The implementation of syntax analysis using ALGOL, and some mathematical applications

By Eric Foxley and Peter King*

This paper describes the development of an ALGOL program to perform syntax analysis, using a set of syntax definitions supplied in a notation similar to Backus Normal Form. Five applications of syntax analysis are then discussed, including algebraic differentiation and compilation.

When the Department of Mathematics at the University of Nottingham initiated a postgraduate course in computing, it was felt that teaching the principles of compiling could be done more satisfactorily if a "Compiler Compiler" program of the type implemented on Atlas by R. A. Brooker (1963) was available. Since this University's computing service was transferred from a data link with Manchester University's Atlas to its own English Electric KDF9, no such program was available. It was therefore decided to attempt an ALGOL program to demonstrate some of the main principles.

Any program of this type consists of two main parts, one to perform syntax analysis on the source program, and the other to use the results of the analysis to perform compilation. The development of a syntax analysis procedure is described below. However, during the development of this part of the program, many applications other than compilation became apparent, and this paper describes a selection of applications, only touching briefly on compilation. It is hoped to produce a further paper giving fuller details of compilation.

Syntax analysis

Any source text which is to be investigated using syntax analysis must be a phrase structure language. The most commonly used method of describing the syntax of phrase structure languages at present is known as "Backus Normal Form". Examples of definitions are

Example 1

⟨digit⟩ ::= 1|2|3|4|5|6|7|8|9|0

meaning

"a (phrase of type) digit is defined as (the symbol) 1 or (the symbol) 2 or . . . or (the symbol) 0".

Example 2

⟨unsigned integer⟩ ::= ⟨digit⟩|⟨unsigned integer⟩⟨digit⟩

meaning

"an unsigned integer is either a digit, or an unsigned integer followed by a digit".

Example 3

⟨integer⟩ ::= ⟨unsigned integer⟩|+⟨unsigned integer⟩|
−⟨unsigned integer⟩

meaning

"an integer is an unsigned integer, preceded by either a + or a − or no sign".

Thus the symbol "::=" means "is defined as", the symbol "|" means "or", and a name within the brackets ⟨ and ⟩ represents a phrase class.

Note that the definition of ⟨unsigned integer⟩ is recursive, and many alternative definitions are possible, for example

⟨unsigned integer⟩ ::= ⟨digit⟩⟨unsigned integer⟩|⟨digit⟩

If a source string satisfies the ith of the possible alternative definitions of a certain phrase, we will say that it is of category i. Thus in our first phrase definitions above, +26 is an integer of category 2, 26 is an integer of category 1 and an unsigned integer of category 2 consisting of an unsigned integer of category 1 and a digit of category 6.

In anticipation of typing these phrase definitions on a Flexowriter, we are using the symbols $<$ and $>$ instead of ⟨ and ⟩ as in standard Backus Normal Form, since the latter are not available on the Flexowriter. Also, in the spirit of ALGOL, we ignore all layout characters in the definitions. We therefore need a visible terminator, and will use a semi-colon from now on.

Further examples

⟨operator⟩ ::= +|−|.|/;
⟨variable⟩ ::= x | y | z;
⟨operand⟩ ::= ⟨variable⟩|(⟨expression⟩);
⟨expression⟩ ::= ⟨operand⟩|⟨operand⟩⟨operator⟩
⟨operand⟩;

Note that recursion still exists, but not within a single definition. Expressions satisfying this syntax are

$$x + y$$
$$(x\,.\,y)/z$$
$$((x + y)\,.\,z) + (x - y)$$

but the following expressions do not satisfy the

* *Department of Mathematics, The University, Nottingham.*

Downloaded from https://academic.oup.com/comjnl/article/10/4/325/463854 by guest on 19 April 2024

definitions:

$x \,.\, -y$
$x + y + z$
$-z$
$2 \,.\, x + y$
$xy + z$

In the ALGOL 60 Revised Report (1963) the definition of an identifier is

⟨letter⟩ ::= $a|b|c|d|$ . . . . $|z|A|B|$ . . . . $|Z$;
⟨identifier⟩ ::= ⟨letter⟩|⟨identifier⟩⟨letter⟩|
⟨identifier⟩⟨digit⟩;

A definition of an ALGOL variable is

⟨variable⟩ ::= ⟨simple variable⟩|
⟨subscripted variable⟩;
⟨simple variable⟩ ::= ⟨identifier⟩;
⟨subscript list⟩ ::= ⟨arithmetic expression⟩|
⟨subscript list⟩, ⟨arithmetic expression⟩;
⟨subscripted variable⟩ ::= ⟨identifier⟩
[⟨subscript list⟩];

The use of metasymbols in the source string

So far we have used the metasymbols

⟨ ⟩ | ; and ::=

We therefore cannot use this system to describe source statements including these symbols, since to allow us to write $|a|$ for the modulus of a we would require a definition

⟨primary⟩ ::= ⟨variable⟩|(⟨expression⟩)||⟨variable⟩|;
which is obviously open to mis-interpretation.

To overcome this difficulty (which will become more significant as we introduce more metasymbols) we use the notation ⟨|⟩ in phrase definitions to represent the symbol | in a source string, ⟨⟨⟩ to represent ⟨, ⟨;⟩ to represent ; , etc. A list of statements separated by semi-colons can now be defined using

⟨statement list⟩ ::= ⟨statement⟩⟨;⟩⟨statement list⟩|
⟨statement⟩;

and a relation operator as

⟨relation operator⟩ ::= =|≠|⟨⟨ ⟩|⩽|⟨ ⟩⟩|⩾;

Similarly, if the phrase definitions are not to be layout sensitive, special notations are necessary to indicate the presence of spaces, tabs and newlines in the source string, if it is to be layout sensitive.

Outline of a method for syntax analysis

We require first a means of storing a set of phrase definitions, and secondly a means of searching the stored definitions and comparing them with a given source string.

