ETH zürich

ETH Library

Double syntax oriented processing

Journal Article

Author(s):
Alpiar, R.

Publication date:
1971

Permanent link:
https://doi.org/10.3929/ethz-b-000423061

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
Computer journal 14(1), https://doi.org/10.1093/comjnl/14.1.25

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Double syntax oriented processing

R. Alpiar

Computer Centre of the Swiss Federal Institute of Technology, Zürich, Switzerland, and The Swiss Federal Institute for Reactor Research, Würenlingen, Switzerland

A syntax-oriented processing scheme is described in which the syntaxes of both the input and output sentences are given equal importance. Both syntaxes are defined by BN-type production rules, and are connected by giving certain classes identical names in the two languages. This results in a fully symmetric scheme, input can be recovered from output by merely exchanging the two blocks of production rules. 'Simple' and 'Ramified' processing are distinguished—the latter being an extension which allows context sensitive features to be handled. It is suggested that this scheme is a useful tool enabling the non-specialist to effect routine modifications to sentences or whole programmes automatically. The possibilities of the method are illustrated by numerous examples, and reference is made to a field-tested implementation.

(Received January 1970)

1. Introduction and preliminary matters

The aim of this paper is to introduce a processing scheme, with particular application to the processing of programmes written in formal languages. We use the word 'processing', rather than 'translation' deliberately. For under 'processing' we imply not only the construction of a new programme (as in compilation, for instance), but also the construction of any other information relating to the input matter (such as diagnostics, lists of variable names, and so on). Particular applications of the present scheme include:

1. The modification of entire programmes in conformity with a local 'dialect' of the original programming language.
2. The execution of programmes in a temporarily altered form, either as a debugging aid, or to test special features.
3. Translation of programmes written in an extended symbolic language, back into the original symbolic language.

Emphasis is placed on applications of a provisional nature. There are numerous occasions when only a single execution of a temporarily altered programme is necessary, in which the alterations are slight, but occur repeatedly in the programme. A single execution of the modified programme may be all that is required in order to compare its speed and accuracy on a machine of different type and word length. Under such circumstances it becomes economical to employ an automatic processor even when the latter is somewhat inefficient by reason of its generality. It will be argued that the presently described scheme, at least in its simpler form, could be a convenient instrument in the hands of an unsophisticated programmer—enabling him to perform automatically routine modification which would otherwise have to be performed 'by hand'.

1.1. General principles

The scheme which we shall describe will be seen to be an obvious extension of principles which are well known, and which fall under the heading of 'Syntax-Oriented Translation'. In such schemes the input, consisting of a set of sentences, is analysed according to a set of language production rules. The resulting parsing tree is then utilised to generate the desired output sentences, given certain rules of procedure which are appended to the production rules of the input language. The relationship between input and output is asymmetric—since the output (in contrast to the input) is not generally treated as belonging to a formal language specifiable by production rules.

If we do regard input and output in a symmetric way, the source and target languages would each be specified by its own autonomous set of production rules. Input would be *analysed* using the source productions, and this analysis, with the help of the target productions, would be utilised to *synthesise* the desired output. Source and target production rules can be considered as independent entities: in fact, most of the examples in this paper will work backwards—that is output sentences can be processed back into their original input sentences by merely interchanging the blocks of source and target production rules. Further, on adding more blocks of production rules, chain processing through a succession of intermediate languages can be accomplished.

It may be noticed that languages will here be regarded as being sufficiently defined by their syntax production rules. Since we are concerned merely with the processing of strings, the notion of semantics becomes superfluous. If we insist on using the word, we may say that the semantics of the source language is defined by the target language productions, and the mapping between these production blocks. Semantics is always relative, an explanation of meaning in terms of symbols which are assumed to be well understood. Without going outside our present scheme, we might easily construct a target language which printed out natural language 'semantic' interpretations of source language sentences.

The plan of this paper is first to clarify the notation and terminology which will be used here. Next, in Section 2, the philosophy of 'simple processing' is explained, and illustrated by a number of graded examples. Section 3 deals with 'ramified processing', an extension which enables one to handle context sensitive features. Section 4 discusses the setting up of a processor which is easy to use in practice, mentioning a recently-developed programme, named MAMMAL (A *M*odest *Ma*nipulation *L*anguage).

1.2. Terminology and notation

In the following discussion a number of words and signs will be used in a specialised sense. Without wishing to set down a complete terminology of formal languages, it is nevertheless helpful to underline the meanings of certain frequently-used expressions.

A *process* is understood to mean a transformation of input sentences into output sentences, utilising the presently described method. We shall later distinguish between *simple* and *ramified* processing. A production rule consists of two parts: the left hand or *defined* part, and the right hand or *definition* part. The defined part contains a single, non-null identifier: the definition part may be empty, or contain one or more identifiers separated by logical delimiters. An *identifier* is a string of alphanumeric characters, possibly empty. There are two *logical delimiters*, they are the logical 'and' and 'or' signs, neither of which are alphanumeric. A *language* consists of a set of production rules, headed by a unique language identifier. *Class names* are identifiers which appear in the defined part of at least one production rule of the language. If the class name also appears in the definition part of the same production rule, then that class name is *recursive*. An *initial* class name is one which never appears in the definition part of a production, unless it also appears as the defined class name of that production. Thus initial class names may be recursive, though this is unusual. *Terminals* are identifiers which are neither class names, nor language identifiers, they stand either for themselves, or for special sets of characters, called *terminal classes*.

If the definition of a class name contains n logical 'or' delimiters, then that class name is said to have $n + 1$ *alternative definitions*, *alternatives*, or *choices*. If n is zero then that class name is said to be *uniquely defined*, otherwise, it is *multiply defined*.

The signs which will be used in this paper are set out in **Table 1**, and the actual fount characters appearing there constitute the *publication language*. Different sets of consistently defined characters may be used elsewhere, as for instance, as punched card input for a particular implementation of processing. Table 1 will be extended by the use of subscripts when we come to discuss ramified processing. As a reading aid, language production blocks will be indented in the text, headed by an offset language identifier symbol. Blank spaces are employed merely for the sake of appearance, and have no logical significance.

Our notation differs slightly from standard BNF, and the author hopes that readers accustomed to the latter will bear with him. The reason for the change will become clear in the discussion of Ramified Processing in Section 3. Although we could have stuck strictly to BNF in earlier sections, it was felt better to adopt a consistent notation throughout the paper. In our notation fewer supplementary signs are needed. For example:

```
Standard BNF:
<CLASS> :: = <TYPE1> . <END> | <TYPE2> . <END>
                                      (12 signs)
Modified BNF:
<CLASS> TYPE1 ∧.∧ END | TYPE2 ∧.∧ END
                                   (only 7 signs)
```

There is however a drawback. It is not immediately apparent whether a string in the definition part of a rule is a class name or a terminal string. This can only be determined by seeing whether the string also occurs in the defined part of a rule of that language.

The following example serves to illustrate the terminology and conventions which we have adopted. Production rules are set up for a language consisting of non-empty sets of unsigned numbers, separated by commas. The numbers may be any mixture of integers, or fixed and floating point forms.

Table 1
Publication language conventions

SEMANTIC MEANING	PUBLICATION LANGUAGE
Language identifiers:	
source languages	*S* *Sn* } 'n' being an
target languages	*T* *Tn* } integer
Start of the defined part of a production rule	<
Start of the definition part of a production rule	>
Logical 'and'	∧
Logical 'or'	|
Class names	any upper case alphanumeric string
Terminal classes:	
any single letter	l
any single digit	i
any single letter or digit	li
any string of letters	sl
any string of digits	si
any alphanumeric string	sli
any single character	a
any string of characters	sa
Terminal identifiers	Any strings of characters, including the null string, but excluding the special signs < > ∧ |, and lower case letters

Example 1

```
*S* <SET>               NUMBER | NUMBER ∧ , ∧ SET
    <NUMBER>            INTEGER | FIXED POINT |
                        FLOATING POINT
    <INTEGER>           si
    <FIXED POINT>       INTEGER ∧ . | . ∧ INTEGER |
                        INTEGER ∧ . ∧ INTEGER
    <FLOATING POINT>    FIXED POINT ∧ EXPONENT
    <EXPONENT>          E ∧ POSSIBLE SIGN ∧ INTEGER
    <POSSIBLE SIGN>      | + | −
```

In this language *S*, SET, NUMBER, INTEGER, FIXED POINT, FLOATING POINT, EXPONENT, and POSSIBLE SIGN are all class names. The strings , . + − E are five different terminals (in this example, exceptionally, there are no terminals longer than one character). The terminal class si has been used to define the class name INTEGER. Strictly speaking terminal classes are a superfluous concept, for we could have defined

```
<INTEGER> 0 | 1 | 2 | 3 | 4 .......
```

However, the use of terminal classes makes production rules more easily read. Moreover, the parsing algorithm may be so constructed that membership of a terminal class is more efficiently established than membership of the same class name expanded in extenso. The class name SET is recursively defined, and is an initial class. INTEGER, FLOATING POINT and EXPONENT are uniquely defined, but all other class names are multiply defined.

Clearly the actual choice of class names is a dummy choice, consistently altering them does not affect the language being defined. This fact will be made use of later on.

1.3. Parsing, parsing trees and their representation

Given a language defined by a set of production rules, any sentence belonging to it can be analysed, and its analysis

can be represented by a parsing tree. This parsing tree provides complete information as to exactly how the language production rules can generate the given sentence. We shall conceive of a parsing tree as consisting of *nodes* and *pointers*. Each node contains certain internal information, and in addition, points to four other nodes. These four other nodes can be thought of as being respectively the 'father', the 'eldest son', the 'next elder brother' and the 'next younger brother' of an all-male genealogical tree. Some of these pointers may point at a 'zero' or 'dummy' node—as, for instance, the next younger brother pointer of a youngest son. Only one node, the *vertex* will have a zero father pointer.

