ISSN 0084-6198

Algol Bulletin no. 44

MAY 1979

CONTENTS PAGE

The ALGOL BULLETIN is produced under the auspices of the Working Group on ALGOL of the International Federation for Information Processing (IFIP WG2.1, Chairman Robert B. K. Dewar, Courant Institute).

The following statement appears here at the request of the Council of IFIP:

"The opinions and statements expressed by the contributors to this Bulletin do not necessarily reflect those of IFIP and IFIP undertakes no responsibility for any action that might arise from such statements. Except in the case of IFIP documents, which are clearly so designated, IFIP does not retain copyright authority on material published here. Permission to reproduce any contribution should be sought directly from the authors concerned. No reproduction may be made in part or in full of documents or working papers of the Working Group itself without permission in writing from IFIP".

Facilities for the reproduction and distribution of the Bulletin have been provided by Professor Dr. Ir. W. L. van der Poel, Technische Hogeschool, Delft, The Netherlands. Mailing in N. America is handled by the AFIPS office in New York.

The ALGOL BULLETIN is published approximately three times per year, at a subscription of $7 per three issues, payable in advance. Orders and remittances (made payable to IFIP) should be sent to the Editor. Payment may be made in any currency (a list of acceptable approximations in the major currencies will be sent on request), but it is the responsibility of each sender to ensure that cheques etc. are endorsed, where necessary, to conform to the currency requirements of his own country. Subscribers in countries from which the export of currency is absolutely forbidden are asked to contact the Editor, since it is not the policy of IFIP that any person should be debarred from receiving the ALGOL BULLETIN for such a reason.

The Editor of the ALGOL BULLETIN is:
Dr. C. H. Lindsey,
Department of Computer Science,
University of Manchester,
Manchester, M13 9PL,
United Kingdom.

Back numbers, when available, will be sent at $3 each. However, it is regretted that only AB32, AB34, AB35, AB38, AB39, AB41, AB42 and AB43 are currently available. The Editor would be willing to arrange for a Xerox copy of any individual paper to be made for anyone who undertook to pay for the cost of Xeroxing.

AB44.0 EDITOR'S NOTES

The WG, and also its ALGOL 68 Support Subcommittee and its Transput Task Force, met in Summit, New Jersey, at the beginning of April.

On the Algol 68 front, the major event was the acceptance of the Implementation Model of the ALGOL 68 Transput (see AB44.1.1 in this issue). This is intended to make it easier for all implementors of the language to incorporate a correct, compatible and efficient transput system into their compilers. Hans van Vliet worked for one and a half years in modeling and remodeling the transput section of the Revised Report in such a way that it can now be implemented efficiently, still preserving virtually all of the external specifications of the Revised Report. Moreover, his description has the virtues of a good textbook for implementors: it is clear, understandable and precise. The model is based on a buffer concept which provides the proper interface with arbitrary operating systems as they exist nowadays. In recognition of his great achievement, the Model is now always informally referred to as the "Hansput".

Another document accepted at the meeting was the Revised MC Test Set, by D.Grune (see AB44.1.2 in this issue). Originally, this Test Set was prepared as an acceptance test for the CDC ALGOL 68 compiler. It has now been extensively revised, with many new programs added. It is now up to all users who are contemplating purchasing ALGOL 68 compilers to insist that the Test Set be run and an agreed standard for conformance obtained. Only in this way will we gradually force implementors to move towards correct implementations of the language.

Finally, two more Commentaries on the Revised Report were released (see AB44.3.1 in this issue). Now, at last, things seem to have become fairly quiet so far as the appearance of fresh bugs in the Report is concerned.

In spite of all this ALGOL 68 activity, the main part of the meeting was taken up with papers on programming methodology - part of the continuing search for that elusive language "ABSTRACTO". It was decided that the next step should be to prepare an agreed set of example problems against which each proposed "ABSTRACTO" language should be tested. In fact, what we need is lots of concrete ABSTRACTO examples to look at.

At this point, I must make my usual plea for material for the next issue. This issue contains two papers which are more or less in the category of "algorithms", such as I asked for in AB41. More material of the same sort would be particularly welcome.

AB44.1 Announcements

AB44.1.1 An Implementation Model of the ALGOL 68 Transput.

This report has been written by J.C. van Vliet on the request of the Task Force on Transput, which was set up by the Subcommittee on ALGOL 68 Support of IFIP WG2.1. It aims at a precise description of the transput of ALGOL 68, conforming with section 10.3 of the Revised Report. Whereas section 10.3 of the Revised Report describes the intention of transput, the emphasis in this report is on implementability.

A variety of ALGOL 68 implementations exist or are near completion. They all support some kind of transput, although they all differ slightly from each other and from the Revised Report. This diversity renders the transfer of programs from one implementation to the other very difficult, if not virtually impossible.

The existence of so many different transput systems may to some extent be due to the fact that the definition as given in the Revised Report does not really facilitate implementation of the transput. Each implementor again has to struggle his way through the transput section and locate the problems with the particular operating system.

The approach taken is similar to the one in the Revised Report: the transput is described in pseudo-ALGOL 68. The pseudo-ALGOL 68 part can be considered as a language extension which is reasonably implementable. The primitives underlying the model are not defined in ALGOL 68. Instead, their semantics are given in some kind of formalized English, resembling the way in which the semantics of the Revised Report are defined. One advantage of a description in pseudo-ALGOL 68 is that it can largely be tested mechanically. It has been the intention that the ALGOL 68 text, after suitable substitution of the pseudo comments, could be compiled, thereby automatically creating part of the runtime environment.

This report was accepted at the recent meeting of WG2.1 held in Summit, New Jersey, and the WG resolved to ask its parent committee, IFIP TC2, to authorize the following statement for release with it.

> This implementation model of the ALGOL 68 transput has been reviewed by IFIP Working Group 2.1. It has been scrutinized to ensure that it correctly interprets the transput as defined in section 10.3 of the Revised Report. This model is recommended as the basis for actual implementations of the transput.

Copies of the Report can be obtained from the Mathematisch Centrum, 2e Boerhaavestraat 49, 1109 AL Amsterdam at a price of HFl 27 (plus postage). Its full title is

Mathematical Centre Tracts No. 111
ALGOL 68 Transput Part II - An Implementation Model.
by J.C. van Vliet.

(Part I, to be published later, will contain some of the background to and motivations for the model. The two parts taken together will constitute Hans van Vliet's doctoral thesis.) The text of the model is also available in machine-readable form.

AB44.1.2 <u>The Revised MC Test Set.</u>

The Revised MC Test Set comprises 190 ALGOL 68 programs, in part correct ones, in part intentionally incorrect ones. They are designed to explore the full range of ALGOL 68 language features, and include many attempts to trip the compiler up or to uncover incorrect short-cuts.

Many of the programs are pathological and should not be considered as representative of ALGOL 68 programming style. With this in mind, almost all the programs are worth while reading, some as puzzles, some for the good programming features they contain, some for their not widely known programming techniques and a few for their good style.

The test set is not complete, firstly because such a product is never complete: there is no exhaustive testing and one cannot cater for every contingency. Secondly, a few aspects of the language are under-represented (e.g. SHORT and LONG values and bulk I/O). However, if a compiler processes the test set well and also works well on the daily stream of average programs, it may be regarded as a very good compiler. Certainly, all implementors should be encouraged to use it and, especially, to report in their accompanying documentation how it behaved.

The Test Set was accepted at the recent meeting of WG2.1 held in Summit, New Jersey, and the WG resolved to ask its parent committee, IFIP TC2, to authorize the following statement for release with it.

> This ALGOL 68 Test Set has been reviewd by IFIP Working Group 2.1, which wishes to recommend it as a valuable means of testing implementations of ALGOL 68.

Copies of the Test Set will shortly be available from the Mathematisch Centrum, 2e Boerhaavestraat 49, 1109 AL Amsterdam. It will also be available in machine-readable form on most reasonable formats of magnetic tape.

AB44.1.3 TORRIX.

At its meeting in Jablonna, Poland, in August 1978, IFIP WG2.1 authorized the release of the following statement.

> The library package TORRIX comprising definitions for handling vectors and matrices in ALGOL 68, as published in the Mathematical Centre Tracts series, has been scrutinized to ensure that:
>
> a) It strictly conforms to the definition of ALGOL 68.
>
> b) It is consistent with the philosophy and orthogonal framework of that language.
>
> c) It addresses a significant application area in a comprehensive and appropriate manner.
>
> In releasing this statement the intention is to encourage the incorporation of this library package in library preludes of ALGOL 68 implementations.

TORRIX is published as Mathematical Centre Tracts No. 86. Volume 1 is currently available. Volume 2 will be available later in the year. See AB42.1.5 for further information.

AB44.1.4 The Revised Report in German.

The Revised Report on the Algorithmic Language ALGOL 68 has now been translated into German by Prof. Immo O. Kerner of the Paedagogische Hochschule Dresden. It is published by Akademie-Verlag, 108 Berlin, Leipziger Str. 3-4 (list number 202. 100/401/78), under the title "Revidierter Bericht uber die Algorithmische Sprache ALGOL 68".

Although the text is in German, all hyper-rules and paranotions are given in English, as is the standard-prelude (except for comments).

AB44.1.5 An Axiomatic Semantic Definition of ALGOL 68.

This doctoral thesis, by Richard Schwarz, is obtainable from the Computer Science Department, School of Engineering and Applied Science, University of California, Los Angeles, CA 90024, so long as stocks last (after that, it should be obtainable in microfiche from NTIS, Springfield, Virginia 22151).

The report gives a formal axiomatic definition of a major subset of ALGOL 68. The definition, roughly the same length as the axiomatic definition of EUCLID, handles many features generally considered to be serious impediments to program verification. The small set of very general rules governing the semantics of ALGOL 68 leads to a very clean axiomatic definition, defining an extraordinarily expressive language.

It should be required reading for anyone who still believes that side parameters and unrestricted aliasing of names are absolute bars to program verification. They are not. All necessary axioms are given here and, because of the orthogonal structure of the language, the axiomatic definition is surprisingly short.

AB44.1.6. Other reports available.

The following reports are available from the Mathematisch Centrun, 2e Boerhaavestraat 49, 1109 AL Amsterdam.

AFLINK - A new ALGOL 68 - FORTRAN interface, by H.J.Bos and D.T.Winter (Report No. NN 17)

(a tool for use with the CDC ALGOL 68 compiler permitting, especially, ALGOL 68 procedures to be passed as parameters to FORTRAN subroutines).

A Modules and Separate Compilation Facility for ALGOL 68 by C.H.Lindsey and H.J.Boom (Report No. IW 105/78 - HFl 6 plus postage)

(as already published in AB43.3.2).

AB44.3.1

Commentaries on the Revised Report

The following commentaries are issued by the Sub-committee on ALGOL 68 Support, a standing sub-committee of IFIP WG 2.1. They deal with problems which have been raised in connection with the Revised Report on the Algorithmic Language ALGOL 68, and mostly take the form of advice to implementers as to what action they should take in connection with those problems. These commentaries are not to be construed as modifications to the text of the Revised Report.

Note that commentaries are not being published on trivial misprints. Those concerned about such misprints (and especially those preparing new printings of the Report) should apply to the Editor of the ALGOL Bulletin for the latest list of agreed Errata.

{{Commentaries 1 through 30 have already been published (see AB42.3.1 and AB 43.3.1). The two new commentaries published here were accepted by the Support Sub-committee at its meeting in April 1979.}}

31) Overwriting of existing books and control of the write mood.

The Report provides that, where both "put" and "get" are "possible", an existing book with sequential access may be read up to some arbitrary point and overwritten with new information from there onwards. If a given implementation can only overwrite from the beginning of a line, or even from the start of the book, it should be arranged that "put possible" (which is a procedure) only returns TRUE when the current position is at a place from which overwriting may commence.

However, even if "get" is not "possible" (but "put" is), the Report permits such an arbitrary point to be reached by suitable calls of "space", "newline" and "newpage". However, these calls can only be implemented by reading the book from the beginning, counting characters and line and page terminators, and this is impossible by hypothesis. It is indeed strange that even "put(f, newline)" causes the book to be read in order to skip a line. It is even stranger that there is no way in which the first line of an existing book can be overwritten with an empty line. These difficulties all stem from the fact that it is the putting of an actual character which causes the logical end of the file to be retracted to the current position {10.3.3.1.b}. Implementers are therefore advised to test in "set write mood" (10.3.1.4.j) for the case where the logical file end is beyond the current line in a sequential access book, and to retract the logical file end there rather than in "put char". Moreover, it should now be the case that all calls of "put" and "putf", even "put(f, ())", should set the write mood and bring about this effect. To this end, implementors should always call "set write mood" at the start of "put" (10.3.3.1.a) and of "putf" (10.3.5.1.a) {just after the tests for "opened" which are currently provided}. Very few programs will be changed in meaning as a result of this and, moreover, the precise effect defined by the Report can always be obtained by writing "get(f, newline)" (in situations where "get" is "possible", of course).

32) On the scope of the particular-program.

