A note on Foster's Syntax Improving Device

by P. M. Woodward

Summary

The paper describes an ALGOL program called SID, devised and written for RREAC by J. M. Foster. The program is designed to operate on the syntax rules of any programming language of a type and complexity similar to that of ALGOL. It transforms the rules to a form suitable for single-track parsing, and can be made to generate a syntax-directed compiler which does not refer back to the original form of the rules.

1. Introduction

In April 1966, J. M. Foster completed a remarkable computer program which "improves" the syntax rules of a certain class of artificial languages. The program itself is expressed in ALGOL, runs on the RREAC computer (which has 24K words of core) without filling it, and takes advantage of certain list-processing procedures of McCarthy type which are provided as part of the RREAC's software. Foster's program is called SID.

An authoritative paper on SID by its originator has not yet been published, and the present account is based on a lecture which, in the absence of J. M. Foster, was delivered apologetically at Imperial College by the writer, assisted by I. F. Currie end Miss S. G. Bond, on October 26, 1966. It describes in outline what SID does but gives no detail about how it does it. Briefly, SID accepts as input the rules of a context-free grammar expressed in Backus form, and attempts to reformulate them so as to arrive at an equivalent set of rules suitable for use by a simple one-track parsing algorithm. Although the form of the rules is altered, the structure of the language is naturally preserved unchanged. As not all languages are susceptible to single-track parsing technique, SID may fail, and if so, it reports the extent to which it partially succeeded. If, on the other hand, it fully succeeds, it outputs the parsing algorithm. It has thus generated an efficient syntax analyser for the given language. One more important facility is included, the ability to manipulate "functions" or "actions" along with the syntax rules. These can be any actions which the user wishes to associate with the syntax analysis, such as compilation. In a legalistic sense, SID can then be said to provide as its output a complete compiler for the given language. This statement is, however, apt to be misleading, as it can give the impression that there is little more to compiler-writing than provision of the source-language syntax rules. The sophistication of a compiler largely resides in the "actions" to be associated with the analysis of the source program, and their manual construction still demands the skill of a compiler-writer and an intimate knowledge of the properties of the object code. Even so, SID is proving extremely valuable in applications. For the compiler-writer it helps

particularly by segmenting the task as a whole, and by easing tremendously the task of progressive debugging.

The syntax improvement effected by SID is aimed at producing rules in single-track predictive form as explained in section 2 which follows. This requires the application of two types of transformation, outlined in sections 3 and 4, and illustrated by example in section 5. The concept of "actions" to be performed in response to syntax analysis is defined in section 6, and the paper concludes with a trivial example in which SID is applied to the problem of generating a number-input routine from the syntax rules for a signed integer.

2. Single-track predictive analysis

Compilers based on formal methods of syntax analysis are said to be "syntax-directed", but they can be cumbersome in use. Syntax analysis can be a slow business if many alternative routes have to be tried - with repeated back-tracking - until the correct one is found [1]. If alternative possible branches of the analysis are developed in parallel, each being retained until all but one has been contradicted by the input stream, the demands on storage capacity may also become very large. The parsing algorithm employed in the syntax analyser which is output from SID avoids waste of time or storage space if, as is very frequently the case, the structure of the language lends itself to single-track predictive analysis. Otherwise SID fails. It should particularly be noted that failure occurs, not during compilation, but in the compiler-writing stage. No grammatically correct program can give rise to parsing failure in a syntax analyser which has emerged from SID, because SID has automatically vetted the rules.

The syntax transformations which SID carries out are "transpositions" to convert the rules to a suitable form for "predictive" analysis, and "factorizations" to convert the results of the first stage into a form suitable for "single-track" predictive analysis. Before describing what is meant by transposition and factorization (writer's terms, not Foster's), it is necessary to define certain standard forms of syntax rules.

(a) The initial form

All rules supplied to SID must be of the form

$$A = V_1V_2 \ldots V_n \quad | \quad W_1W_2 \ldots W_m \quad | \quad \ldots \qquad (1)$$

where A is a class-name (such as "integer" or "declaration"). The vertical bars on the right separate the alternative strings into which A can be expanded. The V's W's etc. are either terminal symbols* (such as + or **begin**) or further class-names (non-terminals). For example, one might have a rule such as

DECIMALNUMBER = INTEGER | .INTEGER | INTEGER.INTEGER

In this example, V_1 is the class-name INTEGER, W_1 is the terminal symbol "." and W_2 is the class-name INTEGER, etc. In some formal

* A terminal symbol is one which stands for itself, i.e. cannot be further expanded.

linguistic papers, each alternative in (1) would be regarded as a separate rule, but in the present account we shall refer to them as "rule alternatives", and a set of rule alternatives for one class-name as a "rule".

