output the corresponding S-text. Under given conditions it is shown below that S is context-free, that the grammars of S and L are very similar, and that an S to L translator can be derived automatically from the user's L to S translator.

The inversion algorithm could be useful in cases where S is such a large subset of S' that it can be used in place of S'. Once the L to S' translator has been written, an S' to L translator is then available at little extra cost. Whether S and S' are similar enough might be a matter of opinion, but this similarity is likely only if S' is context-free and similar to L. An example might be when L and S' are essentially the same high-level programming language differing only in compiler-dependent features, where only correct programs (which would then conform to a CF-grammar) were being manipulated.

The earliest reference to the inversion of grammars known to the author is in Evey (1963), where it is shown that a push-down transducer may be used to 'translate' from one CF language to another related CF language, and that the inverse translator can be constructed simply. In this paper, however, it is not necessary to assume anything about the internal construction of the parser—only that its input and output are as described in the next section. What is discussed here is a further transformation of the parser output, used by Metcalfe, which is stimulated by special operators, viz. the symbols $\overline{X}$ and $\overline{C}$, in the output language. These operators enable the translator-writer to express concisely a wide range of transformations of the input.

Notation

A CF grammar is defined in the usual way, and attention is restricted to 'reduced' or 'admissible' grammars in which each non-terminal is used individually in the derivation of at least one string of the language. The notation used here is on the lines of Ginsburg (1966). Let the translator be called T_{LS}. It has a set of 'non-terminals' $\{\alpha, \beta, \gamma, \ldots, \tau \ldots,\}$ including a 'top-level' non-terminal σ. Each 'terminal symbol' of T_{LS} actually consists of a pair of symbols: the first symbol is one of a set of input terminals $\{a, b, c, \ldots\}$ the terminal alphabet of L, which also includes the empty input symbol ε. The second symbol of the pair is one of a set of output terminals. For clarity these are denoted by $A, B, C, \ldots$ but there is no necessity for the input and output marks to be distinct. The output alphabet also includes an empty symbol $\in$ and the special output terminal symbols $\overline{X}$ and $\overline{C}$ whose uses are described below. Often the reader will find that the combinations $\varepsilon\overline{X}$ and $\varepsilon\overline{C}$ are contracted to $\overline{X}$ and $\overline{C}$ respectively, since the latter are never paired with any input symbol except ε.

Example

L might have the rules:

$$\sigma \rightarrow ab\alpha c$$
$$\alpha \rightarrow d\alpha$$
$$\alpha \rightarrow \varepsilon\ .$$

The possible input strings are $abc, abdc, \ldots abd^n c$, for all integers n. By adding output symbols and an instance of ε, the following rules could be obtained for a translator T_{LS}:

$$\sigma \rightarrow aAb\in\alpha cD$$
$$\alpha \rightarrow d\in\alpha\varepsilon K$$
$$\alpha \rightarrow \varepsilon F\ .$$

Then for input $abd^n c$ the output is $A\in\in^n FK^n D$ which collapses to $AFK^n D$. In detail for $n = 2$, the input $abddc$ leads to the parse-tree of **Fig. 1**.

The output is the string of symbols $A\in\in\in FKKD$ arising from the terminal pairs occurring in the parse tree.

The T_{LS} grammar is thus a superposition of two CF grammars with the same rule-structure differing only in the particular terminals used. One is the grammar of L and the other is the grammar of an intermediate output language P, say, whose strings in general contain instances of $\overline{X}$ and $\overline{C}$.

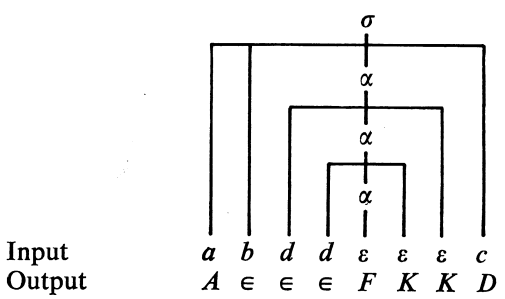

Fig. 1 Parse tree for example input string

The edit stack

Following Metcalfe, the output (in language P) from the parser is loaded symbol by symbol on to a push-down 'edit-stack'. When an $\overline{X}$ is encountered in the parser-output, it is not loaded, however, but is used as a command to interchange the items in the top two cells of the stack. When a $\overline{C}$ is encountered it is not loaded, but is used as a command to concatenate the top two items; the effect is that they become one item and that the stack is nested up one position. Each 'cell' on the stack is capable of holding a string of arbitrary length. At the end of the process, the final output is taken from the bottom of the stack upwards, ignoring the divisions between the items. **Fig. 2** shows an outline of the system. The flow of symbols is shown going from right to left so as to suggest that a language P string $ABC\ldots$ is processed in the order A, then B, then C, and so on.