According to the letter of the Revised Report the first environ created during the elaboration of the ENCLOSED-clause of the particular-program is newer in scope than the environ of the user-task in which it is contained. This would imply that the heap scope (see also Commentary 3) is newer than the scope of the variables (in particular "stand in", "stand out" and "stand back") declared in the particular-prelude. As a consequence, the elaboration of, e.g., the call "open(stand in, "", stand in channel)" in the particular-prelude would result in scope violation and thus be undefined. This is, however, not the intention. In effect, the environ in question should be considered nonlocal, so that the scopes concerned are the same. Also, the meaning of the following particular-program should be well defined:

```
      BEGIN on logical file end(stand in, (REF FILE f)BOOL: GOTO lfe);
           DO STRING s; read((s, newline)); print((s, newline)) OD;
     lfe:  print("***eof***")
     END
```

AB44.4.1

EXPERIENCES WITH ALGOL 68 TRANSPUT

PARTS I, II :

Critical remarks on Revised Report's transput section

by Hanno Wupper* July 1978

Preliminary remark

If, from the following pages, readers not familiar with Algol 68 might get the impression that part 10.3 of the Revised Report is a most useless and confusing document written by complete ignorants, the author must apologize and stress that this was not at all his intention. A useful and comfortable transput system has been defined there. The definition consists of a set of Algol 68 programs describing transput activities down to the handling of single characters in the file (or book, as it is called there), thus making clear beyond doubt what has to happen in any particular situation. Algol 68 transput, as well as its method of definition, is in a much better state than the definitions of I/O systems of various other languages and cannot be ignored by anyone working on operating systems or higher level languages.

Not before we actually started an implementation we noticed several disadvantages. The offensive tone of this paper is a result of disappointment due to expectations too optimistic. If some of the statements in this papers can be proved wrong, all the better. The author wishes to express his thanks to J.C.van Vliet from the Mathematical Centre, Amsterdam and to his colleages G.Baszenski, J.Krieger, M.Peuser, N.Voelker, C.-G.Warlich and, above all, Prof.H.Ehlich, who all found time for long discussions of transput. Source of experiences was the work on an implementation for the TR 440, which would have been impossible without the fine Algol 68C bootstrap-kit from Cambridge University and J.C.van Vliet's machine independent transput system.

* Address: Rechenzentrum der Ruhr-Universitaet Bochum,
Postfach 102148, D-4630 Bochum, W. Germany

Part I

UNDERSTANDING ALGOL 68 TRANSPUT

General considerations

1. Algol 68, defined by programs written in three different languages

In his remarks "On the Revised Algol 68 Report" [Algol Bulletin AB36.4.3] M. Sintzoff says that rewriting the old Report had been organized as a programming project. Indeed the Revised Report can be considered as a program defining the Algol 68 machine. It consists of three parts, each written in a different language:

- Syntax, i.e. the set of program texts accepted by the Algol 68 machine is defined by the rules of a two-level van Wijngaarden grammar. Such grammars in principle are powerful enough to simulate any Turing machine, thus allowing the occurrence of several indecidabilities; but in the Revised Report they have been used with care: Some "general hyper-rules" [RR 1.3] provide useful structured programming tools, and most of the syntax part certainly forms an elegant high level program with well chosen "identifiers" (i.e. paranotions).

- Semantics of the Algol 68 machine is defined by programs "expressed in natural language, but making use of some carefully and precisely defined terms and concepts" [RR 0.1.1] (cf. Sintzoff [AB36.4.3 p31]). The structure of the semantics parts is composed from serial, conditional, and case constructs and is closely related to the structure of the corresponding syntax rules.

- Only a nucleus of the Algol 68 machine, consisting of comparatively few "basic constructs" is defined by these means. The machine then is extended by a program written in Algol 68 itself: The important chapters on the standard prelude [RR 10.2] and transput [RR 10.3] mainly consist of operator- and procedure-declarations.

To implement Algol 68 on an arbitrary machine one should have to do nothing but provide translators for two-level grammars and for the special subset of natural language, and then simply "run" the Report.

Unfortunately this is a bit difficult.

Arbitrary two-level grammars cannot be automatically converted to a deterministic recognizer, and the special grammar in the Report seems not to be of a type for which parser constructors have been developed (cf. Deussen).

At least, however, it does not seem too difficult to hand-translate the rules to e.g. an equivalent affix grammar. (Affix grammars luckily may be automatically converted to an executable program, cf. Koster, Watt.)

Though as we see the definition of syntax and semantics as given in the Report is not suitable for automatic translation, it is clear and precise and tells the implementor exactly what to do - though, of couse, "it may be difficult to understand to the 'uninitiated' reader".

The standard prelude as given in part IV of the Report [RR 10.2] in form of several Algol 68 routine texts serves as a useful implementation model: The method of definition here presents no difficulties; some of the operations might even be implemented by translating parts of the Report.

The transput section, however, [RR 10.3] presents a somewhat different situation. It again consists of Algol 68 declarations and again the method of description is understood easily; but then the implementor is left with long programs the intention of which tends to remain obscure. The additional pragmatic remarks sometimes are more confusing than enlightening. Moreover, some of the procedures even seem to contain bugs. In the following chapter we will have a closer look at why this section may be of little help to the implementor.

2. Initial problems: Finding out what to implement

When an implementor who has not been familiar with Algol 68 transput for years, studies section 10.3 of the Revised Report he finds himself completely lost.

At first he does not understand what books, channels, and files really are and what is the difference between them. He probably knows all about his own operating system and about the transput systems for several other programming languages; but the pragmatics of the Report give little help to match what he knows with what he is to implement for his Algol 68 system. Probably he sooner or later gets some idea that a "book" is more or less what is called file in most operating systems, and he surely will find out that that a "file" is no more than a kind of status vector, describing the momentary situation of transput for an open book. But he gets quite mixed up when he wants to know what a "channel" really is. The pragmatics [RR 10.3.1.2] say something about physical devices, possibly useful in nuclear physics and that "a channel is a structured value whose fields are routines returning truth values which determine the available methods of access to a book linked via that channel" - which latter knowledge may as well be extracted from the definition of mode .CHANNEL, where it is expressed more clearly [RR 10.3.1.2.a]. One also learns that a channel has some "channel number", which slightly reminds the reader of Fortran's unit numbers. (The pragmatics say nothing about it, nor seems it to be used anywhere in the Report.)

A channel "corresponds to one or more physical devices" [RR 10.3.1.2] - but in modern operating systems the user does not need to know about physical devices. Maybe a channel really is a unit number plus some useful enquiry routines? But then one learns that several files may be open at one and the same channel a time. Perhaps it is safest to have just one channel and code everything in the file_idf, or to provide one channel for each Fortran logical unit number and allow the idf to be the empty string only? The purpose of channels remains in the dark.

When after all one feels sure what to do with them one can start to implement a transput system. It will, of course, have to behave exactly as described by the Report. But it turns out to be more than a challenging puzzle to find out what really is described there. What one finally has found out often is rather astonishing and not what programmers might expect.

Of course this is a general programming problem. As soon as a language contains as powerful constructs as arithmetic operations, loops, and conditional expressions it may become indecidable what a program really does and whether it computes a certain given function.

Therefore, care should be taken first to specify the problem, then to write a well structured program containing assertions to prove its correctness. Most programs are written the other way round: The programmer's vague idea of what the code should look like is punched in; during the "testing phase" several conditional statements are added, and finally the problem is adjusted to the behaviour of the program.

Several important languages are known to be defined as "what the compiler accepts", and some unexpected unorthogonal restrictions in the Fortran standard may have resulted from the same technique.

Sadly enough, the transput section of the Revised Report gives a similar impression - it even seems to contain serious bugs. Several presumed errors and other problems are listed in an interesting paper compiled by van Vliet [DWA 11/1]. We will mention only a few of them:

- The .BEYOND operator used in establish [10.3.1.4.b] does not test whether a given position is "beyond" another one.

- Even though a special "primal environ" has been introduced some transput calls will violate scope restrictions.

- By definition of mode .INTYPE [10.3.2.2.d] it is not possible to input values of e.g. mode .STRUCT(.BOOL b, .STRING s).

- Input and output are incompatible in several cases.

Hopefully, the programming errors and misprints will be corrected by some official document; more dangerous are the cases where the behaviour of the routines is obscure, or unexpected, or not realistic: Implementors will be likely to deviate slightly or less slightly from the Report. There are already several implementations of different transput systems, not all of them being super- or sublanguages.

But even if one wants to stick to the Report absolutely and implement the transput routines exactly as they are printed there one is in a mess:

Most parts of transput activities are performed by the operating system. What remaines to be done by language dependent routines, besides conversion of values, is the testing of conditions (or "events"), calling event routines, providing default actions, etc. In Algol 68 these actions are strongly connected with routines provided by the user. They are quite complicated logically, but not critical in the sense of CPU time, so it would be sensible to copy them from the Report as they are. However, it turns out to be impossible to draw a line dividing chapter 10.3 of the Report into two appropriate parts:

Especially the layout routines are a conglomerate of actions belonging to all the different layers in an operating system, from user level right down to the physical device.

One cannot copy the routines from the Report; so one has to write a completely new transput system; so one is forced to understand what is defined by the Report, or, more precisely, what is i n t e n d e d there.

Studying the transput section and finding out its intention turns out to be a source of not "innocent merriment" but innumerable surprises.

Possibly the authors had in mind a special, limited machine or operating system with some very particular restrictions. The decision to use the same set of layout routines for both reading and writing causes additional restrictions and problems.

PART II

ALGOL 68 TRANSPUT CONSIDERED INSUFFICIENT

The following chapters try to show that even if all implementation problems are solved one cannot feel too happy because the result will be disappointing.

While implementing Algol 68 transput on our TR440 we came along a number of problems far more serious than the above mentioned difficulties in connection with the method of description: transput, implemented exactly as it is defined in the Report contains a number of unorthogonal restrictions that will surprise the user and possibly give him the impression that Algol 68 possesses one of the most old-fashioned transput systems he finds on his machine. Handling of random access will be inefficient and undesirable, even in a " superlanguage" where some of the unnecessary restrictions have been dropped.

4. Superfluous restrictions

The attributes set_possible, get_possible, put_possible, and bin_possible ought to be mutually independent. There is no reason why changes between char_mood and bin_mood should be forbidden whenever .NOT set possible, or why reading and writing may not be alternated in the singular case of a sequential file used for binary transput.

5. Important types of books not considered at all

The texts of books for Algol 68 always have to resemble a .REF .FLEX [] .FLEX [] .FLEX [] .CHAR; each character being identified by a triple of integers, the page, line, and character numbers. Positions inside the logical file run from .POS(1,1,1) and never contain gaps.

Modern operating systems provide books of a somewhat different structure (sometimes called "index sequential" or "sequential keyed"): There are "record keys" rather than line numbers, taken from an ordered set of, in general, strings or sometimes integers. Arbitrary gaps between keys inside the logical file are possible. Such books may be read sequentially or set to a certain line or to the first existing line after a certain key. Lines may be of arbitrary length. Usually there are no pages, but the concept may be generalized by allowing arbitrary tree structure rather than adding just one more dimension for pages.

Books of that kind or at least useful special cases can be handled by several programming languages and operating systems. The special case with integral keys is quite common, even to certain Basic implementations. Due to its limited nature, standard Fortran I/O can be extended to allow handling of such books in a straightforward way. The more complicated Algol 68 transput, however, suggests more than one way to extend the effects of newline, space, or set, and certainly there are several possibilities for additional layout routines and events (on_undefined_record, delete). Some official document with appropriate recommendations is needed urgently, otherwise there soon will be all kinds of differing implementations.

Another type of book not provided for is the good old magnetic tape: Books on tape may be read backwards! (In most operating systems that is extended to all sequential files on background storage. In Algol 68 the routines space/backspace are not really symmetric and there are no reverse versions of newline or newpage. The effect of Fortran's BACKSPACE can be achieved by no means. Last not least there are no hints how to make use of a dialogue terminal.

6. Random access condemned to inefficiency

The main difference between sequential and random access is that in the latter case the logical_file_end always is close to the character written last, while with random access lines may be overwritten without destroying the information on subsequent lines. The Report tries to make users benefit from the advantages of random access as much as possible - with the result that after each call of newline or set, the old contents of the new line must be available. There are no means to express that one wants to overwrite an entire line without reading it first, though that is the case in most applications.

If the text of a book actually is a multiple value accessible to the program this is, of course, no problem. In a real life operating system, however, where records have to be read from background storage ("or even from a set up in nuclear physics"), it can be too expensive. Perhaps pairs of routines like set_to(p,l,c) vs. get_from(p,l,c) might be what is needed (with space writing blanks if a line was positioned by set_to).

7. Unrealistic behaviour of compressible random access books

Except for the restrictions and inefficiencies mentioned above, all incompressible and all sequential books behave sensibly. Now in [RR10.3.1.6.aa] a most interesting pragmatic remark can be found:

> "Although the effect of a channel whose books are both compressible and of random access is well defined, it is not anticipated that such a combination is likely to occur in actual implementations."

At first glance no harm seems to be in that statement, though, of course, assumptions on what manufacturers might provide in an operating system could be hazardous. Compressibility is orthogonal to and independent of random accessibility. Users of our TR440 in fact have been declaring nearly all their books compressible with random access for the last eight years. Both properties are implemented efficiently, and when the price is low one usually chooses the best available.

The shock occurred when, after starting an implementation of Algol 68 transput, we had a closer look at what actually is well defined there: Once a line has been written to a random access book its length cannot be changed any more (exept by scratch), even if the book is compressible. Just look at the last example of [RR 10.3.1.3]: It will not work if set_possible (f1) .AND compressible(f1) and if the lines of the book happen to be shorter than int width.

This is a severe restriction and quite against the philosophy of random access.

CONCLUSION

If Algol 68 is meant to be a serious alternative to other existing languages, a revision and extension of transput is needed.

It should be stated here that C.H.Lindsey's recent article "Algol 68 and your friendly Neighbourhood Operating System" [AB 42.4.4 p.22] probably will be of great help to all implementors. Lindsey gives a comprehensive overview of implementation problems and possible solutions. Indeed many of the problems mentioned occurred to us during the last year. We found similar solutions, but not without long discussions and a lot of thinking. Most of the problems we felt unable to solve, however, are not dealt with by Lindsey.

