Description of a syntax-directed translator

By C. M. Reeves

An extension of the Backus notation used in the ALGOL report is described which permits both the syntax and semantics of general languages to be specified readily and compactly. A mechanism based on that of Metcalfe is explained for performing the indicated operations automatically, and a description is given of an ALGOL program which simulates this mechanism on a KDF9 computer.

The paper will be of interest to students and teachers of computing science who desire facilities for practical experimentation in the design of formal languages. The KDF9 ALGOL program is available on request.

1. Introduction

This paper is submitted in the belief that students and teachers of computer science may find it of value to have readily available facilities for practical experimentation in the definition and manipulation of formal languages. A rather detailed though deliberately elementary account is given of the philosophy and mode of action of one particular approach to this problem which has been implemented as an ALGOL program running on a KDF9 computer. The program is available on request.

The ALGOL program simulates a special-purpose stored-program digital computer. A program for this machine represents the structure (syntax) and required interpretation (semantics) of some language, the source language. The purpose of the machine is to read as data, under control of its program, a text allegedly written in the source language. The decision whether or not the source text is grammatical in the source language is made on the basis of the rules of syntax for the source language embodied in the program. The machine will give an error indication if the source text is ungrammatical, otherwise it will produce an output determined by the semantics in the program. It may be appropriate to regard the output as a text written in another language, the target language. The machine acts in this case as a translator. Alternatively the output may be some rearrangement or modification of the input and the operation of the machine is that of an editor.

There is a wide range of possible applications, achieved in the main by differences in the programs which are run on the machine. At the same time we may consider possible hardware extensions to the machine, simulated by changes in the ALGOL program.

The overall organization of the machine is based on the very clear description given by Metcalfe (1964). It is felt that the present account will be most useful if it is as much as possible self-contained. Accordingly a summary of Metcalfe's proposals is given together with the various modifications and extensions that have been made, and an account is also given of the implementation through the ALGOL program. The use of ALGOL is justified, not by its efficiency at run time certainly, but by its universality and by the expectation that users will wish to make their own modifications to the program.

2. A meta-syntactic language

Most readers will be aware that the official document which defines the ALGOL language (Backus et al., 1963) makes use of a special notation, the so-called Backus normal form, for defining the grammar, or rules of syntax, of ALGOL. This meta language is simple, powerful and general. It is the basis for most formal specifications of current programming languages. We shall take it as our own starting point.

Backus notation represents the grammar of a language as a set of definitions of grammatical structures. Each definition has three components: first the name of the structure being defined, second the symbol ::= which is read as "is defined as", third an expression which specifies the permitted forms of the structure. The symbol ::= belongs to the meta language, not to the language whose grammar is being defined, and so is termed a meta symbol. The names of the structures being defined may be spelt using symbols which do belong to the defined language, and so in order to avoid any possible confusion, these names, wherever they occur, are enclosed between the angled brackets ⟨ and ⟩. These brackets are thus further meta symbols. Now we come to the expression part of the definition. An expression is a list of alternative permitted forms of a structure, pairs of alternatives being separated by the meta symbol | which is read as "or". Each alternative is a list of elements written one following another. The sense of an alternative is obtained by reading "followed by" between adjacent pairs of elements. Finally an element is one of two types, a constant or a variable. A constant is any symbol belonging to the character set or "alphabet" of the defined language, and represents itself. A variable has the form of the name of a structure and represents any permitted form as defined for that structure.

† The ALGOL report uses \.
As an example, we find in the ALGOL report
\[
\langle \text{digit} \rangle ::= 0 \uparrow 1 \uparrow 2 \uparrow 3 \uparrow 4 \uparrow 5 \uparrow 6 \uparrow 7 \uparrow 8 \uparrow 9
\]
which reads

“A digit is defined as the symbol 0 or the symbol 1 or . . . or the symbol 9”

and

\[
\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \uparrow \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle
\]

which reads

“An unsigned integer is defined as a digit or an unsigned integer followed by a digit”.

Thus in parsing the structure 943, we recognize 9 as a digit and, taking the first alternative, as an unsigned integer also. 4 is also a digit and hence from the second alternative, 94 is an unsigned integer. Similarly 3 is a digit and so 943 is an unsigned integer.

A definition in which the structure being defined appears within the defining expression is termed recursive. The ALGOL report makes widespread use of recursive definitions. An alternative to recursion may often be obtained by an extension of the meta language. We introduce another meta symbol ★ which may precede any element in an expression and is read as “a sequence of none or many”. Thus

\[
\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \ ★ \langle \text{digit} \rangle
\]

is read as

“An unsigned integer is defined as a digit followed by a sequence of none or many digits”.

Let us look again at the ALGOL report. We find

\[
\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \uparrow
\]

\[
\langle \text{identifier} \rangle \langle \text{letter} \rangle \uparrow \langle \text{identifier} \rangle \langle \text{digit} \rangle
\]
The repetition of \langle \text{identifier} \rangle on the right-hand side can be avoided by introducing the meta symbols ( and ) whose effect when enclosing an expression is to produce a structure syntactically equivalent to an element. Thus

\[
\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \uparrow
\]

\[
\langle \text{identifier} \rangle \langle \text{letter} \rangle \uparrow \langle \text{identifier} \rangle \langle \text{digit} \rangle
\]
or yet more compactly

\[
\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \ ★ \langle \text{letter} \rangle \uparrow \langle \text{digit} \rangle
\]

which reads

“An identifier is defined as a letter followed by a sequence of none or many letters or digits”.

Our meta language is now quite complicated. We have described it informally in English but it would be convenient to have some more formal and compact definition. To invent a meta-meta-language in which to describe our meta-language would be to embark on a

\[\uparrow \text{Metcalfe uses } \uparrow\]

journey that has no end. Let us instead consider the much more satisfying task of using our meta language itself to describe itself. As things stand this is impossible because our meta symbols have explicitly been excluded from the character set of the defined language. This obstacle is removed by introducing a meta symbol = with the significance that whatever symbol follows it is to be interpreted as a constant and which for consistency must be placed in front of every constant. Thus \uparrow on its own is a meta symbol, but \uparrow = denotes the actual symbol “\uparrow =”.