Example

Suppose T_{LS} has the one rule:

$$\sigma \rightarrow aAbB\overline{X}cC\overline{C}dD\overline{X}eE\ .$$

Then from the L-string *abcde* the parser machine yields $AB\overline{X}C\overline{C}D\overline{X}E$ which is then applied symbol by symbol to the edit stack as follows:

Stack Contents (top at right)	*String yet to be applied to the stack*
	$AB\overline{X}C\overline{C}D\overline{X}E$
A	$B\overline{X}C\overline{C}D\overline{X}E$
$A\ B$	$\overline{X}C\overline{C}D\overline{X}E$
$B\ A$	$C\overline{C}D\overline{X}E$
$B\ A\ C$	$\overline{C}D\overline{X}E$
$B\ AC$	$D\overline{X}E$
$B\ AC\ D$	$\overline{X}E$
$B\ D\ AC$	E
$B\ D\ AC\ E$	

The final output is $BDACE$.

The effect of the editing is thus to execute a permutation of the ordinary output symbols in the P-language string and to delete the special $\overline{X}$ and $\overline{C}$ symbols.

It is a useful convention that the instances of ϵ in the P-string are not annihilated until they are combined with other items on the edit stack by the use of $\overline{C}$, or if this does not occur, until the edit stack is unloaded, when they are ignored.

Some string operations

Now defined are some operations on strings of symbols such as might arise on the right-hand side of a rule of T_{LS}.

$N(\ldots)$: Delete all the instances of the pairs $\epsilon\overline{X}$ or $\epsilon\overline{C}$, (or of the single symbols $\overline{X}$, $\overline{C}$ if no input symbols occur in the operand-string).

$I(\ldots)$: Delete all output symbols from the terminal pairs.

$O(\ldots)$: Delete all input symbols from terminal pairs.

$O''(\ldots)$: As $O(\ldots)$ but also add the suffix $''$ to each non-terminal.

$R(\ldots)$: Reverse the L-symbol and the S-symbol in each non-special terminal pair.

Now if a rule of T_{LS} be written

$$\tau \rightarrow f_1 \ldots f_i \ldots f_m$$

where each of the f_i is a terminal pair or a non-terminal, then the corresponding rule of L is

$$\tau \rightarrow I(N(f_1 \ldots f_m))$$

and the corresponding rule of P is

$$\tau \rightarrow O(f_1 \ldots f_m)\ .$$

Suppose now that the string $O(f_1 \ldots f_m)$ be loaded on to the edit stack, with the non-terminal f_i just being regarded as individual symbols. The result is a permutation π, say, of the ordinary symbols of $O(f_1 \ldots f_m)$ i.e.

$$\pi(N(O(f_1 \ldots f_m)))\ .$$

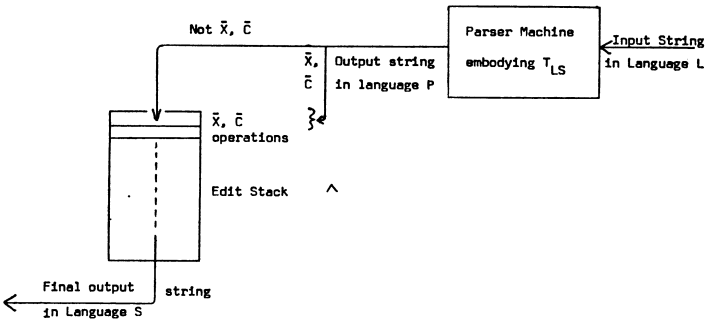

Fig. 2 L to S translation

Example

Suppose T_{LS} contains a rule

$$\alpha \rightarrow aA\beta\overline{X}\overline{C}\ .$$

Then the corresponding rule of L is

$$\begin{aligned}\alpha &\rightarrow I(N(aA\beta\overline{X}\overline{C}))\\ &= I(aA\beta)\\ &= a\beta\ .\end{aligned}$$

The corresponding rule of P is

$$\begin{aligned}\alpha &\rightarrow O(aA\beta\overline{X}\overline{C})\\ &= A\beta\overline{X}\overline{C}\ .\end{aligned}$$

The expression $\pi(N(O(aA\beta\overline{X}\overline{C})))$

$= \pi(N(A\beta\overline{X}\overline{C}))$

$= \pi(A\beta)$

$= \beta A$, because $\overline{X}\overline{C}$ yields an interchange of A and β.

'Unitary' translators

The inversion algorithm is first considered for a subset of all translators which is now defined. Later it is shown to apply more widely.

A non-terminal is said to be 'unitary' if each of its terminal strings, when loaded on to the edit stack, forms exactly one item and does not change the previous contents of the stack apart from nesting them down one cell.

