A syntax improving program

By J. M. Foster*

A program called SID has been written which accepts as data a grammatical definition of a language and attempts to transform it into an equivalent grammar which can be parsed by a simple one-track parsing algorithm. Not all grammars can be so transformed, and if SID fails it reports the reason for its failure. If it succeeds, it further transforms the grammar rules that it has produced into a fast-running compiler written in machine code.

(First received September 1967)

Compilers which use a syntactic description of the language that they are translating are desirable because of their convenience and flexibility, but they are often slow. Syntax analysis programs which are capable of working for all possible phrase-structure grammars are unlikely to be fast, and so, in order to achieve a tolerable rate of translation, various classes of well-behaved grammars have been used (Floyd, 1963; Johnson, 1965). This is sometimes unsatisfactory because the way in which the grammatical rules of the language have to be written conflicts with the natural structure, and because a grammatical definition suitable for one system may not transfer unchanged to another.

In order to reduce some of these difficulties, a program called SID (Syntax Improving Device) has been written which attempts to transform a grammatical definition into an equivalent grammar which can be parsed by a simple one-track parsing algorithm. If it succeeds, it transforms the grammar rules that it has produced into a fast-running compiler written in machine code.

The ability to recognize that a string of characters belongs to a language does not constitute a compiler, and so SID carries out its transformations on the syntax and translation mechanism together, thus obviating the need for the compiler writer to work in terms of the transformed rules.

If SID succeeds, the grammar that it produces is capable of analysing a string at any point in terms of the next character and the previous analysis record, without any back-tracking (Floyd, 1964). The grammar is also unambiguous. Being sure of unambiguity is not easy by hand, and this has proved one of the more useful features of the program.

SID assumes little about the nature of the translating functions, and so it can be used for a variety of languages. It has been used to produce part of an ALGOL compiler, to help in the development of a new language and a compiler for it, and in the development of a pseudo-English query system.

The efficiency aimed at can be illustrated with the syntactic description of a number, discussed below, for which the final output of SID corresponds to the following ALGOL program, though the output is actually in machine code.

```
        t := character;
        if t ⩾ 10 then goto failure;
        t1 := t; t := character;
        n := t1;
label:  if t < 10 then begin t1 := t; t := character;
                             n := n × 10 + t1;
                             goto label
                       end
```

The function "*character*" reads successive items from the input.

Part of an ALGOL compiler produced by SID has been compared with a hand-written compiler. The version produced by SID was about 50% longer and ran at about the same speed, but it checked the syntax completely, which the hand-written one did not.

SID is written in ALGOL, and uses list-processing features developed on RREAC for ALGOL.

The translating routines

A syntax is generally considered to be a passive description of the legal sentences of a language. The syntax

$$\langle sentence \rangle ::= p\ q\ r \mid s\ t\ u$$

means that the two possible sentences are *p q r* and *s t u*. Alternatively, the syntax can be considered to represent an active recognizer for the sentences of the languages. Instead of the *p* in the definition representing the letter *p*, it means a program which reads a character and makes sure that it is *p*. The grammatical class names (the non-terminal symbols) are likewise active; the ⟨*sentence*⟩ in the example is a recognizer which reads characters and is satisfied by either *p q r* or *s t u*. So the whole syntax can be regarded as an active recognizer. This is equivalent to the method known as top-to-bottom analysis.

It is a simple extension to allow the symbols occurring in the syntactic definition to stand for not merely recognition programs, but any programs. They are understood to be obeyed at the moment in the recognition process corresponding to their placing in the syntax. If these routines are those which produce the output from the analysis, then the difficulty about transforming the translation mechanism has been removed. Any trans-

* *Royal Radar Establishment, Malvern*; (now at *Aberdeen University*).

formations on the syntax which would produce an equivalent grammar if the routines were all ordinary basic symbols, will give a recognizer which will produce the same translation as the original. For such transformations do not affect the order in which the routines are obeyed when a particular string of characters is recognized.