We have so far spoken of the various symbols forming the character set of the defined language without in any way specifying what they are. In fact it is very convenient not to have to list them explicitly. To allow for this we introduce two more meta symbols, U and \#. U denotes “any symbol” and \# preceding an element denotes “any symbol other than”. This is most easily understood when the element is a constant: thus \# = 3 means any symbol other than the digit 3. It is possible to have more complex forms such as \#(=a=\#b\#a=\#c) which denotes (i) any symbol other than “a” or (ii) “a” provided that it is not followed by “b” or “c”. Here he acceptability of “a” depends upon its context.

All the meta symbols so far introduced are available in the KDF9 paper tape code. In particular U is obtained by means of the non-escaping underscore code. We have adopted the convention for our ALGOL simulation on KDF9 that the alphabet for any particular language being defined must be the set or a subset of the marks that can be printed in a single carriage position using this tape code. We only recognize symbols which put ink on the paper: spaces, tabs, case shifts and newlines are ignored. We are left with the set of normal- and shifted-case symbols any of which may be underlined: “A”, “a”, “\u211a” and “\u211aa” are four distinct symbols.

We are now ready to define the syntax of our meta syntactic language as follows, where we use the meta symbol \# to separate pairs of definitions.

\[
\langle \text{grammar} \rangle ::= \langle \text{definition} \rangle \ ★ \langle ; \langle \text{definition} \rangle \rangle
\]
\[
\langle \text{definition} \rangle ::= \langle \text{variable} \rangle ::= ::= ::= \langle \text{expression} \rangle
\]
\[
\langle \text{expression} \rangle ::= \langle \text{alternative} \rangle \ ★ \langle \uparrow \langle \text{alternative} \rangle \rangle
\]
\[
\langle \text{alternative} \rangle ::= * \langle \text{element} \rangle
\]
\[
\langle \text{element} \rangle ::= \langle \text{variable} \rangle \uparrow \langle \text{constant} \rangle \uparrow = \# \langle \text{element} \rangle
\]
\[
\uparrow = \langle \text{expression} \rangle \uparrow = \# = \# \langle \text{element} \rangle
\]
\[
\langle \text{constant} \rangle ::= = = \# U
\]

3. A parsing mechanism

This section describes a simple process for determining whether a given source text is grammatical in a language whose syntax is specified as has been described above. The process is performed by a special-purpose stored-program computer. We shall first describe the computer and then how to write programs for it.
Syntax-directed translator

The data for this parser machine is the ordered stream of symbols forming the source text. The program which the machine obeys is the parsing algorithm for the source language. The program is an ordered stream of instructions, and instructions are normally obeyed in sequence. Both the data and the program are scanned one element at a time. The two points of scan may be moved in either direction. Their positions may be recorded and subsequently reset if desired.

Besides storage space for the data and program, the machine has two registers “inpt” (input pointer) and “pcr” (parser control register) for recording the current points of scan, a push-down stack to record pairs of these pointer positions, and a Boolean register “flag” whose values true and false indicate the success or failure of different stages in the parsing process. Initially the points of scan are set to the positions of the leading elements of the two streams, the stack is empty, and “flag” is set true. The control cycle of the machine consists of inspecting the instruction of the program specified by “pcr” carrying out the indicated operations which will include changing “pcr” (normally to the position of the successor in sequence), and returning to commence a new cycle.

The instruction code is of single-address type and has the structure (F, A) where F, the function part, specifies the operation, and A, the address part, the operand. The address part may be either a symbol of the defined language or a pointer value identifying some particular instruction of the program, or it may be null. The list of available functions F and their effects is as follows. Except where the contrary is explicitly stated it is implicit that their effects include advancing “pcr” to the location of the next instruction in the program sequence.

MATCH If the source symbol at the current point of scan is the same as that in the address part A, set “flag” true and advance “inpt” to the position of the successor symbol. Otherwise set “flag” false.

NEXT Set “flag” true and advance “inpt” to the successor position.

CALL Record the value of “inpt” in the stack together with, if the address part A is null, then null else the location of the next instruction in the program sequence. If A is not null then set A in “pcr”. Set “flag” true.

RETURN If “flag” is false then restore to “inpt” the value from the top of the stack. If the control register setting on top of the stack is not null then restore this value to “pcr”. Delete this top pair of values from the stack.

NOT Reverse the setting of “flag”. Restore to “inpt” the value from the top of the stack. If “flag” is now true then advance “inpt” to its successor position. Delete the top of the stack.

TRUE If “flag” is true then copy the address part A into “pcr”. Otherwise set “flag” true and restore to “inpt” the value from the top of the stack, leaving the stack unaltered.

FALSE If “flag” is false then copy the address part A into “pcr”.

FLAG If “flag” is true then copy the address part A into “pcr”. Otherwise set “flag” true.

STOP Display the value of “flag” and stop the machine.

The purpose of these nine functions will become clear as we describe their synthesis into complete programs. Briefly, however, CALL and RETURN are subroutine cue and link instructions which use the stack to administer the return addresses; TRUE, FALSE and FLAG are conditional jump instructions; MATCH, NEXT and NOT are input instructions; STOP is an output instruction. Before discussing the programming of the machine it will be helpful to describe informally the basis of the parsing process.

The fundamental operation is to test whether any portion of the input text immediately following the current point of scan satisfies the definition of the syntax structure which is currently of interest: if so the point of scan is advanced to the end of this portion. In making this test, each alternative is tested in turn in the order listed, and whichever is first satisfied is accepted. In testing an alternative, each element is tested in turn until either one is rejected or the list is exhausted.