The internal information contained within a node consists of four items: a class name, an integer representing one of the possible alternative definitions of that class name, the identifier of one of the components of that alternative definition, finally the section of the input sentence covered by that component. The vertex node however contains only two items of information, being the language identifier, and the entire input sentence.

In this paper parsing trees are represented with the aid of both a diagram and also a tabulation. The diagram names the various nodes, *n*0, *n*1, *n*2, . . . and illustrates their genealogical relationships graphically. The tabulation sets out the internal information contained within each node, in four columns.

As an illustration of this convention, we will set up a parsing diagram and a parsing tabulation of the parsing tree of the following sentence

17, 3.89, 7.E-3

parsed according to the production rules of Example 1.

We shall often wish to traverse a parsing tree—that is—to visit each of its nodes once and only once in some logical manner. There are several possible ways of doing this. The particular order which we have chosen to enumerate the nodes of **Fig. 1** is called 'preorder traversal'. Briefly stated, down pointers take priority over right pointers, and the latter take priority over up pointers. We shall adhere to preorder traversal throughout this paper.

Our attention will be mainly focused on the structure of the parsing tree: the mechanism of the parsing algorithm which creates it is here of secondary importance. However, the mechanism of parsing will touch our argument at three points, and it may be as well to mention these now.

First comes the question of the speed and efficiency of the parsing algorithm. We have already seen how the concept of terminal classes was introduced with this factor in mind.

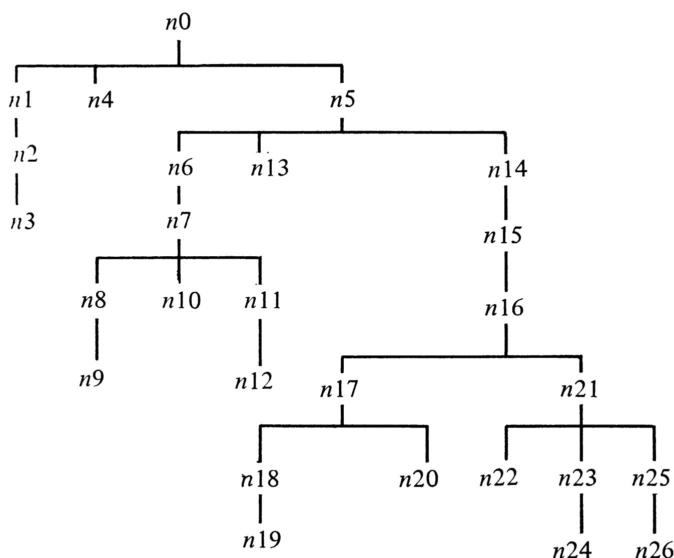


Fig. 1. Diagram of the parsing tree of a simple sentence.

Secondly, the matter of parsing ambiguity. It may be possible to generate a given sentence from the same set of production rules in two different ways. When this is so, we adopt the convention that the first possible valid parsing is the one which is required of the parsing algorithm. By 'first possible parsing' we mean that valid parsing obtained by taking the first possible alternative definition of each multiply defined class name in the parsing tree traversed in preorder. This priority convention is rather important in the present scheme. Suppose that we are interested in strings of a certain type appearing in the input sentence: we set up the following production rules

<STRING> CERTAIN TYPE | ANY OTHER
<ANY OTHER> sa
<CERTAIN TYPE> etc. etc.

The priority convention ensures that a CERTAIN TYPE stringlet is always so parsed, even though it could always be alternatively parsed as ANY OTHER. One application is when one wishes to treat certain types of strings in a special way, and ignore all other types of strings. Again, one may wish to take specific action on encountering ungrammatical strings: this would be effected by adding an alternative definition ANY OTHER as the last alternative definition of the initial class name of the language. Appropriate action would be taken if the parsing algorithm were forced to choose that alternative. Thus the order of alternatives in the definition part of production rules is significant, although the ordering of the production rules themselves is not. It is assumed that the parsing algorithm will consider each alternative definition of a class name in the order in which these alternatives are written in the language.

Lastly, the problem of 'left recursion'. Consider the following production rules:

S <STRING> INTEGER | ANY OTHER
<INTEGER> INTEGER $\wedge$ DIGIT | DIGIT
<DIGIT> i
<ANY OTHER> sa

Any input string would cause an unsophisticated parser to hang up—continually searching for confirmation of the class name INTEGER. We therefore amend the definition of INTEGER:

<INTEGER> DIGIT | INTEGER $\wedge$ DIGIT

The production rules will now parse input strings consisting of digits successfully. However ANY OTHER strings will still cause a hang up, which can be cured by a further amendment:

<INTEGER> DIGIT | DIGIT $\wedge$ INTEGER

and this works perfectly for all possible input strings. However there are instances (and specific examples will be provided below) in which only the original or first amendment of the production rules will produce a parsing tree of the 'shape' desired for a particular translating application. Of course, parsing algorithms can be constructed which accept left recursive production rules—but these may carry heavy penalties in computation time. It is therefore interesting to note, as will be demonstrated further on, that the left recursion problem can be neatly sidestepped within the framework of the proposed processing scheme.

2. Simple processing

2.1. Basic principles

The production rules of a language are rules which enable us to produce all valid strings of that language. Different strings are the result of making different choices from the possible alternative definitions of multiply defined class names. Apart from the measure of indeterminacy implied

in terminal classes, each valid string can be considered as characterised by a certain set of choices of the multiply defined class names.

Let us emphasise this point by carrying out a little experiment. Suppose we have a 'defective parsing tree', that is, a parsing tree of which the diagram, and also the 3rd and 4th items in the tabulation have been lost. How would one then set about recovering the missing information, and finally the entire original input string? One would scan each node of the tabulation in turn. Since these are tabulated in preorder, the position of each node in a reconstructed parsing diagram can be determined. Each node which represents a multiply defined class name can now have its component item filled in—referring to the production rules, the correct alternative choice for that class name, and the parsing diagram under reconstruction. Once the component item is filled in, the immediate progeny of that node can be identified. Each son node must fall under one of four types. Either it is a terminal, in which case the production rule enables us to fill in the 4th item of the node immediately. Or it is a terminal class, and in this case we are permitted to enter any stringlet belonging to that terminal class. Or again, it is a uniquely defined class name—and then we can add a set of daughter nodes as specified in the production rules. Or lastly it is a multiply defined class name—and in this case we refer to the second item of the tabulation to tell us which of the possible alternative definitions is to be chosen.

Perhaps a practical example will help to make this point clearer. We take the production rules of Example 1, and the first three columns only of **Table 2**. Our aim is to recover the input string from this information. Readers wishing to take part in this exercise should cover over the last two columns of Table 2.

Running down the table, node *n*0 merely tells us that we are at the vertex of a parsing tree. Node *n*1 says that the next nodes of the parsing tree are given by the production rule for the class name SET, taking the second alternative. We now have the beginning of a parsing tree, see **Fig. 2**(*a*). Further reference to the production rules indicates that the second component is a terminal string. This branch of the parsing tree cannot therefore grow any further. For clarity we enclose such terminal nodes in a ring. Both the class names NUMBER and SET are multiply defined. These branches of the parsing tree must therefore continue to grow, and we indicate the fact by a vertical bar beneath the node, as in **Fig. 2**(*b*).

We return to Table 2 at the place where we broke off. Node *n*2 tells us to take the first alternative definition of the class name NUMBER. The production rules specify that this definition contains only one component (the class name INTEGER). Further this class name is also uniquely defined as the terminal class 'si'. Entering all this information into our parsing tree we obtain **Fig. 2**(*c*).

Returning again to Table 2, nodes *n*3, *n*4 and *n*5 contain redundant information which we have already obtained from the production rules. Node *n*6 says that the second alternative definition of the class name SET must again be selected. Using the production rules as far as they can be unambiguously followed, we get **Fig. 2**(*d*). Node *n*7 now tells us to take the second alternative definition of the class name NUMBER (that is FIXED POINT). And node *n*8 says we must take the third alternative definition of FIXED POINT. The parsing tree has grown to **Fig. 2**(*e*).

By now we have grasped the rules of the game. We persevere to the end of Table 2, and then have a completed parsing tree. All branches end with either a terminal string, or a terminal class. Conflating these together in preorder we get the following reconstruction of the input string:

si , si . si , si . E - si

Table 2

Tabulation of the parsing tree of a simple sentence

NODE	CLASS NAME	ALT.	COMPONENT	INPUT SENTENCE
*n*0	S			17,3.89,7.E-3
*n*1	SET	2	NUMBER	17
*n*2	NUMBER	1	INTEGER	17
*n*3	INTEGER	1	si	17
*n*4	SET	2	,	,
*n*5	SET	2	SET	3.89,7.E-3
*n*6	SET	2	NUMBER	3.89
*n*7	NUMBER	2	FIXED POINT	3.89
*n*8	FIXED POINT	3	INTEGER	3
*n*9	INTEGER	1	si	3
*n*10	FIXED POINT	3	.	.
*n*11	FIXED POINT	3	INTEGER	89
*n*12	INTEGER	1	si	89
*n*13	SET	2	,	,
*n*14	SET	2	SET	7.E-3
*n*15	SET	1	NUMBER	7.E-3
*n*16	NUMBER	3	FLOATING POINT	7.E-3
*n*17	FLOATING POINT	1	FIXED POINT	7.
*n*18	FIXED POINT	1	INTEGER	7
*n*19	INTEGER	1	si	7
*n*20	FIXED POINT	1	.	.
*n*21	FLOATING POINT	1	EXPONENT	E-3
*n*22	EXPONENT	1	E	E
*n*23	EXPONENT	1	POSSIBLE SIGN	-
*n*24	POSSIBLE SIGN	3	-	-
*n*25	EXPONENT	1	INTEGER	3
*n*26	INTEGER	1	si	3

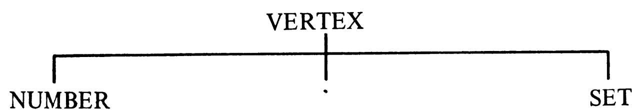


Fig. 2(a). Start of a reconstructed parsing tree.