Examples

σ is unitary in the following grammars:

$$\sigma \rightarrow aA$$
$$\sigma \rightarrow aAbB\overline{X}\overline{C}$$
$$\begin{cases}\sigma \rightarrow aA\\ \sigma \rightarrow xXyYzZ\overline{C}\overline{C}\end{cases}$$

σ is not unitary in the following:

$$\sigma \rightarrow aAbB$$
$$\begin{cases}\sigma \rightarrow aA\\ \sigma \rightarrow \overline{C}aAbB\ .\end{cases}$$

A translator is said to be unitary if all its non-terminals are unitary. When writing a translator it is natural to think in terms of items on the edit stack, and hence in terms of non-terminals which form single-items. Unitary translators therefore form an important subset of all translators.

Two simple algorithms are now presented which together determine whether a given translator is unitary. The possibility of recursive rules makes this problem non-trivial. The proofs of the algorithms are long enough to make a major digression if given here—another article which includes them is in preparation.

Firstly, for any terminal string s define:

(*a*) its 'length', denoted by $l(s)$, to be the net increase in the number of items on the edit stack as counted before and

immediately after the deposition of s

(*b*) its degree of interference, denoted by $d(s)$, to be the number of items previously on the edit stack which are changed, perhaps in their order only, during the deposition of s.

The simplest string s is a single terminal, for which the $l(s)$ and $d(s)$ values are given in **Table 1**.

Table 1

Type of terminal	*l-value*	*d-value*
Ordinary terminal	1	0
$\overline{C}$	−1	2
$\overline{X}$	0	2

A non-terminal τ possesses a set of l and d values, which includes one l-value and one d-value for every terminal expansion of τ. It is easy to see that τ is unitary if for every terminal expansion of τ, $l(\tau) = 1$ and $d(\tau) = 0$.

To test a translator for 'unitary-ness'

(*a*) For every rule $\tau \rightarrow f_1 \ldots f_i \ldots f_m$ form the sum

$$\sum_{i=1}^{m} l(f_i)$$

on the assumption that if f_i is a non-terminal then $l(f_i) = 1$. If the sum is 1 for every rule then $l(\tau) = 1$ for every τ.

(*b*) Assume that $l(\tau) = 1$ for every τ. Now for each rule in turn calculate

$$\max_{1<i\leqslant m} \left(d(f_1), d(f_i) - \sum_{j=1}^{i-1} l(f_j)\right)$$

on the assumption that if f_k is a non-terminal then $d(f_k) = 0$, $1 \leqslant k \leqslant m$. If the expression is zero for every rule then $d(\tau) = 0$ for every τ and the translator is unitary.

Theorem 1:
If T_{LS} is a unitary translator, then S, the set of all possible output strings, is a CF language. There exist related grammars for L and S such that for each rule $\tau \rightarrow f_1 \ldots f_m$ of T_{LS} there corresponds an L-rule $\tau \rightarrow I(N(f_i \ldots f_m))$ and an S-rule $\tau \rightarrow \pi(N(O(f_1 \ldots f_m)))$.

Proof:
The proof is taken in two stages:

Lemma 1:
Each string in S conforms to the given CF grammar so that S is a subset of a CF language S''.

Lemma 2:
Every string in S'' is shown to be the possible output of some input string, and is therefore in S, so that S'' is a subset of S.

Having proved these lemmas it follows at once that S and S'' are the same set.

Proof of Lemma 1:
Choose a string s in S. Then there is a parse-tree of σ, the top-level non-terminal of T_{LS}, whose output terminals, when processed on the edit stack, form the string s. Consider a particular instance of a non-terminal τ which occurs in some node of the tree. The next nodes down from this τ contain the individual symbols $f_1, f_2, \ldots f_m$ of some rule $\tau \rightarrow f_1 \ldots f_i \ldots f_m$ of T_{LS}.

Now consider what happens when the parser output string is loaded on to the edit stack. Because T_{LS} is unitary then so is τ, and the terminal substring which is an expansion of τ loads on to the edit stack as a single item. Call this item τ''. Again because T_{LS} is unitary, all the non-terminal f_i are unitary and yield single items f_i'', say, so that τ'' is a constant permutation (and concatenation) of the f_i'', whether they are ordinary terminals or non-terminals i.e.

$$\tau'' = \pi(N(O''(f_1 \ldots f_m))) \ .$$

Then since the grammars of L and P are reduced, every τ must appear in the parse tree for some eventual output string s. Thus for every rule $\tau \rightarrow I(N(f_1 \ldots f_m))$ of L there is a corresponding relationship $\tau'' = \pi(N(O''(f_1 \ldots f_m)))$ of substrings in the final output. Thus s is a string of a CF-language S'' whose rules, by a trivial change of notation, are as required.