As an illustration let us use the definitions

\[
\langle \text{digit} \rangle ::= \text{0} \uparrow \text{1} \uparrow \text{2} \uparrow \text{3} \uparrow \text{4} \uparrow \text{5} \uparrow \text{6} \uparrow \text{7} \uparrow \text{8} \uparrow \text{9}
\]

\[
\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \uparrow \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle
\]

of the ALGOL report to test whether the sequence \(\rightarrow 12\) is acceptable as an unsigned integer where we use \(\rightarrow\) to denote the point of scan. The process goes as follows:

1. ? \langle unsigned integer \rangle \rightarrow 12 ;
2. ? \langle digit \rangle \rightarrow do —
3. ? \langle digit \rangle \rightarrow do —
4. Reject “0” \rightarrow do —
5. ? \langle digit \rangle \rightarrow do —
6. Accept “1” \rightarrow 1 \rightarrow 2 ;
7. Accept \langle digit \rangle \rightarrow do —
8. Accept \langle unsigned integer \rangle \rightarrow do —

This is not what was intended: the leading digit has been accepted on its own, whereas we should have accepted the whole stream of digits up to the first non-digit. This is an example of what Metafile calls mis-ordered alternatives. Let us try again with

\[
\langle \text{unsigned integer} \rangle ::= \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle \uparrow \langle \text{digit} \rangle
\]

1. ? \langle unsigned integer \rangle \rightarrow 1 2 ;
2. ? \langle unsigned integer \rangle \rightarrow do —

246
Syntax-directed translator

This is no good because we are stuck in a loop. Metcalfe terms this a circular definition. Let us try

\( \langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \langle \text{unsigned integer} \rangle \uparrow \langle \text{digit} \rangle \)

Steps (1) to (7) are as above and we then continue

(8) \( \langle \text{unsigned integer} \rangle \quad 1 \rightarrow 2 \); (9) \( \langle \text{digit} \rangle \quad \text{do} \quad \langle \text{unsigned integer} \rangle \quad \text{do} \)
(10) \( ? \langle \text{digit} \rangle \quad \text{do} \)
(11) Reject "0" \quad \text{do} -
(12) \quad \text{do} -
(13) Reject "1" \quad \text{do} -
(14) \quad \text{do} -
(15) Accept "2" \quad 1 \rightarrow 2 ;
(16) Accept \langle \text{digit} \rangle \quad \text{do} -
(17) \langle \text{digit} \rangle \quad \text{do} -
(18) \langle \text{unsigned integer} \rangle \quad \text{do} -
(19) Test and reject each of ten digits
(20) \langle \text{digit} \rangle \quad 1 \rightarrow 2 ;
(21) ? \langle \text{digit} \rangle \quad \text{do} -
(22) \langle \text{digit} \rangle \quad \text{do} -
(23) Same as (19)–(38)
(24) Reject \langle \text{digit} \rangle \quad \text{do} -
(25) Reject \langle \text{unsigned integer} \rangle \quad \text{do} -
(26) Finally, let us examine the iterative—i.e. non-recursive—form
\( \langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle * \langle \text{digit} \rangle \)
(27) \langle \text{digit} \rangle \quad \text{do} -
(28) \langle \text{digit} \rangle \quad \text{do} -
(29) \langle \text{digit} \rangle \quad \text{do} -
(30) \langle \text{digit} \rangle \quad \text{do} -
(31) Try again \quad 1 \rightarrow 2 ;
(32) Accept \langle \text{digit} \rangle \quad \text{do} -
(33) \langle \text{digit} \rangle \quad \text{do} -
(34) \langle \text{digit} \rangle \quad \text{do} -
(35) Do not try again but accept * term \quad \text{do} -
(36) Accept \langle \text{digit} \rangle \quad \text{do} -
(37) Accept \langle \text{unsigned integer} \rangle \quad \text{do} -

We are now in a position to implement the parsing process on our computer. Each syntactic definition of the meta language is represented by a subroutine in the computer program. The location of the head of the subroutine corresponds to the name of the structure defined, and its instructions to the defining expression. These instructions are partitioned into sequences corresponding to each alternative. The function of each such sequence is to set the flag corresponding to the acceptability of that alternative. Pairs of sequences are separated by a TRUE instruction causing a conditional jump to the end of the subroutine where the link instruction RETURN is placed. Thus as soon as an acceptable alternative is found, the rest are skipped.

\[ \text{--- name} \]
\[ \text{[ALTERNATIVE]} \]
\[ \text{TRUE link} \]
\[ \text{[ALTERNATIVE]} \]
\[ \text{TRUE link} \]
\[ \ldots \]
\[ \text{[ALTERNATIVE]} \]
\[ \text{RETURN} \]

For each alternative, the program examines each element in turn. The group of instructions for each element sets the flag according to the acceptability of that element. Between pairs of such groups is placed a FALSE instruction causing a conditional jump to the end of the alternative. Thus if any element is rejected, the flag is set false and the rest are skipped.

\[ \text{[ELEMENT]} \]
\[ \text{FALSE L1} \]
\[ \text{[ELEMENT]} \]
\[ \text{FALSE L1} \]
\[ \ldots \]
\[ \text{[ELEMENT]} \]

In fact, if the element is of * type, we can omit the FALSE instruction following it since we know it is
always acceptable. Such a term is called passive as opposed to those which require a test and are called active. \( U \) is another passive element.

The instruction sequences for each type of element are shown diagrammatically below. They are reasonably self-evident.

\[
\begin{align*}
\text{CALL} & \quad \text{name} \quad \text{for} \langle \text{variable} \rangle \\
\text{MATCH} & \quad \text{const} \quad \text{for} \langle \text{constant} \rangle \\
\end{align*}
\]

The address part is the location of the head of the relevant subroutine. The address part is the constant under test.

\[
\begin{align*}
L2 & \rightarrow \\
\text{[ELEMENT]} & \quad \text{for} \ast \langle \text{element} \rangle \\
\text{FLAG} & \quad \text{L2} \\
\text{CALL} & \quad \text{for} \langle \text{expression} \rangle \\
\text{RETURN} & \quad \text{Note that the address part of \text{CALL} is null.} \\
\text{NEXT} & \quad \text{for} \ U \\
\text{CALL} & \quad \text{for} \neq \langle \text{element} \rangle \\
\text{NOT} & \quad \text{Note that the address part of \text{CALL} is null.}
\end{align*}
\]