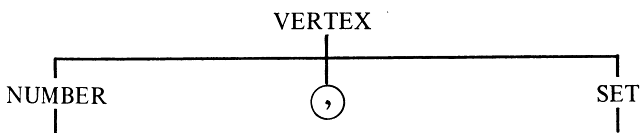


Fig. 2(b). Next stage of reconstruction.

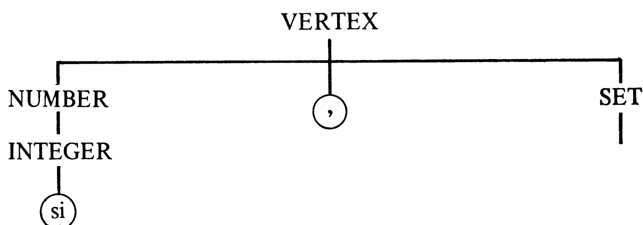


Fig. 2(c). Entering the information from node *n*2.

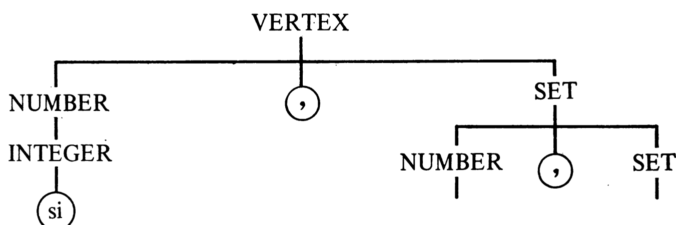


Fig 2(d). Including information from node *n*6.

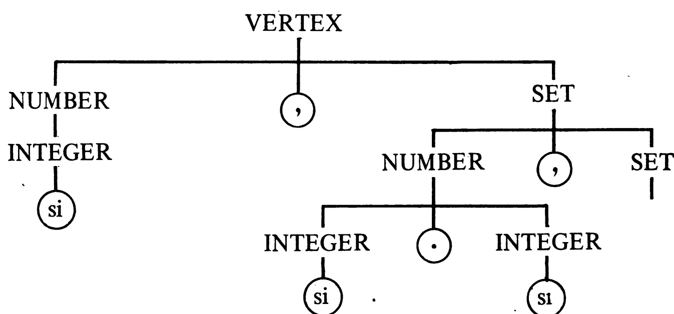


Fig. 2(e). Information from nodes *n*7 and *n*8 included.

We have thus recovered the form of the original string (it was 17, 3.89, 7. E-3). Because we defined INTEGER as the terminal class 'si', the particular digits present in the input string have been lost. Had we chosen to define

<INTEGER> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

in the production rules, then the parsing table would have been longer, but the exact contents of the input string would have been recoverable.

This apparently pointless thought-experiment has been conducted solely in order to introduce the basic principles of simple processing: for in the suggested processing scheme all we do is to use a *new set of production rules* in the above reconstruction algorithm. We now have two sets of production rules: those which were used to parse the original input string are termed the source production rules. And the new set of rules used to 'reconstruct' the original string are called the target production rules. Of course if the two sets of production rules are identical, then the reconstruction algorithm will merely reconstruct the original sentence, apart from differences in the terminal classes. Otherwise a different string will be reconstructed, which is a valid string of the target language, and may be thought of as a translation of the original string from the source to the target language.

Three further points need to be cleared up.

First, how are alternative choice numbers to be transmitted from the source to the target parsing tree under construction? The solution adopted here is that this transmission will occur only when the source and target nodes carry the same (multiply defined) class name. We have already seen that the choice of class names in a language is a dummy choice, so that no restrictions are imposed on a new language in insisting that certain of its class names should be identical with certain ones of another language. The alternative choice number of the source parsing tree is thus transferred to the target parsing tree under construction, when the two nodes contain the same class name. We must therefore insist that the number of alternative definitions of identically named class names in the two languages be the same. Once again, this imposes no real restriction upon the target language. It merely ensures that the target language is neither incomplete nor over abundant. This assignment of corresponding alternative definitions to identically named class names will be referred to as the 'principle of correspondence'. When an as yet undecided target tree node is assigned one of its alternative definitions by means of the principle of correspondence, we shall say that it has been 'determined'. The source tree node responsible for this 'determination' will be called the 'activating node'.

The second point is this: what happens if the target tree contains several undecided nodes bearing the same class name? Which of these nodes will be subject to determination by an identically named source tree activating node? Each time an activating node in the source tree determines an undecided node of the target tree, the latter node is 'filled out'—by this we mean that the target productions are invoked to create progeny for the newly determined target tree node. These progeny in turn breed further progeny, so that the newly determined target node becomes the vertex of a subtree. The leaves of this subtree are either terminals, terminal classes, or undecided nodes, that is nodes bearing a multiply defined class name. The latter undecided nodes are strung together in a 'chain of undecided nodes' and the source tree activating node is made to point at the first member of this chain.

The rule which we shall always apply is that an undecided target tree node can only be determined by activating nodes which are correctly positioned in the source tree. This correct positioning insists that the activating node must be a member of a subtree whose vertex is the header of an undecided chain of nodes to which the candidate for determination belongs. Intuitively, we may think of this as insisting that the 'meaning' of any node can only be found in the contents of its subtree of nodes, and never in any collateral subtree: loosely speaking, this means that our languages are context-free. This rule, the 'principle of paternity' ensures that recursively defined class names are determined by the correctly nested activating nodes of the source tree. Example 6, below, illustrates this point nicely. But a further question occurs. What if two nodes belonging to the same undecided chain of nodes carry the same class name? Our convention will be that only the first of these will be determined by an activating source tree node. The fact that one node and only one is determined by each activating node is again associated with the idea of context

freedom. In Section 3, we shall see that relaxing this rule, and linking alternative choices together, results in a degree of context sensitive behaviour. The fact that the only node of the undecided chain to be determined is the first suitable candidate in that chain is of no great significance, since it can be bypassed by writing out the components of production rules in a different order.

The third and last point is fairly trivial, and concerns the treatment of terminal classes in the target parsing tree under construction. The convention is that the terminal class stringlet will be transmitted unaltered from the terminal class source tree node to the target tree node bearing the same terminal class name, and in the corresponding position of the target language tree.

2.2. Graded examples

One gets a better feel for what is happening by following a number of examples, than by trying to assimilate many pages of abstract explanation. Some of the following cases have no particular application, other than to demonstrate special processing techniques.

Example 2
The output sentence is to be an exact copy of the input sentence.

```
*S* <STRING>        CHARACTER | CHARACTER ∧ STRING
    <CHARACTER> a
*T* <STRING>        CHARACTER | CHARACTER ∧ STRING
    <CHARACTER> a
```

The above process analyses the source sentence character by character, and sets up the target tree, again character by character. Since the class name CHARACTER is a terminal class, stringlets (in this case single characters) in the source tree CHARACTER nodes are copied unaltered into the corresponding target tree nodes. The same copying operation would have been more simply and efficiently effected by the process

```
*S*  <STRING>  sa
*T*  <STRING>  sa
```

Example 3
The output sentence is to be the reversal of the input sentence—e.g. PQRS becomes SRQP

```
*S*  <STRING>  a | a ∧ STRING
*T*  <STRING>  a | STRING ∧ a
```

We notice that the definition of STRING in the target productions is left-recursive. But this causes no trouble in target productions, since the construction of the target parsing tree (as opposed to the parsing analysis of the source sentence) requires no 'back up' operations.

Example 4
The output sentence is to be a copy of the input sentence but with the final character missing.

```
*S*  <STRING>  a | a ∧ STRING
*T*  <STRING>    | a ∧ STRING
```

Here we notice that the first alternative definition of the target production for the class name STRING is a terminal which is the null string.

Example 5
The output sentence is to be the tail of the input sentence—that is, the input sentence with the first character missing.

```
*S*  <STRING>  a | STRING ∧ a
*T*  <STRING>    | STRING ∧ a
```

and here STRING is left-recursively defined in the source productions. In case the parsing algorithm is unable to cope, source production left-recursion can be circumvented with the following two-link process

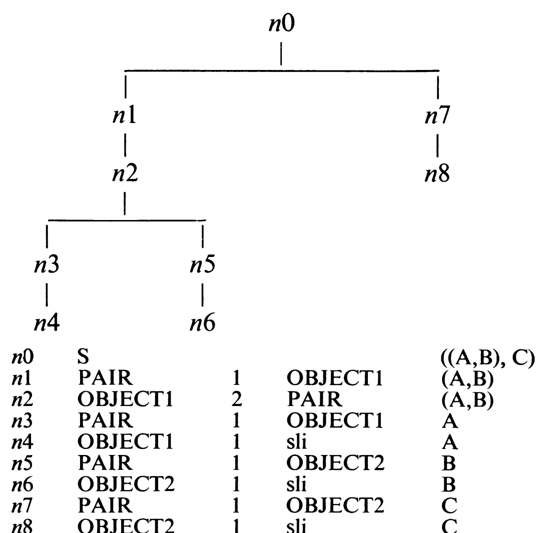
```
*S1*  <STRING>  a | a ∧ STRING
*T1*  <STRING>  a | STRING ∧ a
*S2*  <STRING>  a | a ∧ STRING
*T2*  <STRING>    | STRING ∧ a
```

The first link of the process reverses the order of the original string. This reversed string is now an input string for the second link, which chops off the first character, and reverses the remaining characters. A typical string ABCDEF becomes first FEDCBA and finally BCDEF.