Consider the process of reading and forming the value of a number. When the first digit is read its value is assigned to a variable. Subsequent digits cause the variable to be multiplied by ten and the new digit added to it. This could be denoted in a particular example by

$$1\ \alpha\ 2\ \beta\ 7\ \beta\ 3\ \beta$$

where α is the operation

$$n := t$$

t being the value of the last character, and β is

$$n := n \times 10 + t$$

The interpretation is that α and β are carried out at the points in the string of characters where they occur. An active syntax with operations can be written for this string.

$$\langle number \rangle ::= \langle digit \rangle\ \alpha \mid \langle number \rangle\ \langle digit \rangle\ \beta$$

It happens that in this case the translating operations are at the end of the productions, but this is not so in the equivalent productions

$$\langle number \rangle ::= \langle digit \rangle\ \alpha\ \langle continue \rangle$$
$$\langle continue \rangle ::= \phi \mid \langle digit \rangle\ \beta\ \langle continue \rangle$$

where ϕ represents the empty string of characters. This is an example of the sort of change which SID carries out.

Transformations on the syntax

The first transformations which SID applies remove left recursion. Such productions as

$$\langle number \rangle ::= \langle digit \rangle \mid \langle number \rangle\ \langle digit \rangle$$

give rise to difficulty because the top-to-bottom algorithm does not terminate unless precautions are taken. In fact this syntax could cause the $\langle number \rangle$ recognizing routine to be continuously activated. To avoid this the rules are transformed so that it is not possible to call a recognition program while it is still active, unless a character has been read in between. This is just the transformation needed to turn the syntax into the form suitable for predictive analysis. Such a transformation always exists, and can be made to preserve ambiguity (Greibach, 1964).

The transformation can be illustrated on $\langle number \rangle$ which yields

$$\langle number \rangle ::= \langle digit \rangle\ \langle newclass \rangle$$
$$\langle newclass \rangle ::= \phi \mid \langle digit \rangle\ \langle newclass \rangle$$

where $\langle newclass \rangle$ is a new non-terminal symbol. This is the only transformation needed in practical examples which have occurred in using SID, but a generalization which deals with every situation is possible and is used by SID.

The more general case arises because the potential loop can involve more than one non-terminal symbol.

$$\langle a \rangle ::= \langle b \rangle \mid x$$
$$\langle b \rangle ::= \langle a \rangle \mid y$$

There may be a set of non-terminal symbols involved in a loop, $\langle n_i \rangle\ (i = 1 \ldots s)$. The general case can be written

$$\langle n_i \rangle ::= \langle a_i \rangle \mid \langle n_j \rangle\ \langle b_{ji} \rangle$$

In this the symbols $\langle a_i \rangle$ and $\langle b_{ji} \rangle$ can stand for any collection of other symbols.
The set $\langle a_i \rangle$ are assumed not to begin with any of the $\langle n_i \rangle$. A convention similar to the normal summation convention is used: if a suffix is repeated, then the disjunction of all possible values is implied.

The transformed version of this general case is

$$\langle n_i \rangle ::= \langle a_j \rangle\ \langle x_{ji} \rangle$$
$$\langle x_{rs} \rangle ::= \delta_{rs} \mid \langle b_{rt} \rangle\ \langle x_{ts} \rangle$$

The symbol δ_{rs} has the value ϕ if r is equal to s and the value ω if r is not equal to s.

The symbol ϕ, the empty string, is such that when it occurs the string of characters is equivalent to the same string with the ϕ left out. The symbol ω, the illegal symbol, is such that any string in which it occurs is not a legal string of the language. If both ϕ and ω are allowed to occur in the original definitions for the $\langle n_i \rangle$ then the transformation described is perfectly general, as is easily proved by examination of the possible legal sentences.

In practice, however, it is best to transform the smallest linked sets first, and the effect of this is that the more complex multiple transformations do not have to be used.

In its second stage SID takes the set of grammatical rules, which can now be parsed by a top-to-bottom method without danger of an indefinite loop, and attempts to change them so that they can be parsed quickly. Suppose that the definition of some class name is

$$\langle a \rangle ::= \langle b \rangle\ \langle c \rangle \ldots \mid \langle d \rangle\ \langle a \rangle \ldots$$

and that the set of characters which possibly begin $\langle b \rangle$ is disjoint from the set of characters which can possibly begin $\langle d \rangle$. Then if, during the parsing, the algorithm is attempting to recognize an $\langle a \rangle$, it can tell from the next character which of the two alternative productions must be used. The other possibility can be discarded immediately. The second set of transformations therefore attempts to ensure that the sets of characters starting each of the alternative productions for a class name are disjoint. This process, unlike the first transformation, cannot always be done.