Finally, the program is completed by placing at its head a call of the principal subroutine followed by a \text{STOP} order.

\[
\begin{align*}
\text{CALL} & \quad \text{principal} \\
\text{STOP} & \quad \\
\end{align*}
\]

We give two illustrations. First the program corresponding to the syntax

\[
\langle \text{digit} \rangle ::= 0 \uparrow 1 \uparrow 2 \uparrow = 3 \uparrow = 4 \uparrow = 5 \uparrow = 6 \uparrow = 7 \uparrow = 8 \uparrow = 9
\]

\[
\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \ast \langle \text{digit} \rangle
\]

\[
\begin{align*}
\text{CALL} & \quad \text{ui} \\
\text{STOP} & \quad \\
\end{align*}
\]

\[
\begin{align*}
\text{ui} & \quad \text{CALL} \ d \\
& \quad \text{FALSE} \ LB \\
& \quad \text{LC} \quad \text{CALL} \ d \\
& \quad \text{FLAG} \ LC \\
& \quad \text{LB} \quad \text{RETURN}
\end{align*}
\]

The reader may find it helpful to work through the operation of the program starting with the input pointer set at \( \rightarrow 1 \ 2 \); and convince himself that it does parallel the moves already outlined.

The second illustration is the program corresponding to the syntax of the meta language itself as set out at the end of Section 2. This program will check the validity of the rules of syntax of an arbitrary language when they are expressed in terms of the meta language.

\[
\begin{align*}
\text{CALL} & \quad \text{grammar} \\
\text{STOP} & \quad \\
\end{align*}
\]

\[
\begin{align*}
\text{grammar} & \quad \text{CALL} \quad \text{definition} \\
& \quad \text{FALSE} \ A1 \\
& \quad \text{CALL} \quad \text{A3} \\
& \quad \text{MATCH} \quad ; \\
& \quad \text{FALSE} \ A2 \\
& \quad \text{CALL} \quad \text{definition} \\
& \quad \text{RETURN} \quad \text{A2} \\
& \quad \text{FLAG} \quad \text{A3} \\
& \quad \text{RETURN} \\
\end{align*}
\]

\[
\begin{align*}
\text{definition} & \quad \text{CALL} \quad \text{variable} \\
& \quad \text{FALSE} \ B1 \\
& \quad \text{MATCH} \quad ; \\
& \quad \text{FALSE} \ B1 \\
& \quad \text{MATCH} \quad ; \\
& \quad \text{FALSE} \ B1 \\
& \quad \text{MATCH} \quad = \\
& \quad \text{FALSE} \ B1 \\
& \quad \text{CALL} \quad \text{expression} \\
& \quad \text{RETURN} \quad \text{B1} \\
\end{align*}
\]

\[
\begin{align*}
\text{expression} & \quad \text{MATCH} \quad \langle \text{digit} \rangle \langle \text{digit} \rangle \\
& \quad \text{FALSE} \ C1 \\
& \quad \text{CALL} \quad \text{C2} \\
& \quad \text{MATCH} \quad \rangle \\
& \quad \text{NOT} \quad \text{C1} \\
& \quad \text{FLAG} \quad \text{C2} \\
& \quad \text{MATCH} \quad \rangle \\
\end{align*}
\]

\[
\begin{align*}
\text{variable} & \quad \text{MATCH} \quad \langle \text{digit} \rangle \langle \text{digit} \rangle \\
& \quad \text{FALSE} \ C1 \\
\end{align*}
\]

\[
\begin{align*}
\text{alternative} & \quad \text{CALL} \quad \text{alternative} \\
& \quad \text{false} \ D1 \\
\end{align*}
\]

\[
\begin{align*}
\text{alternative} & \quad \text{MATCH} \quad \langle \text{digit} \rangle \langle \text{digit} \rangle \\
& \quad \text{FALSE} \ D2 \\
& \quad \text{CALL} \quad \text{alternative} \\
\end{align*}
\]

\[
\begin{align*}
\text{RETURN} & \quad \text{D2} \\
\text{FLAG} & \quad \text{D3} \\
\text{RETURN} & \quad \text{D1}
\end{align*}
\]
Syntax-directed translator

<table>
<thead>
<tr>
<th>alternative, E1</th>
<th>CALL</th>
<th>element</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FLAG</td>
<td>E1</td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>element</th>
<th>CALL</th>
<th>variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TRUE</td>
<td>F1</td>
</tr>
<tr>
<td></td>
<td>CALL</td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>F1</td>
</tr>
<tr>
<td></td>
<td>MATCH</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>F2</td>
</tr>
<tr>
<td></td>
<td>CALL</td>
<td>element</td>
</tr>
<tr>
<td>F2</td>
<td>TRUE</td>
<td>F1</td>
</tr>
<tr>
<td></td>
<td>MATCH</td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>F3</td>
</tr>
<tr>
<td></td>
<td>CALL</td>
<td>expression</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>F3</td>
</tr>
<tr>
<td></td>
<td>MATCH</td>
<td>)</td>
</tr>
</tbody>
</table>

| F3               | TRUE | F1       |
|                  | MATCH | U       |
|                  | TRUE | F1       |
|                  | MATCH | ≠       |
|                  | FALSE | F4      |
|                  | CALL | element  |

| F1, F4           | RETURN |        |

| constant          | MATCH | =       |
|-------------------| FALSE | G1      |
| NEXT              | RETURN |        |

the end of the previous section automatically from the meta grammar given at the end of Section 2.

There is an evident similarity between the two texts. Each syntactic structure in the meta grammar gives rise to a characteristic sequence in the program. This suggests that the parser output should be caused by output instructions which are obeyed at points in the program closely following recognition of the corresponding structures in the input text. Provision must be made for rearranging the output stream. For example, the symbol ≠ generates an instruction NOT which follows the instructions generated by the structure <element> which it precedes. In the next section we describe Metcalfe's two-stage mechanism for achieving this.