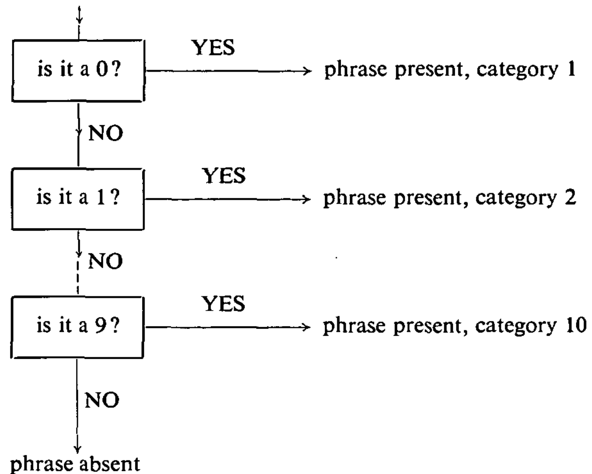
To search for a phrase type "digit", defined as

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9;

we search by comparing the given string (a single character in this case) initially with the character 0, then if it is not the same, comparing it with the character 1, and so on until we find either

(*a*) it agrees with one of the given alternative characters, or (*b*) all the alternatives have been tried, and none agrees.

The flow diagram is



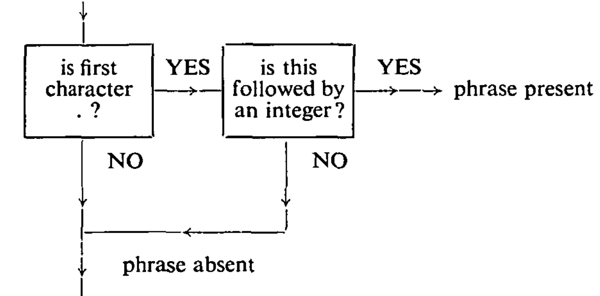
Similarly, with the definition

⟨type⟩ ::= **real**|**integer**|**Boolean**;

the flow diagram is shown in **Fig. 1** overleaf.

A definition involving another type of phrase is

⟨decimal part⟩ ::=. ⟨integer⟩



This involves searching for an integer during the search for a decimal part. This is easily catered for by using a recursive procedure for the searching.

Requirements of phrase definitions

The standard definitions from the ALGOL report are no longer suitable if searching is to be performed as above, since the following two failures can easily occur.

(1) *Infinite recursion*

The definition

⟨*u*/*s* integer⟩ ::= ⟨*u*/*s* integer⟩ ⟨digit⟩| . . . .

will cause continuous searching for an unsigned integer in a closed loop.

Downloaded from https://academic.oup.com/comjnl/article/10/4/325/463854 by guest on 19 April 2024

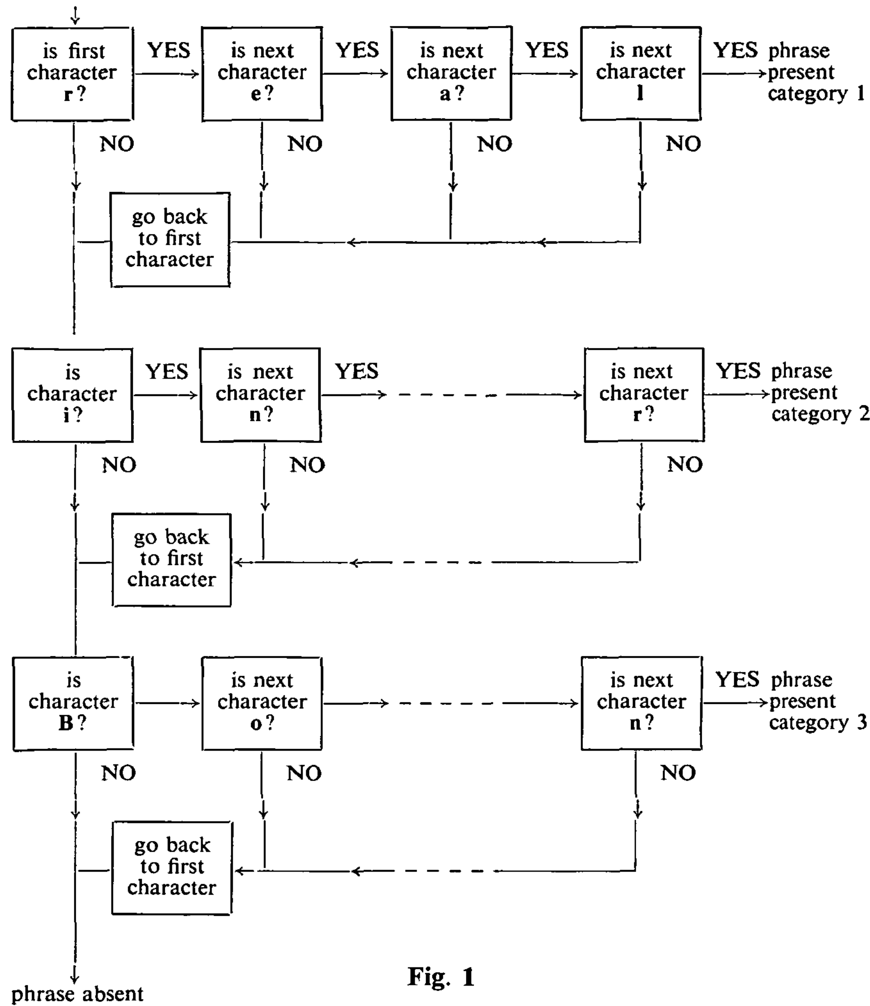


Fig. 1

Such loops may also occur between two or more definitions, i.e.

$$\langle a\rangle ::= \langle b\rangle \ldots.$$
$$\langle b\rangle ::= \langle a\rangle \ldots.$$

The necessary conditions for avoiding such loops are that

(*a*) no definition must start with the phrase it is defining.

(*b*) no set of definitions (say $P_1, \ldots, P_n$) must exist such that one of the alternative definitions of P_1 starts with P_2, P_2 with $P_3, \ldots, P_n$ with P_1.

These conditions are known as "cyclic non-nullity", and require that each time a loop in a rule-chain is traversed in analysing a source string, at least one element is recognized in the source string.

(2) *Non-maximum search*

If we define

⟨*u*/*s* integer⟩ ::= ⟨digit⟩|⟨digit⟩⟨*u*/*s* integer⟩;

(which avoids infinite recursion) the presence of an integer of several digits will not be found: given the string 326, the method of searching described above will find the digit 3 and exit. The alternative definitions must therefore be arranged such that they are in order of decreasing size of phrase which may be found.