Proof of Lemma 2:
Let s be a string of S''. Then a parse-tree of s exists according to the grammar of S''. By applying the inverse permutations π^{-1} to the rule-instances in this tree, we derive a parse-tree of P and hence of T_{LS} from which the final output string s could be derived. By using the input symbols in the terminal pairs in the T_{LS}-tree an input string is obtained from which s could be derived. Hence s is in S.

Note on Theorem 1: The word 'possible' in the statement of Theorem 1 is necessary when L is ambiguous. For then one input string could give rise to two distinct output strings and whether or not one of them ever appeared would be a matter of parser design.

Inverse translators

An implementation is now developed for the inverse translation process of Lemma 2, which was:

(1) Parse the input string s with respect to the grammar of S.

(2) Apply the appropriate inverse permutations to the tree branches.

(3) Replace the symbols of S with the corresponding symbols of L and take the resulting L-string as the output.

Notice that the same L-string is obtained whether the S-symbols are placed by L-symbols before or after permutations of the branches. Thus the above scheme may be re-written:

(1) Parse the input string s.

(2) Replace S-symbols by L-symbols.

(3) Apply the inverse permutations to give the output string.

The first two steps are now exactly what the parser does when using T_{LS} except that the roles of L and S are interchanged. Call this new translator T_{SL}. Then for every rule $\tau \rightarrow f_1 \ldots f_m$ of T_{LS} there is a rule $\tau \rightarrow \pi(N(O(f_1 \ldots f_m)))$ of S, which, by reversing pairs instead of just selecting the output symbols yields a rule $\tau \rightarrow \pi(N(R(f_1 \ldots f_m)))$ for T_{SL}. This T_{SL} is not yet completely correct, because, although parsing and symbol replacement are done, there is as yet no way of performing the inverse permutations of step 3.

Example
Suppose the rules of T_{LS} are

$$\begin{cases} \sigma \rightarrow aA\delta\overline{X}\overline{C}bB\overline{X}\overline{C} \\ \delta \rightarrow eEgG\overline{X}\overline{C} \\ \delta \rightarrow fF \ . \end{cases}$$

Then T_{LS} is unitary and the input string *aegb* of L leads to parser output $AEG\overline{X}\overline{C}\overline{X}\overline{C}B\overline{X}\overline{C}$ which when edited becomes $BGEA$. Now by applying $R(\ldots)$ to all rules and $N(\ldots)$ and $\pi(\ldots)$ to the first two rules the following rules for T_{SL} are obtained:

$$\begin{cases} \sigma \rightarrow Bb\delta Aa \\ \delta \rightarrow GgEe \\ \delta \rightarrow Ff \ . \end{cases}$$

Input $BGEA$ to this T_{SL} would result in output *bgea* which is not the same as the input to T_{LS}.

One might hope that it would be possible to add $\overline{X}$ and $\overline{C}$ symbols to the rules of a T_{SL} so as to obtain the required inverse permutations. However this cannot be done in general because there exist permutations which can be implemented using $\overline{X}$ and $\overline{C}$ but whose inverses cannot. An example is $\sigma \rightarrow AaBb\overline{X}Cc\overline{C}Dd\overline{X}Ee\overline{X}Ff\overline{X}\overline{C}\overline{X}\overline{C}\overline{C}\overline{C}$. By trying every combination of $\overline{X}$ and $\overline{C}$ it is easy to show that the inverse rule cannot be constructed. Some other technique is therefore necessary, and the author's proposal is to make reversible the process of applying a string to the edit stack. The stack is redefined so that one cell holds only one symbol. The new action of the $\overline{C}$ operator is just to add a new $\overline{D}$ symbol to the stack. The new action of the $\overline{X}$ operator is to interchange the top two items on the stack and then to add a new $\overline{Y}$ symbol to the stack. A stack item is now defined by:

⟨item⟩::=⟨normal output symbol, possibly empty⟩
|⟨item⟩⟨item⟩ $\overline{D}$
|⟨item⟩ $\overline{Y}$

where the top of the stack is at the right. An item can thus be determined by considering its symbols one by one starting at the top of the stack.

Examples of items

Unedited string	A	$AB\overline{C}$	$AA\overline{C}BB\overline{C}\overline{C}$	$AB\overline{X}\overline{C}$	$EB\overline{X}\overline{C}E\overline{C}A\overline{X}\overline{C}$
Item on stack	A	$AB\overline{D}$	$AA\overline{D}BB\overline{D}\overline{D}$	$BA\overline{Y}\overline{D}$	$ABE\overline{Y}\overline{D}E\overline{D}\overline{Y}\overline{D}$

It is easy to see that items created like this contain the same sequence of normal output symbols as they would have done under the former stack definition.