5. Parser output and the editor machine

We suppose that the parser machine is provided with an output facility in addition to the flag indicator. This consists of an ordered sequence of positions. This can be scanned in either direction, the current point of scan being recorded in a register "outpt" (output pointer). Each position may record a symbol or other single item. An output instruction loads the position indicated by the current point of scan with an indicated item and advances "outpt" to the successor position. Certain instructions cause "outpt" to be restored to an earlier value and have the effect of erasing all intermediate items from the output stream. Initially the output stream is empty and "outpt" indicates the leading position.

The basic output instructions of the parser are as follows

| PRINT | Output the symbol in the address part of the instruction. |
| COPY  | Output the symbol just read in—i.e. the one immediately preceding that specified by "inpt" as the next to be read. |
| NULL  | Output a special null symbol. Note that this occupies a space in the output stream and causes "outpt" to be advanced—unlike a null input. |
| EDIT  | Output the edit code specified by the address part of the instruction. |

We distinguish between symbols and edit codes on the parser output. PRINT, COPY and NULL output symbols. Before discussing this distinction, we list the effect of the introduction of the output stream upon the other instructions of the parser. The parser stack is augmented at each level by a location which records a value of "outpt". These are processed as follows

| CALL | Record the value of "outpt" in the stack. |
| RETURN | If "flag" is false then restore to "outpt" the value from the top of the stack and delete it from the stack. |
| NOT  | Restore to "outpt" the value from the top of the stack and delete it from the stack. |

4. Semantics

Thus far we have considered only the structure and analysis of the grammar of a language: in this section we turn to its meaning or interpretation. Putting this objectively, we shall now consider how to provide our parsing machine with more extensive output facilities. The sorts of output that we may require are as flexible as the sorts of input and we need similar means of specifying them. Thus our meta-semantic language must be extended to encompass a meta-semantic language. We shall henceforth call it a meta language and understand that it will specify the structure of both the input and output streams. This extension to the meta language will of course be reflected in an extension to the parsing machine itself. In describing these extensions it is convenient to have a specific problem in mind to illustrate the motivation.

The specific problem is to enable the extended machine to read a description of the syntax and semantics of an arbitrary language written in the extended meta language, and to produce as output the corresponding program for the extended machine which will read a message in the specified language and produce the indicated output. Thus we wish to produce a compiler which will spare the user the chore of programming in the machine code of the parser machine. To be yet more specific, we wish to produce the program given at
TRUE If "flag" is false restore to "outpt" the value from the top of the stack, leaving the stack unaltered.

It is seen that the effect of these added operations is to erase any output that may have taken place in the course of an unsuccessful recognition attempt.

We come next to the interpretation of the parser output stream. This consists of mixed symbols and so-called edit codes. In fact this stream acts as combined data and program for a second computer, the editor machine. The symbols are the data elements and the edit codes the instructions for this second machine.

The editor machine has its own independent push-down stack. At each level the elements of the stack are variable length ordered sequences or strings of symbols. There is also an output device capable of emitting an ordered stream of symbols.

The operation of the machine is initiated by the execution by the parser machine of an EDIT W instruction where W is one of the edit codes, as will be described. At this stage the parser-editor interface contains a stream of mixed symbols and edit codes terminated by the W which has just been output by the parser. The editor stack is emptied, and the input stream is scanned one element at a time from its head to the final W. At each stage, if the element is a symbol it is placed on top of the editor stack, pushing down the previous entries. If the element is an edit code the action depends upon the particular code. The basic codes are

X Exchange the entries in the top two levels of the stack.
C Combine the two top entries in the stack into one by attaching the head of the top string to the tail of the next. Nest up the rest of the stack.
W Output the elements in the editor stack in sequence level by level from the bottom up and from head to tail within each level, ignoring the null symbol whenever it appears.

For illustrative purposes on paper a convenient linear representation of the state of the stack is given by writing the elements of each level in order with the head on the left. The levels are separated by commas (say) and are ordered such that the bottom level is on the left. The code W then produces an output obtained by reading the symbols from left to right omitting the level separators. C merely deletes the right-most comma. Thus in order to change toga into goat we make up an input stream by inserting edit codes as follows:

t o g X C a C X W

The state of the stack after each step is as follows:

<table>
<thead>
<tr>
<th>input</th>
<th>stack</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t, o</td>
</tr>
</tbody>
</table>

6. A meta-semantic language

In this section we link up the production of output from the parser with the meta language by defining additional meta symbols which correspond to the output instructions in the parser program. The correspondence is as follows

<table>
<thead>
<tr>
<th>meta language</th>
<th>program</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b . . . z</td>
<td>PRINT a</td>
</tr>
<tr>
<td></td>
<td>PRINT b</td>
</tr>
<tr>
<td></td>
<td>. . .</td>
</tr>
<tr>
<td></td>
<td>PRINT z</td>
</tr>
</tbody>
</table>

where a b . . . z is any sequence of symbols

| K      | COPY   |
| O      | NULL   |
| W      | EDIT W |
| C      | EDIT C |
| X      | EDIT X |

As an illustration, we consider the translation into Reverse Polish notation of a simple form of arithmetic expression in three variables p, q, and r.

\[
\begin{align*}
\langle\text{data}\rangle & ::= \langle\text{arithmetic expression}\rangle W ; \\
\langle\text{arithmetic expression}\rangle & ::= \langle\text{term}\rangle \times \langle\text{adding operator}\rangle K \langle\text{term}\rangle X C C ; \\
\langle\text{adding operator}\rangle & ::= + \uparrow = - ; \\
\langle\text{term}\rangle & ::= \langle\text{factor}\rangle \times \langle\text{multiplying operator}\rangle K \langle\text{factor}\rangle X C C ; \\
\langle\text{multiplying operator}\rangle & ::= \times \uparrow = / ; \\
\langle\text{factor}\rangle & ::= \langle\text{variable}\rangle K \uparrow = \langle\text{arithmetic expression}\rangle = ; \\
\langle\text{variable}\rangle & ::= p \uparrow = q \uparrow = r
\end{align*}
\]

The corresponding program begins as follows

<table>
<thead>
<tr>
<th>data</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td>CALL ae</td>
</tr>
</tbody>
</table>

The reader is advised to complete the program and to verify that from an input \(p + p/(q + r \times p) - q\); it will produce the output \(p p q r p x + + q -\).
Syntax-directed translator

| FALSE  | B2 |
| COPY   |    |
| CALL   | term |
| FALSE  | B2 |
| EDIT   | X  |
| EDIT   | C  |
| EDIT   | C  |
| B2 RETURN |    |
| FLAG   | B3 |
| B1 RETURN |   |

7. Integers and labels

In all our examples of programs for the parser machine we have used the mnemonic codes for the function parts of instructions and have made free use of symbolic labels in the address parts wherever one instruction referred to the location of another. If we are to succeed in compiling such programs automatically from their meta language form, we need to be more explicit over defining the structure of a program.