Hence we must define an unsigned integer as

⟨*u*/*s* integer⟩ ::= ⟨digit⟩⟨*u*/*s* integer⟩|⟨digit⟩;

and a variable either as

⟨variable⟩ ::= ⟨subscripted variable⟩|⟨simple variable⟩;

or as

⟨variable⟩ ::= ⟨identifier⟩⟨parameter part⟩;
⟨parameter part⟩ ::= [⟨subscript list⟩]|⟨empty⟩;

For identifiers we use

⟨identifier⟩ ::= ⟨letter⟩⟨rest of identifier⟩;
⟨rest of identifier⟩ ::= ⟨letter⟩⟨rest of identifier⟩|
⟨digit⟩⟨rest of identifier⟩|
⟨empty⟩;

Notice that our definitions of phrases such as ⟨integer⟩

Downloaded from https://academic.oup.com/comjnl/article/10/4/325/463854 by guest on 19 April 2024

are now the reverse of the corresponding definitions in the ALGOL 60 Revised Report. If our "search" procedure were altered to scan both the definitions and the source string from the end forwards, we could use most of the standard ALGOL Report definitions.

Implementation using ALGOL

We assume that each typed symbol can be read in by the computer to produce a unique code number (the "input code" of the symbol) by an instruction such as

$$i := readch;$$

The actual code numbers of each symbol in any particular input system will be irrelevant, since the definitions and the source string will both be read into the computer using this procedure, so that comparing characters in the definitions with characters in the source string is independent of their numerical values. The only assumption we will make is that the code representations of the digits 0 to 9 are consecutive.

The IFIP ALGOL input/output procedures include

i := *insymbol* (*stream*, "*string*");

which sets *i* to the position of the next symbol from the given input stream in the given string. Thus using

i := *insymbol* (*stream*, "0123456789$_{10}$. *ABCDEFGHIJKLMNOP*");

would give the character "0" the code number 1, "," the code number 12, *E* the code number 17, and so on. An appropriate string would then allow most input systems to be simulated, assuming that all codes were positive. The phrase definitions and source string can thus be stored in an **integerarray**. It is of interest to note that an input string such as

real | **integer**

could in practice be stored in three different ways on our Atlas-compatible KDF9 input system, for example:

<table>
<tr><td>Atlas readch (visible characters reconstructed)</td><td>r</td><td>e</td><td>a</td><td>l</td><td>|</td><td>i</td><td>n</td><td>t</td><td></td></tr>
<tr><td>KDF9 charin (non-advance underline)</td><td>_</td><td>r</td><td>_</td><td>e</td><td>_</td><td>a</td><td>_</td><td>l</td><td></td></tr>
<tr><td>KDF9 inbasicsymbol (ALGOL basic symbols)</td><td colspan="3">real</td><td>|</td><td colspan="3">integer</td><td>|</td><td></td></tr>
</table>

When the phrase definitions are stored in the array, the code for | must be replaced by some code which cannot represent an input character, so that the metasymbol | cannot be confused with a symbol | occurring in the form ⟨ | ⟩ in a phrase definitions, i.e. as a symbol which is required in a source string. We will specify the code representing the metasymbol by the value contained in an **integer** variable "or".

To search for a phrase defined by

⟨trig fn⟩ ::= *sin*|*cos*|*tan*|*sec*|*cosec*|*cot*;

which has been stored in an **interarray** *def* as

s	*i*	*n*	*or*	*c*	*o*	*s*	*or*	*t*	*a*	*n*	

we take the first symbol of the definition, *s*, and compare it with the first symbol of the source string (which we will assume has been read in and stored in an **integer-array** *ss*). If it agrees, we continue fetching symbols from the definition array, and comparing each one with the next symbol of the source string array. If we find the metasymbol *or* in the array *def* before any disagreement occurs, the phrase is present. If a disagreement occurs, we skip through the definition array to the next occurrence of *or*, that is we try the next category, and start at the beginning of the source string again. For consistency, we add an *or* at the end of the definition, so that whenever an *or* is reached in the definition array, the phrase definition is satisfied. Following the last *or* we insert a termination marker *term*. If this is reached, all possible categories have been tried and failed, so the phrase is absent. The values of both *or* the *term* must be chosen so that they cannot represent an input character.

A set of instructions to compare definitions stored in the array *def* from *def*[*dpoint*] onwards with a source string stored in the array *ss* from *ss*[*spoint*] onwards, and using the method outlined above, is as follows:

```
        sstart := spoint; category := 1;
next: d := def[dpoint];
      if d = or then
               begin search := true; goto end end success;
      if d = term then
               begin search := false; goto end end failure;
      if d = ss [spoint] then
               begin spoint := spoint + 1;
                              dpoint := dpoint + 1;
               goto next end one symbol found;

trynextcategory:
      d := dpoint;
      for dpoint := dpoint + 1 while def[dpoint] ≠ or
                                          do d := dpoint;
      dpoint := d + 2;
      category := category + 1;
      spoint := sstart;
      goto next;
end:
```

To be able to refer to other phrase types in the definition of a particular phrase type, we give each phrase type a number, and for simplicity of typing, always refer to phrase types by numbers. The notation in use at present is typified by the definitions.

1 ⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9;
2 ⟨*u*/*s* integer⟩ ::= ⟨1⟩⟨2⟩|⟨1⟩;

Thus any number contained between the metasymbols "⟨" and "⟩" is interpreted as referring to a phrase of type indicated by that number. The number at the

Downloaded from https://academic.oup.com/comjnl/article/10/4/325/463854 by guest on 19 April 2024

beginning of the line gives the number of that particular phrase type. All symbols between this number and the metasymbol "::=" are ignored; normally a verbal indication of the name of the phrase type is inserted. The use of names instead of numbers would be simple to implement, but the system using numbers has been found quite satisfactory in practice, and involves a minimum of typing effort.

Within the integer array *def*, a definition such as

⟨trig fn⟩ ::= *sin*(⟨13⟩)|*cos*(⟨13⟩);

is stored as

s	*i*	*n*	(	−13	)	*or*	*c*	*o*	*s*	(	−13	)	*or*	*term*

Each occurrence of a "constituent phrase" in the definition is represented by the negation of that phrase's number. Since we are assuming that all representations of symbols are positive, no confusion of phrase types and actual characters can occur. We now add to our syntax analysis instructions after the test **if** $d = term$ **then**, the instruction

```
if d < 0 then
  begin comment this element of def represents a phrase
                                                   type;
  if search (− d) then
  begin dpoint := dpoint + 1; goto next end;
  goto trynextcategory end;
```

where the instruction *search*(*i*) is assumed to activate a search for a phrase of type *i*.