Example 6
The source language consists of ordered pairs of objects (Object 1, Object 2): each object is either an alphanumeric string or itself an ordered pair. The output strings are to reverse the order of objects in each pair: for example, the input string ((A, B), (C,(D, E))) is to be processed to give (((E, D) ,C), (B,A)).

```
*S*  <PAIR>       (∧ OBJECT1 ∧ , ∧ OBJECT2 ∧)
     <OBJECT1>    sli | PAIR
     <OBJECT2>    sli | PAIR
*T*  <PAIR>       (∧ OBJECT2 ∧ , ∧ OBJECT1 ∧)
     <OBJECT1>    sli | PAIR
     <OBJECT2>    sli | PAIR
```

This example illustrates the importance of the principle of paternity. Consider the processing of an input sentence ((A, B),C). The source language tree, stripped of the irrelevant brackets and commas, can be illustrated by the following diagram and tabulation:



*n*0	S			((A,B), C)
*n*1	PAIR	1	OBJECT1	(A,B)
*n*2	OBJECT1	2	PAIR	(A,B)
*n*3	PAIR	1	OBJECT1	A
*n*4	OBJECT1	1	sli	A
*n*5	PAIR	1	OBJECT2	B
*n*6	OBJECT2	1	sli	B
*n*7	PAIR	1	OBJECT2	C
*n*8	OBJECT2	1	sli	C

The initial node of the parsing tree for the target language is filled out, producing an undecided chain of two nodes with class names OBJECT2 and OBJECT1. As node *n*2 of the source tree is scanned, OBJECT1 of the target tree becomes determined: its second alternative definition is selected, thus producing two further undecided nodes, OBJECT2 and OBJECT1. The contents of the latter is then determined by node *n*4. When we get to node *n*6, the target tree contains two undecided nodes with the same class name OBJECT2. The principle of paternity ensures that the correct target tree node is determined—that is the one which belongs to an undecided chain headed by the activating node *n*2, and not that belonging to the chain headed by *n*8.

Incidentally this operation could have been performed merely by interchanging the brackets (and) in a reversed string:

```
*S*  <STRING>     CHARACTER | CHARACTER ∧ STRING
     <CHARACTER>  sli | DELIMITER
     <DELIMITER>  ( | ) | ,
*T*  <STRING>     CHARACTER | STRING ∧ CHARACTER
     <CHARACTER>  sli | DELIMITER
     <DELIMITER>  ) | ( |
```

However, if the processing requires the reorganisation of triplets or larger multiplets, it cannot be so neatly debunked.

Example 7
Source sentences consist of function statements of the form
'function name' ('parameters separated by commas').
Four function names are available, the functions JUP and JDOWN are each called with one parameter only, whereas the functions IUP and IDOWN must be called with two parameters. The parameters may be either alphanumeric identifiers, or themselves function calls.

In the target language only a single function name NODE is present, but is always called with four parameters, with the following meaning:

Parameter 1 : identical with the first parameter of input function calls
Parameter 2 : value 1 for functions JUP or IUP, value 2 for functions JDOWN or IDOWN
Parameter 3 : value 0 for functions JUP or JDOWN value 1 for functions IUP or IDOWN
Parameter 4 : same as second parameter of functions IUP or IDOWN, set zero for functions JUP or JDOWN

Thus the input sentence IUP(JDOWN(P), JUP(JUP (XYZ)) would become on output

NODE(NODE(P,2,0,0),1,1,NODE(NODE(XYZ,1,0,0),1,0,0))

The following process will perform the required transformation:

```
*S* <SENTENCE> OP1 ∧ (∧ EXP1 ∧) |
               OP2 ∧ (∧ EXP1 ∧ , ∧ EXP2 ∧)
    <OP1>      JUP | JDOWN
    <OP2>      IUP | IDOWN
    <EXP1>     sli | SENTENCE
    <EXP2>     sli | SENTENCE
*T* <SENTENCE> NODE(∧ EXP1 ∧ , ∧ OP1 ∧ ,0,0) |
               NODE(∧ EXP1 ∧ , ∧ OP2 ∧ ,1, ∧ EXP2 ∧)
    <OP1>      1 | 2
    <OP2>      1 | 2
    <EXP1>     sli | SENTENCE
    <EXP2>     sli | SENTENCE
```

Incidentally, this process (as well as those of preceding examples) is reversible. On exchanging the two blocks of production rules, former output will be processed back again to the original input.

Example 8
The input statements are FORTRAN instructions, possibly containing calls for library functions of the form SINF ( ), COSF( ), EXPF( ), etc. These calls may occur recursively and repeatedly in the same instruction. The output statements are to consist of the same instructions modified by removing the final 'F' from all such function calls. The trick is so to set up the source language productions that function calls are selectively recognised—we must beware of recognising other identifiers, for example PQCOSF(. . . as library calls.

```
*S* <INSTRUCTION> ANY ∧ SPECIAL ∧ INSTRUCTION |
                  ANY
    <ANY>         | a ∧ ANY
    <SPECIAL>     DELIMITER ∧ LIBRARY ∧ F(
    <DELIMITER>   + | − | * | / | = | ( | ,
    <LIBRARY>     SIN | COS | EXP | LOG | etc.
*T* <INSTRUCTION> ANY ∧ SPECIAL ∧ INSTRUCTION |
                  ANY
    <ANY>         | a ∧ ANY
    <SPECIAL>     DELIMITER ∧ LIBRARY ∧ (
    <DELIMITER>   + | − | * | / | = | ( | ,
    <LIBRARY>     SIN | COS | EXP | LOG | etc.
```

This example illustrates an application of the priority convention. INSTRUCTION is non-uniquely deconcatenable, since any string could be parsed under the second alternative definition, ANY. The priority convention ensures that the instruction is first tested for the presence of a SPECIAL sub-string, which is defined as one containing function calls.

But why has the class name ANY been defined as | a ∧ ANY, instead of simply <ANY> sa? For two reasons. First, the parser might be so constructed that sa was not treated as a class name with unlimited alternative definitions of 0 characters, 1 character, 2 characters . . . . Thus the whole instruction would be parsed as ANY, leaving no characters over to be parsed under SPECIAL ∧ INSTRUCTION: thus the parser would always pass to the second alternative definition of INSTRUCTION. Secondly: even if the parser algorithm recognised sa as a multiply defined class name, it would probably 'work backwards', that is, as first alternative associate all characters of the string with ANY, as second alternative, all characters but the last with ANY, and so on, until enough characters were left at the end of the string to be recognised as SPECIAL ∧ INSTRUCTION. Exhaustive as this is, it is only a tiny inner loop of a still wider search, since, at this stage, only the last occurrence of SPECIAL in the input string has been recognised. The above definition of ANY ensures that the parser always searches forwards, and minimises the length of the input string falling under this class name.

Example 9
Certain dialects of FORTRAN, and certain compilers, require that the titles of all programmes, subroutines and functions commence with the string FORTRAN VI, followed by the normal name and possible formal parameters. The following process will convert title cards to this format, and copy all other strings unaltered.

```
*S* <STATEMENT> TITLE | sa
    <TITLE>     INITIAL ∧ sa
    <INITIAL>   PROGRAM | SUBROUTINE | FUNCTION
*T* <STATEMENT> TITLE | sa
    <TITLE>     FORTRAN VI ∧ INITIAL ∧ sa
    <INITIAL>   PROGRAM | SUBROUTINE | FUNCTION
```

Example 10
This set of examples is included to demonstrate the versatile nature of simple processing. Possible applications include the extension of source language syntax to include matrix expressions, the manipulation of algebraic expressions, or the systematic modification of source statements to take advantage of the idiosyncracies of a given compiler.

The problem is to set up a process for the conversion of simple assignment statements into simple machine code. We will first define what we mean by these two phrases. By simple machine code, we mean machine code limited to the following set of instructions, all but the last of which are followed by an 'address', consisting of an alphanumeric upper case identifier, which may be identical to a corresponding identifier appearing in the input assignment statement:

CLA	clear accumulator and add contents of address
SUB	subtract contents of address from accumulator
ADD	add contents of address to accumulator
MUL	multiply accumulator by contents of address
DIV	divide accumulator by contents of address
POW	raise accumulator to power of contents of address
STO	store contents of accumulator in address
CHS	change sign of accumulator

The address following an instruction may be an erasible storage location, denoted by a lower case letter, or string, *x*, *y*, *abc*. If necessary, a pushdown stack is also available, and is addressed simply as 'stack'.

The questions of where the input strings come from, and what is to be done with the corresponding output strings, have been ignored up to now. These considerations, in so far as they are relevant to the present study, will be

touched on in Section 4. But the present example serves as a convenient introduction to the policy which will be later advocated. Let us assume that the machine code output is to be punched on cards, one instruction per card, in certain columns. The target productions must generate output strings of instructions with special markers, which indicate where each instruction and each address part starts. It will then be the duty of a specially written dispatcher routine to recognise these markers, and to take appropriate action. In the same way a specially written dispatcher routine would be written to recognise which fields of an input card constitute an input string for the source language, to deal with the matter of continuation cards, to reject comment cards, and so on.