We distinguish between the stored form of a program and the so-called assembly language form. These differ in that the assembly language permits symbolic labels and addresses whereas addresses are held in absolute form in the stored form of the program. We shall compile from the meta language form into the assembly language form. For the present we discuss only this stage, leaving the description of the assembly process to a subsequent section.

The assembly language form of a program is a stream of signed integers. The syntax may conveniently be expressed in terms of the meta language as follows, where explicit integers are within quotation marks indicating for example that “999” is one symbol rather than a sequence of three digits.

\[
\begin{align*}
\langle \text{program} \rangle & ::= * \langle \text{routine} \rangle = \text{“999”}; \\
\langle \text{routine} \rangle & ::= * \langle \text{instruction} \rangle = \text{“0”}; \\
\langle \text{instruction} \rangle & ::= * \langle \text{label} \rangle \langle \text{function} \rangle \langle \text{address} \rangle; \\
\langle \text{label} \rangle & ::= \langle \text{internal reference} \rangle \langle \text{external reference} \rangle; \\
\langle \text{internal reference} \rangle & ::= \langle \text{integer} x, -999\leq x \leq -1 \rangle; \\
\langle \text{external reference} \rangle & ::= \langle \text{integer} x, x < -1000 \rangle; \\
\langle \text{function} \rangle & ::= \langle \text{integer} x, 1 \leq x \leq 998 \rangle; \\
\langle \text{address} \rangle & ::= \langle \text{integer} \rangle
\end{align*}
\]

The convention is that a negative integer represents a label or symbolic address. The mnemonic codes for the functions are replaced† by integers in the range 1 to 998. Therefore if when reading an instruction we encounter “0” or “999” before a function part, we know that we have reached the end of a routine or of the complete program respectively. An instruction may have several labels, and these precede the function part. A negative address part denotes a symbolic address whose absolute form will be the location in store of the instruction with the corresponding label. It is assumed that all absolute addresses will be strictly positive so that zero denotes a null address in a CALL or other instruction such as RETURN or COPY etc. Further, each symbol of the source stream and output stream must be identified by a unique integer so that the addresses of MATCH and PRINT instructions shall be determined. In our case we have used the standard internal KDF9 tape-code representation, increased by 128 for underlined symbols.

It is also necessary to specify the representation in terms of integers of the parser-editor interface.

\[
\begin{align*}
\langle \text{interface stream} \rangle & ::= * \langle \text{symbol} \rangle \uparrow \langle \text{edit code} \rangle; \\
\langle \text{symbol} \rangle & ::= \langle \text{integer} x, x > 0 \rangle; \\
\langle \text{edit code} \rangle & ::= \langle \text{integer} x, x < 0 \rangle
\end{align*}
\]

The mnemonic edit codes are replaced† by negative integers. The null symbol output from the parser by the NULL instruction is represented by the integer 500 000 000 000.

The distinction between an integer and the digits of its representation is an important one and enters the meta language in two ways. Firstly the compilation of PRINT instructions is extended to permit general integers as well as symbols to be specified as address parts in the meta language. Thus \([12]\) is compiled as PRINT 12 whereas \([12]\) is compiled as PRINT 17 followed by PRINT 18 where, as it happens, 17 and 18 are the internal representations of the digits 1 and 2 in the KDF9 tape code. In general the syntax of the sequence permitted within square brackets is

\[
* ( (= (\langle \text{integer} \rangle = ) \uparrow \neq = ) )
\]

where \langle \text{integer} \rangle ::= \langle \text{digit} \rangle \ast \langle \text{digit} \rangle.

The second reference to integers in the meta language is in the action of two new edit codes \(V\) and \(N\). These compile into the instructions EDIT \(V\) and EDIT \(N\) (or rather their numerical equivalents*). Their effects are as follows when interpreted by the editor:

- **V** On the assumption that the top level of the editor stack contains a string of digits, replace the string by the integer which it represents in the decimal scale, the digit at the head being the most significant.
- **N** On the assumption that the top level of the editor stack contains a single integer, change the sign of (i.e. negate) this integer.

The use of the \(V\) code is seen in the full specification of an integer in the meta language.

\[
\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \ast \langle \text{digit} \rangle \text{K} \circ \langle \text{digit} \rangle \text{K} \circ \langle \text{digit} \rangle \text{K} \circ V
\]

As successive digits are recognized they are copied and combined with the string already read, and finally the whole string is replaced by the value of the integer.

We are now ready to discuss the generation of negative integers as labels for cross references within the parser.
program. There are two sorts of label which we have called internal and external. Internal labels are for local use within a routine and external labels are for global use between routines. It will be seen from the syntax of a compiled program that internal labels have values \(x\) in the range \(-999 < x < -1\) and external ones in \(x < -1000\).

External labels correspond to the names of variables in the meta language or, equivalently, to the locations of the heads of routines in the parser program. This correspondence is set up by means of a further edit code, \(L\) in the meta language which compiles as EDIT \(L\). This code is associated with additional storage in the editor machine. Independently of the editor stack is provided a list called “name”. This is a list of sub-lists. The first element of each sub-list is an external label \(< -1000\): the second element is a list of symbols making up a variable name. The list “name” is initially empty. There is a register “index” initially set to \(-1000\). The action of the code is as follows:

\(L\) On the assumption that the top level of the editor stack contains a string of symbols forming a variable name, scan the list “name” for this name. If found, replace the entry in the stack, by the corresponding external label. If not found, enter the name from the stack in the list “name” with the current value of “index” as external label and replace the stack entry as before—also reduce the value of “index” by unity.