(b) Greibach's standard predictive form [2]

All rule alternatives are of the form

$$A = vV_2V_3 \ldots V_n \qquad (2)$$

where v is a terminal symbol and $V_2 \ldots V_n$ are non-terminal. The advantage of this form is easily seen, when we remember that syntax analysis always begins with the question "Is my given string of symbols a legal sentence?" In attempting to answer this question, we are led to further questions of the same type applying to parts of the input string, the class-name "sentence" being progressively replaced by more and more restricted class-names until finally the analysis is complete and the first question can be answered. If all rule alternatives are in Greibach's form, the algorithm for analysis is immediately obvious. Thus, given a string *abcd* we simply compare the first symbol, a, with the starting symbol for every rule alternative under the class name "sentence" (or "ALGOL Program"). As soon as we find a rule alternative beginning with *a*, we have generated a sub-problem of exactly the same type for the remainder of the string, *bcd*. If there is more than one rule-alternative which starts with a, more than one line of analysis must be followed up until (hopefully) by the time the input string is finished, a single analysis remains.

Example: Analyse the sentence *abcd*. by the following grammar (in which ϕ denotes a null alternative):

S = aX

X = bZ | dX | bY | ϕ

Y = c | ϕ

Z = cX

Method

abcd	S aX
bcd	X bZ | bY
cd	Z | Y cX | c
d	X | ϕ dX
ϕ	X ϕ

Hence the analysis is (a (b (c (d ( )))))

S X Z X X

(c) Single-track standard predictive form

All rules are of the form

$$A = vV_2V_3 \ldots V_n \mid wW_2W_3 \ldots W_m \mid \ldots \qquad (3)$$

where the terminal symbols v, w, etc. are all **different**. (If one of the rule alternatives is a null, it is required that the next rule to be applied should not contain any alternatives starting with one of the symbols v, w etc. which are alternatives to the null.) It is easily seen that this more restricted grammar permits a straight analysis, without the need to retain alternatives as in the line bcd of the previous example.

Broadly, SID's transformations are designed to convert from form (a) to form (b) by a process of "transposition" and then from (b) to (c) by a process of "factorization". However, the forms (b) and (c) are unnecessarily restrictive, and the following relaxations are permitted. First, it is unnecessary to insist that the second and subsequent symbols in a rule-alternative should be non-terminal; any class-names or terminal symbols can be permitted. And secondly, it is not insisted that each rule-alternative should begin with a terminal symbol provided that the non-terminal symbol can be directly expanded, by its own rule and by any resulting further expansions, into a set of alternatives which actually do begin with terminal symbols. Terminal symbols reached in this way are treated as though explicitly present in the original rule. This may appear to complicate slightly the parsing algorithm, as a stack becomes necessary to keep track of the expansions, but the stack is simple and linear and the storage of the rules themselves is simplified.

3. Transposition

The first task of SID is to eliminate from all rule-alternatives any starting class-names which cannot be directly expanded to terminal symbols. This means, in effect, that cyclic non-terminals must be sought out and removed. The simplest example of a cyclic rule may be written

$$A = a \mid Ab \qquad (4)$$

which is the syntax rule for an *a* followed by any number of b's. Using algebraic notation, we can write

$$A = a + Ab$$

where + denotes **or**. Hence

$$A = a + ab + abb + \ldots$$
$$= aX(say)$$

where $X = 1 + b + bb + \ldots$

$$= 1 + bX$$

Thus the original single rule may be re-written as the pair of rules

$$A = aX \qquad (5)$$
$$X = bx \mid \phi$$

(Note that unity in the algebra corresponds to a null in the syntax.) If small letters denote terminal symbols, each of these rules is now in prediotive form.

In general, cyclic non-terminals may give rise to loops which embrace more than one rule, as for example in the pair of rules

$$A_1 = a_1 \mid A_1 b_{11} \mid A_2 b_{21}$$
$$A_2 = a_2 \mid A_1 b_{12} \mid A_2 b_{22}$$

This pair of rules is like a pair of "simultaneous equations" in non-commutative algebra, and is in fact a generalization of the equation

$$A = a + Ab$$

provided that each of these symbols is treated as a matrix, thus

$$A = [A_1 \quad A_2]$$
$$a = [a_1 \quad a_2]$$
$$b = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

in which we interpret 1 as a 2×2 unit matrix. To express the solution in syntax rule form, let

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}$$

and we obtain

$$A_1 = a_1 X_{11} \mid a_2 X_{21}$$
$$A_2 = a_1 X_{12} \mid a_2 X_{22}$$

where

$$X_{11} = b_{11} X_{11} \mid b_{12} X_{21} \mid \phi$$
$$X_{21} = b_{21} X_{11} \mid b_{22} X_{21}$$
$$X_{12} = b_{11} X_{12} \mid b_{12} X_{22}$$
$$X_{22} = b_{21} X_{12} \mid b_{22} X_{22} \mid \phi$$

The original cyclic pair of rules for A_1 and A_2 has now been expanded into six rules, **all in predictive form**. In general the matrices can be of any order, and the a's and b's need not be terminal symbols.