In the present set of examples, we shall make the target productions insert a − sign between the function and address parts of an instruction, and place a period at the end of each instruction.

By a 'simple assignment statement' we mean here a statement of the form 'identifier' = 'expression', where 'identifier' is an alphanumeric string, and 'expression' consists only of unsubscripted identifiers together with the arithmetic operators + − * / ** ( ), interpreted normally. 'Expression' may contain neither subscripted variables, nor pure numbers.

The problem will be handled one step at a time, first considering the simplest of assignment statements, in which only the operators + and − may appear, till finally a scheme for fully bracketed expressions is set up.

Example 10(*a*)
The very simplest case assumes that the expressions contain the operators + and − only.

```
*S*  <ASSIGNMENT>   sli ∧ = ∧ EXPRESSION
     <EXPRESSION>   POSS UNARY ∧ sli ∧ POSS CONT
     <POSS UNARY>    | + | −
     <POSS CONT>     | PM ∧ sli ∧ POSS CONT
     <PM>           + | −
*T*  <ASSIGNMENT>   EXPRESSION ∧ STO- ∧ sli ∧ .
     <EXPRESSION>   CLA- ∧ sli ∧. ∧ POSS UNARY ∧
                    POSS CONT
     <POSS UNARY>    | | CHS-
     <POSS CONT>     | PM ∧ sli ∧ . ∧ POSS CONT
     <PM>           ADD- | SUB-
```

Under this process, an input assignment statement, say A = −B + C − D, will result in the following output string.

CLA-B. CHS. ADD-C. SUB-D. STO-A.

Example 10(*b*)
When the input expressions are allowed to contain all four operators +, −, *, /, two alternative policies may be followed. Either, regardless of the efficiency of the compiled output, we may insist on the operations being performed in their input string order: alternatively, laws of commutation and association may be invoked to produce compacter coding. The first process follows the former alternative:

```
*S*  <ASSIGNMENT>   sli ∧ = ∧ EXPRESSION
     <EXPRESSION>   POSS UNARY ∧ CLAUSE ∧ POSS
                    CONT
     <POSS CONT>     | POSS CONT ∧ PM ∧ CLAUSE
     <POSS UNARY>    | + | −
     <PM>           + | −
     <CLAUSE>       sli | CLAUSE ∧ MD ∧ sli
     <MD>           * | /
*T*  <ASSIGNMENT>   EXPRESSION ∧ STO- ∧ sli ∧ .
     <EXPRESSION>   CLAUSE ∧ POSS UNARY ∧
                    POSS CONT
     <POSS CONT>     | POSS CONT ∧ STO-y. ∧ CLAUSE ∧
                    STO-x. CLA-y. ∧ PM ∧ x.
     <POSS UNARY>    | | CHS.
     <PM>           ADD- | SUB-
     <CLAUSE>       CLA- ∧ sli ∧ . | CLAUSE ∧ MD ∧
                    sli ∧ .
     <MD>           MUL- | DIV-
```

A typical input string reading, say, Z = A + B*C − D/E, will result in the following output:

CLA-A. STO-y. CLA-B. MUL-C. STO-x. CLA-Y. ADD-x. STO-y. CLA-D. DIV-E. STO-x. CLA-y. SUB-x. STO-Z.

The above source language production rules are viciously left-recursive. Consider the parsing of a simple string Z = A + B + C. In order to identify A + B + C as an EXPRESSION, A must first be identified as a CLAUSE: then +B + C has to be identified as a POSS CONT. But before this can happen, the parser must *fail* in an attempt to identify +B + C as a CLAUSE. Unless the parser is quite sophisticated it will hang. The author's parser, after some modification, was able to control the situation, but at the expense of a frightening expenditure of machine time. By applying the lesson learnt in Example 5, left-recursive source language productions can be circumvented entirely, using the following 2-link process:

```
*S1*  <STRING>       sli | sli ∧ OP ∧ STRING
      <OP>           + | − | * | / | = | =+ | =−
*T1*  <STRING>       sli | STRING ∧ OP ∧ sli
      <OP>           + | − | * | / | = | = | −=
*S2*  <ASSIGNMENT>   EXPRESSION ∧ = ∧ sli
      <EXPRESSION>   POSS CONT ∧ CLAUSE ∧ POSS UN
      <POSS CONT>     | CLAUSE ∧ PM ∧ POSS CONT
      <POSS UN>       | −
      <PM>           + | −
      <CLAUSE>       sli | sli ∧ MD ∧ CLAUSE
      <MD>           * | /
*T2*  <ASSIGNMENT>   EXPRESSION ∧ STO- ∧ sli ∧ .
      <EXPRESSION>   POSS CONT ∧ CLAUSE ∧ POSS UN
      <POSS CONT>     | POSS CONT ∧ STO-y. ∧ CLAUSE
                     ∧STO-x. CLA-y. ∧ PM ∧ x.
      <POSS UN>       | CHS.
      <PM>           ADD- | SUB-
      <CLAUSE>       CLA- ∧ sli ∧ . | CLAUSE ∧ MD ∧
                     sli ∧ .
      <MD>           MUL- | DIV-
```

In link 1, language S1, we have distinguished between identifiers (sli) and operators (OP), in order to avoid the reversal of component characters of identifiers, as would have occurred without this distinction. Also, the monary + sign has been eliminated in link 1, leaving only a possible unary − to be dealt with in link 2.

Some improvements in output coding can be expected if we allow the productions to distinguish between simple clauses (e.g. PQRT) and compound ones (e.g. P*Q/R*T): as it stands a simple clause +A generates the coding STO-y. CLA-A. STO-x. CLA-y. ADD-x. !

Example 10(*c*)
However, the normal way of generating efficient coding is to re-order the clauses so that structured clauses are always handled before simple identifier clauses: thus all temporary storing in registers x and y is avoided. This could either be accomplished in a two link process, of which the first produces a rearranged output string as input for the second link—or in a one link process:

```
*S*  <ASSIGNMENT>   sli ∧ = ∧ EXPRESSION
     <EXPRESSION>   POSS UN ∧ NAME ∧ POSS NAMES |
                    POSS UN ∧ CLAUSE ∧ MIXTURE |
                    POSS UX ∧ NAME ∧ POSS NAMES ∧
                    PNC ∧ CLAUSE ∧ MIXTURE
     <POSS NAMES>    | PNN ∧ NAME ∧ POSS NAMES
     <MIXTURE>       | PNN ∧ NAME ∧ MIXTURE |
                    PNC ∧ CLAUSX ∧ MIXTURE
     <POSS UN>       | + | −
     <POSS UX>       | + | −
     <PNN>          + | −
     <PNC>          + | −
     <NAME>         sli
     <CLAUSE>       sli ∧ MD ∧ sli ∧ POSS MORE
     <POSS MORE>     | POSS MORE ∧ MD ∧ sli
     <CLAUSX>       CLAUSE
     <MD>           * | /
*T*  <ASSIGNMENT>   EXPRESSION ∧ STO- ∧ sli ∧ .
     <EXPRESSION>   CLA- ∧ NAME ∧ . ∧ POSS UN ∧
                    POSS NAMES | CLAUSE ∧ POSS UN
```

```
                  ∧ MIXTURE | CLAUSE ∧ PNC ∧
                  POSS UX ∧ NAME ∧ POSS NAMES
                  ∧ MIXTURE
<POSS NAMES>       | PNN ∧ NAME ∧ . ∧POSS NAMES
<MIXTURE>          | MIXTURE ∧ PNN ∧ NAME |
                  CLAUSX ∧ PNC ∧ MIXTURE
<POSS UN>         | | CHS-
<POSS UX>         ADD- | ADD- | SUB-
<PNN>             ADD- | SUB-
<PNC>              | CHS-
<NAME>            sli
<CLAUSE>          CLA- ∧ sli ∧ . ∧ MD ∧ sli ∧ . ∧
                  POSS MORE
<POSS MORE>        | POSS MORE ∧ MD ∧ sli ∧ .
<CLAUSX>          STO-x. ∧ CLAUSE ∧ PNC ∧ ADD-x.
<MD>              MUL- | DIV-
```

The above productions, utilising the principle of priority, first test for expressions composed entirely of simple clauses, next for mixtures commencing with a structured clause, and lastly for any other mixture of simple and structured clauses. In the third case the first occurring structured clause in the mixture is singled out for special treatment. Notice the use of repeated identical productions for the class names PNN and PNC, since a + or − occurring before a simple clause has to be treated differently from the same signs occurring before structured clauses. The class names POSS MORE is left recursive—we have already seen how this nuisance can be avoided.

If expressions are now allowed to contain the power operator **, the translation of CLAUSE becomes a little more complicated. In order to produce efficient coding, the same kind of ordering search has to take place within CLAUSE as within EXPRESSION in the above example. The details are not particularly instructive.