Note that the pointer *spoint*, which indicates the current point at which the source string is being searched, must be non-local to the searching procedures, so that it will be advanced during the operation *search*(−*d*) if this search is successful. Because of this, and because a later part of the search may fail within a particular category, it is essential that the variable *sstart* be preserved during the operation *search*(−*d*). Similarly, both *dpoint* and *category* must be preserved.

The definitions are stored in a one-dimensional array, with another array whose *i*th element points to the start of the definition of the phrase of type *i*. In our program this second array is called *ilf*, (since it is essentially an Iliffe vector, i.e. a vector whose elements are all addresses of other elements).

The searching instructions can now be made into a procedure which satisfies all the required conditions, by containing them within

```
Booleanprocedure search(i); value i; integer i;
  comment takes the truth-value true if a phrase of type
  i is present starting at positions "spoint" of array "ss";
  begin integer sstart, dpoint, category;
  dpoint := ilf[i];
  . . . (searching instructions as above) . . .
end: end search;
```

The procedure for reading in phrase definitions to the array *def* must operate as follows, where *dpoint* is used to indicate the next free space available for storing definitions:

1. Read an integer, say *type*, indicating the number of this phrase type, and store the current value of *dpoint* in *ilf*[*type*].
2. Skip intervening material until the metasymbol "::=" is reached.
3. Read characters to the array *def* until the terminator is reached, replacing

   ⟨integer⟩ by the negative of the integer
   ⟨symbol⟩ by the symbol itself
   | by the code *or*
   ; by the codes *or*, *term*.
4. Repeat, until a suitable terminator of the definitions is reached, such as the occurrence of a phrase of type 0.

The significance of the non-local identifiers is as follows:

integer variables:

equals: the input code of the symbol "::="
termin: the input code of the symbol terminating each phrase definition, ";" above
orin: the input code for the symbol "|" in the phrase definitions
or: the internal representation for the symbol "|"
term: the internal representation of the terminator
zero: the input code for the symbol "0"
ob: the input code for the symbol "⟨"
cb: the input code for the symbol "⟩"

integer procedures

readno: reads the value of the next number on the input stream
readch: reads the code of the next character on the input stream

Boolean procedure

digit(*i*): takes the value **true** if *i* is a code number which represents a digit, and **false** otherwise

The procedure is:

```
procedure readphrasedefinitions;
  begin integer i, j, k, dpoint; dpoint := 1;
  for k := readno while k ≠ 0 do
    begin ilf[k] := dpoint;
    for i := readch while i ≠ equals do;
    for i := readch while i ≠ termin do
      begin if i = orin then i := or
      else if i = ob then
        begin i := readch; if digit(i) then
          begin i := i-zero;
          for j := readch while j ≠ cb do
                    i := i*10 + j-zero;
          i := − i end
        else j := readch end;
```


Downloaded from https://academic.oup.com/comjnl/article/10/4/325/463854 by guest on 19 April 2024

```
        def[dpoint] := i; dpoint := dpoint + 1 end;
    def[dpoint] := or; def[dpoint + 1] := term;
    dpoint := dpoint + 2 end
  end readphrasedefinitions;
```

Assembling the analysis record

Having analysed a program, or source statement, and found that it is syntactically correct, we will presumably wish to refer to the constituent parts which have themselves been analysed. In order to store details of each phrase found to be present, we declare

```
integerarray type, cat, from, to [1 : length],
                         link [1 : links, 1 : length];
```

where *length* is dependent upon the complexity of the source statement being analysed in any particular case. The *i*th phrase is detailed by entries stored in

type[*i*]	giving the type of the phrase
cat[*i*]	giving its category
from [*i*]	giving its starting position in the source string
to[*i*]	giving its finishing position in the source string
link[1, *i*] *link*[2, *i*] *link*[3, *i*]	indicating the position of the details of the constituents

The entries *from*[*i*] and *to*[*i*] specify the portion of the source string which is occupied by the phrase, in case we need to refer to it at some later time. The link entries indicate the position in store of any phrases which were required as constituents of the original phrase. As an example, we will use the definitions

```
1 ⟨variable⟩ ::= a|b|c|d|e|f|g|h|i|j|k|l;
2 ⟨primary⟩  ::= ⟨1⟩|(⟨4⟩);
3 ⟨term⟩     ::= ⟨2⟩ − ⟨3⟩|⟨2⟩;
4 ⟨sae⟩      ::= ⟨3⟩ + ⟨4⟩|⟨3⟩;
```

to analyse the source string

$$a + b - c$$

Note that we instruct our searching program to search for a phrase of type 4. We do not instruct it to look for a phrase type which satisfies the source string. The analysis record will appear as follows:

i	*type*[*i*]	*cat*[*i*]	*from*[*i*]	*to*[*i*]	*link*[*j*, *i*] $j = 1$	$j = 2$
1	4	1	1	5	2	5
2	3	2	1	1	3	–
3	2	1	1	1	4	–
4	1	1	1	1	–	–
5	4	2	3	5	6	–
6	3	1	3	5	7	9
7	2	1	3	3	8	–
8	1	2	3	3	–	–
9	3	2	5	5	10	–
10	2	1	5	5	11	–
11	1	3	5	5	–	–

The first entry reads:

A phrase of type 4 was found (as required), of category 1, occupying positions 1 to 5 of the source string, and the records of the constituent parts are to be found in rows 2 (for the ⟨3⟩) and 5 (for the ⟨4⟩).

The second entry (which gives details of the ⟨3⟩ just mentioned) reads:

A phrase of type 3 was found, of category 2, occupying only position 1 of the source string, for further details see row 3.

The number of links required for any particular entry depends on the number of phrase types which appear as constituent parts of the definition of that entry. Thus the definitions

```
1 ::= ⟨2⟩⟨3⟩⟨2⟩;
2 ::= ⟨3⟩⟨2⟩;
3 ::= + ⟨2⟩;
```

require respectively 3, 2 and 1 links. Thus the width of the declared array *link* must equal the maximum number of constituent phrases in any definition. Usually we find 3 to be sufficient.