In a complete. set of syntax rules, there may be several cyclic definitions, and SID employs a simple algorithm. to determine where the cycles occur. After applying transpositions to remove the cycles, any remaining class-names which occur as starters of rule-alternatives can be traced, by a direct route through the rules, to a set of alternative terminal symbols. However, if the rules were transformed no further,

it would not be possible, when analysing a "sentence", to predict uniquely, as each symbol was read, which rule to apply next. There would be a number of branching alternatives to choose from, and new branches might at first be created more rapidly than old ones died. Though temporary, this situation generally gives rise to more computing than is necessary for syntax analysis.

4. Factorization

Given a syntax rule such as the following (in which small letters are terminal symbols and capitals may be class-names),

S = aX | aY | bZ (6)

it would be necessary, after reading the symbol a to test for both X and Y, which are alternatives. These could lead to still further branching. To remove the ambiguity, we could make the construction

S = aP | bZ (7)

P = X | Y

so that, after reading the symbol a, the next stage is to test for P only. Algebraically, this corresponds to the factorization

aX + aY = a(X+Y) = aP (say)

If X and Y are class-names, SID checks that they lead to disjoint sets of alternative starting symbols, and if not, carries out further factorizations. Though in principle a simple idea, the greater part of SID is devoted to this process, which, from a programming point of view, is not by any means straightforward.

5. An artificial example

Consider a given pair of rules

S = a | Sc | Tb (8)

T = ϕ | aT

in which small letters are terminal symbols. SID will first discover the cycle in the S-rule and remove it, thus

S = aX | TbX

X = ϕ | cX

T = ϕ | aT

Observing that T may start with an a, which leads to ambiguity, SID substitutes for T in S, thus

S = aX | bX | aTbX

and factorizes giving

(9.1) S = aY | bX (9)

(9.2) Y = X | TbX

(9.3) X = ϕ | cX

(9.4) T = ϕ | aT

In this example, no further factorization is necessary. This is clear from the lists of starting symbols which SID will have constructed for each alternative:

(10.1) S a | b (10)

(10.2) Y – , c | b , a

(10.3) X – | c

(10.4) T b | a

A hyphen stands for the end of the input string which is to be analysed as an S. In no rule does either of the alternative sets shown at (10) share a common symbol. Thus SID has completed the transformation of rules (8) yielding the improved set (9).

To illustrate the use of (9) and (10), consider how to parse the string abc as an S.

```
abc     Testing for S
        Refer to 10.1, find a on left.
        Refer to 9.1 (left)
        aY

 bc     Testing for Y
        Refer to 10.2, find b on right
        Refer to 9.2 (right)
        TbX

        Enter the position T↑bX on stack

        Testing for T
        Refer to 10.4, find b on left
        Refer to 9.4 (left)
        ϕ

        Unstack the position reached
        bX

  c     Testing for X
        Refer to 10.3, find c on right
        Refer to 9.3 (right)
        cX

  -     Testing for X
        Refer to 10.3, find - on left
        Refer to 9.3 (left)
        ϕ
        No positions on stack.
        Analysis complete.
```

6. Actions

If the syntax rules provided to SID are supplemented by "actions" embedded at appropriate places in the rules) these will be carried along through SID's transformations and re-appear at the right places in the improved rules. There is thus no need for the syntax-directed compiler to refer back to the original form of the rules before carrying out the actions [3]. As a simple example, we may consider a rule such as the following, in which **x** and **y** denote actions, or "outputs" from the rule:

I = D **x** | ID **y** (11)

This could be a description of the structure of an integer (I = integer, D = digit). The analysis of 365 would be

(((3 **x**) 6 **y**) 5 **y**)
I I I

After improvement (11) becomes

I = D **x** T (12)

T = ϕ | D **y** T

and the "improved analysis" of 365 would be

(3 **x** (6 **y** (5 **y** ( ))))
I T T T

The actions **x** and **y** are thus carried out in the same place as before. Typically, **x** might represent the instruction to store the first digit, whilst **y** would represent instructions to multiply the accumulated answer by ten and add the new digit.