Example 10(*d*)

As a final example, we set up a two-link process for compiling fully bracketed expressions. Since it is not output optimised it is fairly easy to follow what is happening

```
*S1*  <SEQ>          sli | sli ∧ OPS ∧ SEQ
      <OPS>          OP | OP ∧ OPS
      <OP>           + | − | * | / | ** | ( | ) | =
*T1*  <SEQ>          sli | SEQ ∧ OPS ∧ sli
      <OPS>          OP | OPS ∧ OP
      <OP>           + | − | * | / | ** | ) | ( | =
*S2*  <ASSIGNMENT>   EXPRESSION ∧ = ∧ sli
      <EXPRESSION>   CLAUSE ∧ POSS SIGN |
                     sli ∧ PM ∧ EXPRESSION |
                     CLAUSE ∧ PM ∧ EXPRESSION
      <CLAUSE>       ATOM | sli ∧ MD ∧ ATOM |
                     ATOM ∧ MD ∧ CLAUSE
      <ATOM>         sli | (∧ EXPRESSION ∧) |
                     NAM1 ∧ ** ∧ NAM2 |
                     (∧ EXPRESSION ∧)** ∧ sli |
                     (∧ EXPRESSION ∧)**
                     (∧ EXPRESSION ∧)
      <POSS SIGN>     | + | −
      <PM>           + | −
      <MD>           * | /
      <NAM1>         sli
      <NAM2>         sli
*T2*  <ASSIGNMENT>   EXPRESSION ∧ STO- ∧ sli ∧ .
      <EXPRESSION>   CLAUSE ∧ POSS SIGN |
                     EXPRESSION ∧ PM ∧ sli |
                     EXPRESSION ∧ STO-stack. ∧
                     CLAUSE ∧ STO-x. CLA-stack. ∧
                     PM ∧ x.
      <CLAUSE>       ATOM | ATOM ∧ MD ∧ sli |
                     CLA- ∧ NAM2 ∧ POW- ∧
                     NAM1 ∧ . | EXPRESSION ∧
                     STO-stack. CLA- ∧ sli ∧
                     POW-stack. | EXPRESSION ∧
                     STO-stack. ∧ EXPRESSION ∧
                     POW-stack.
      <POSS SIGN>     | | CHS.
      <PM>           ADD- | SUB-
      <MD>           MUL- | DIV-
      <NAM1>         sli
      <NAM2>         sli
```

3. Ramified processing

3.1. Basic principles

Versatile as simple processing has shown itself to be, there are a number of operations (some of which seemingly trivial) which cannot be performed at all, or only with great inelegance, within the framework already set up. These operations will be recognised as demanding context sensitive understanding of the production rules.

To give two examples: firstly, the selective recognition of strings of duplicated characters (e.g. AA11BB22), or, the processing of a given string, say A1B2, into such a duplicated string. Although the problem is soluble within the framework of simple processing, the solution is about as inelegant as can be envisaged:

```
*S*  <STRING>      CHARACTER | CHARACTER ∧
                   STRING
     <CHARACTER>   A | B | . . . . | 0 | 1 | 2 . . . .
*T*  <STRING>      CHARACTER | CHARACTER ∧
                   STRING
     <CHARACTER>   AA | BB | . . . . | 00 | 11 | . . . .
```

As a second example, leading to an even uglier solution, consider the conversion of an arbitrary string, say A1B2, into a palindromic string A1B22B1A.

```
*S*  <STRING>   A | B | . . . . | A ∧ STRING |
                B ∧ STRING | C ∧ STRING | . . . . . .
*T*  <STRING>   AA | BB | . . . . | A ∧ STRING ∧ A |
                B ∧ STRING ∧ B | C ∧ STRING ∧ C | . . . .
```

These problems are characterised by the need to 'understand' certain class names, in terms of a previous analysis of class names. We distinguish two cases: (a) A class name depends for its interpretation upon the analysis of another class name *within the same parsing tree*, and (b) A class name depends for its interpretation on other parsing trees, that is, on the parsing trees of previously parsed sentences.

A small extension to simple processing will allow us to deal with problems falling under type (a). It is just a matter of linking alternative definitions, so that linked class names which are also brothers in the parsing tree are compelled to follow the same alternative definition. In the publication language, class names are denoted by upper case alphanumeric strings. We now introduce a new publication language sign, §, to link class names together. This sign will appear in the definition part of a production rule only. A string of the form say, ABC§P12 in the definition part of a production rule will mean that, when utilising the production rule for the class name ABC, look for an elder brother in the parsing tree under construction, with the same name ABC§P12. If one is found, then take the same alternative definition for ABC, as was taken by that elder brother. In particular, if ABC is a terminal class, use the same terminal class stringlet as that elder brother. This convention may be used in both the source, and the target productions.

With this convention in use the production rule

```
<PROD>   AB§3 ∧ PQ ∧ AB ∧ RS ∧ AB§4 ∧ AB§3 ∧ AB§4
```

would insist that the alternative definitions chosen for the first and sixth sons of PROD were the same—and also those chosen for the fifth and seventh sons. If it so happened that AB were defined by the production rule

```
<AB>   + | −
```

then there would be just four possible 'expansions' for PROD, namely

```
<PROD>   + ∧ PQ ∧ AB ∧ RS ∧ +++ |
         + ∧ PQ ∧ AB ∧ RS ∧ −+− |
         − ∧ PQ ∧ AB ∧ RS ∧ +−+ |
         − ∧ PQ ∧ AB ∧ RS ∧ −−−
```

Were, however, AB defined by the production rule

```
<AB>   i
```

then some 100 possible expansions of PROD might be generated:

```
<PROD>  0 ∧ PQ ∧ AB ∧ RS ∧ 000 |
        0 ∧ PQ ∧ AB ∧ RS ∧ 101 |
        ........................
        ........................
        3 ∧ PQ ∧ AB ∧ RS ∧ 030 | and so on
```

The two little problems of generating duplicated and palindromic strings can now be solved.

```
*S*  <STRING>       CHARACTER | CHARACTER ∧
                    STRING
     <CHARACTER>    a
*T*  <STRING>       CHARACTER§1 ∧ CHARACTER§1 |
                    CHARACTER§1 ∧ CHARACTER§1 ∧
                    STRING
     <CHARACTER>    a
*S*  <STRING>       CHARACTER | CHARACTER ∧
                    STRING
     <CHARACTER>    a
*T*  <STRING>       CHARACTER§1 ∧ CHARACTER§1 |
                    CHARACTER§1 ∧ STRING ∧
                    CHARACTER§1
     <CHARACTER>    a
```

Using linked alternatives in the source productions, we can also selectively recognise palindromic strings:

```
*S*  <STRING>         PALINDROMIC | sa
     <PALINDROMIC>    CHARACTER§X ∧ CHARACTER§X|
                      CHARACTER§Z ∧ PALINDROMIC
                      ∧ CHARACTER§Z
```

Nice though this arrangement appears to be, we shall not devote much space to its development here, since it has already been discussed in the literature, and it unfortunately cannot deal with problems of type (b).

Problems of the latter type are not far to find. Within the FORTRAN language for instance, TYPE and DIMENSION statements are glaring examples, since they create rules for the understanding of strings which will only be encountered later on.

We are looking for some mechanism whereby information can be conveyed from earlier parsing trees, to ones which will be set up later. The storage and recovery of this information might be accomplished in two ways. First by conserving certain parsing trees, at least in skeletal form, as a list or set of tables for future reference. Unfortunately this would mean very radical departure from the simple processing concepts of Section 2. The second alternative is to insist that production rules, and these alone, are to convey all necessary information. We therefore need a mechanism whereby production rules can be manipulated, modified, added to. We are led a step further towards the proposed viewpoint when we consider that the output from simple processing is an output string, to be dealt with by a specially written output dispatcher routine. If we wish to remain within the boundaries set by simple processing, or at least to avoid straying too far outside them, we must arrange for the processing output to be, on occasion, a string which is a new production rule. We are then led to distinguish between production rules whose output is a genuine output string, and those whose output is raw material of production rules. This can be done hierarchically. All production rules so far set up, whose output is sent to the output dispatcher, will be termed *zero type* production rules. We then define a type-n production rule as one whose output string is to be considered as raw material for a type-$(n-1)$ production rule.

A type-0 production rule is unable to generate an output string which is itself a type-0 production rule, since it is unable to generate the production rule delimiters < > ∧ |. Since these signs are necessary to the setting up of production rules, and will have to appear in the output strings of higher type productions, a distinction must be made between their appearance as genuine delimiters or as parts of output strings.

Further, the distinction must be hierarchical, to correspond with the hierarchy of production types set up. The most natural way of making this distinction is by appending digital suffixes to the four signs < > ∧ |. These signs will appear suffixed with o in type-0 productions: though this suffix may be omitted without causing confusion, in order to retain compatibility with simple processing. In type-1 productions these signs will appear suffixed with $_1$ when they are used as delimiters. When they are otherwise suffixed, then they are simply terminal characters as far as that production rule is concerned. The output from a type-n production becomes a type-$(n-1)$ production rule to be entered in the same language block (source or target) in which that type-n production occurred. If that block already contains a type-$(n-1)$ production bearing the same class name, then the newly created production replaces the old.

The activation of higher type productions is subject to just the same rules as for type-0 productions: that is, a production will be activated provided that it is part of the correct undecided chain and bears the same class name as the activating node. The output from a higher type production is entered as a new production as soon as it is complete, that is, when the higher type parsing tree being generated contains no more undecided nodes. A rule must also be stated to decide when a node is to set up a new parsing tree, rather than merely determine undecided nodes in an already existing tree. A new parsing tree will only be set up when the vertex is an initial class name.

In cases when a class name has more than one type of definition, and also appears in the definition part of a production, we must indicate which type of production is intended there. This can be done by subscripting the class name itself with the type-digit of the production to which reference is intended. All this is entirely consistent with simple, or type-0, processing.