Literature

P.Deussen: A decidability criterion for van Wijngaarden Grammars, Acta Informatica 5, 353-375 (1975)

C.H.A.Koster: Affix grammars, in: Algol68 Implementation, North Holland, 1971

D.A.Watt: Analysis-oriented two-level grammars, Ph.D.thesis, TU Berlin 1974

van Wijngaarden et al.: Revised Report on the Algorithmic Language Algol68, Acta Informatica 5, Fasc.1-3, 1975

AB44.4.2

Changing Line Lengths in Random Files.

C.J.Cheney (University of Cambridge).
C.H.Lindsey (University of Manchester).
L.G.L.T.Meertens (Mathematical Centre, Amsterdam).
H.Wupper (Ruhr University).

In a book that is both random access and compressible, the lines (and pages) may become of any length up to the physical limit when it is first written by sequential means (the only way to move the logical file end away from the position (1,1,1) where it is left by "establish"). After that, there is no way in which individual lines (or pages) can be shortened or lengthened. Implementers whose operating system provides a convenient system for doing this may care to provide an environment enquiry "clip line possible", and to implement the following "clipping" superlanguage feature.

In addition to the logical end of file pointer {10.3.1.6.cc} maintained by the implementer, let there be an additional "local logical end" pointer which points to some position in the current line and which normally points to the end of the line unless the logical end of file is in that line, in which case the two pointers coincide.

Let a procedure "clip line", of mode PROC(REF FILE)VOID, be provided whose effect, on books and channels where its use is permitted, is to set "write mood" and then to expand the current line to some physical limit (as in the pseudo comment in "put char" {10.3.3.1.b+25}) and to set the logical end to the current position. Subsequent calls of "put" within this line push the local end forward just as is done with the logical end of file in "put char" at present {10.3.3.1.b+21}. The contents of the line between the local logical end and the physical end are inaccessible (calls of the "on line end" event would be made), and calls of "space" in this area write blanks. Whenever the logical file end is within the current line, the two logical file pointers are always moved together.

As soon as the line ceases to be current (due to a call of "newline" or "set" or even "close"), the line is compressed to wherever the local logical end has now reached (or, if the book is not "compressible", it is filled with blanks to its physical end). Thus, a user who wishes to rewrite a line in the middle of his random-access file "set"s to the start of the line (or to the middle of it if he only wishes to rewrite the last part of it) and calls "clip line". He then "put"s new characters as required and in due course when he calls "newline" or "set"s elsewhere, the line will be rewritten with its new contents and (if "compressible") its new length.

Although it is not necessarily suggested that the "clip line" facility should necessarily be possible on books and channels other than those for which "compressible", "put" and "set" are all "possible", it may be observed that its properties are in fact well defined in other cases (and useful applications can even be imagined). Also, it is clear that "clip page" and "clip file" facilities could also be defined (and even implemented) in an entirely analagous manner.

AB44.4.3 # A Finite State Lexical Analyzer for the Standard Hardware Representation of ALGOL 68.

by H.B.M. Jonkers
(Mathematisch Centrum, Amsterdam)

ABSTRACT

A finite state lexical analyzer for ALGOL 68 programs written in the standard hardware representation is described. The analyzer is written in a very simple language, allowing semi-mechanical translation to an arbitrary language. The whole language, including format-texts, is dealt with.

KEY WORDS & PHRASES: ALGOL 68, lexical analysis, finite state machine, semi-mechanical translation.

1. INTRODUCTION

For two reasons the lexical analysis of ALGOL 68 programs is not as trivial as might be expected. First of all at some places (e.g., TAO-symbols) the lexical structure of ALGOL 68 is rather awkward. Secondly ALGOL 68 programs can be represented in different stropping regimes [1]. A lexical analyzer for ALGOL 68 featuring all three stropping regimes has already been published [2]. Apart from the deviations from [1] mentioned in the next paragraph, the lexical analyzer described here differs from [2] in the following points:

(1) It basically is a finite state machine. This allows a wide range of implementation methods to be applied and adds to efficiency.
(2) It is described in a very simple language, allowing semi-mechanical translation to an arbitrary language (e.g., machine language). The lexical analyzer was in fact tested by translating it into an ALEPH program using a text editor.
(3) All parts of programs are dealt with, including format-texts.
(4) The description is hopefully more accessible and more readable than [2].

The lexical analyzer takes as its input program texts representing ALGOL 68 particular-programs in the standard hardware representation [1], allowing the following deviations from [1]:

(1) Besides worthy characters all characters occurring in section 9.4.1. of [3] are allowed; for a list of all characters accepted by the lexical analyzer see appendix 1. If only worthy characters are to be accepted, this can be achieved by adding a preprocessor to the lexical analyzer accepting worthy characters only.
(2) Besides the three stropping regimes defined in [1], a fourth regime is provided, the STROP regime. In the STROP regime, tags and bolds are represented as they are in POINT stropping, with the

addition of the following rule:

- A bold word may be written as a strop ("′"), followed, in order, by the worthy letters or digits corresponding to the bold-faced letters or digits in the word, followed by a strop. If the bold word is followed by a disjunctor other than a strop, the last strop may be omitted.

(3) In the RES regime the point may be omitted from a bold word if it is preceded by a digit from an integral-, real- or bits-denotation (cf. [4]).

The output of the lexical analyzer consists of "tokens", which we shall call "words" (as in [2]) to prevent confusion, since there already is an ALGOL 68 paranotion "token". The exact definition of a "word" is given in section 3. Roughly speaking a "word" corresponds to an ALGOL 68 denotation, comment or NOTION-symbol. Each time the lexical analyzer is activated, it delivers a word. By repeated activation of the lexical analyzer the program text will be transformed into a stream of words. If the program text corresponds to an ALGOL 68 particular program in the standard hardware representation (augmented as above), the stream of words will correspond to this particular program in a way more fully described in section 2. If the program text does not satisfy the specifications of the standard hardware representation, the lexical analyzer will generate one or more error messages. Otherwise the program text, and consequently the stream of words, does not correspond to an ALGOL 68 program. If the lexical analyzer is part of a compiler, this will lead to an error message at a higher level in the compiler.

The lexical analyzer itself consists of four separate lexical analyzers, one for each stropping regime. The first advantage of this is an increase of efficiency: it is no longer necessary to inspect the environment continually during lexical analysis to determine which stropping regime we are in. Second, if we don′t want to allow all of the stropping regimes, we can simply omit the lexical analyzers for one or more of the stropping regimes. In this way, we are not burdened with the details of stropping regimes which are not allowed anyway, as would be the case with a lexical analyzer in which all stropping regimes are integrated. A disadvantage seems to be the size of such a lexical analyzer when allowing more than one stropping regime. However, since the lexical analyzers for the different stropping regimes differ from each other at only a limited number of places, large parts of them can be combined. This combination of the separate lexical analyzers is not difficult and is left to the implementer (see also note 1 in section 7). The coordination of the separate lexical analyzers during lexical analysis must be taken care of by the global routine using them (e.g., a parser). We shall call this routine the "supervisor".

As the lexical analyzer is composed of four lexical analyzers, one for every stropping regime, so is in turn each lexical analyzer made up of two analyzers: the "unit level lexical analyzer" and the "format level lexical analyzer". The unit level lexical analyzer is designed to analyze program text at the unit and pragmat level, assuming that the interior of pragmats has a somewhat ALGOL 68-like structure. Comments are automatically skipped by the unit level lexical analyzer. The format level lexical analyzer is designed to analyze program text at the format-text level, comments also being skipped automatically. A considerable part of the unit and format level lexical analyzer coincides, so they can partially be combined. The supervisor must coordinate the unit and format level lexical analyzer. We shall often use "the lexical analyzer" to mean one of the separate (unit or format level) lexical analyzers.

For reasons of efficiency, the model of a finite transducer has been chosen for the lexical analyzer, i.e., the lexical analyzer can be viewed as a program for a finite state machine. The description of this machine is found in section 3. The machine is completely described in ALGOL 68 by a number of data structures and a number of operations on these data structures, which we shall call "instructions". Moreover, a number of predicates on these data structures is given, which we shall call "conditions". These "conditions" are used to enable conditional state transitions. We point out here beforehand, that this method of description has only been chosen for the sake of clarity and is not the best way to implement the machine (see section 7). To describe the program which is to run on this machine, we use a mini language called ALEX, defined in sections 4 and 5. Programs in ALEX are closely related to right-linear (transduction) grammars. The entire lexical analyzer was in fact constructed by transforming context-free grammars for the different words into right-linear grammars and subsequently combining these into an ALEX program. The lexical analyzer program itself is listed in section 6.

2. WORDS

A word is a value with a structure (a "mode") described by the following ALGOL 68 declaration:

```
mode word = struct (int mark, string info);
```

The words generated by the lexical analyzer are described below. For each value of the mark field the corresponding paranotion(s) is (are) given. For each value of the info field the corresponding representation of the paranotion in the reference language is given, omitting typographical display features (the Greek letter "ξ" is used to indicate a character). Values of the mark field are indicated by names in upper case letters. Values of the info field are indicated by strings (without embracing quotes), "ε" indicating the empty string.

Remarks:

(1) It is not always possible for a finite state machine to determine whether an "=" at the end of a TAO-symbol belongs to this TAO-symbol or not (see also [2]). In case of doubt the TAO-symbol and the "=" are packed together into one word with mark = SHORTOP EQUALSETY (in contrast with the algorithm in [2]). Words with mark = SHORTOP EQUALSETY are the only words that may correspond to a sequence of more than one symbols (see the second column in the table below).

(2) For some applications the filling of the info field of some words might have to be changed. For example, if comments should not be discarded, the info field of a word with mark = COMMENT could be filled with the comment text. In general, no fundamental changes in the lexical analyzer are needed for this. In most cases the insertion and/or deletion of a few "instructions" in the lexical analyzer program will suffice.

(3) For the value "EOF" of the mark field no corresponding paranotion is given since there is none. A word with mark = EOF is used to indicate the end of the word stream.

mark	paranotion	info	representation
TAG	TAG-symbol.	$\xi_1 \ldots \xi_n$	$\xi_1 \ldots \xi_n$
BOLD	bold-TAG-symbol.	$\xi_1 \ldots \xi_n$	$\xi_1 \ldots \xi_n$
	except:		
	bold-comment-symbol;		
	style-i-comment-symbol;		
	bold-pragmat-symbol;		
	style-i-pragmat-symbol.		
INT	integral-denotation.	$\xi_1 \ldots \xi_n$	$\xi_1 \ldots \xi_n$
REAL	real-denotation.	$\xi_1 \ldots \xi_n$	$\xi_1 \ldots \xi_n$
BITS	bits-denotation.	$\xi_1 \ldots \xi_n$	$\xi_1 \ldots \xi_n$
SHORTOP	DOP-BECOMESETY-symbol.	$\xi_1 \ldots \xi_n$	$\xi_1 \ldots \xi_n$
	except:		
	equals-symbol;		
	tilde-symbol.		
SHORTOP	DYAD-cum-equals-symbol;	$\xi_1 \ldots \xi_n=$	$\xi_1 \ldots \xi_n=$
EQUALSETY	DYAD-symbol,		
	is-defined-as-symbol;		
	DYAD-cum-equals-cum-		
	becomes-symbol;		
	DYAD-cum-assigns-to-symbol,		
	is-defined-as-symbol.		
STRING	string-denotation.	$\xi_1 \ldots \xi_n$	"$\xi_1 \ldots \xi_n$"
CHAR	character-denotation.	ξ	"ξ"
BECOMES	becomes-symbol.	ε	:=
IS	is-symbol.	ε	:=:
ISNOT	is-not-symbol.	ε	:≠:
		ε	:/=:
STICKCOLON	brief-else-if-symbol;	ε	|:
	brief-ouse-symbol.		
EQUALS	equals-symbol;	ε	=
	is-defined-as-symbol.		
TILDE	tilde-symbol;	ε	~
	skip-symbol.		
STICK	brief-then-symbol;	ε	|
	brief-else-symbol;		
	brief-in-symbol;		
	brief-out-symbol.		
COLON	label-symbol;	ε	:
	colon-symbol;		
	up-to-symbol;		
	routine-symbol.		
COMMA	and-also-symbol.	ε	,
SEMICOLON	go-on-symbol.	ε	;
OPEN	brief-begin-symbol;	ε	(
	brief-if-symbol;		
	brief-case-symbol;		
	style-i-sub-symbol.		
CLOSE	brief-end-symbol;	ε	)
	brief-fi-symbol;		
	brief-esac-symbol;		
	style-i-bus-symbol.		
SUB	brief-sub-symbol.	ε	[
BUS	brief-bus-symbol.	ε	]

AT	at-symbol.	ε	@
NIL	nil-symbol.	ε	∘
DOLLAR	formatter-symbol.	ε	$
COMMENT	comment.	¢	$¢\xi_1\ldots\xi_n¢$
		comment	$\underline{\text{comment}}\,\xi_1\ldots\xi_n\,\underline{\text{comment}}$
		co	$\underline{\text{co}}\,\xi_1\ldots\xi_n\,\underline{\text{co}}$
		#	$\#\xi_1\ldots\xi_n\#$
PRAGSYM	bold-pragmat-symbol;	pragmat	$\underline{\text{pragmat}}$
	style-i-pragmat-symbol.	pr	$\underline{\text{pr}}$
EOF		ε	

The following words can be generated by the format level lexical analyzer exclusively:

CHARROW	string-denotation; character-denotation.	$\xi_1\ldots\xi_n$	$"\xi_1\ldots\xi_n"$
FIXNUM	fixed-point-numeral.	$\xi_1\ldots\xi_n$	$\xi_1\ldots\xi_n$
ASYM	letter-a-symbol.	ε	a
BSYM	letter-b-symbol.	ε	b
CSYM	letter-c-symbol.	ε	c
DSYM	letter-d-symbol.	ε	d
ESYM	letter-e-symbol.	ε	e
FSYM	letter-f-symbol.	ε	f
GSYM	letter-g-symbol.	ε	g
ISYM	letter-i-symbol.	ε	i
KSYM	letter-k-symbol.	ε	k
LSYM	letter-l-symbol.	ε	l
NSYM	letter-n-symbol.	ε	n
PSYM	letter-p-symbol.	ε	p
QSYM	letter-q-symbol.	ε	q
RSYM	letter-r-symbol.	ε	r
SSYM	letter-s-symbol.	ε	s
XSYM	letter-x-symbol.	ε	x
YSYM	letter-y-symbol.	ε	y
ZSYM	letter-z-symbol.	ε	z
POINT	point-symbol.	ε	.
PLUS	plus-symbol.	ε	+
MINUS	minus-symbol.	ε	-

3. MACHINE

The lexical analyzer programs are described in a language called ALEX (see sections 4 and 5). ALEX programs describe a series of actions of a "machine". This machine is described below by a set of ALGOL 68 declarations. The machine consists of a number of data structures, a number of actions on the data structures, called "instructions", and a number of predicates on the data structures, called "conditions". The "instructions" are used in ALEX programs to denote primitive actions of the machine. The "conditions" are used to make decisions dependent upon the value of the machine data structures.

```
# 1. Data structures. #

struct (int state, string buffer) status;

# The variable "status" represents the status of the machine.
  The "state" field holds the current state of the machine.
  The "buffer" field is used to cope with lookahead. #

string input;
char head;

# The variable "input" represents the input file.
  The variable "head" is used to temporarily save the first character of
  "input". #

struct (int mark, string info) word;

# The variable "word" is used to pass information on the token which has
  been read to the outside world. #

int match index;
bool match possible;

# The variables "match index" and "match possible" are used for pattern
  matching purposes inside comments, thus allowing an efficient skipping of
  comments. #

# 2. Auxiliary definitions. #

char eof = ...;

# "eof" is used as an end of file marker and must be some character that
  cannot occur in the input. #

op norm = (char ch) char:
   if   ch = "A" then "a"
   elif ch = "B" then "b"
   .
   .
   .
   elif ch = "Z" then "z"
   else ch
   fi;

# We need the operator "norm" because of the fact that with a few
  exceptions the two cases of a letter are equivalent. #

proc write = (string s) void: info of word +:= s;

op head = (string s) char: s[1];
op tail = (string s) string: s[2 : upb s];

proc reserved = (string s) bool:
   (s = "at" or s = "begin" or ... or s = "while");
proc comment = (string s) bool:
   (s = "co" or s = "comment");
proc pragmat = (string s) bool:
   (s = "pr" or s = "pragmat");

# 3. Instructions. #

proc put = void: write(norm head);
proc putitem = void: write(head);
proc save = void: buffer of status +:= norm head;
proc clear = void: buffer of status := "";
proc append = void: begin write(buffer of status); clear end;
proc reread = void: begin buffer of status +=: input; clear end;
proc read = void: head := if input = "" then eof else head input fi;
proc next = void: input := tail input;

proc point = void: write(".");
proc zero = void: write("0");
proc quote = void: write("""");
proc strop = void: write("´´");
proc equals = void: write("=");
proc tilde = void: write("~");
proc colon = void: write(":");
proc differs = void: write("≠");
proc divided = void: write("/");

proc reset = void: begin match index := 0; match possible := true end;
proc match = void:
   if match possible
   then if match index < upb info of word
        then match index +:= 1;
             match possible := norm head = info[match index]
        else match possible := false
        fi
   fi;

proc error = (int n) void: ...;

# What should be done when an error occurs is left to the implementer.
  For error diagnostics, see appendix 2. #

# 4. Conditions. #
```

```
proc reservedinfo = bool: reserved(info of word);
proc reservedbuffer = bool: reserved(buffer of status);
proc commentinfo = bool: comment(info of word);
proc commentbuffer = bool: comment(buffer of status);
proc pragmatinfo = bool: pragmat(info of word);
proc pragmatbuffer = bool: pragmat(buffer of status);
proc two = bool: info of word = "2";
proc four = bool: info of word = "4";
proc eight = bool: info of word = "8";
proc sixteen = bool: info of word = "16";
proc sizeone = bool: upb info of word = 1;
proc sizetwo = bool: upb info of word = 2;
proc sizethree = bool: upb info of word = 3;

proc matching = bool:
   (match possible and match index = upb info of word);
```

4. SYNTAX OF ALEX

ALEX programs syntactically resemble right-linear grammars. The only difference is that to every production rule a (possibly empty) "action", and to every "single production" rule a (possibly empty) "condition" is associated. If we omit the "actions" and "conditions", what remains is a pure right-linear grammar. In the case of the lexical analyzer described here, this grammar generates an (infinite) stream of ALGOL 68 symbols in the standard hardware representation. The syntax of ALEX is given by a van Wijngaarden grammar. The van Wijngaarden grammar is used here only in its most simple form, viz. as an abbreviation mechanism for a context free grammar. The syntax introduces a terminology, which is used in the next section to define the semantics of ALEX.

```
PRODUCTIVITY::
     productive;
     nonproductive.
program:
     transduction rule sequence.
transduction rule:
     PRODUCTIVITY transduction rule.
PRODUCTIVITY transduction rule:
     defined state, colon symbol, PRODUCTIVITY transduction rule body.
defined state:
     state.
PRODUCTIVITY transduction rule body:
     PRODUCTIVITY alternative sequence option, out alternative.
PRODUCTIVITY alternative:
     PRODUCTIVITY condition, transduction, go on symbol.
productive condition:
     charset.
nonproductive condition:
     sub symbol, condition, bus symbol.
transduction:
     curly open symbol, action, curly close symbol, applied state.
action:
     empty;
     mark;
     instruction list;
     instruction list, and also symbol, mark.
applied state:
     state.
out alternative:
     transduction.
```

Some notions are not defined in the syntax; we define them informally below.

state	: a state of the machine.
charset	: a set of characters.
condition	: a predicate on the machine data structures.
instruction	: an operation on the machine data structures.
mark	: a value of the mark field of a word.

In addition, an ALEX program must satisfy the following conditions:

(1) All charsets in a productive transduction rule are disjoint.
(2) All conditions in a nonproductive transduction rule are mutually exclusive.
(3) All defined states are different.
(4) All applied states occur as a defined state.

Remarks:

(1) A termination condition for ALEX programs could be added without great difficulty. However, since we only use ALEX for the description of the lexical analyzer, we shall omit this. Termination of the constituent programs of the lexical analyzer (see section 6) can be verified rather easily.
(2) A transduction rule with a body consisting of an out alternative only can be parsed as a productive as well as a nonproductive transduction rule. Since in this case both kinds of transduction rules are semantically equivalent, the ambiguity causes no harm.

5. SEMANTICS OF ALEX

We shall define the semantics of an ALEX program by translating it into a pseudo ALGOL 68 procedure operating on the machine described in section 3.

TRANSLATION OF A PROGRAM:

Let P be an ALEX program.
P = "R1 ... Rn",
where R1, ... , Rn are transduction rules.
The translation TRANS(P) of P is defined as:

```
TRANS(P) = "proc p = void:
            begin word := (skip, "");
               goto state of status;
               TRANS(R1);
               .
               .
               .
               TRANS(Rn);
               exit:
            end"
```

TRANSLATION OF A TRANSDUCTION RULE:

Let R be a transduction rule.

(1) R is a productive transduction rule.
R = "S: C1 T1; ... ; Cn Tn; T0.",
where S is a state,
C1, ... , Cn are charsets,
T0, ... , Tn are transductions.
The translation TRANS(R) of R is defined as:

If n = 0:

```
TRANS(R) = "S: TRANS(T0)"
```

If n > 0:

```
TRANS(R) = "S: read;
               if   head in C1 then next; TRANS(T1)
               elif head in C2 then next; TRANS(T2)
               .
               .
               .
               elif head in Cn then next; TRANS(Tn)
               else TRANS(T0)
               fi"
```

N.B.
The instruction "read" does not remove a character from the string

"input" ("next" does). It merely assigns the head of "input" to "head".

(2) R is a nonproductive transduction rule.
R = "S: [B1] T1; ... ; [Bn] Tn; T0.",
where S is a state,
B1, ... , Bn are conditions,
T0, ... , Tn are transductions.
The translation TRANS(R) of R is defined as:

If n = 0:

```
TRANS(R) = "S: TRANS(T0)"
```

If n > 0:

```
TRANS(R) = "S: if   B1 then TRANS(T1)
               elif B2 then TRANS(T2)
               .
               .
               .
               elif Bn then TRANS(Tn)
               else TRANS(T0)
               fi"
```

TRANSLATION OF A TRANSDUCTION:

Let T be a transduction.

(1) T = "{I1, ... , In} S",
where I1, ... , In are instructions,
S is a state.
The translation TRANS(T) of T is defined as:

```
TRANS(T) = "I1; ... ; In;
            state of status := S;
            goto S"
```

(2) T = "{I1, ... , In, M} S",
where I1, ... , In are instructions,
M is a mark,
S is a state.
The translation TRANS(T) of T is defined as:

```
TRANS(T) = "I1; ... ; In;
            mark of word := M;
            state of status := S;
            goto exit"
```

6. PROGRAMS

There are eight ALEX programs constituting the lexical analyzer, one for each pair (level, regime), where level is UNIT or FORMAT and regime is POINT, UPPER, RES or STROP. Large parts of these programs are textually equal. Rather than listing them all in their full length, we shall combine them in a single listing and use two variables "level" and "regime" inside the text to indicate what part of the text belongs to what program. So the program for level = l and regime = r can be constructed by simply erasing all text with level ≠ l or regime ≠ r.

Remarks:

(1) The names of the states have been chosen so as to indicate the string of characters that has been read so far.
(2) All charsets occurring in the transduction rules are listed in appendix 1, except for the charset "other". The latter is not a fixed charset but, if it occurs in a transduction rule T, it is equal to the set of all characters (except "eof") that are not element of a charset of T (other than "other").
(3) The state "STRINGESCAPE" has been provided to enable the use of the strop character as an escape character inside character and string denotations. If the strop character is to be used this way, the transduction rule for this state must be modified.
(4) Before the first activation of a program the machine must be initialized properly. This initialization should be done by the supervisor and should read:

```
     status := (EMPTY, "");
```