Lemma 3:
The editing process is now reversible.

Proof:
The editing process consists of a sequence of actions, one for each symbol of the string to be edited. The whole sequence is reversible if each action in the sequence can be proved reversible. There are three types of action, depending on whether the next symbol is a $\overline{C}$, $\overline{X}$ or a normal symbol. Each type leaves on the top of the stack a distinguishing symbol which indicates what the reverse action should be. This can always be performed.

Definition
Let $\pi_D(\ldots)$ denote the editing process just described, with the new $\overline{Y}$ and $\overline{D}$ symbols *retained* in the final output string.

Theorem 2:
If T_{LS} is unitary with a typical rule $\tau \rightarrow f_1 \ldots f_m$, then the rules of T_{SL} are the corresponding rules $\tau \rightarrow \pi_D(R(f_1 \ldots f_m))$.

Proof:
The translation scheme to be validated is illustrated in **Fig. 3.** The output from the parser is loaded *en bloc* on to the stack and then is unloaded using the $\overline{Y}$ and $\overline{D}$ operators. The string thus unloaded, with $\overline{C}$ and $\overline{X}$ removed, is the alleged final output string.

Now the input language to the T_{SL} parser is S because for each rule, $\pi_D(\ldots)$ and $\pi(N(\ldots))$ give the same permutation of the non-special f_i. It remains to be shown that the parser output is reverse-edited into the appropriate L-string, i.e. that for each non-terminal instance in the parser tree the correct inverse permutation is carried out.
This is true provided that:

(*a*) the editing process is reversible—proved by Lemma 3.

(*b*) any expansion of a non-terminal f_i (of T_{SL}) acts as one item on the stack so that the Y operator works correctly. This is true because the corresponding f_i in T_{LS} is unitary from the condition of the theorem. All expansions of this latter f_i become single items on the stack. However these items *are* the expansions of the f_i in T_{SL} (except that S-symbols appear in place of L-symbols, but this does not affect the number of items).

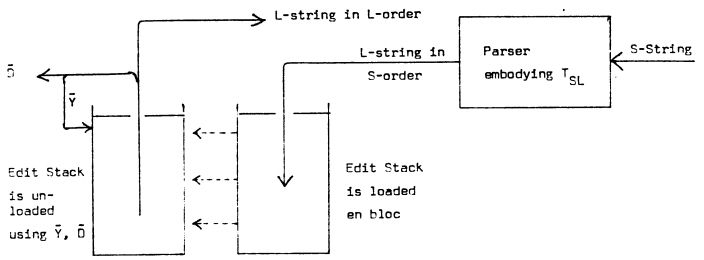

Fig. 3 S to L translation

Example
As before, let the rules of T_{LS} be

$$\sigma \rightarrow aA\delta\overline{X}\overline{C}bB\overline{X}\overline{C}$$
$$\delta \rightarrow eEgG\overline{X}\overline{C}$$
$$\delta \rightarrow fF\ .$$

Applying $\pi_D(R(\ldots))$ to each rule gives

$$\sigma \rightarrow Bb\delta Aa\overline{Y}\overline{D}\overline{Y}\overline{D}$$
$$\delta \rightarrow GgEe\overline{Y}\overline{D}$$
$$\delta \rightarrow Ff\ .$$

as the grammar of T_{SL}.
Input *aegb* to T_{LS} gives final output *BGEA*. Using this as input to T_{SL}, the parser output is $bge\overline{Y}Da\overline{Y}D\overline{Y}D$ which is placed on the stack. The unloading goes as follows:

Stack contents (*top at right*)	*Final output*
$bge\overline{Y}Da\overline{Y}D\overline{Y}D$	—
$bge\overline{Y}Da\overline{Y}D\overline{Y}$	—
$ge\overline{Y}Da\overline{Y}Db$	—
$ge\overline{Y}Da\overline{Y}D$	b
$ge\overline{Y}Da\overline{Y}$	b
$age\overline{Y}D$	b
$age\overline{Y}$	b
aeg	b
—	$aegb$

Automation of the algorithm
The algorithm of Theorem 2 can itself be expressed as a translator, the input and output of which are an arbitrary unitary T_{LS} and its corresponding T_{SL} respectively. Such a T-grammar is referred to as an 'inverter' I, and the part of it which manipulates the right-hand side of one rule is given below. To improve readability a BNF-like notation is used:

⟨RULE RHS⟩ ::= $\varepsilon\in\varepsilon\in\varepsilon\in$⟨ELEMENT STRING⟩$\overline{C}\overline{C}\overline{C}\overline{C}$
⟨ELEMENT STRING⟩ ::= ⟨TERMINAL PAIR⟩
⟨ELEMENT STRING⟩$\overline{C}$
|⟨NON-TERMINAL⟩⟨ELEMENT STRING⟩$\overline{C}$
|$\overline{X}$⟨$\overline{X}$⟩$\overline{C}$⟨ELEMENT STRING⟩
|$\overline{C}$⟨$\overline{C}$⟩$\overline{C}$⟨ELEMENT STRING⟩$\varepsilon\in$
|$\varepsilon\in$
⟨TERMINAL PAIR⟩ ::= ⟨*L*-TERMINAL⟩
⟨*S*-TERMINAL⟩$\overline{X}\overline{C}$

where

⟨*L*-TERMINAL⟩, ⟨*S*-TERMINAL⟩	accept any L or S terminal respectively and generate the same terminal.
⟨NON-TERMINAL⟩	accepts and generates any non-terminal.
⟨$\overline{X}$⟩	accepts $\overline{X}$ and generates $\overline{Y}$
⟨$\overline{C}$⟩	accepts $\overline{C}$ and generates $\overline{D}$, i.e., there is assumed to be a distinction between the edit codes $\overline{X}$ etc. of I and the data-

symbols $\overline{X}$, etc. of T_{LS} and $T_{SL,\text{ suffixed }D}$ below for clarity.

The action of ⟨ELEMENT STRING⟩ when processing a rule $\tau \to f_1 \ldots f_m$ is to examine each $f_1, f_2 \ldots$ in turn and send them to the edit stack suitably modified so that $\pi_D(R(f_1 \ldots f_m))$ is performed.

It is interesting to consider inverting I itself whose grammar contains one rule with $d = 2$ (for X) and one with $d = 1$ (for C).

$$\overline{X}\langle\overline{X}\rangle\overline{C}\langle\text{ELEMENT STRING}\rangle$$

leads to parser machine output

$$\in\in\in\in\overline{X}\,\overline{Y}_D\overline{C}\langle\overline{X}\rangle\overline{C}\overline{D}_D\overline{C}\langle\text{ELEMENT STRING}\rangle\in\overline{C}\in\overline{C}\overline{C}\overline{C}\overline{C}\overline{C}$$

which leads to final output

$$\overline{Y}\langle\overline{X}\rangle\overline{D}\langle\text{ELEMENT STRING}\rangle$$

where $\langle\overline{X}\rangle$ now accepts $\overline{Y}_D$ and generates $\overline{X}_D$.

Similarly

$$\overline{C}\langle\overline{C}\rangle\overline{C}\langle\text{ELEMENT STRING}\rangle\varepsilon\in$$

leads to final output

$$\overline{D}\langle\overline{C}\rangle\overline{D}\langle\text{ELEMENT STRING}\rangle\in\varepsilon$$

where $\langle\overline{C}\rangle$ now accepts $\overline{D}_D$ and generates $\overline{C}_D$ and ε proceeds to the stack before being regarded as a dummy.

Since these inverted rules work perfectly well, it can be considered how to relax the condition $d = 0$ for the working of the algorithm in Theorem 2. However, consider

$$\alpha \to aAbB\beta$$
$$\beta \to \overline{X}$$

with which the input string ab yields output BA.

The algorithm inverts the first rule into

$$\alpha \to AaBb\beta$$

which does not accept BA. The trouble is that the $\overline{X}$ is not explicit and is therefore not taken into account by the inversion algorithm. This leads to the simple (but not all-embracing) rule that where an $\overline{X}$ interchanges items already on the stack before the activation of the rule containing the $\overline{X}$, they must have been placed there by two identical elements for which $l = 1$ and $d = 0$, e.g.

$$\begin{cases}\alpha \to aAaA\beta \\ \beta \to \overline{X}\end{cases} \text{ or } \begin{cases}\alpha \to \gamma\gamma\beta \\ \beta \to \overline{X} \\ \gamma \to aA\end{cases}.$$

(Note that the rule ⟨RULE RHS⟩ . . . is written to cater for the inversion of right-hand sides with $d \leqslant 3$. The reader is invited to attempt its use for $\tau \to \overline{C}\overline{C}\overline{C}$ and $\tau \to \overline{C}\overline{C}\overline{C}\overline{C}$).

An extension to Theorem 2

Another relaxation of the 'unitary-ness' condition is given below as Theorem 3 and is simple to prove.

Definition:
In a rule $\tau \to f_1 \ldots f_m$, a symbol f_i 'is in the range of an $\overline{X}$' if there exists j, $m \geqslant j > i$ for which $f_j = \overline{X}$ and such that when the symbols $f_1 \ldots f_{j-1}$ are applied to the stack, f_j then interchanges two items, one of which includes f_i.