While we are searching, we must therefore have a pointer *rpoint* to indicate how much of the analysis record arrays are already in use. Whenever the procedure *search* is entered, we must note the value of *rpoint* (this is where details of this phrase will be entered) and advance *rpoint* by 1 position to ensure that these details are not overwritten. If the search is unsuccessful, the pointer *rpoint* is moved back to its original position. If the search is successful, all the details of this phrase must be entered in the appropriate places. The necessary instructions are:

```
On entry to search: this := rpoint; rpoint := rpoint + 1;
At failure exit: rpoint := this;
At success exit: type[this] := i; cat[this] := category;
                 from[this] := sstart; to[this] := spoint
                                                     − 1;
```

The only remaining items to store are the link entries, which indicate the position of the analysis records of the other phrases involved in the definition. To do this we set *lpoint* (the link pointer) to 1 on entry to *search*, and alter the recursive statement to read:

```
if d < 0 then
  begin link[lpoint, this] := rpoint; if search (− d) then
    begin lpoint := lpoint + 1; dpoint := dpoint + 1;
    goto next end;
  goto trynextcategory end;
```

We must also add the resetting instruction *lpoint* := 1; after the label *trynextcategory* to reset the link pointer if one category fails after some constituents have been found.

The procedure is now complete, and can be used for producing analysis records for arbitrary phrase definitions and source strings. We will now describe three additional features which have been found helpful.

Downloaded from https://academic.oup.com/comjnl/article/10/4/325/463854 by guest on 19 April 2024

(1) *Built-in phrases*

Searching for such commonly occurring phrase types as letter, digit, identifier and integer using the procedure *search* in the form described above is inefficient in two respects. First, to find the ⟨letter⟩ *Z* may have involved 51 unsuccessful searches, when we probably know that the input code for any letter lies between two known limits. It would be more efficient to define a letter as a character whose input code was between two given numbers.

Secondly, if an integer is defined as

1. ⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9;
2. ⟨integer⟩ ::= ⟨1⟩⟨2⟩|⟨1⟩;

the analysis record corresponding to the integer 12345 would be

i	*type*[*i*]	*cat*[*i*]	*from*[*i*]	*to*[*i*]	*link*[*j*, *i*]	
1	2	1	1	5	2	3
2	1	2	1	1	–	–
3	2	1	2	5	4	5
4	1	3	2	2	–	–
5	2	1	3	5	6	7
6	1	4	3	3	–	–
7	2	1	4	5	8	9
8	1	5	4	4	–	–
9	2	2	5	5	10	–
10	1	6	5	5	–	–

Such long analysis records are inconvenient and wasteful. To overcome these disadvantages, we follow the Compiler Compiler and have special definitions for certain phrase types such as those mentioned above, known as Built-in Phrases, or BIPs. In our version, we have reserved phrase types with numbers up to 10 for BIPs. Early in the procedure *search* we insert

if $i < 10$ **then goto** *bips*;

and at the end of the procedure insert either

bips: **if** $i = 1$ **then begin** . . . (search for BIP of type 1) . . . **end**
else if $i = 2$ **then begin** . . . (search for BIP of type 2) . . . **end**

or

bips: **begin switch** *s* := *digit*, *letter*, *integer*, *ident*; **goto** *s*[*i*];
digit: (search for a digit); **goto** *end*;
letter: (search for a letter); **goto** *end*;
integer: . . .

We have at present implemented BIP's for the following phrase types:

PHRASE TYPE	CATEGORY ALLOCATED
⟨letter⟩	1 to 26, for *A* to *Z* 27 to 52 for *a* to *z*
⟨digit⟩	The value of the digit
⟨integer⟩	The value of the integer
⟨identifier⟩ ⟨rest of identifier⟩ ⟨letter string⟩	An integer uniquely determined by the first six characters
⟨string⟩	The internal code of the first character after the opening quote
⟨; or **end** or **else**⟩	1, 2 or 3 respectively.

Further, the procedure to read in the phrase definitions has been amended to interpret ⟨*l*⟩ to represent the phrase number of a letter, ⟨*d*⟩ for ⟨digit⟩, ⟨*i*⟩ for ⟨integer⟩, ⟨*n*⟩ for ⟨identifier⟩ (name) and ⟨*s*⟩ for ⟨string⟩. This enables us to write, for example,

⟨decimal number⟩ ::= ⟨*i*⟩. ⟨*i*⟩|⟨*i*⟩.⟨*i*⟩;

(2) *Simplifying the definitions*

A very common type of phrase definition is

⟨sae⟩ ::= ⟨term⟩⟨addop⟩⟨term⟩|⟨term⟩;

If the phrase we are searching turns out to be of category 2, the search for a ⟨term⟩ is carried out twice, once in the unsuccessful category 1, then again for category 2. Since the definition of a ⟨term⟩ is similar, an ⟨sae⟩ which happens to be a ⟨factor⟩ will involve 4 searches for a ⟨factor⟩; an ⟨sae⟩ which is a ⟨primary⟩ will involve 8 searches for a ⟨primary⟩. With the standard definition of a ⟨simple Boolean⟩ matters are even worse. These difficulties can be overcome by defining, for example,

⟨sae⟩ ::= ⟨term⟩⟨rest of sae⟩;
⟨rest of sae⟩ ::= ⟨addop⟩⟨term⟩|⟨empty⟩;

To avoid the use of so many "rest of . . ." definitions, we have introduced the notation

⟨sae⟩ ::= ⟨term⟩[⟨addop⟩⟨sae⟩|⟨empty⟩];

which is read as: "an ⟨sae⟩ is a ⟨term⟩ followed by either ⟨addop⟩⟨sae⟩ or ⟨empty⟩". Any alternatives either ⟨addop⟩⟨sae or ⟨empty⟩". Any alternatives between the metasymbols "[" and "]" must be preceded by whatever precedes "[" and followed by whatever follows "]".