LISTING OF THE PROGRAMS

```
level = UNIT
|
| regime = POINT
| |
| | EMPTY:
| |     letter {put} TAG;
| |     point {} POINT;
| |     digit {put} FIX;
| |     quote {} QUOTE STRING;
| |     equals {} EQUALS;
| |     tilde {} TILDE;
| |     dyad {put} DYAD;
| |     stick {} STICK;
| |     colon {} COLON;
| |     comma {COMMA} EMPTY;
| |     semicolon {SEMICOLON} EMPTY;
| |     open {OPEN} EMPTY;
| |     close {CLOSE} EMPTY;
| |     sub {SUB} EMPTY;
| |     bus {BUS} EMPTY;
| |     at {AT} EMPTY;
| |     nil {NIL} EMPTY;
| |     dollar {DOLLAR} EMPTY;
| |     cent {put} BRIEFCOMMENT;
| |     cross {put} STYLEIICOMMENT;
| |     typo {} EMPTY;
| |     other {error(1)} EMPTY;
| |     {EOF} EMPTY.
|
| regime = UPPER
| |
| | EMPTY:
| |     lowerletter {put} TAG;
| |     upperletter {put} POINTETY UPPERBOLD;
| |     point {} POINT;
| |     digit {put} FIX;
| |     quote {} QUOTE STRING;
| |     equals {} EQUALS;
| |     tilde {} TILDE;
| |     dyad {put} DYAD;
| |     stick {} STICK;
| |     colon {} COLON;
| |     comma {COMMA} EMPTY;
| |     semicolon {SEMICOLON} EMPTY;
| |     open {OPEN} EMPTY;
| |     close {CLOSE} EMPTY;
| |     sub {SUB} EMPTY;
| |     bus {BUS} EMPTY;
| |     at {AT} EMPTY;
| |     nil {NIL} EMPTY;
| |     dollar {DOLLAR} EMPTY;
| |     cent {put} BRIEFCOMMENT;
| |     cross {put} STYLEIICOMMENT;
| |     typo {} EMPTY;
| |     other {error(1)} EMPTY;
| |     {EOF} EMPTY.
```

```
|
| regime = RES
| |
| | EMPTY:
| |    letter {put} TAGBOLD;
| |    point {} POINT;
| |    digit {put} FIX;
| |    quote {} QUOTE STRING;
| |    equals {} EQUALS;
| |    tilde {} TILDE;
| |    dyad {put} DYAD;
| |    stick {} STICK;
| |    colon {} COLON;
| |    comma {COMMA} EMPTY;
| |    semicolon {SEMICOLON} EMPTY;
| |    open {OPEN} EMPTY;
| |    close {CLOSE} EMPTY;
| |    sub {SUB} EMPTY;
| |    bus {BUS} EMPTY;
| |    at {AT} EMPTY;
| |    nil {NIL} EMPTY;
| |    dollar {DOLLAR} EMPTY;
| |    cent {put} BRIEFCOMMENT;
| |    cross {put} STYLEIICOMMENT;
| |    typo {} EMPTY;
| |    other {error(1)} EMPTY;
| |    {EOF} EMPTY.
|
| regime = STROP
| |
| | EMPTY:
| |    letter {put} TAG;
| |    point {} POINT;
| |    strop {} STROP;
| |    digit {put} FIX;
| |    quote {} QUOTE STRING;
| |    equals {} EQUALS;
| |    tilde {} TILDE;
| |    dyad {put} DYAD;
| |    stick {} STICK;
| |    colon {} COLON;
| |    comma {COMMA} EMPTY;
| |    semicolon {SEMICOLON} EMPTY;
| |    open {OPEN} EMPTY;
| |    close {CLOSE} EMPTY;
| |    sub {SUB} EMPTY;
| |    bus {BUS} EMPTY;
| |    at {AT} EMPTY;
| |    nil {NIL} EMPTY;
| |    dollar {DOLLAR} EMPTY;
| |    cent {put} BRIEFCOMMENT;
| |    cross {put} STYLEIICOMMENT;
| |    typo {} EMPTY;
| |    other {error(1)} EMPTY;
| |    {EOF} EMPTY.
|
| regime = POINT, STROP
| |
| | TAG:
| |    letgit {put} TAG;
| |    typoscore {} TAG TYPOSCORE;
| |    {TAG} EMPTY.
| |
| | TAG TYPOSCORE:
| |    letgit {put} TAG;
| |    typo {} TAG TYPOSCORE;
| |    {TAG} EMPTY.
|
| regime = UPPER
| |
| | TAG:
| |    lowerletgit {put} TAG;
| |    underscore {} TAG UNDERSCORE;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
| |
| | TAG UNDERSCORE:
| |    lowerletgit {put} TAG;
| |    upperletter {save, error(5), TAG} POINTETY UPPERLETTER;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
| |
| | TAG TYPO:
| |    lowerletgit {put} TAG;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
|
| regime = RES
| |
| | TAGBOLD:
| |    letgit {put} TAGBOLD;
| |    underscore {} TAG UNDERSCORE;
| |    typo {} TAGBOLD TYPO;
| |    {} TAGBOLD END.
| |
| | TAGBOLD TYPO:
| |    [reservedinfo] {} BOLD;
| |    {} TAG TYPO.
| |
| | TAGBOLD END:
| |    [reservedinfo] {} BOLD;
| |    {TAG} EMPTY.
| |
| | TAG:
| |    letgit {put} TAG;
| |    underscore {} TAG UNDERSCORE;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
| |
| | TAG UNDERSCORE:
| |    letgit {put} TAG;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
| |
| | TAG TYPO:
| |    letter {save} TAG BOLDETY;
| |    digit {put} TAG;
| |    typo {} TAG TYPO;
| |    {TAG} EMPTY.
| |
| | TAG BOLDETY:
| |    letgit {save} TAG BOLDETY;
| |    underscore {append} TAG UNDERSCORE;
| |    typo {} TAG BOLDETY TYPO;
| |    {} TAG BOLDETY END.
| |
| | TAG BOLDETY TYPO:
| |    [reservedbuffer] {TAG} SAVEDBOLD;
| |    {append} TAG TYPO.
| |
| | TAG BOLDETY END:
| |    [reservedbuffer] {TAG} SAVEDBOLD;
| |    {append, TAG} EMPTY.
| |
| | SAVEDBOLD:
| |    {append} BOLD.
|
| regime = POINT, RES, STROP
| |
| | POINT:
| |    letter {put} POINT BOLD;
| |    digit {point, put} VAR;
| |    typo {} POINT TYPO;
| |    {error(3)} EMPTY.
| |
| | POINT TYPO:
| |    digit {point, put} VAR;
| |    typo {} POINT TYPO;
| |    {error(3)} EMPTY.
| |
| | POINT BOLD:
| |    letgit {put} POINT BOLD;
| |    underscore {error(6)} BOLD;
| |    {} BOLD.
|
| regime = UPPER
| |
| | POINT:
| |    lowerletter {put} POINT LOWERBOLD;
| |    upperletter {put} POINTETY UPPERBOLD;
| |    digit {point, put} VAR;
| |    typo {} POINT TYPO;
| |    {error(3)} EMPTY.
| |
| | POINT TYPO:
| |    digit {point, put} VAR;
| |    typo {} POINT TYPO;
| |    {error(3)} EMPTY.
| |
| | POINT LOWERBOLD:
| |    lowerletgit {put} POINT LOWERBOLD;
| |    underscore {error(6)} BOLD;
| |    {} BOLD.
| |
| | POINTETY UPPERBOLD:
| |    upperletgit {put} POINTETY UPPERBOLD;
| |    underscore {error(6)} BOLD;
| |    {} BOLD.
|
| regime = STROP
| |
| | STROP:
| |    letter {put} STROP BOLD;
| |    {error(4)} EMPTY.
| |
| | STROP BOLD:
| |    letgit {put} STROP BOLD;
| |    strop {} BOLD;
| |    underscore {error(6)} BOLD;
| |    {} BOLD.
|
| regime = POINT, UPPER, RES, STROP
| |
| | BOLD:
| |    [commentinfo] {} BOLDCOMMENT;
| |    [pragmatinfo] {PRAGSYM} EMPTY;
| |    {BOLD} EMPTY.
|
| regime = POINT, RES, STROP
| |
| | FIX:
| |    digit {put} FIX;
| |    point {} FIX POINT;
| |    ten {put} STAG POWER;
| |    letter e {save} FIX E;
| |    letter r {save} FIX R;
| |    typo {} FIX;
| |    {INT} EMPTY.
| |
| | FIX POINT:
| |    digit {point, put} VAR;
| |    letter {save, INT} POINT LETTER;
| |    typo {point} FIX POINT TYPO;
| |    {point, zero, error(8)} VAR.
|
| regime = UPPER
| |
| | FIX:
| |    digit {put} FIX;
| |    point {} FIX POINT;
| |    ten {put} STAG POWER;
| |    lowerletter e {save} FIX E;
| |    lowerletter r {save} FIX R;
| |    typo {} FIX;
| |    {INT} EMPTY.
| |
| | FIX POINT:
| |    digit {point, put} VAR;
| |    lowerletter {save, INT} POINT LOWERLETTER;
| |    upperletter {save, INT} POINTETY UPPERLETTER;
| |    typo {point} FIX POINT TYPO;
| |    {point, zero, error(8)} VAR.
|
| regime = POINT, UPPER, RES, STROP
| |
| | FIX POINT TYPO:
| |    digit {put} VAR;
| |    typo {} FIX POINT TYPO;
| |    {zero, error(8)} VAR.
| |
| | FIX E:
| |    digit {append, put} FLO;
| |    sign {append, put} STAG POWER SIGN;
| |    typo {append} STAG POWER;
| |    {INT} LEGGLE.
|
| regime = POINT, UPPER, STROP
| |
| | FIX R:
| |    [two] {} RADIX R(1);
| |    [four] {} RADIX R(2);
| |    [eight] {} RADIX R(3);
| |    [sixteen] {} RADIX R(4);
| |    {INT} LEGGLE.
|
| regime = RES
| |
| | FIX R:
| |    [two] {} RADIX R(1);
| |    [four] {} RADIX R(2);
| |    [eight] {} RADIX R(3);
| |    [sixteen] {} HEXBITS LEGGLE;
| |    {INT} LEGGLE.
|
| regime = POINT, RES, STROP
| |
| | VAR:
| |    digit {put} VAR;
| |    ten {put} STAG POWER;
| |    letter e {save} VAR E;
| |    typo {} VAR;
| |    {REAL} EMPTY.
|
| regime = UPPER
| |
| | VAR:
| |    digit {put} VAR;
| |    ten {put} STAG POWER;
| |    lowerletter e {save} VAR E;
| |    typo {} VAR;
| |    {REAL} EMPTY.
|
| regime = POINT, UPPER, RES, STROP
| |
| | VAR E:
| |    digit {append, put} FLO;
| |    sign {append, put} STAG POWER SIGN;
| |    typo {append} STAG POWER;
| |    {REAL} LEGGLE.
| |
| | STAG POWER:
| |    digit {put} FLO;
| |    sign {put} STAG POWER SIGN;
| |    typo {} STAG POWER;
| |    {zero, error(9), REAL} EMPTY.
| |
| | STAG POWER SIGN:
| |    digit {put} FLO;
| |    typo {} STAG POWER SIGN;
| |    {zero, error(9), REAL} EMPTY.
| |
| | FLO:
| |    digit {put} FLO;
| |    typo {} FLO;
| |    {REAL} EMPTY.
|
| regime = POINT, RES, STROP
| |
| | RADIX R(n):
| |    radigit(n) {append, put} BITS(n);
| |    noradletgit(n) {save, INT} LEGGLE;
| |    typo {append} RADIX R TYPO(n);
| |    {append, zero, error(10), BITS} EMPTY.
| |
| | RADIX R TYPO(n):
| |    radigit(n) {put} BITS(n);
| |    typo {} RADIX R TYPO(n);
| |    {zero, error(10), BITS} EMPTY.
| |
| | BITS(n):
| |    radigit(n) {put} BITS(n);
| |    typo {} BITS(n);
| |    {BITS} EMPTY.
|
| regime = UPPER
| |
| | RADIX R(n):
| |    lowerradigit(n) {append, put} BITS(n);
| |    lowernoradletgit(n) {save, INT} LEGGLE;
| |    typo {append} RADIX R TYPO(n);
| |    {append, zero, error(10), BITS} EMPTY.
| |
| | RADIX R TYPO(n):
| |    lowerradigit(n) {put} BITS(n);
| |    typo {} RADIX R TYPO(n);
| |    {zero, error(10), BITS} EMPTY.
| |
| | BITS(n):
| |    lowerradigit(n) {put} BITS(n);
| |    typo {} BITS(n);
| |    {BITS} EMPTY.
|
| regime = RES
| |
| | HEXBITS:
| |    digit {put} HEXBITS;
| |    hexletter {save} HEXBITS LEGGLE;
| |    nohexletter {save} HEXBITS LEGGLE END;
| |    typo {} HEXBITS;
| |    {} HEXBITS END.
| |
| | HEXBITS END:
| |    [sizethree] {zero, error(10), BITS} EMPTY;
| |    {BITS} EMPTY.
| |
| | HEXBITS LEGGLE:
| |    digit {append, put} HEXBITS;
| |    hexletter {save} HEXBITS LEGGLE;
| |    nohexletter {save} HEXBITS LEGGLE END;
| |    typo {append} HEXBITS;
| |    {append} HEXBITS END.
| |
| | HEXBITS LEGGLE END:
| |    [sizetwo] {INT} LEGGLE;
| |    [sizethree] {zero, error(10), BITS} LEGGLE;
| |    {BITS} LEGGLE.
|
| regime = POINT, UPPER, STROP
| |
| | LEGGLE:
| |    {append} TAG.
|
| regime = RES
| |
| | LEGGLE:
| |    {append} TAGBOLD.
|
| regime = POINT, RES, STROP
| |
| | POINT LETTER:
| |    {append} POINT BOLD.
|
| regime = UPPER
| |
| | POINT LOWERLETTER:
| |    {append} POINT LOWERBOLD.
| |
| | POINTETY UPPERLETTER:
| |    {append} POINTETY UPPERBOLD.
|
| regime = POINT, UPPER, RES, STROP
| |
| | STRINGRETURN:
| |    [sizeone] {CHAR} EMPTY;
| |    {STRING} EMPTY.
| |
| | EQUALS:
| |    equals {equals, equals} DYAD EQUALS;
| |    nomad {equals, put} DYAD NOMAD;
| |    colon {} EQUALS COLON;
| |    {EQUALS} EMPTY.
| |
| | EQUALS COLON:
| |    equals {equals, colon, equals, SHORTOP} EMPTY;
| |    {EQUALS} COLON.
| |
| | TILDE:
| |    equals {tilde, equals} DYAD EQUALS;
| |    nomad {tilde, put} DYAD NOMAD;
| |    colon {tilde} DYAD NOMADETY COLON;
| |    {TILDE} EMPTY.
| |
| | DYAD:
| |    equals {equals} DYAD EQUALS;
| |    nomad {put} DYAD NOMAD;
| |    colon {} DYAD NOMADETY COLON;
| |    {SHORTOP} EMPTY.
| |
| | DYAD EQUALS:
| |    equals {} DYAD NOMAD EQUALS;
| |    colon {colon} DYAD EQUALS COLON;
| |    typo {} SHORTOP EQUALSETY TYPOSETY;
| |    {SHORTOP EQUALSETY} EMPTY.
| |
| | DYAD EQUALS COLON:
| |    equals {equals} SHORTOP EQUALSETY TYPOSETY;
| |    {SHORTOP} EMPTY.
| |
DYAD NOMAD:
   equals {} DYAD NOMAD EQUALS;
   colon {} DYAD NOMADETY COLON;
   {SHORTOP} EMPTY.

DYAD NOMAD EQUALS:
   colon {equals, colon, SHORTOP} EMPTY;
   {SHORTOP} EQUALS.

DYAD NOMADETY COLON:
   equals {colon, equals, SHORTOP} EMPTY;
   {SHORTOP} COLON.

SHORTOP EQUALSETY TYPOSETY:
   equals {SHORTOP} EQUALS;
   typo {} SHORTOP EQUALSETY TYPOSETY;
   {SHORTOP EQUALSETY} EMPTY.

STICK:
   colon {STICKCOLON} EMPTY;
   {STICK} EMPTY.

COLON:
   equals {} COLON EQUALS;
   differs {} COLON DIFFERS;
   divided {} COLON DIVIDED;
   {COLON} EMPTY.

COLON EQUALS:
   colon {IS} EMPTY;
   {BECOMES} EMPTY.

COLON DIFFERS:
   colon {} COLON DIFFERS COLON;
   {COLON} DIFFERS.

COLON DIFFERS COLON:
   equals {COLON} DIFFERS COLON EQUALS;
   {ISNOT} EMPTY.

COLON DIVIDED:
   equals {} COLON DIVIDED EQUALS;
   {COLON} DIVIDED.

COLON DIVIDED EQUALS:
   colon {ISNOT} EMPTY;
   {COLON} DIVIDED EQUALS.

DIFFERS:
   {differs} DYAD.

DIFFERS COLON EQUALS:
   {differs, colon, equals, SHORTOP} EMPTY.
```

```
| | DIVIDED:
| |    {divided} DYAD.
| |
| | DIVIDED EQUALS:
| |    {divided, equals} DYAD EQUALS.
```