We have, of necessity, introduced a subscripting extension of the simple publication language of Table 1. We now further introduce, for pure convenience rather than from necessity, another little extension. It is at times convenient to perform simple integer arithmetic during processing. In other words, given a string of integers and the usual arithmetic symbols + − * ( ), we may wish to construct a second string of digits only (possibly preceded by a unary −) with the same numerical value. This could actually be done by setting up production rules! However, these would constitute rather a clumsy mess of operations. We therefore introduce the pair of brackets { } with the convention that any characters between them are limited to the digits and the operators + − * ( ), as normally understood—and that an equivalent string of digits is to replace the whole structure. This replacement may be thought of as being performed by a specially written subroutine, which (for further convenience) will not complain on encountering such strings as, say, {− −3 + −4}, but will quietly replace such a string by −1. This act of evaluation will be considered as taking place immediately before the completed output string is handed over to the output dispatcher, or incorporated as a lower type production rule, as the case may be.

A procedural problem arises when a higher type production occurs in a source language, and concerns the backing-up which most parsers have to perform when they follow a false trail. What is to happen if one of these false trails has generated alterations in the source production rules? The parser, in backing up, may then be using a false set of production rules. There appear to be three different approaches to meet this difficulty: firstly, to construct such a clever parsing algorithm that back-up is never necessary; secondly, insist that whenever a production rule results in modifications, that the old state of the productions be stored alongside the activating node, and be restored if the parser ever backs-up past that node. The third approach is

to take this peculiarity explicitly into account in setting up production rules.

For example: suppose that the class name LANGUAGE has two alternative definitions

<LANGUAGE>   FRENCH | GERMAN

and further, that the class name FRENCH activates a type-1 production reading

<$_1$FRENCH$_1$>   .... VERB STEM $\wedge_1 \wedge \wedge_1$ PAST PART

which finally generates a new type-0 production, reading

<VERB>   AVOIR $\wedge$ PAST PART

At a later stage, the parser decides that FRENCH was the wrong alternative definition of LANGUAGE, and that it must try for GERMAN. We now have a false type-0 production for the class name VERB, which may invalidate the proper parsing of the GERMAN branch. We now wish to replace this false production rule for VERB, with either (a) a *known* new production, or (b) by a known (possibly non-existent) previous production, or (c) by a production whose text we don't know, but which was created by a known activating node.

Cases (a) and (b) are readily handled by the introduction of a type-1 production

<$_1$GERMAN>$_1$ <VERB> .... (known replacement definition)

This will become active as a type-0 production as soon as the parsing tree node GERMAN is set up, since it contains no further undecided nodes.

But case (c) requires the use of a type-2 production. Suppose that the node which set up the definition of VERB which we wish to recover, has the class name ACTIVE. Then the type-2 production required reads:

<$_2$ACTIVE>$_2$ <$_1$GERMAN>$_1$<VERB> $\wedge_2$ VERB

When the parsing node FRENCH is set up, we can arrange for the node ACTIVE to be activated, and to set up the vertex of a new parsing treelet. This will result in a type-1 production being added to the language

<$_1$GERMAN>$_1$ <VERB> .... (old type-0 definition) ...

Later, if backup occurs, and the node GERMAN is set up in the main tree, the above type-1 production will be activated to yield the desired, but not specifically known production text.

Notice the interpretation which must be placed upon the definition part of the above type-2 production. The string <$_1$GERMAN>$_1$ <VERB> is a single terminal string, since no production rule exists with such a string in its defined part. However the string VERB is a class name, since a type-0 production commencing <VERB> exists.

3.2. Graded examples

Example 11

Given an arbitrary input string ABC... construct as output the corresponding palindromic string ABC....CBA.

S	<STRING>	ANY | ANY $\wedge$ STRING
	<ANY>	a
T	<STRING>	ANY $\wedge$ SAME | ANY $\wedge$ STRING $\wedge$ SAME
	<ANY>	a
	<$_1$ANY>$_1$	<SAME> $\wedge_1$ ANY

Perhaps some comments may help to explain what is happening. Whenever a node with class name ANY is scanned in the source language parsing tree, two things happen. First, a target tree node bearing the class name ANY is 'determined'. Since ANY is a terminal class, according to our rules, the input stringlet covered by this node is conveyed to the target tree node—suppose that this stringlet is the character X. The next thing to happen is that a first order target parsing tree with vertex bearing the class name ANY is set up. This vertex has two sons, with class names <SAME> and ANY. The first of these is simply a terminal string, the second a terminal class, which as before receives the input stringlet X. This first order tree now contains no undecided nodes, and therefore immediately generates the type-0 production

<SAME>   X

The process of filling out the target tree continues. The next node to be encountered bears the class name STRING, and is left undecided. The last node at this stage of filling out bears the class name SAME. This class name has a zero order production consisting of the terminal character X. Thus the latter character again enters the target tree. The scanning of the source parsing tree now continues, and the next node to be encountered bears the class name STRING, and will determine the target tree node bearing that class name which was left undecided. The next time a source tree node bearing the class name ANY is scanned, a new zero order production for the class name SAME will result, and will replace the previously generated production for this class name.

Example 12

To construct a source language which selectively discriminates sentences of the form:

'string of characters' * 'same string of characters'.

S	<REPEAT>	STRING $\wedge$ * $\wedge$ SAME STRING | sa
	<STRING>	sli
	<$_1$STRING>$_1$	<SAME STRING> $\wedge_1$ STRING

Example 13

Input sentences consist of strings of alphanumeric identifiers separated by commas. Output sentences are similar, but any duplicated identifiers are suppressed.

S	<LIST>	IDENT | IDENT $\wedge_1 \wedge$ LIST
	<IDENT>	OLD | NEW
	<OLD>	*
	<NEW>	sli
	<$_1$NEW>$_1$	<OLD> $\wedge_1$ OLD $\wedge_1$ | $\wedge_1$ NEW
T	<LIST>	IDENT | IDENT $\wedge$ LIST
	<IDENT>	| NEW $\wedge_1$
	<NEW>	sli

On encountering the first identifier in the list, say AB1, the parser must take the second alternative definition of IDENT, since the first leads to a terminal * which cannot be an identifier. This second alternative has class name NEW which results in setting up a type-1 production, in turn generating a type-0 production reading

<OLD>   * | AB1

If any later identifier in the input string is identical with an earlier one, then the first alternative definition of IDENT will be followed, as the production rule for OLD will already contain an alternative definition leading to that identifier.

The target language production rules ensure that a new identifier is added in to the target tree only when the second alternative definition of IDENT in the source tree has been followed.

Notice the various interpretations which must be placed on the four components in the definition part of <$_1$NEW>$_1$. The first component is <OLD>—this is a terminal string since no production rule for <OLD> exists: the next

component is OLD—and this is a class name, since a production rule for OLD does exist. The third component is |, which is a terminal string in a type-1 production rule. The last component is NEW which is a class name, since there is a production rule for this identifier.

Example 14
The input sentences contain imbedded alphanumeric identifiers. In the output strings these identifiers are to be replaced by standardised identifiers, say JNO, JN1, JN2, . . . such that the same input identifier, wherever it appears, is to be replaced by the same standardised identifier. The production rules written out here are purposely incomplete, since we are not concerned with the question of how the identifiers are to be teased out of input sentences.

S <IDENT> OLD | NEW
<OLD> *
<NEW> sli
<$_1$NEW>$_1$ <OLD> $\wedge_1$| ∧, NEW
T <IDENT> OLD | NEW
<JC> 0
<OLD> *
<$_1$NEW>$_1$ <NEW>sli<OLD> $\wedge_1$ OLD $\wedge_1$ JN $\wedge_1$ JC ∧
<JC> {∧ $\wedge_1$ +1}

When a NEW node is scanned in the source tree, a new type-0 source production, and three new type-0 target productions are generated. After encountering three different input string identifiers, say ABC, PQR, XYZ, the state of the production rules is:

S <IDENT> OLD | NEW
<OLD> * | ABC | PQR | XYZ
<NEW> sli
<$_1$NEW>$_1$ remains unaltered
T <IDENT> OLD | NEW
<JC> 3
<OLD> * | JNO | JN1 | JN2
<NEW> sli
<$_1$NEW>$_1$ remains unaltered

Example 15
The simple machine code set up in Example 10, allowed the use of stack operations of the form STO-stack. , ADD-stack. and so on. We wish to convert all such input sentence operations into conventional operations, using sequential identifiers, say LC0, LC1, LC2, in place of 'stack'.

S <INSTRUCTION> STACK | OTHER
<OTHER> sa
<STACK> STO-stack. ∧ INC |
DEC ∧ OP ∧ -stack.
<OP> ADD | SUB | MUL | DIV | POW
<INC>
<DEC>
T <INSTRUCTION> STACK | OTHER
<OTHER> sa
<STACK> STO-LC ∧ INT ∧ . |
OP ∧ -LC ∧ INT ∧ .
<OP> ADD | SUB | MUL | DIV | POW
<INT> 0
<$_1$INC>$_1$ <INT> {$\wedge_1$ INT $\wedge_1$ +1}
<$_1$DEC>$_1$ <INT> {$\wedge_1$ INT $\wedge_1$ −1}

The two class names INC and DEC, with null source language productions, have been introduced solely to increment or decrement the target production for INT at the right times. The right times being to increment INT after each STO-stack operation and to decrement INT before any other stack operation.

Example 16
As a final exercise we show how an extension to standard FORTRAN syntax can be implemented. Input consists of sentences in an 'extended-FORTRAN' dialect. In this dialect arrays may be subscripted with unlimited number of subscripts, instead of just three subscripts as in normal FORTRAN. Each subscript may range between lower and upper bounds set in a DIMENSION statement. Upper and lower bounds are integers, possibly negative. A comma separates the lower from the upper bound, and a $ separates the latter from the next lower bound. A typical DIMENSION statement might read

DIMENSION ABC(10,20$-3,0$-10,+100),PQR(7,21),
XYZ(−1,+0 $ -2,+2 $ −3.+3 $ −4,+4)

Output from the processor is to consist of equivalent instructions in standard FORTRAN, in which all arrays are singly subscripted, with positive subscripts, and the appropriately altered DIMENSION statements.