Other examples are

⟨program⟩ ::= **begin**[⟨declaration list⟩|⟨empty⟩] ⟨statement list⟩**end**;
⟨declaration list⟩ ::= ⟨declaration⟩⟨;⟩[⟨declaration list⟩| ⟨empty⟩];
⟨statement list⟩ ::= ⟨statement⟩[⟨;⟩⟨statement list⟩| ⟨empty⟩];
⟨for list element⟩ ::= ⟨arith exprn⟩[**step** ⟨arith exprn⟩ **until** ⟨arith exprn⟩|**while**⟨boolean expression⟩|⟨empty⟩];

The basic difference now is that if a definition fails between "[" and "]", the various pointers need only be reset to their positions when "[" was passed, instead of being reset as if to start a new category. The category of a phrase is easily obtained in most cases as the number of times a search has failed plus one. For example a block would be a ⟨program⟩ of category 1, and a compound statement would be a ⟨program⟩ of category 2

Downloaded from https://academic.oup.com/comjnl/article/10/4/325/463854 by guest on 19 April 2024

in the example above. Ambiguities of category arise in a definition such as

⟨*a*⟩ ::= ⟨*b*⟩[⟨*c*1⟩|⟨*c*2⟩][⟨*d*1⟩|⟨*d*2⟩];

where 4 possible categories exist. Such ambiguities are ignored, so that if a unique and meaningful category is to be obtained, then there must be only one occurrence of "[" and "]" between any two occurrences of "::=", ";" or "|" not within these symbols.

The essential additional features of the search procedure are now

1. If failure occurs before [is reached, skip the definitions up to the next occurrence of | not contained between [and], adding one to the category for each | passed contained between [and].
2. If [is reached, note all pointer positions.
3. If success occurs after [, skip to the next] and continue searching, but note how many occurrences of | were passed, in case a failure occurs after the].
4. If failure occurs after |, skip to the next occurrence of |, resetting the various pointers to the values noted in step 2 above.
5. If] is reached, none of the alternatives between [and] is satisfied, so skip to the next occurrence of |.
6. If failure occurs after], skip to the next occurrence of |, and update the category by the number of occurrences of | passed in stage 3 above.

The counting in step 3 is necessary in definitions such as

⟨*a*⟩ ::= ⟨*b*⟩[⟨*c*⟩|⟨*d*⟩|⟨*e*⟩]⟨*f*⟩|⟨*g*⟩;
⟨*g*⟩ ::= ⟨*b*⟩⟨*c*⟩⟨*h*⟩;

to ensure a correct category for ⟨*g*⟩ in the first definition.

(3) *Control phrases*

So far, our syntax searching has been a "passive" operation, merely setting up the analysis record. It is sometimes necessary to execute certain chosen instructions during the course of syntax searching. Examples of occasions when such extra instructions are required occur frequently during the syntax analysis of ALGOL programs, such as

1. Whenever a declaration is found, the identifier and its properties must be noted for reference in later compilation.
2. If the search for a ⟨**for** list⟩ fails, the fixed word symbol **for** must have been present, so a ⟨**for** list⟩ must have been intended, but wrongly constructed or typed. For the benefit of the programmer, the faulty ⟨**for** list⟩ can be printed out with appropriate comment, and the source program skipped to the next **do**.
3. Similarly, if a search for a ⟨statement⟩ fails (excluding the dummy statement) all source program up to the next **end**, **else** or semi-colon can be printed out, and the syntax analysis continued.

The present means of implementing this requirement is to use phrase numbers greater than 1000 to represent what we will call "control phrases". At the head of the procedure *search* we insert the instruction

if *i* > 1000 **then goto** *control*;

where *control* labels a section of the procedure similar to the BIPs section, giving instructions to be obeyed for each control number. Using the above examples, the phrase definitions might be

```
⟨type declaration⟩ ::= [real|integer|Boolean]⟨name list⟩
                                                  ⟨1007⟩;
⟨for list⟩         ::= ⟨for list element⟩[,⟨for list⟩|]|
                                                  ⟨1009⟩;
⟨statement⟩        ::= ⟨non dummy statement⟩|⟨1001⟩;
```

In this case the label *control* could be followed by instructions such as

```
    if i = 1007 then addtopropertylist (1, j)
else if i = 1009 then
       begin write text ("for list not recognized"); . . . end
else if i = 1001 then
       begin if not search(terminator) then
         begin writetext("statement not recognized");
                                                . . . end;
       . . .
       end;
  search :=true;
       . . .
```

where the parameters of the procedure *addtopropertylist* indicate the properties of the names to be added and the position of the name list, and the integer *terminator* gives the phrase number of the phrase definition

terminator ::= **end**|**else**|⟨;⟩;

Interpretation of the analysis record

Once the source string has been satisfactorily analysed, we will in most cases want to interpret the analysis record according to some given semantic rules, and thus obtain a target string. To perform this "semantic interpretation" we use another recursive procedure (which we will call *compile* for the moment) which has the following heading:

procedure *compile*(*i*, *j*); **value** *i*, *j*; **integer** *i*, *j*;

This procedure interprets the *i*th link of the phrase detailed by the *j*th entries in the analysis record. Details of this phrase are obtained using the instructions

j := *link*[*i*, *j*]; *i* := *type*[*j*]; *k* := cat[*j*];

The variables *i* and *k* now give respectively the type and category of the phrase to which the original *i*th link of the *j*th entry referred. We can now give a separate set of instructions for interpreting every possible type and category of phrase, including, if required, interpretation of the constituent phrases by recursive calls such as

compile(2, *j*);

Downloaded from https://academic.oup.com/comjnl/article/10/4/325/463854 by guest on 19 April 2024

to interpret the second constituent. (Notice that the value of j has now been updated to the position of the current phrase we are interpreting in the analysis record.) Thus if a particular definition of one category of one phrase is

⟨operand⟩⟨operator⟩⟨operand⟩

and we wish to interpret this into reverse Polish notation (with the operator following the operands) the necessary instructions are:

compile(1, *j*); *compile*(3, *j*); *compile*(2, *j*);

i.e. compile the first constituent, then the third, then the second. The complete operation of this method of interpretation is easiest explained by using examples for illustration. We will use the additional procedures

- *write*(*i*): to write the value of i as in integer in the target string.
- *writetext*("*string*"): to add the given string to the target string.
- *copyphrase*(*j*): to copy the phrase given by the jth entry of the analysis record to the target string, i.e. to copy out that part of the source string from position *from*[*j*] to position *to*[*j*].
- *copylink*(*i*, *j*): to copy the ith constituent of the phrase defined by the jth entry of the analysis record to the target string. This could be written *copyphrase*(*link*[*i*, *j*]).