<u>level = FORMAT</u>

```
|
| regime = POINT
| |
| | EMPTY:
| |    letter {save, reread} LETGITS;
| |    point {} POINT;
| |    digit {put} FIX;
| |    quote {} QUOTE STRING;
| |    plus {PLUS} EMPTY;
| |    minus {MINUS} EMPTY;
| |    comma {COMMA} EMPTY;
| |    open {OPEN} EMPTY;
| |    close {CLOSE} EMPTY;
| |    dollar {DOLLAR} EMPTY;
| |    cent {put} BRIEFCOMMENT;
| |    cross {put} STYLEIICOMMENT;
| |    typo {} EMPTY;
| |    other {error(2)} EMPTY;
| |    {EOF} EMPTY.
|
| regime = UPPER
| |
| | EMPTY:
| |    lowerletter {save, reread} LETGITS;
| |    upperletter {save} POINTETY UPPERTAGGLE;
| |    point {} POINT;
| |    digit {put} FIX;
| |    quote {} QUOTE STRING;
| |    plus {PLUS} EMPTY;
| |    minus {MINUS} EMPTY;
| |    comma {COMMA} EMPTY;
| |    open {OPEN} EMPTY;
| |    close {CLOSE} EMPTY;
| |    dollar {DOLLAR} EMPTY;
| |    cent {put} BRIEFCOMMENT;
| |    cross {put} STYLEIICOMMENT;
| |    typo {} EMPTY;
| |    other {error(2)} EMPTY;
| |    {EOF} EMPTY.
|
| regime = RES
| |
| | EMPTY:
| |    letter {save} TAGGLE;
| |    point {} POINT;
| |    digit {put} FIX;
| |    quote {} QUOTE STRING;
| |    plus {PLUS} EMPTY;
| |    minus {MINUS} EMPTY;
| |    comma {COMMA} EMPTY;
| |    open {OPEN} EMPTY;
| |    close {CLOSE} EMPTY;
| |    dollar {DOLLAR} EMPTY;
| |    cent {put} BRIEFCOMMENT;
| |    cross {put} STYLEIICOMMENT;
| |    typo {} EMPTY;
| |    other {error(2)} EMPTY;
| |    {EOF} EMPTY.
|
| regime = STROP
| |
| | EMPTY:
| |    letter {save, reread} LETGITS;
| |    point {} POINT;
| |    strop {} STROP;
| |    digit {put} FIX;
| |    quote {} QUOTE STRING;
| |    plus {PLUS} EMPTY;
| |    minus {MINUS} EMPTY;
| |    comma {COMMA} EMPTY;
| |    open {OPEN} EMPTY;
| |    close {CLOSE} EMPTY;
| |    dollar {DOLLAR} EMPTY;
| |    cent {put} BRIEFCOMMENT;
| |    cross {put} STYLEIICOMMENT;
| |    typo {} EMPTY;
| |    other {error(2)} EMPTY;
| |    {EOF} EMPTY.
|
| regime = RES
| |
| | TAGGLE:
| |    letgit {save} TAGGLE;
| |    {} TAGGLE END.
| |
| | TAGGLE END:
| |    [commentbuffer] {append} POINTETY COMMENT;
| |    [pragmatbuffer] {append} POINTETY PRAGMAT;
| |    {reread} LETGITS.
|
| regime = POINT, RES, STROP
| |
| | LETGITS:
| |    letter a {ASYM} LETGITS;
| |    letter b {BSYM} LETGITS;
| |    letter c {CSYM} LETGITS;
| |    letter d {DSYM} LETGITS;
| |    letter e {ESYM} LETGITS;
| |    letter f {FSYM} LETGITS;
| |    letter g {GSYM} LETGITS;
| |    letter i {ISYM} LETGITS;
| |    letter k {KSYM} LETGITS;
| |    letter l {LSYM} LETGITS;
| |    letter n {NSYM} LETGITS;
| |    letter p {PSYM} LETGITS;
| |    letter q {QSYM} LETGITS;
| |    letter r {RSYM} LETGITS;
| |    letter s {SSYM} LETGITS;
| |    letter x {XSYM} LETGITS;
| |    letter y {YSYM} LETGITS;
| |    letter z {ZSYM} LETGITS;
| |    hjmotuvw {error(2)} LETGITS;
| |    digit {put} FIX;
| |    {} EMPTY.
|
| regime = UPPER
| |
| | LETGITS:
| |    lowerletter a {ASYM} LETGITS;
| |    lowerletter b {BSYM} LETGITS;
| |    lowerletter c {CSYM} LETGITS;
| |    lowerletter d {DSYM} LETGITS;
| |    lowerletter e {ESYM} LETGITS;
| |    lowerletter f {FSYM} LETGITS;
| |    lowerletter g {GSYM} LETGITS;
| |    lowerletter i {ISYM} LETGITS;
| |    lowerletter k {KSYM} LETGITS;
| |    lowerletter l {LSYM} LETGITS;
| |    lowerletter n {NSYM} LETGITS;
| |    lowerletter p {PSYM} LETGITS;
| |    lowerletter q {QSYM} LETGITS;
| |    lowerletter r {RSYM} LETGITS;
| |    lowerletter s {SSYM} LETGITS;
| |    lowerletter x {XSYM} LETGITS;
| |    lowerletter y {YSYM} LETGITS;
| |    lowerletter z {ZSYM} LETGITS;
| |    lowerhjmotuvw {error(2)} LETGITS;
| |    digit {put} FIX;
| |    {} EMPTY.
|
| regime = POINT, RES, STROP
| |
| | POINT:
| |    letter {save} POINT TAGGLE;
| |    {POINT} EMPTY.
| |
| | POINT TAGGLE:
| |    letgit {save} POINT TAGGLE;
| |    {} POINT TAGGLE END.
| |
| | POINT TAGGLE END:
| |    [commentbuffer] {append} POINTETY COMMENT;
| |    [pragmatbuffer] {append} POINTETY PRAGMAT;
| |    {reread, POINT} LETGITS.
|
| regime = UPPER
| |
| | POINT:
| |    lowerletter {save} POINT LOWERTAGGLE;
| |    upperletter {save} POINTETY UPPERTAGGLE;
| |    {POINT} EMPTY.
| |
| | POINT LOWERTAGGLE:
| |    lowerletgit {save} POINT LOWERTAGGLE;
| |    {} POINT LOWERTAGGLE END.
| |
| | POINT LOWERTAGGLE END:
| |    [commentbuffer] {append} POINTETY COMMENT;
| |    [pragmatbuffer] {append} POINTETY PRAGMAT;
| |    {reread, POINT} LETGITS.
| |
| | POINTETY UPPERTAGGLE:
| |    upperletgit {save} POINTETY UPPERTAGGLE;
| |    {} POINTETY UPPERTAGGLE END.
| |
| | POINTETY UPPERTAGGLE END:
| |    [commentbuffer] {append} POINTETY COMMENT;
| |    [pragmatbuffer] {append} POINTETY PRAGMAT;
| |    {clear, error(7)} EMPTY.
|
| regime = POINT, UPPER, RES, STROP
| |
| | POINTETY COMMENT:
| |    underscore {error(6)} BOLDCOMMENT;
| |    {} BOLDCOMMENT.
| |
| | POINTETY PRAGMAT:
| |    underscore {error(6), PRAGSYM} EMPTY;
| |    {PRAGSYM} EMPTY.
|
| regime = STROP
| |
| | STROP:
| |    letter {save} STROP TAGGLE;
| |    {error(4)} EMPTY.
| |
| | STROP TAGGLE:
| |    letgit {save} STROP TAGGLE;
| |    {} STROP TAGGLE END.
| |
| | STROP TAGGLE END:
| |    [commentbuffer] {append} STROP COMMENT;
| |    [pragmatbuffer] {append} STROP PRAGMAT;
| |    {reread, error(4)} LETGITS.
| |
| | STROP COMMENT:
| |    strop {} BOLDCOMMENT;
| |    underscore {error(6)} BOLDCOMMENT;
| |    {} BOLDCOMMENT.
| |
| | STROP PRAGMAT:
| |    strop {PRAGSYM} EMPTY;
| |    underscore {error(6), PRAGSYM} EMPTY;
| |    {PRAGSYM} EMPTY.
|
| regime = POINT, UPPER, RES, STROP
| |
| | FIX:
| |    digit {put} FIX;
| |    typo {} FIX TYPO;
| |    {FIXNUM} LETGITS.
| |
| | FIX TYPO:
| |    digit {put} FIX;
| |    typo {} FIX TYPO;
| |    {FIXNUM} EMPTY.
| |
| | STRINGRETURN:
| |    {CHARROW} EMPTY.

level = UNIT, FORMAT
|
| regime = POINT, UPPER, RES, STROP
| |
| | QUOTE STRING:
| |    quote {} QUOTE STRING QUOTE;
| |    strop {} QUOTE STRING STROP;
| |    item {putitem} QUOTE STRING;
| |    control {} QUOTE STRING;
| |    other {error(11)} QUOTE STRING;
| |    {error(13)} STRINGRETURN.
| |
| | QUOTE STRING QUOTE:
| |    quote {quote} QUOTE STRING;
| |    typo {} QUOTE STRING QUOTE TYPO;
| |    {} STRINGRETURN.
| |
| | QUOTE STRING QUOTE TYPO:
| |    quote {} QUOTE STRING;
| |    typo {} QUOTE STRING QUOTE TYPO;
| |    {} STRINGRETURN.
| |
| | QUOTE STRING STROP:
| |    strop {strop} QUOTE STRING;
| |    {} STRINGESCAPE.
| |
| | STRINGESCAPE:
| |    {strop, error(12)} QUOTE STRING.
| |
| | BRIEFCOMMENT:
| |    cent {COMMENT} EMPTY;
| |    other {} BRIEFCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | STYLEIICOMMENT:
| |    cross {COMMENT} EMPTY;
| |    other {} STYLEIICOMMENT;
| |    {error(14), COMMENT} EMPTY.
|
| regime = POINT
| |
| | BOLDCOMMENT:
| |    point {} BOLDCOMMENT POINT;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT POINT:
| |    letter {reset, match} BOLDCOMMENT POINT TAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT POINT TAGGLE:
| |    letgit {match} BOLDCOMMENT POINT TAGGLE;
| |    underscore {} BOLDCOMMENT;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT ENDTEST:
| |    [matching] {COMMENT} EMPTY;
| |    {} BOLDCOMMENT.
|
| regime = UPPER
| |
| | BOLDCOMMENT:
| |    upperletter {reset, match} BOLDCOMMENT POINTETY UPPERTAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT POINT:
| |    lowerletter {reset, match} BOLDCOMMENT POINT LOWERTAGGLE;
| |    upperletter {reset, match} BOLDCOMMENT POINTETY UPPERTAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT POINT LOWERTAGGLE:
| |    lowerletgit {match} BOLDCOMMENT POINT LOWERTAGGLE;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT POINTETY UPPERTAGGLE:
| |    upperletgit {match} BOLDCOMMENT POINTETY UPPERTAGGLE;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT UNDERSCORE:
| |    upperletter {} BOLDCOMMENT UNDERSCORE UPPERTAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT UNDERSCORE UPPERTAGGLE:
| |    upperletgit {} BOLDCOMMENT UNDERSCORE UPPERTAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    underscore {} BOLDCOMMENT UNDERSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT ENDTEST:
| |    [matching] {COMMENT} EMPTY;
| |    {} BOLDCOMMENT.
|
| regime = RES
| |
| | BOLDCOMMENT:
| |    letter {reset, match} BOLDCOMMENT TAGGLE;
| |    digiscore {} BOLDCOMMENT LETGITSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT TAGGLE:
| |    letgit {match} BOLDCOMMENT TAGGLE;
| |    underscore {} BOLDCOMMENT LETGITSCORE;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT LETGITSCORE:
| |    letgitscore {} BOLDCOMMENT LETGITSCORE;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT ENDTEST:
| |    [matching] {COMMENT} EMPTY;
| |    {} BOLDCOMMENT.
|
| regime = STROP
| |
| | BOLDCOMMENT:
| |    point {} BOLDCOMMENT POINT;
| |    strop {} BOLDCOMMENT STROP;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT POINT:
| |    letter {reset, match} BOLDCOMMENT POINT TAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    strop {} BOLDCOMMENT STROP;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT POINT TAGGLE:
| |    letgit {match} BOLDCOMMENT POINT TAGGLE;
| |    underscore {} BOLDCOMMENT;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT STROP:
| |    letter {reset, match} BOLDCOMMENT STROP TAGGLE;
| |    point {} BOLDCOMMENT POINT;
| |    strop {} BOLDCOMMENT STROP;
| |    other {} BOLDCOMMENT;
| |    {error(14), COMMENT} EMPTY.
| |
| | BOLDCOMMENT STROP TAGGLE:
| |    letgit {match} BOLDCOMMENT STROP TAGGLE;
| |    strop {} BOLDCOMMENT ENDTEST;
| |    underscore {} BOLDCOMMENT;
| |    {} BOLDCOMMENT ENDTEST.
| |
| | BOLDCOMMENT ENDTEST:
| |    [matching] {COMMENT} EMPTY;
| |    {} BOLDCOMMENT.
```

7. IMPLEMENTATION NOTES

Essentially the lexical analyzer described here is a finite state machine. Implementation techniques for finite state machines are well known, so we shall not discuss them here. Nevertheless there are some details, largely pertaining to the method of description of the lexical analyzer, that should get some attention. We discuss them below.