Notice here that the f_i are treated as symbols, so that the predicate 'is in the range of an $\overline{X}$' is readily evaluated by inspection of the individual rule involved.

Theorem 3:
A translator T_{LS} is invertible using the algorithm of Theorem 2 if

(*a*) $d(\tau) = 0$ for every τ

(*b*) $l(\tau) = 0$ for all τ which are in the range of an $\overline{X}$ in any rule.

Proof:
To prove invertibility it is sufficient to show that $\pi_D(\ldots)$ applies the same permutation to $R(f_1 \ldots f_m)$ at the time of inversion of T_{LS} as π does to an expansion of $f_1 \ldots f_m$ at the time of use of T_{LS}. The only rearrangement operator is $\overline{X}$ and the permutations are the same if $l(\tau) = 1$ and $d(\tau) = 0$ in the range of every instance of $\overline{X}$ in $f_1 \ldots f_m$. The condition $d(\tau) = 0$ for every τ not in the range of an $\overline{X}$ simply avoids the complications of the last section.

The use of ambiguous grammars

The processes described above need clarification if either or both of L and S have ambiguous grammars, for then translation followed by inverse translation will not necessarily result in the original input string. It is supposed that the writer of the translators is aware of all the ambiguities. He allows them only because the different outputs derived from different parses of a given string 'mean the same' in every case, in some sense known to him. This can be developed as follows:

Definition:
Consider a (possibly infinite) sequence of language strings $s_1, s_2 \ldots s_r, \ldots$ in which the odd-indexed strings are in L and the even-indexed strings are in S. Let translations $s_r \to s_{r+1}$ exist for each $r > 0$ for which s_{r+1} exists. Then for any integers p, q such that $p < q$ and s_q exists, the relationship between s_p and s_q is 's_q means the same as s_p'.

Corollary 1:
Given any strings s, s' and s'' such that
(*a*) s'' means the same as s'
(*b*) s' means the same as s
then s'' means the same as s. This follows immediately from the definition.

Corollary 2:
If s' means the same as s then s means the same as s'.
It is sufficient to prove the inverse for each translation $s_r \to s_{r+1}$, (whether r is odd or even).
But this is what the main part of the paper demonstrates, using the given algorithm and edit stack arrangement.

Thus each string of the sequence $s_1, s_2, \ldots$ means the same as all the others. (Note however that if an L-string and an S-string mean the same there is not necessarily a translation between them—for consider the strings 'a' and 'B' and the translator $\sigma \to aA$, $\sigma \to bA$, $\sigma \to bB$).

If, for example, from input s_1 the parser derives output s_2, it is of no great concern whether the inverse parser derives s_1 or s_3 from s_2. (The only concern would be brevity and human readability).

Example:
Using a form of BNF again for clarity and letting both input and output characters be chosen from the FORTRAN Set:

$$\langle\text{exp}\rangle ::= \langle\text{op}\rangle{+}{+}\overline{X}\overline{C}\overline{C}$$
$$\langle\text{op}\rangle ::= \langle\text{letter}\rangle|(\in\langle\text{op}\rangle)\in CC$$
$$\langle\text{letter}\rangle ::= AA|BB\ .$$

Then all the inputs $A + B$, $(A) + (B)$, $((A)) + B$ and many more all translate into $AB+$. The inverse grammar is

$$\langle\text{exp} ::= \langle\text{op}\rangle\,\langle\text{op}\rangle + + \overline{Y}\overline{D}\overline{D}$$
$$\langle\text{op}\rangle ::= \langle\text{letter}\rangle|\varepsilon(\langle\text{op}\rangle\varepsilon)\overline{D}\overline{D}$$
$$\langle\text{letter}\rangle ::= AA|BB$$

which is ambiguous. Given an input string (such as $AB+$) the different parses lead to different outputs depending on how many times the second alternative of ⟨op⟩ is used. All these outputs, $A + B$, $(A) + (B)$ etc., have an obvious constant meaning. $A + B$ is the shortest and most readable.

Implementation

Following Professor Reeves' clear description of his parser and editor machines, the author implemented a similar system in POP-2. Little change to the meta-language was found necessary to permit the writing of an invertible meta-grammar and an invertible inverter. With only a few alterations, all the various elements of the metalanguage were found to be invertible, including some context-sensitive features. Further information can be given on request.

Acknowledgements

The author wishes to thank the Journal's referees and also Professor Reeves and Dr. R. Housden for their constructive and very helpful criticism. He is also grateful to Professor Reeves for his kind provision of extra documentation, and to Mr. R. Bowman for his help with the POP-2 language and its ICL 4130 implementation. Thanks are also due to Mrs. C. Goulding and Miss A. Kozaryn for their patient typing of many drafts.