Example 1: Translating ordinary arithmetic expressions to Polish notation.

We will consider the restricted system given by the phrase definitions

```
1 ⟨variable⟩ ::= x|y|z;
2 ⟨primary⟩  ::= ⟨1⟩|(⟨4⟩);
3 ⟨term⟩     ::= ⟨2⟩*⟨3⟩|⟨2⟩;
4 ⟨rae⟩      ::= ⟨3⟩+⟨4⟩|⟨3⟩;
```

The procedure to translate any such restricted arithmetic expression into Polish notation is

```
procedure Polish(i,j); value i,j; integer i,j;
  begin integer k; switch s := vble,prim,term,rae;
  j := link[i,j]; i := type[j]; k := cat[j];
  goto s[i];
vble: copyphrase(j); goto end;
prim: Polish(1,j); goto end;
term:
  rae: if k=1 then
    begin if i=3 then writetext("*") else writetext("+");
    Polish (1,j); Polish(2,j) end
  else Polish(1,j);
  end: end Polish;
```

Thus with an analysis record stored in *type*[1], *cat*[1], etc., onwards, and with link[1,0] set to 1, the instruction

Polish(1,0);

will cause the Polish version of the source string to be output.

Example 2: Algebraic differentiation.

Algebraic differentiation of the restricted arithmetic expressions defined above can be performed if we redefine a primary as

⟨primary⟩ ::= ⟨indept vble⟩|⟨dept vble⟩|⟨constant⟩|(⟨4⟩);

where the rules for differentiation are

- ⟨indept vble⟩ : derivative is 1
- ⟨dept vble⟩ : derivative is D followed by the variable name
- ⟨constant⟩ : derivative is 0

We will name the procedure *diff*, and write down only the instructions required for each value of i and k. The ways of entering and leaving each such set of instructions will be omitted. It is assumed that the phrases ⟨indept vble⟩, ⟨dept vble⟩ and ⟨constant⟩ have been defined elsewhere. Their numbers are immaterial, since they are not required in the interpretation procedure.

```
i=2:       k=1: writetext("1");
           k=2: writetext("D"); copylink(1,j);
           k=3: writetext("0");
           k=4: writetext("("); diff(1,j); writetext(")");
i=3 or 4: k=2: diff(1,j);
i=3:       k=1: diff(1,j); writetext("*"); copylink(2,j);
                                             writetext("+");
                diff(2,j); writetext("*"); copylink(1,j);
i=4:       k=1: diff(1,j); writetext("+"); diff(2,j);
```

Example 3: Expansion of complex number expressions

Suppose we wish to interpret the source string

z := *z*1*(*z*2+*p*);

as though z, $z1$ and $z2$ were complex, and produce two assignment statements

```
zreal := z1real*(z2real+p)−z1imag*(z2imag+0);
zimag := z1real*(z2imag+0)+z1imag*(z2real+p);
```

This process is of benefit to users who wish to write expressions in complex arithmetic, and have this "macrogenerator" translate them into the equivalent ALGOL expressions acting on the real and imaginary parts of each variable. We now alter the definition of a primary to

⟨primary⟩ ::= ⟨complex vble⟩|⟨real vble⟩|(⟨4⟩);

and require two interpreting procedures to obtain the real and imaginary parts of any given expression. Naming these two procedures *real* and *imag*, the main instructions of *real* are

```
i=2:      k=1: copylink(1,j); writetext("real");
          k=2: copylink(1,j);
          k=3: writetext("("); real(1,j); writetext(")");
i=3 or 4:k=2: real(1,j);
```


Downloaded from https://academic.oup.com/comjnl/article/10/4/325/463854 by guest on 19 April 2024

```
i=3:      k=1: real(1,j); writetext("*"); real(2,j);
                   writetext("—");
                   imag(1,j); writetext("*"); imag(2,j);
i=4:      k=1: real(1,j); writetext("+"); real(2,j);
```

while the body of *imag* must include

```
i=2:      k=1: copylink(1,j); writetext("imag");
          k=2: writetext("0");
          k=3: writetext("("); imag(1,j); writetext(")");
i=3 or 4: k=2: imag(1,j);
i=3:      k=1: real(1,j); writetext("*"); imag(2,j);
                   writetext("+");
                   imag(1,j); writetext("*"); real(2,j);
i=4:      k=1: imag(1,j); writetext("+"); imag(2,j);
```

Example 4: Evaluation of expressions

A different application of the analysis record is its use to calculate the value of an expression given as a source string. To do this, we alter the procedure heading to

realprocedure *value*(*i*,*j*); **value** *i*,*j*; **integer** *i*,*j*;

with the initial instructions as before. Using the first set of phrase definitions (those of Example 1) the body of the procedure (bearing in mind that it is now a type procedure) must allow execution of the following instructions:

```
i=1:      k=1: value := x;
          k=2: value := y;
          k=3: value := z;
i=2:           value := value(1,j);
i=3 or 4: k=2: value := value(1,j);
i=3:      k=1: value := value(1,j)*value(2,j);
i=4:      k=1: value := value(1,j) + value(2,j);
```

This procedure can now be used to calculate the value of any analysed expression for given values of the variables x, y and z. Similar methods to those indicated above can be used to calculate the value of the derivative of the expression. Expressions such as

$$x+(\textbf{if } y<0 \textbf{ then } y \textbf{ else } z)$$

can be evaluated by including extra phrase definitions for Boolean expressions, and having an extra procedure

Booleanprocedure *bool*(*i*,*j*); **value** *i*, *j*; **integer** *i*, *j*;

to calculate the value of the Boolean expressions. The phrase definition

⟨arith exprn⟩ ::= **if** ⟨Bool exprn⟩ **then** ⟨arith exprn⟩ **else** ⟨arith exprn⟩

would then have a corresponding entry in the procedure *value* as

value := **if** *bool*(1,*j*)**then** *value*(2,*j*) **else** *value*(3,*j*)

Example 5: Compilation of ALGOL programs.