(1) The lexical analyzer consists of eight separate programs, one for each pair (level, regime). If more than one such program is needed, one might wish to combine coinciding parts of these programs. An obvious way to do this, is to turn common sets of states representing a submachine of the finite state machine into procedures or subroutines. Such sets of states are, for instance, the sets of states for the reading of short operators, strings and comments. The degree of interweaving can even be increased by combining "similar" states, such as the "EMPTY" states, into a single state. Pushing this interweaving too far, however, can easily lead to a loss of efficiency, because it requires a frequent inspection of the current regime and/or level.

(2) The "append" instruction can be implemented by copying the "buffer" to the "info" and subsequently clearing the buffer (as described in section 3). However, it can be seen that if the buffer is not empty, the only instructions executed on info and buffer are "save", "append" and "clear". So the concatenation of info and buffer behaves like a stack. Therefore we can implement them as:

```
string infobuff;
int sep;
```

where

```
infobuff[1 : sep]
```

represents the info, and

```
infobuff[sep+1 : upb infobuff]
```

represents the buffer. An "append" instruction now boils down to:

```
sep := upb infobuff;
```

(3) In the description of the machine the input is represented as a string, while in fact it most likely is a file. This can give some problems implementing the "reread" instruction. The "reread" instruction appends the contents of the buffer to the head of the input and clears the buffer. This instruction is only used in the format level programs (so if we only need the unit level programs, the problem does not exist). It can be implemented by copying the buffer to a special lookahead buffer and (after clearing the buffer) start reading from this lookahead buffer instead of the input file. It can easily be seen that as long as the lookahead buffer is not empty, no characters are "saved", i.e. put in the buffer. So one might be tempted not to copy the buffer at all and

use the buffer itself as the lookahead buffer. By doing so, however, the stack behavior of the concatenation of info and buffer will get lost, because it is possible that a "put" instruction must be executed with a nonempty buffer (it is possible to restore the stack behavior though, but this is rather tricky). So if the info and buffer are concatenated as described in (2), one should not use the buffer as the lookahead buffer.

REFERENCES

[1] HANSEN, W.J. and H. BOOM,
Report on the Standard Hardware Representation for ALGOL 68,
Algol Bulletin 40 (1976) 24-43.

[2] BELL, R.,
A Token Recognizer for the Standard Hardware Representation of ALGOL 68,
Algol Bulletin 41 (1977) 47-70.

[3] WIJNGAARDEN, A. VAN, et al. (eds.),
Revised Report on the Algorithmic Language ALGOL 68,
Acta Informatica 5 (1975) 1-236.

[4] HANSEN, W.J.,
Trouble Spots in the Standard Hardware Representation for ALGOL 68,
Algol Bulletin 42 (1978) 11-13.

APPENDIX 1: CHARSETS

All worthy characters (including both upper and lower case letters) plus all characters of the reference language (including some control characters) may occur in the input, i.e. the following characters are allowed:

a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 . 10 " space . ∨ ∧ & ≠ < ≤ ≥ > /
÷ % ⌷ ⌊ ⌈ ⊥ ⌉ ˜ ↓ ↑ + − = × * , ; ( ) | : [ ] @ ∘ $
¢ # ′ _ newline newpage

The charsets applied in section 6 are defined below. A set of characters is denoted by a list of its elements surrounded by curly brackets, each element separated by a blank. Furthermore we use "+" for set union and "-" for set difference. The charset "item" is not defined; it must be equal to the set of all characters that are allowed as a string item.

at	= {@}
bus	= {]}
cent	= {¢}
close	= {)}
colon	= {:}
comma	= {,}
control	= {newline newpage}
cross	= {#}
differs	= {≠}
digiscore	= digit + underscore
digit	= {0 1 2 3 4 5 6 7 8 9}
divided	= {/}
dollar	= {$}
dyad	= nomad + {∨ ∧ & ≠ ≤ ≥ ÷ % ⌷ ⌊ ⌈ ⊥ ⌉ ↓ ↑ + −}
equals	= {=}
hexletter	= {a b c d e f A B C D E F}
hjmotuvw	= {h j m o t u v w H J M O T U V W}
letgit	= letter + digit
letgitscore	= letgit + underscore
letter	= lowerletter + upperletter
letter a	= {a A}
letter b	= {b B}
letter c	= {c C}
letter d	= {d D}
letter e	= {e E}
letter f	= {f F}
letter g	= {g G}
letter i	= {i I}
letter k	= {k K}
letter l	= {l L}
letter n	= {n N}
letter p	= {p P}
letter q	= {q Q}
letter r	= {r R}

letter s	=	{s S}
letter x	=	{x X}
letter y	=	{y Y}
letter z	=	{z Z}
lowerhjmotuvw	=	{h j m o t u v w}
lowerletgit	=	lowerletter + digit
lowerletter	=	{a b c d e f g h i j k l m n o p q r s t u v w x y z}
lowerletter a	=	{a}
lowerletter b	=	{b}
lowerletter c	=	{c}
lowerletter d	=	{d}
lowerletter e	=	{e}
lowerletter f	=	{f}
lowerletter g	=	{g}
lowerletter i	=	{i}
lowerletter k	=	{k}
lowerletter l	=	{l}
lowerletter n	=	{n}
lowerletter p	=	{p}
lowerletter q	=	{q}
lowerletter r	=	{r}
lowerletter s	=	{s}
lowerletter x	=	{x}
lowerletter y	=	{y}
lowerletter z	=	{z}
lowernoradletgit(n)	=	lowerletgit - lowerradigit(n)
lowerradigit(1)	=	{0 1}
lowerradigit(2)	=	{0 1 2 3}
lowerradigit(3)	=	{0 1 2 3 4 5 6 7}
lowerradigit(4)	=	{0 1 2 3 4 5 6 7 8 9 a b c d e f}
minus	=	{-}
nil	=	{∘}
nohexletter	=	letter - hexletter
nomad	=	{< > / × *}
noradletgit(n)	=	letgit - radigit(n)
open	=	{(}
plus	=	{+}
point	=	{.}
quote	=	{"}
radigit(1)	=	{0 1}
radigit(2)	=	{0 1 2 3}
radigit(3)	=	{0 1 2 3 4 5 6 7}
radigit(4)	=	{0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F}
semicolon	=	{;}
sign	=	{+ -}
stick	=	{|}
strop	=	{′}
sub	=	{[}
ten	=	{$_{10}$}
tilde	=	{~}
typo	=	{space .̲} + control
typoscore	=	typo + underscore
underscore	=	{_}
upperletgit	=	upperletter + digit
upperletter	=	{A B C D E F G H I J K L M N O P Q R S T U V W X Y Z}

APPENDIX 2: ERROR DIAGNOSTICS

error

1 Illegal character at the unit level.
Character skipped.
2 Illegal character at the format level.
Character skipped.
3 Unidentified point.
Point skipped.
4 Unidentified strop.
Strop skipped.
5 Bold preceded by underscore.
Underscore skipped.
6 Bold followed by underscore.
Underscore skipped.
7 Illegal bold word at the format level.
Bold word skipped.
8 No digits in fractional part of real denotation.
Zero inserted.
9 No digits in exponent part of real denotation.
Zero inserted.
10 No radix digits in bits denotation.
Zero inserted.
11 Illegal string item.
Character skipped.
12 Strop not followed by strop in character or string denotation.
Strop inserted.
13 End of file in character or string denotation.
Quote inserted.
14 End of file in comment.
Comment symbol inserted.

AB44.4.4 The Use of Algol 68 Pattern Matching to Describe a Formal Logic System

by V.J. Rayward-Smith,

School of Computing Studies and Accountancy,

University of East Anglia, Norwich, NR4 7TJ.

Abstract

Axioms of formal logic cannot be defined in a context-free manner and thus standard parsing techniques cannot be used in their recognition. This paper describes how SNOBOL-like pattern matching techniques are applied to overcome the parsing problem in a package of routines used in the teaching of formal arithmetic. The routines are written in Algol 68 using pattern matching facilities described by Housden and Kotarski (1977).

§1. Introduction

In his paper McGettrick (1976) discusses the problems of dealing with logical expressions in a computer aided learning environment. In this paper we describe the Logic Teaching Package (LTP), a package of Algol 68 routines used to teach students the concepts of formal arithmetic. LTP is designed for the use of third year undergraduates undertaking a course in Mathematical Logic. This course uses the well-known book by Kleene (1952) as its text and, in this paper, we discuss how LTP covers those concepts introduced in Chapter IV of the book.

It is not difficult to recognise strings conforming to Kleene's definitions of *variable, term* or *formula*. In fact, since all these can be described in a context-free manner, standard parsing techniques can be used. However, this is not the case for the majority of the axioms defined by Kleene and thus such standard parsing techniques are not generally applicable.

The problem is overcome by using SNOBOL-like character string pattern matching techniques (Griswold, Poage and Polonsky, 1971) to recognise the various constructs. Because of the author's personal preference for Algol 68, LTP is written in the language and the pattern matching facilities used are those described in Housden and Kotarski (1977). This album of modes, operators and procedures which provides SNOBOL-like pattern matching facilities for the Algol 68 programmer has been well tested by successive classes of Computing Studies undergraduates on a Data Structures course taught in the second year of a B.Sc. programme at U.E.A. (Housden and Rayward-Smith, 1975).

The recognition of a correct proof in formal arithmetic demonstrates the immense power of pattern matching techniques. There have been some attempts to determine precisely the class of languages recognised by restricted sets of the pattern matching facilities but the problem remains unsolved for the general case. Fleck (1971) shows that the context-free languages can be recognised using just the primitive pattern null, string constants and variables together with alternation, concatenation, unevaluated expression and assignment operators. In later works, Fleck (1975, 1978) describes the two classes of languages recognised by these pattern matching facilities with the addition of (a) complementation and (b) immediate value assignment. If both complementation and immediate assignment are added then pattern matching can be used to recognise the extended context-free languages as defined in Liu (1977). In this paper, we describe the use of pattern matching to recognise a language that is not an extended context-free language. This is achieved by the use of deferred evaluation of pattern procedures. The theoretical limitations of such a technique have yet to be explored fully although a start has been made (Fleck, 1978).

§2. Variables, terms and formulae

Clearly, patterns can be defined which correspond to the definitions of *variable*, *term* and *formula* found in Kleene (1952). However, one of the niggling problems in handling logical expressions is that of removal and insertion of parentheses. The conventions for omission of parentheses are given in Kleene (1952) together with rules for restoring an expression to its fully parenthesised form. Rather than cope with all the problems inherent in this, LTP firstly translates any term or formula presented to it into an equivalent Polish form (Łukasiewicz, 1951). For example,

both $a \supset b \vee c \& d$

and $a \supset ((b \vee c) \& d)$

are translated into $\supset (a) \& \vee (b)(c)(d)$

Note that during translation several checks are carried out. Firstly, all variables are recognised; unquantified variables are surrounded by (and) but quantified variables together with the quantifier remain in square brackets. For example,

$[\exists c](c' + a = b)$

is translated to $[\exists c] = +'(c)(a)(b)$

Because variables can consist of an arbitrary number of characters, these two conventions simplify later processing. Secondly, during the translation phase the sprurious space characters are removed since it is assumed that spaces are everywhere insignificant in the student's input. Thirdly, the existence of any illegal input symbols is reported to the student user by suitable error messages. Lastly, the algorithm to translate an infix expression into its equivalent prefix form enables any mismatching of parentheses to be reported during the translation phase. The operators are ranked in the order $\supset$, &, $\vee$, $\sim$, $\forall x$, $\exists x$, $=$, $+$, $\cdot$, $'$, where x is any variable and the tighter the operator binds, the further to the right it appears in this list.

After completion of the translation phase, *variable, term* and *formula* are defined using the following patterns and procedures, the notation being that of Housden and Kotarski (1977).

```
string letter = "abcdefghijklmnopqrstuvwxyz", digit = "0123456789",
       zero = "0", ob = "(", cb = ")", os = "[", cs = "]";
proc max no = (pattern p) pattern: (ref pattern q = heap pattern;
       q:= p + *q or null);
pattern number:= span (digit),
       quantifier:= exists or forall c the strings exists and forall
                                                represent ∃ and ∀ c,
       connective:= and or or or imply c the strings and, or, imply
                                                represent ∧, ∨, ⊃ c,
       variable := ob + break (cb) + cb,
       qvariable:= os + break (cs) + cs,
       term:= max no (patt prime) +
              (zero or variable or (plus or dot) + *term + *term),
       formula:= equals + *term + *term
                 or *connective + *formula + *formula
                 or not + *formula
                 or *qvariable + *formula
```

The operator * when applied to an object of mode ref pattern produces a pattern which preserves the "name" of its pattern argument and not its value. This deferred evaluation of patterns is used both to avoid unnecessary copying during pattern construction and also in the construction of recursive pattern expressions. The recursive definition in *max no* enables one to construct a pattern q to match as many occurrences as possible of the pattern p in any input string.