The transformations, which are very simple, resemble the distributive law of ordinary algebra.

$$\langle a\rangle ::= \langle b\rangle \mid \langle c\rangle$$
$$\langle b\rangle ::= \langle d\rangle \mid \langle e\rangle$$

can always be replaced by

$$\langle a\rangle ::= \langle d\rangle \mid \langle e\rangle \mid \langle c\rangle$$

and

$$\langle a\rangle ::= \langle b\rangle\ \langle c\rangle \mid \langle b\rangle\ \langle d\rangle$$

can always be replaced by

$$\langle a\rangle ::= \langle b\rangle\ \langle x\rangle$$
$$\langle x\rangle ::= \langle c\rangle \mid \langle d\rangle$$

Consider this part of the syntax of ALGOL.

⟨*statement*⟩ ::= ⟨*block*⟩ | ⟨*compound statement*⟩

⟨*block*⟩ ::= **begin** ⟨*declaration list*⟩ ; ⟨*statement list*⟩ **end**

⟨*compound statement*⟩ ::= **begin** ⟨*statement list*⟩ **end**

This is not satisfactory because ⟨*statement*⟩ can start with **begin** in two analyses which are only resolved later. The following is a form which needs no back-tracking.

⟨*statement*⟩ ::= **begin** ⟨*statement continuation*⟩

⟨*statement continuation*⟩ ::= ⟨*statement list*⟩ **end** |
⟨*declaration list*⟩ ; ⟨*statement list*⟩ **end**

The difficulties of the process, as in automatic algebraic simplification, lie not in carrying out the transformations, but in deciding which must be done and in ensuring that the method terminates.

For processing a particular class name, the set of characters which can begin each of the productions must be formed. It should be remembered that during this process the class names and the characters occurring in the productions are considered, not the translating actions (which are transformed but do not cause transformations). Each production is examined. If a character occurs first then this in itself makes up the required set. If a class name occurs first then all the characters which can begin it make up the required set; but if this class name contains an empty production then the process of examination continues and new items may be added to this set. If the production is empty, then the set is formed from all the characters which can occur immediately to the right of the class name in any production. Calculating these sets presents no difficulty of principle, but takes a major part of SID's time.

If the sets are disjoint then nothing further need be done; but if a group of them intersect, then the definition must be altered. If the first item in each of the productions is the same, they can be treated as **begin** was in the preceding example. If not, then the class name of highest order is replaced by its expanded definition, and the process is repeated. The order of a class name is the maximum number of substitutions which can be made, always on the first item of an example of the definition. Thus the order of ⟨*block*⟩ is 1 and the order of ⟨*statement*⟩ before transformation is 2. Expanding a class name can never give one of higher order at the beginning.

If one of the intersecting sets results from a ϕ, SID reports a failure. For example,

⟨*word*⟩ ::= ⟨*number*⟩ *space space*
⟨*number*⟩ ::= *digit* | ⟨*number*⟩ *digit* | ⟨*number*⟩ *space digit*

which says that two spaces terminate a *number* but one does not, results in

⟨*word*⟩ ::= ⟨*number*⟩ *space space*
⟨*number*⟩ ::= *digit* ⟨*number continuation*⟩
⟨*number continuation*⟩ ::= ϕ | *digit* ⟨*number continuation*⟩ | *space digit* ⟨*number continuation*⟩

by the first transformations, and failure now occurs because *number continuation* contains ϕ and *space* can occur to its right as well as starting its third production.

SID may also report a failure because its attempt to transform the grammar causes it to loop.

A third set of transformations is applied to improve the efficiency. Classes with identical definitions are detected, and uses of them are made to refer to a single class. Class names defined by single productions and occurring in only one place are substituted into this place.

After all this processing an ALGOL syntax, which originally contained 54 class names, contained 82 class names.