The processing scheme which we shall set up to solve this problem, is not the only possible solution, nor necessarily the simplest. Furthermore, certain implicit assumptions are made, for instance that the source strings contain one DIMENSION statement only. These simplifications could have been avoided at the cost of making the production rules more complex, and the whole exercise less valuable as a demonstration. On the other hand, further elaborations could have been incorporated, such as making the scheme compatible with standard FORTRAN, or constructing built-in execution checks against dimensioning outside limits. These refinements have been avoided for the same reason.

The problem divides neatly into simple and ramified processing tasks. The reconstruction of the DIMENSION statement is a matter of simple processing. All that is really required is to replace $ signs by multiplications, add brackets and +1 stringlets, and use curly brackets to evaluate the results arithmetically. The above written dimension statement is to appear in output as:

DIMENSION ABC({(20−10+1)*(0− −3+1)*(+100− −10+1) }),
PQR({(21−7+1)}),
XYZ({(+1− −1+1)*(+2− −2+1)*(+3− −3+1)*
(+4− −4+1)})

We have already assumed that our arithmetic processor can deal with the eccentric appearance of these expressions, and replace the whole string by

DIMENSION ABC(4884),PQR(15),XYZ(945).

The task of the ramified process is to set up new type-0 production rules in both the source and target languages. These will be production rules for a new class name, DNA (standing for 'dimensioned name'). This class name will explain how dimensioned variables are to be recognised in the source language, and how they are to be processed in the target language. Given a dimension statement item, say ABC(l1, u1 $ 12, u2 $ 13, u3 $ . .) and a source string dimensioned array, say ABC(d1, d2, d3 . . .), the following type-0 production for DNA will be set up in the source language

<DNA> . . | ABC(∧ .OTS ∧ , ∧ OTS ∧ , ∧ OTS . . .) . .

where OTS is also a type-0 class name ('other statement'), defined by a source language production. The above production for DNA will contain as many OTS entries in ABC, as the latter has subscripts in its DIMENSION declaration.

In the target language, the following corresponding type-0 production must be generated:

<DNA> . . | ABC(1+ ∧ d1 ∧ {− ∧ l1 ∧ }+({u1 ∧
− ∧ l1 ∧ +1})*({∧ d2 ∧ − ∧ 12 ∧ }
+({∧ u2 ∧ − ∧ 12 ∧ +1})*(∧
d3 ∧ {− ∧ 13 ∧ } . . . .)))) . . . .

In the above typical dimensioned variable ABC, on substituting the actual upper and lower limits of the subscripts, we would get

<DNA> ABC(1+ ∧ d1 ∧ −10 +11*(∧ d2 ∧ +3+111*
∧ d3 ∧ +10)))

The subscripts d1, d2 and d3 are simply strings of characters which may themselves contain subscripted variables, which will undergo the same processing operations recursively.

Function names pass unaltered through the processor, since their identifiers will not have appeared in the DIMENSION statement. Arrays differently dimensioned to the number of dimensions allocated in the DIMENSION statement, will not be recognised under the class name DNA. The upper and lower subscript limits appearing in the DIMENSION statement may be integral arithmetic expressions, since they are in any case passed through the arithmetic processor by the use of { }.

For the sake of brevity we have used a new terminal class name, ssi, to stand for a set of digits possibly preceded by a unary + or − sign.

```
*S*  <STATEMENT>         DST | OTHER
     <DST>               DIMENSION ∧ LIST
     <LIST>              LIST ITEM | LIST ITEM ∧ , ∧ LIST
     <LIST ITEM>         sli ∧ (∧ DIMENSIONS ∧)
     <DIMENSIONS>        LOD ∧ , ∧ HID |
                         LOD ∧ , ∧ HID ∧ $ ∧ DIMENSIONS
     <OTHER>             sa ∧ DNA ∧ OTHER | sa
     <LOD>               ssi
     <HID>               ssi
     <_1DST>_1           <DNA> ∧, LIST,
     <_1LIST>_1          LIST ITEM_1 |_1 LIST ITEM_1 ∧_1 |
                         ∧_1 LIST_1
     <_1LIST ITEM>_1     sli ∧, (∧, DIMENSIONS, ∧, )
     <_1DIMENSIONS>_1    ∧ OTHER ∧ |_1
                         ∧ OTHER ∧ , ∧_1 DIMENSIONS,
*T*  <STATEMENT>         DST | OTHER
     <DST>               DIMENSION ∧ LIST
     <LIST>              LIST ITEM | LIST ITEM ∧ , ∧ LIST
     <LIST ITEM>         sli ∧ ({∧ DIMENSIONS ∧ })
     <DIMENSIONS>        (∧ HID ∧ − ∧ LOD ∧ +1) |
                         (∧ HID ∧ − ∧ LOD ∧ +1)* ∧
                         DIMENSIONS
     <OTHER>             sa ∧ DNA ∧ OTHER | sa
     <LOD>               ssi
     <HID>               ssi
     <_1DST> _1          <DNA> ∧_1 LIST_1
     <_1LIST> _1         LIST ITEM_1 |_1 LIST ITEM_1 ∧_1 |
                         ∧_1 LIST_1
     <_1LIST ITEM>_1     sli ∧_1 (1+ ∧ OTHER {− ∧, LOW ∧,}
                         ∧_1 DIMENSIONS ∧,)
     <_1DIMENSIONS>_1    |, +({∧_1 HID ∧_1 − ∧_1 LOD ∧_1
                         +1})*( ∧ OTHER {− ∧_1 LOW ∧_1}
                         ∧_1 DIMENSIONS ∧_1)
     <_1LOD>_1           <LOW> ∧_1 LOD
```

Processing in practice

Some thought has been given to the question of how the above described scheme might be presented in a form acceptable to the average user. At one extreme one might incorporate all the above ideas, with additional refinements into a new formal language. At the other extreme we might imbed all the processing operations into an existing formal language making use of subroutine calls. On the whole a compromise seems preferable. The language production rules will be regarded as strings in a new formal language, or as input for a processor-compiler. All other needs will be imbedded in a symbolic language. In practice the user would have to write out his source and target production rules in the specified program input language (that is, the punched card corollary of the publication language set up in Table 1, and its extension). The user would also write his own input and output dispatcher routines, called say INPUT and OUTPUT. In these the user will have to decide how to cut up input into meaningful strings, whether to ignore blank columns, how to deal with continuation cards, what to do with the output, and so on. The target productions might have been so written that the first character of the output string was really an instruction to the OUTPUT routine (as, analogously, the first character of each line of machine output is commonly an instruction to the line printer). By this means one might select format statements, print out messages (for instance 'INPUT STRING UNGRAMMATICAL'), decide whether to print, punch or otherwise store output. All these are matters of conventional programming.

The author has set up a field-tested programme for performing simple processing, called MAMMAL. It accepts blocks of production rules in a form closely similar to the publication language of Table 1. A small main program, and input and output dispatcher routines complete the system. Most of the examples of Section 2 have been tested out, and run correctly, though they are inefficient and time-consuming. The slow running is due to the employment of rather a naïve parsing algorithm, based on a general tree manipulation scheme (MESH). Most of the parsing algorithm instructions are calls on the various MESH subroutines. The use of MESH made MAMMAL easy to write, but execution times are an order of magnitude greater than if the processing scheme had been properly programmed.

5. Conclusion

It was our aim to present a compact scheme enabling a wide variety of manipulation operations to be specified and performed. It is believed that, at the simple processing level at any rate, the scheme is quite easy to use. It involves little more than a slight extension of normal notation for syntax for production rules. We suggest that simple processing could be a useful tool in the hands of a fairly unsophisticated programmer. Ramified processing however, calls for a more informed approach: the language is hardly more elaborate, but its application to the solution of specific problems does require very careful thought.

6. References

By far the greatest fraction of the literature associated with syntax processing is devoted to the construction of parsing algorithms, and the formal studies of syntax. A most informative review article of this material, with numerous references is to be found in Feldmann and Gries (1968). The author has made extensive use of Ingermann (1966) and Cheatham and Sattley (1964) in the initial stage of learning about syntax parsing. The present paper is however concerned more with the use made of parsing trees, than their construction. Here the literature is relatively sparse. The author has been unable to discover any papers in which the output strings are obtained from output production rules, in a scheme so symmetric, that input and output blocks of production rules can be actually interchanged. The system with the greatest similarity appears to be Brooker, Morris and Rohl (1967): in the latter, productions for input strings are called *formats* or *phrase definitions*, with which output productions named *format routines* are associated. However the latter productions do not appear to be in a form such as could be used by a parsing algorithm. Other systems contain actual output instructions, or pointers to such instructions, imbedded within the input production rules. Yet another scheme, in Reynolds (1965) makes use of list processing operations to define the output strings.

References

Brooker, R. A., Morris, D., and Rohl, J. S. (1967). Compiler-compiler facilities in Atlas autocode, *The Computer Journal*, Vol. 9, pp. 350-352.

Cheatham, T. E., and Sattley, K. (1964). Syntax directed compiling, *Proc. AFIPS* 1964 *SJCC*, Vol. 25, pp. 31-57.

Feldmann, J., and Gries, D. (1968). Translator writing systems, *CACM*, Vol. 11, No. 2, pp. 77-113.

Ingerman, P. Z. (1966). *A Syntax Oriented Translator*. Academic Press Inc.: New York.

Reynolds, J. C. (1965). An introduction to the COGENT programming system, *Proc. ACM 20th Natl. Conf.*, pp. 422-436.