The user of LTP is not made aware of the internal representation but is simply provided with routines allowing him to test whether the strings he defines represent variables, terms or formulae and to test his understanding of the conventions for omission of parentheses. Routines are also provided to check the student's understanding of scope. The scope of a binary operator is found from the internal form by searching from the operator for the first two occurrences of formula-patterns. Hence, in ⊃(a)&∨(b)(c)(d), the scope of & is found to be ∨(b)(c)(d). If the operator is unary, the scope is simply the first such formula-pattern discovered. Since quantified variables are regarded as unary operators, the scope of a quantified variable can be similarly found, enabling LTP to check whether a given occurrence of a variable in a formula or term is bound or free.

A procedure *markbound* is available which marks with a dollar sign every bound occurrence of a variable in a formula. So, if *markbound* is applied to [∃c]=+'(c)(a)(b), both occurrences of c will be marked resulting in [∃c$]=+'(c$)(a)(b), but if *markbound* is applied to ∨[∃c]=+'(c)(a)(b)=(c)(a), the third occurrence of c will not be marked. When substituting terms for variables, it is important to distinguish the free and bound occurrences of variables since the substitution of a term t for a variable x in a formula A consists of simultaneously replacing only the free occurrences of x in A by occurrences of t. A term t is free for x in A(x) if no free occurrence of x in A(x) is in the scope of a quantifier ∀y or ∃y, where y is a variable of t. To check for this property, we simply mark all bound occurrences of variables in A(x) using *markbound*;

t is then substituted for all unmarked occurrences of x in A to produce a new formula A(t). It t were free for x in A(x), applying *markbound* to A(t) would cause no new marking.

§3. Postulates of the theory

Kleene defines the following postulates.

GROUP A. Postulates for the predicate calculus.

GROUP A1. Postulates for the propositional calculus.

1a. $A \supset (B \supset A)$.

1b. $(A \supset B) \supset ((A \supset (B \supset C)) \supset (A \supset C))$.

2. $\dfrac{A,\ A \supset B}{B.}$

3. $A \supset (B \supset A \,\&\, B)$.

4a. $A \,\&\, B \supset A$.

4b. $A \,\&\, B \supset B$.

5a. $A \supset A \vee B$.

5b. $B \supset A \vee B$.

6. $(A \supset C) \supset ((B \supset C) \supset (A \vee B \supset C))$.

7. $(A \supset B) \supset ((A \supset \neg B) \supset \neg A)$.

8. $\neg\neg A \supset A$.

GROUP A2. (Additional) Postulates for the predicate calculus.

9. $\dfrac{C \supset A(x)}{C \supset \forall x A(x).}$

10. $\forall x A(x) \supset A(t)$.

11. $A(t) \supset \exists x A(x)$.

12. $\dfrac{A(x) \supset C}{\exists x A(x) \supset C}.$

GROUP B. (Additional) Postulates for number theory.

13. $A(0) \,\&\, \forall x(A(x) \supset A(x')) \supset A(x)$.

14. $a' = b' \supset a = b$.

15. $\neg a' = 0$.

16. $a = b \supset (a = c \supset b = c)$.

17. $a = b \supset a' = b'$.

18. $a + 0 = a$.

19. $a + b' = (a + b)'$.

20. $a.0 = 0$.

21. $a.b' = a.b + a$.

For Postulates 1-8, A, B and C are formulae. For Postulates 9-13, x is a variable, A(x) is a formula, C is a formula which does not contain x free and t is a term which is free for x in A(x). For Postulates 14-21, A is a formula, a, b, c and x are variables. Postulates 2, 9 and 12 are known as rules and all the other postulates are known as axioms.

Patterns corresponding to axioms 1a, 1b, 3-8 and 14-21 are simple to write. For example, the pattern corresponding to axiom 1a is defined in the following way:

<u>pattern</u> axiom 1a = imply + *formula @ a + imply + *formula + *a

In this example, the operator * is applied to the string variable, a, and this causes a pattern to be produced in which the reference to a is kept. When pattern matching takes place, the operator @ ensures that the variable a is associated with the first occurrence of a formula in the input string and thus *a ensures a match with an exact repetition of the same formula.

Patterns corresponding to the remaining axioms (10, 11 and 13) are not so straightforward and stress the importance of the deferred evaluation of procedures. The difficulty with these three axioms arises from substitution. We will illustrate how these difficulties are solved by constructing a pattern corresponding to axiom 10, i.e. $\supset[\forall x]A(x)A(t)$, where t is a term free for x in A(x) and A(t) denotes the formula achieved from A(x) by replacing every free occurrence of x in A by t. The pattern required uses the operator * applied to a procedure which defers deproceduring until the routine is encountered during pattern matching. Considerable care must be taken, when using such deferred procedures, to ensure that any variables used in the body of the procedure are in scope when the procedure is executed. This is why procedures delivering patterns used in Housden and Kotarski (1977) are all of mode <u>proc</u> <u>pattern</u>. Problems of handling procedures with parameters are similar to those discussed by Rayward-Smith (1977).

The pattern for axiom 10 is:

```
pattern axiom 10 = imply + os + forall + break (cs) @ x +
      cs + *formula @ a + *formula @ b +
      *(pattern:(pattern p; string adol:= markbound (a);
                            while ob + x + cb
                            isin adol replaceby "*" do skip od;
                            while dollar isin adol replaceby "" do skip od;
                            int j,i:= index ("*", adol);
                            if i=0 then p:= patt a
                            else p:= adol [1:i-1] + term @ t;
                                 while
                                 (j:= index ("*", adol [i+1:upb adol]))≠0
                                 do p:= p + adol [i+1:j-1] + * t;
                                    i:= j
                                 od;
                            if i<upb adol then p:=p+adol[i+1:upb adol]fi
                            fi;
      if (pos(1) + p + rpos(0)) isin b then
                            if upb markbound(a)-upb a=upb markbound(b)-upb b
                            then null else fail
                            fi
      else fail
      fi))
```

Having identified the formula A(x) (called a), the variable x and A(t) (called b), the procedure constructs a pattern which will match any string which is equal to a except that every free occurrence of x is replaced consistently by a term t. It then checks that b is an example of this pattern and that t is a term free for x in A(x). Note that it is possible to incorporate into the pattern error messages informing the student user of LTP why some particular string under consideration might not be an example of the pattern.

Rules 2, 9 and 12 require more than one input string. In a proof, these rules are used to deduce a string from previous strings. For example, the proof of a = a given in Kleene (1952) commences

1. $a = b \supset (a = c \supset b = c)$ - Axiom 16.
2. $0 = 0 \supset (0 = 0 \supset 0 = 0)$ - Axiom 1a.
3. $(a = b \supset (a = c \supset b = c)) \supset ((0 = 0 \supset (0 = 0 \supset 0 = 0)) \supset (0 = 0 \supset (0 = 0 \supset 0 = 0)))$ - Axiom 1a.
4. $(0 = 0 \supset (0 = 0 \supset 0 = 0)) \supset (a = b \supset (a = c \supset b = c))$ - Rule 2, 1, 3.

The first 3 lines of this proof can be easily checked in isolation but the 4th. line needs reference to lines 1 and 3. Hence, when proof checking, every line is stored. If it is an axiom, it is checked using the pattern definitions described above but if it is a rule, the corresponding pattern definition refers to previous lines of the proof. LTP checks any student proof and outputs suitable error messages.

§4. Conclusion

The major criticism of LTP is the relatively large amount of store required (~ 40 K) and although work is continuing in an effort to reduce this, significant reductions are not anticipated. On the credit side, however, LTP is a useful package in two respects. Firstly, the unsophisticated user can have his understanding of predicate calculus thoroughly checked and can receive useful output giving guidance as to the cause of his errors. Secondly, the student familiar with the pattern matching album can find a new insight into the meaning of concepts such as "t is a term free for x in A(x)". By approaching a formal logic system through patterns, many previously difficult concepts are considerably simplified.

Acknowledgement

The author would like to thank Mrs. B. Roper and Miss P. Newby for their programming assistance during the development of LTP. Professor Housden gave much useful advice on the use of his patterns album together with continuous encouragement.

References

FLECK, A.C. (1971). Towards a theory of data structures, *Journal of Computing and System Sciences*, Vol. 5, No. 5.

FLECK, A.C. (1975). Recent developments in the theory of data structures, in *Proceedings of the 4th Texas Conference on Computing Systems*, Austin, Texas.

FLECK, A.C. (1978). Formal models for string patterns, in *Current Trends in Programming Methodology*, Vol. 4; *Data Structuring*, edited by R. Yeh, Prentice-Hall.

GRISWOLD, R.E., POAGE, J.F. and POLONSKY, I.P. (1971). *The SNOBOL 4 Programming Language*, Prentice-Hall.

HOUSDEN, R.J.W. and KOTARSKI, N. (1977). Character String Pattern Matching in Algol 68, in *Proceedings of the Strathclyde Algol 68 Conference, SIGPLAN*, Vol. 12, No. 6.

HOUSDEN, R.J.W. and RAYWARD-SMITH, V.J. (1975). An Information Structures Course based on Algol 68-R presented at the *Conference on Experience with Algol 68, Liverpool University*. (Copies available from the authors).

KLEENE, S.C. (1952). *Introduction to Metamathematics*, North-Holland.

LIU, K.-C. (1977). *An Efficient Algorithm for String Pattern Matching*, Ph.D. Thesis, Iowa..

LUKASIEWICZ, J. (1951). *Aristotle's Syllogism from the Standpoint of Modern Formal Logic*, Oxford University Press.

McGETTRICK, A.D. (1976). Teaching Mathematics by Computer, *The Computer Journal*, Vol. 20, No. 3.

RAYWARD-SMITH, V.J. (1977). Using Procedures in List Processing, in *Proceedings of the Strathclyde Algol 68 Conference, SIGPLAN*, Vol. 12, No. 6.

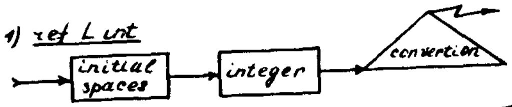
A Schema for reading data in Formatless Input

by Vera Ajueva, A.N. Maslov and V.B. Yakovlev
(Faculty of Computer Science, Moscow State University)

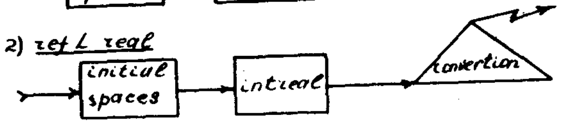
Definitions

1. Ω = {worthy characters, as Standard Hardware Representation},
 $\mathcal{T}$: {characters of 'termstring'}
 flip = {F},
 flop = {T},
 digit = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.
2. $\rightarrow \textcircled{S} \rightarrow$ {a character of the set S is got.}
3. $\rightarrow \textcircled{\tfrac{S}{c}} \rightarrow$ {a character is read; if $c \in S$ then c is got else the event routine of 'on char error' is called, the suggestion being c}
4. $\rightarrow\!\!+\!\!-$ {the line ended; the event routine of 'line mended' is called (and, may be, 'page manded' or/and 'logical file mended' is called')}.
5. $\rightarrow\!\!\not\!\rightarrow$ {the line ended; if the call of the event routine of 'line mended' yields true then $\rightarrow$ else $\rightarrow$}

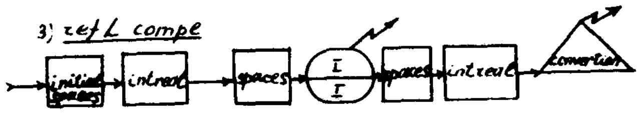
1) ref L int



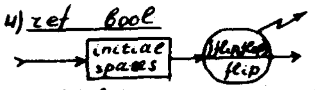
2) ref L real



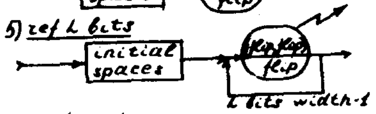
3) ref L compl



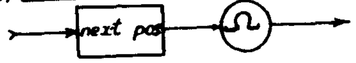
4) ref bool



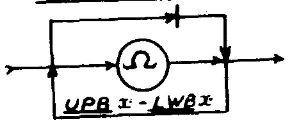
5) ref L bits



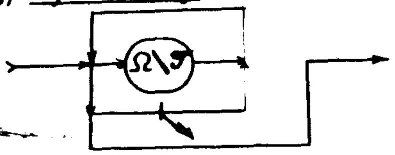
6) ref char



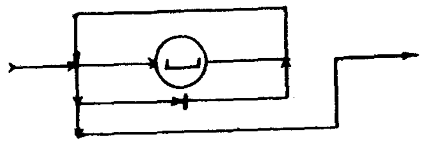
7) ref [ ] char



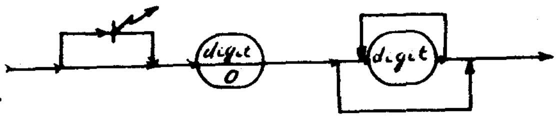
8) ref string



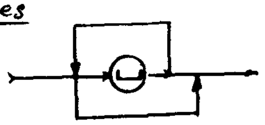
9. initial spaces



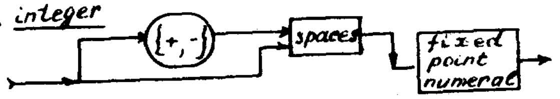
10. fixed point numeral



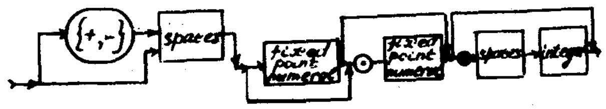
11. spaces



12. integer



13. intreal


{+,-}
spaces
fixed point numeral
fixed point numeral
spaces
integer