The actual compilation of ALGOL programs can be performed using this technique, except that various extra procedures are necessary, for example, to set up the property list of a variable. To give a simple example of compilation into Usercode (the KDF9 assembly language) we will illustrate the compilation of phrases of the type

if⟨Boolean exprn⟩**then**⟨phrase1⟩**else**⟨phrase2⟩;

where ⟨phrase1⟩ and ⟨phrase2⟩ may be arithmetic, Boolean or designational expressions, or statements, depending on context. In Usercode, labels are written simply ⟨integer⟩; and a jump on negative is, for example, J87<Z;.

Using a non-local counter *label* to indicate the next available integer for use as a label in the compiled program, and representing **true** and **false** by 0 and —1 respectively, we compile this type of phrase using the instructions

```
i := label; label := label+2;
compile(1, j);
writetext("J"); write(i); writetext("<Z;");
compile(2,j);
writetext("J"); write(i+1); writetext(";");
write(i); writetext(";");
compile(3,j);
write(i+1); writetext(";");
```

This will produce as target string, with $i = 95$, for example,

```
    (compiled Boolean exprn)
    J95<Z;
    (compiled phrase1)
    J96;
95; (compiled phrase2)
96;
```

Conclusions

Of the five applications of syntax analysis and semantic interpretation illustrated above, the second, third and fourth are all in use at Nottingham. They incorporate a full set of phrase definitions which allow any ALGOL arithmetic expression to be used, including any of the standard functions, and the use of conditional expressions (or sub-expressions). For differentiation, the derivative of *abs* is taken to be 1/*sign*, to give a correct result away from the origin, but fail if the derivative of *abs* is requested at the origin. The functions *sign* and *entier* are treated similarly. For complex macro-generation, various extra complex functions are allowed, as in the Atlas Autocode compiler on Atlas. The fourth application is used in a program to read in functions, and plot them on the Calcomp plotter, while producing on the line printer a summary of the interpolated roots, and the positions and values of any maxima, minima, etc. One version reads in three-dimensional functions and plots their projections. In addition to the ALGOL standard functions, various others such as *sigma* for summing series are available.

The authors wish to express their grateful thanks to the Science Research Council for their financial support during the work on this project.

(*References overleaf*)

Downloaded from https://academic.oup.com/comjnl/article/10/4/325/463854 by guest on 19 April 2024

References

BROOKER, R. A., MACCALLUM, I. R., MORRIS, D., and ROHL, J. S. (1963). The Compiler Compiler, *Annual Review in Automatic Programming*, Vol. 3, London: Pergamon.

NAUR, P., *et al.* (1963). Revised report on the algorithmic language ALGOL 60, *Computer Journal*, Vol. 5, p. 349.

Book Review

Most Notorious Victory, by Ben B. Seligman, 1966. (New York: *The Free Press*.)

"With the victory of the machine—a most notorious victory—the attainment of human autonomy is at best moot." With these words the last sentence of the book explains its title, and encapsulates its theme. The author, an economist concerned with labour relations, sets out to attack the facile utopianism of those who see automation in offices and factories leading inevitably towards a better life, where all will have more goods *and* more leisure to enjoy them, and be delivered from dirt, danger and drudgery in their daily tasks.

The book is divided into four parts, each having three chapters. The first part surveys the developments of industrialization, and in particular of computers listing their uses from elementary book-keeping to artificial intelligence. Readers of this *Journal* could skip Part 1, for it is clearly the part in which the ratio of the author's knowledge to their own is at its lowest. At times, data are regurgitated rather than shown to be significant. For example, what does it profit a layman to learn that "The transistor is a device for transferring signals through a varistor", with no explanation of varistor? The snark was a boojum, you see. Again, as an example of the pitfalls of addressing expert and lay readers simultaneously, after explaining what a computer word is, and that it can represent either an instruction or a quantity, the author adds, darkly: "There are, of course, other systems as well, like those in IBM's 7080".

The second part of the book is devoted to demonstrating by examples that in America automatic machines—computers especially—are steadily permeating all areas of men's work; and that, contrary to what is widely believed, their use *is* giving rise to unemployment, even if this sometimes appears as premature retirement or delayed recruitment of school leavers. A surfeit of cases is quoted to illustrate employers' callousness in their relentless pursuit of efficiency and productivity.

The third part deals with various attempts to solve the labour problems posed by automation: so far unsuccessful, and in the author's view, likely to remain so. Neither the trade unions' work-sharing and retraining, nor the employers' insurance and relocation funds, nor the government's attempts to increase demand or to help depressed areas, have been effective. And, Professor Seligman clearly doubts whether they or their economic advisers yet appreciate that their methods are not geared high enough to tackle the speed of technological change. The technologists and systems analysts who promote this change are taken to task for leaving their humanity at home when they go to work. The "neutral artificers" of top management he calls them, whose interests lie solely in efficiency and securing the widest scope for the satisfactory display of their technical skills.

Professor Seligman would no doubt agree with that other American critic of the affluent society that business has become an end in itself; that giant corporations now foster their own growth, in expanding the power and the careers of their managers, and do so independently of their customers and their owners alike; that in this situation technology acts as a sorcerer's apprentice; and that those economists who should be sounding the alarm are still reading the signs of the times in terms of concepts appropriate to those halcyon textbook days of free-markets and the barter of shoes for cabbages.

In Part 4, the book restates its theme. Professor Seligman challenges the comfortable concepts that automation is only more of the old mechanization; that it creates more, and more interesting, work than it destroys; that it will advance so slowly that all will be well with a little redeployment and the gentle expansion of leisure. This part is the most philosophical, and in it the author shows his keen concern that men's work should help to bring meaning to their lives, and to integrate them into society. He notes that a man's work is widely used as a indication of his worth and his status, and argues that automation is a major threat in our age, by depriving men of the opportunity to work—or to work to create a product rather than to add infinitesimally to an uncomprehended process. And, he is concerned that the pursuit of affluence has dulled our concern for these things—that having eyes we see not.

The author's method of piling example upon example, reference upon reference (there must be more than a thousand at the end of the book, and many ephemeral), may irritate; and he would have done well to heed the advice of his countryman Oliver Wendell Holmes that "A moment's insight is sometimes worth a life's experience". Nevertheless, his subject is serious and seriously treated, and computer men would do well to read this challenge to their almost automatic assumption that computers are good, and more computers better.

F. J. M. Laver (London)

Downloaded from https://academic.oup.com/comjnl/article/10/4/325/463854 by guest on 19 April 2024