The use of the \(L\) code is seen in the full specification of a variable in the meta language.

\[
\langle\text{variable}\rangle ::= = \langle U K \ast (\neq \Rightarrow K C) \Rightarrow L\rangle
\]

The syntax here differs slightly from that in Section 2 but the change is not important.

For generating internal references within routines, we introduce additional functions into the parser instruction code, together with associated hardware—a register “tag” to hold an integer, and a push down list “label” whose elements are values of “tag”. The extra instructions are as follows:

- SETLAB: Set “tag” = 1 and clear “label” list.
- INCLAB: Add current “tag” setting to “label” list and increase “tag” setting by unity.
- LABEL: Output the top entry in the “label” list, followed by the \(N\) edit code.
- DECLAB: Remove top entry in “label” list and if “flag” is \texttt{false} copy into “tag”, otherwise discard.
- RESETLAB: Clear “label” list.

For convenience, these five instructions are in fact selected in the program by the address part of a function “reference”.\(^\dagger\) SETLAB and RESETLAB should be obeyed at the beginning and end of an instruction sequence outputting a routine. INCLAB and DECLAB should be obeyed at the beginning and end of an instruction sequence containing \(LABEL\) instructions corresponding to one particular label value. \(LABEL\) is obeyed whenever a label is to be output. Note that the label value cannot be output from the parser directly as a negative number else the editor would treat it as an edit code. It is the purpose of the \(N\) code to get round this difficulty. The requirement for one of these instructions in the program is denoted in the meta language by the construction \(= R \langle\text{digit}\rangle\) as will be seen in the full specification of the meta grammar in Appendix 2. These five instructions are sufficient to ensure that the correct value for a label is output even though other values may intervene in the compiled program.

8. The use of markers

Metcalfe describes a further facility which can be helpful in economizing the description of a process in the meta language. As a simple illustration, consider the syntax

\[
\langle\text{item}\rangle ::= = a \uparrow = b \uparrow = c ;
\langle\text{group}\rangle ::= = \langle\text{item}\rangle \langle\text{item}\rangle
\]

and suppose that the required output when parsing a group is “d” if the first item of the group is “b”, otherwise nothing. It is no good changing \(\langle\text{item}\rangle\) to

\[
\langle\text{item}\rangle ::= = a \uparrow = b[d] \uparrow = c
\]

because this would cause an output whenever “b” appears in either first or second position in a group. The obvious solution is unattractive, namely

\[
\langle\text{first item}\rangle ::= = a \uparrow = b[d] \uparrow = c;
\langle\text{second item}\rangle ::= = a \uparrow = b \uparrow = c;
\langle\text{group}\rangle ::= = \langle\text{first item}\rangle \langle\text{second item}\rangle
\]

Our preferred solution based on Metcalfe’s facility is

\[
\langle\text{item}\rangle ::= = a \uparrow = b \text{M1} \uparrow = c;
\langle\text{group}\rangle ::= = \langle\text{item}\rangle \text{T1} \langle\text{[d]}\rangle \langle\text{item}\rangle
\]

and in the remainder of this section we shall describe in detail how it works.

The parser is provided with three new instructions and some extra hardware. There is a register “mark” which can hold a sequence of 39 binary digits. (This is the number of bit positions in a KDF9 ALGOL integer.) The parser stack is augmented at each level so that the setting of “mark” can be recorded together with “inpt”, “outpt” and “pcr”.

Existing functions are modified as follows:

- CALL: Add “mark” into top of stack and set zeros in “mark”.
- TRUE: If “flag” is \texttt{false} reset zeros in “mark”.
- RETURN: Remove record from head of stack and copy into “mark”.
- NOT: The three new functions are given below

\[\text{MARK}\]

Insert a 1 digit, in the bit position specified by the address part, in the record at the head of the stack.
SELECT  Copy the address part into a temporary location called “hold”.
TEST   Transfer control (i.e. copy address part into “pcr”) if the digit position of “mark” specified by “hold” contains 0.

We see that each activation of a routine has a set of marker bits associated with it in “mark”. This record is clear at entry to the routine. It may be varied by MARK instructions within routines which it calls but is reset to zeros whenever an alternative is rejected, ready for a fresh start on the next alternative. The current values may be tested by SELECT and TEST instructions. Similarly the routine may itself contain MARK instructions which set values in the record associated with the routine which has called it.

In the meta language a mark is made by a sequence $= M \langle \text{integer} \rangle$. Thus $M3$ compiles as MARK 3. The analogous testing sequence has the syntax

$= T \langle \text{integer} \rangle = (* \langle \text{passive} \rangle =)$

where $\langle \text{passive} \rangle$ is used in the sense of Section 3 to denote an element for which “flag” is known to be true. Such a sequence is compiled as

```
| SELECT  | <integer> |
| TEST    | AAA       |
|         | [ \ldots \langle \text{passive} \rangle \ldots ] |
| AAA     |
```

so that the instructions generated by $* \langle \text{passive} \rangle$ are skipped unless the appropriate marker bit has been set.

To complete our illustration, the compiled form of the example at the head of the section is given below:

```
| CALL group |
| STOP       |
| item       |
| MATCH a    |
| TRUE A1    |
| MATCH b    |
| FALSE A2   |
| MARK l     |
| A2         |
| TRUE A1    |
| MATCH c    |
| A1         |
| RETURN     |
| group      |
| CALL item  |
| FALSE B1   |
```

It should be noted that this implementation differs from that of Metcalfe in that TEST does not disturb “flag”.

9. Miscellaneous facilities

The full specification of the compiler, written in the meta language itself and so capable of compiling itself, is given in Appendix 2. It contains a few features which have not been mentioned.