Forming the compiler

The translation of the syntax into a compiler in machine code is straightforward. Each operation symbol is replaced by the piece of program for which it stands, and the other symbols are replaced by recognition routines for the classes and characters. For example, the syntax for a number enclosed in any number of pairs of brackets

⟨*number*⟩ ::= *digit* α | ⟨*number*⟩ *digit* β
⟨*sentence*⟩ ::= (⟨*sentence*⟩) | ⟨*number*⟩
$\alpha = n := t1$
$\beta = n := n \times 10 + t1$
digit $= t1 < 10$

which is transformed into

⟨*number*⟩ ::= *digit* α ⟨*x*⟩
⟨*x*⟩ ::= ϕ | *digit* β ⟨*x*⟩
⟨*sentence*⟩ ::= (⟨*sentence*⟩) | ⟨*number*⟩
$\alpha = n := t1$
$\beta = n := n \times 10 + t1$
digit $= t1 < 10$

will yield

```
number: subroutine entry
number2: check that the character just read is a digit and
         fail if it is not
         read the next character
         n := t1
       x: if the character just read is a digit, jump to l
         subroutine exit
       l: read the next character
         n := n × 10 + t1
         jump to x
sentence: subroutine entry
         if the character just read is "(" then jump to m
         if the character just read is a digit then jump to
         number 2

         fail
       m: read the next character

         call the subroutine sentence

         if the character just read is not ")" then fail

         read the next character

         subroutine exit
start:   read the next character
         call the subroutine sentence
```

This contains the program for ⟨*number*⟩ given in the introduction.

In order that the comparison of a newly read character with the set of characters which can begin a production should be fast, SID forms a word for each production in which the presence of a bit denotes membership of the set. Checking the character is then done by looking for the appropriate bit for the character just read.

References

FLOYD, R. W. (1963). Syntactic Analysis and Operator Precedence, *J. Assoc. Comp. Mach.*, Vol. 10, p. 316.
FLOYD, R. W. (1964). Bounded Context Syntactic Analysis, *Comm. Assoc. Comp. Mach.*, Vol. 7, p. 62.
GREIBACH, SHEILA (1964). Formal Parsing Systems, *Comm. Assoc. Comp. Mach.*, Vol. 7, p. 499.
JOHNSON, J. B. (1965). A Class of unambiguous Computer Languages, *Comm. Assoc. Comp. Mach.*, Vol. 8, p. 147.

Book Review

Selecting the Computer System, by D. N. Chorafas, 1967; 336 pages. (London: *Gee and Co.*, 70s.)

The book deals with the selection of a computer in a commercial environment, and where the author goes into detail, as in Chapter 2 on "unit runs and file specifications", it is all to do with accounting, sales records, stock planning and so on. The message he wants to convey seems to be as follows:

(1) Computers are tremendous, but getting your office organization on to a machine is a very tricky business with pitfalls everywhere.
(2) Make a detailed and exhaustive study of all your activities and be sure that you know exactly what you want to do.
(3) Involve the top management and get their confidence.
(4) Look very critically at any proposal from a manufacturer, make him give you all the details about the machine he is trying to sell you and make sure that he can provide all the supporting services you need.
(5) Look equally critically at any recommendations you have had from a consultant, for their ranks include rogues and incompetents.

One would have thought that this could have been put across in less than 330 pages. I found the book verbose, platitudinous, often irritating and very expensive.

There is a fair amount of information about computers and their use for commercial tasks between its covers, but the author can never leave facts to speak for themselves. In fact, I got the impression the whole book was written to display how clever is Dr. Chorafas. He takes pains to tell us that he has carried out exhaustive researches. He points out to us from time to time (for example, footnote 2 on page 19 and footnote 1 on page 254) the superiority of his own ideas. And of the eleven technical references he gives nine are to his own books and two to other books published by Messrs. Gee. This is the sort of thing which would cause raised eyebrows in the scientific world. Incidentally, in one of these references (page 42) the author is quite exceptionally unfair, for he uses some private technical terms and refers the reader to another of his own books for their definitions.

To conclude, let me quote the remark on page 47 where Dr. Chorafas has been discussing the probable boom in computers.

> "The English market will most probably share this boom, but it is doubtful how much United Kingdom's own computer manufacturers will benefit—the United States manufacturers' invasion has caused several home companies to fold."

J. HOWLETT (Chilton).