References

ALPIAR, R, (1971). Double Syntax oriented processing. *The Computer Journal*, Vol. 14, pp. 25-37.

EVEY, R. J. (1963). The Theory and Application of Pushdown Store Machines, in *Mathematical Linguistics and Automatic Translation*, Harvard University Computation Laboratory Report, NSF-10, pp. 2.31-2.64.

GINSBURG, S. (1966). *Mathematical Theory of Context-free Languages*, New York: McGraw Hill Book Co.

METCALFE, H. H. (1964). A parameterised compiler based on mechanical linguistics, *Annual Review in Automatic Programming*, Vol. 4, pp. 125-165. (R. Goodman, Editor), Pergamon Press.

REEVES, C. M. (1967). Description of a syntax-directed translator. *The Computer Journal*, Vol. 10, pp. 244-255.

Book reviews

Definition of Programming Languages by Interpreting Automata, by Alexander Ollongren, 1975; 290 pages. (Academic Press, £9·00)

The major part of this work is devoted to the semantic definition of programming languages, and is based very closely on the work of IBM at Vienna. There is, in addition, an introductory chapter on mathematical foundations, and two chapters on syntactic aspects. These two chapters give a brief survey of formal languages, finite automata and parsing. The intended readership includes teachers and students of computer science, mathematics graduate students and senior programmers. The book is claimed 'to adopt a gentle tutorial style throughout,' and I read it as someone who wishes to learn rather than as an expert.

From this standpoint, I found the book very hard to read. The main problem is with the use of English, which is presumably not the author's native language. At best, the sentence structure is awkward and, at worst, ambiguous or tending towards incomprehensibility. Sometimes, moreover, there is a looseness of definition that appears to go deeper than a maladroit use of English. The following extract from the book, which is taken from the very start of Chapter I, may serve as an illustration: 'A basic notion in all branches of logic is an *assertion* which is either true or false. An assertion can take the form of a sequence of words in a natural language; it is then a sentence, which necessarily must be declarative and non-ambiguous. An assertion can take the form of a sequence of mathematical symbols (standing for variables, relational operators and so on), for instance if the assertion is a theorem. Assertions which are either true or false are called statements or *propositions*, independent of the form in which they are presented. All previous sentences are propositions'. (To give the author due credit, some examples follow this explanation, and these help clarify the concept.)

Another problem for the non-expert reader is that the book is uneven in what it assumes he knows. For example, grammars are explained from scratch, but it is assumed that the reader knows what a 'rooted acyclic directed graph' is.

For the expert reader, the book may have some merit. Some of the material towards the end of the book may be of particular interest. Topics covered include: (*a*) the separation of control information from other data used by interpreting automata; (*b*) the relationship between interpreting automata and real compilers; and (*c*) parallelism. At the end of each chapter is an extensive survey of the relevant literature, many of the citations being to IBM Vienna work.

P. J. BROWN (Boulder, Colorado)

Fortran Codes for Mathematical Programming, by A. Land and S. Powell, 1973; 249 pages. (Wiley, £4.75)

The authors have realised that there are a considerable number of people, both inside and outside the universities, who do not require the standard mathematical programming algorithms but want to modify these algorithms for their own specific purposes. As most users are unable to get inside commercially available mathematical programming packages, a large variety of generally inefficient, 'home-grown' programs have been developed and also a lot of untested new algorithms have been published. The rationale for this book is therefore simple; namely to provide detailed descriptions and listings of robust, well-tested computer programs of the linear, quadratic and discrete programming algorithms.

The first chapter briefly describes the mathematical background to each algorithm. The next five chapters detail the listings of the individual algorithms, i.e. a revised simplex algorithm with upper bounding for linear programs, Beale's algorithm for quadratic programming, Gomory's method of integer forms for integer programs, Land and Doig's algorithm for mixed integer programs and finally, a parametric linear programming algorithm. These chapters also contain complete descriptions of the inter-relationships between the computer programs, the exact function of each subroutine together with lists of each routine called and the necessary COMMON variables. The final chapter is all-important as it deals with problems that may be encountered in either moving the routine onto a new computer (the programs were developed on a CDC 6600) or altering the size of problems that may be solved. This chapter contains sections on suggested overlay structures, setting of the various tolerances and the places that they appear through the programs, storage requirements for each array as a function of the size of the problem, and some indication of the algorithms' performances on some test problems. The four appendices contain indexes of the routines and the COMMON variables, a description of the data input format and some further test problems.

The impression that the reader must immediately get from this book is that the authors have taken enormous pain to smooth the path of potential users. Reliable and adaptable are the keywords to describe the programs and the documentation is superb. The book should be on the shelves of all mathematical programmers to dissuade them from ever having to start programming an algorithm on a computer from scratch again.

R. B. FLAVELL (London)