The description of the W edit code was incomplete. When the edit stack has been printed out, it and the interface are cleared and the operations of the parser are resumed at the instruction following the EDIT W which activated the editor. It will be seen from Appendix 2 that the compilation process is segmented into routines in this way, thus economizing in storage.

It will be seen also that a grammar is formally delimited by the sequence BEGIN and END and that following the initial BEGIN is the name of the principal definition. It is this which causes compilation of the program call sequence.

Comment facilities analogous to those of ALGOL are also included. Anything which fails to satisfy the definition of a definition is treated as comment and ignored up to the next semi-colon separator or the terminal END.

10. An implementation in ALGOL

The particular implementation to be described uses list processing techniques throughout. An earlier version based on separate ALGOL arrays failed because dynamic reallocation of storage was necessary but impossible.

Storage for lists consists of the integer arrays $H, T [i : \text{storesize}]$ where storesize is an input parameter. In general a list element with address $i$ consists of the pair $H[i], T[i]$ containing respectively the address of the subsequent element and the element itself.

A list has the following structure

```
\[ v \rightarrow H_1 \rightarrow x_1 \rightarrow H_2 \rightarrow x_2 \rightarrow 0 \wedge x_n \]
```
where \( v \) is the list identifier, \( (x_1, x_2 \ldots x_n) \) are the list elements any of which may be a list identifier, and where the first pseudo-element is the address of the tail. The tail element has a zero H-part. A null or empty list has the structure

\[
\begin{array}{c}
\vspace{1em}
\end{array}
\]

Storage allocation is handled in a conventional manner by means of a list of available space “lavs”, which has no initial pseudo-element. An ALGOL procedure “initialise” assigns the whole list store to “lavs” initially. The elements of “lavs” are set to 500 000 000 001 to facilitate monitoring the storage area.

Locations may be withdrawn from “lavs” by the procedure “take (i)” which assigns the freed address to the integer \( i \). Locations \( i \) no longer required must be restored to “lavs” by the procedure “put (i)”. There is no automatic garbage collection.

A number of general purpose list-processing procedures are declared. In particular “push \((x, v)\)” adds the element \( x \) to the list \( v \) and “pull \((x, v)\)” assigns to \( x \) the head element of the list \( v \).

The stored form of a parser program is a list “program” whose elements are the instructions of the program. An instruction is represented by the ALGOL integer

\[
100 000 \times \text{<function>} + \text{<address>}
\]

This decimal packing was chosen in order to facilitate monitoring.

The assembly of a program in the store is performed one element at a time by the procedure “assemble \((x)\)”. Successive arguments are the successive signed integers of the assembly language form as set out in Section 7. The process makes use of global lists “target”, “int ref” and “ext ref” which are initially empty. The program is assembled in “target” one instruction at a time, symbolic addresses being entered as zero in “target” but having their values (negative integers) and locations in “target” entered in either “int ref” or “ext ref” according as they are internal references or external references. Similarly labels of instructions and their location in “target” are entered into “int ref” or “ext ref”.

Both “int ref” and “ext ref” have the same structure. Their elements are themselves sublists. The head element of a sublist is a label \( l \), i.e. a negative integer. The next element is the address \( t \) in “target” of the instruction labelled \( l \). The remaining elements are the addresses \( f_1, f_2 \ldots \) in “target” of the instructions whose symbolic address part was \( l \).

Whenever the argument of “assemble” is recognized to be the zero which terminates a routine, the list “int ref” is scanned and for each sublist the \( t \) element is added into each of the \( f_1, f_2 \ldots \) locations in “target”. “int ref” is then cleared ready for the next routine. Similarly the list “ext ref” is scanned when the argument “999” which terminates the program is recognized.

Thus at the end of the assembly stage the assembled program is in the list “target”. The first action of the ALGOL program after reading “storesize” is to read a parser program in assembly language form: this will normally be the compiler program. This program is assembled as it is read and the list “target” is transferred to “program” and obeyed. In the event that this program is the compiler, its output from the editor is the compiled program in assembly language form. This will be punched onto paper tape for subsequent use and it may also optionally be processed by “assemble” to yield the stored form in “target” which is then transferred to “program” and itself obeyed. On this second occasion the parser program will not normally produce a program as output, and normally the output from the editor will be printed and the ALGOL program will terminate.

A complete data tape for the ALGOL program has the following structure

\[
\langle \text{storesize} \rangle \langle \text{device number} \rangle
\]

\[
\langle \text{assembly language program} \rangle
\]

\[
\ast \langle \text{route} \rangle \langle \text{device number} \rangle \langle \text{input stream} \rangle \times
\]

Here \( \langle \text{device number} \rangle \) is a KDF9 ALGOL convention: 70 for line printer, 10 for paper tape punch. The first occurrence is required in case of error messages during assembly of the initial program. The form of \langle \text{assembly language program} \rangle \ is as given in Section 7. Individual integers for input to KDF9 ALGOL have the form

\[
(= \downarrow \uparrow \uparrow \downarrow \uparrow)(\text{digit})* (\text{digit})=;
\]

The term \langle \text{route} \rangle is an integer which selects the course of the calculation as follows:

2. The output is a stream of signed integers: the assembly language form of a parser program. No recycling.
3. As for 2 but with recycling.

Each time recycling is called for, a fresh set of \langle \text{route} \rangle \langle \text{device number} \rangle \langle \text{input stream} \rangle \times is required on the data tape. The \langle \text{input stream} \rangle is a stream of symbols with the appropriate structure to be accepted by the parser program which reads it. A final terminating character \( \times \) (underlined multiply) is required.

The grammar of the meta language is given in Appendix 2. This was initially itself compiled by hand into assembly language form (405 instructions) and punched manually onto paper tape. This was checked by compiling itself on KDF9.
Appendix 1

Internal representation of parser functions and edit codes

1 CALL
2 RETURN
3 TRUE
4 FALSE
5 FLAG
6 NOT
7 MARK
8 SELECT
9 TEST
10 reference

11 MATCH
12 NEXT
13 COPY
14 NULL
15 EDIT

16 PRINT
17 STOP

Note: The code printed by the EDIT function is the negated address part.

11. References