7. The output from SID

The output from SID is a skeleton compiler expressed in a macro-language which can be expanded into any machine code for which individual expansions of about eight macros have already been prepared. "Actions" are supplied to SID merely as identifiers, and reappear in the output in the right places. The macro-generation applied to the output can then be used to expand the actions along with the syntax analyser, provided that they too have been programmed.

As an example, we give below an actual input to SID with the corresponding output. However, liberties have been taken with the manner of writing, to avoid the need for macro expansion. The input actions are written out in full, and the output is expressed in more-or-less plain language.

INPUT TO SID

NUMBER = SIGN INTEGER $\underline{n \leftarrow n \times s}$ SPACE

SIGN = + $\underline{s \leftarrow 1}$ | − $\underline{s \leftarrow -1}$ | ϕ $\underline{s \leftarrow 1}$

INTEGER = DIGIT $\underline{n \leftarrow t}$ | INTEGER DIGIT $\underline{n \leftarrow 10 \times n + t}$

OUTPUT FROM SID

```
number:   if h = '+' or '−' or DIGIT
          then [1]sign, [2]integer, n ← n × s, [3]read, exit else fail

  sign:   if h = '+' then [4] read, s ← 1, link else
          if h = '−' then [5] read, s ← −1, link else
          if h = DIGIT then s ← 1, link else fail

integer:  if h = DIGIT then [6] read, n ← t, aux else fail

   aux:   if h = SPACE then link else
          if h = DIGIT then [7] read, n ← 10 × n + t, aux else fail
```

ROUTINE SUPPLIED BY USER†

```
  read:   t ← h, h ← tape, link
```

Explanation

The input is self-explanatory; it represents the syntax of a signed integer and contains actions (underlined) which will assign the integer to the variable n. The output is the syntax analyser, complete except fcr a simple read routine to obtain the next symbol from the tape, which is supplied by the user. The notation should be interpreted as follows:

Each of the items separated by commas are to be treated as commands. For example,

[2] integer	means	Place the marker [2] on the stack and jump to the label integer.
link	means	Refer to stack to find the last marker and proceed to next command.

The program can be tested by assuming that the tape contains the string

```
- 2 7 apace ... ... ...
  ↑
```

and the reader is at the arrow. The state of the variables is initially

h	t	n	s	stack
−				

† Symbols on data string are read as soon as they can but tested as late as possible so read after current look-ahead char found then perform all actions then test for new char

After obeying the program we find the reader at

```
-   2   7   apace   ...   ...   ...
                      ↑
```

and the state of the variables as

h	t	n	s	stack
...	space	-27	-1	

having arrived at n = -27 as required. The interested reader may be tempted to follow through the intermediate steps for himself.

It will be noticed in this example that the most up-to-date symbol, assigned to the variable h, is the one which is used to steer the syntax analysis, whereas the "actions" may operate on any past information from the stream of input symbols up to (but not including) h. That the actions should avoid using the current value of h is purely a matter of convenience associated with the positions at which they are inserted in the input syntax rules. If the analyser is permitted to look ahead by one symbol, the actions can be placed at what seem to be their most natural positions. In large-scale applications of SID, it is convenient to pre-process the input string so that composite symbols such as **begin** and **end** may be treated as the single characters they really represent.

8. Acknowledgement and References

The writer is principally indebted to J. M. Foster, now at Aberdeen University, regretting only that it has not proved possible at present to quote an original reference to the work. Miss S. G. Bond assisted with the preparation of the example in section 7 and with explanations generally. The textual references are

[1] T. V. Griffiths and S. R. Petrick, "On the Relative Efficiences of Context-Free Grammar Recognizers", Comm. A.C.M., vol. 8, p. 289 (May 1965).

[2] Sheila Greibach, "Formal Parsing Systems", Comm. A.C.M., vol. 7, p. 499 (August 1964).

(3) Susumu Kuno, "The Augmented Predictive Analyser for Context-Free Languages – Its Relative Efficiency", Comm. A.C.M., vol. 9, p. 810 (November 1966).

DISTRIBUTION

Library	4
Central Office (for Director)	1
Head of Department (Dr E.V.D. Glazier)	1
Head of Group (Dr. D.H. Parkinson)	1
Author	12
D.S.R.(L), Castlewood House	1
Mr. A.I. Llewelyn, Ministry of Technology	1
Abell House, John Islip street, London, S.W.1.	1
Miss Beryl Kitz, Admiralty Research Laboratory,	1
Teddington, Middlesex	1
Dr. D.P. Jenkins	1
Dr. A.J. Fox	1
Mr. P.W. Edwards	1
Mr. P.R. Wetherall	1
Mr. I.F. Currie	1
Miss S.G. Bond	1
Mr. J.G. Gibbs	1
Mr. M. Griffiths	6
Spares for Miss Vernon, RRE South	10
Total	47