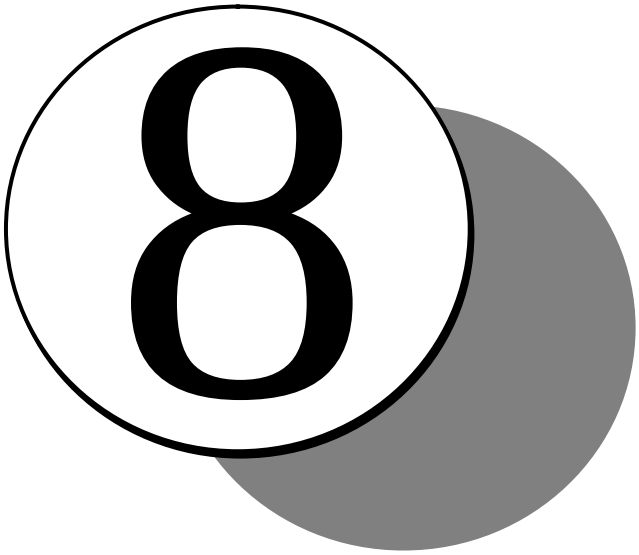
THE ABC NEWSLETTER

ISSN 0922-8055

Issue 8, August 1995



CONTENTS

The ABC Newsletter exists to provide information about ABC and to provide a forum for discussions.

You are encouraged to submit any articles you see fit. Articles don't have to contain fully thought-out ideas, but may be yet undeveloped thoughts intended to stimulate discussion. The kinds of articles we have in mind are: interesting programs, either written or suggestions; unusual applications; letters, discussions on points of the language, proposed improvements, experience with the language, and so on.

If you are fortunate enough to be able to send email, you can submit articles and send mail to:

`abc@cwi.nl`

Otherwise to:

The ABC Newsletter
CWI/AA
Postbox 94079
1090 GB Amsterdam
The Netherlands

This newsletter is available by ftp from:

ftp.cwi.nl

in directory

/pub/abc/newsletter/

THE ABC NEWSLETTER

NEWS

Yes! at long last a new Newsletter! ABC is no longer a funded research group, and therefore is a spare-time activity for us; this means that there is much less activity, both writing and programming.

Here is a summary of the current state of things.

NEWSLETTER

From now on the newsletter will principally be available electronically as a PostScript file from the ABC ftp archive (see below). For people without the resources to get hold of a copy electronically, we will send them on request. The directory is /pub/abc/newsletter.

THE BOOK

Four misprints have been found, alas:

Page 6, 6th line from the bottom

```
keys count item 2
```
should read
```
(keys count) item 2
```

Page 37, 7th line from bottom

```
REMOVE p FROM s
```
should read
```
REMOVE p FROM set
```

Page 55, 9th line from bottom

```
WHILE group IN keys followers:
```
should read
```
WHILE group in keys followers:
```

Page 65 line 3, a ? is missing at the end of the line.

A revised version of the book will shortly be available *free* from the ftp archive (see below). Watch this space.

CHANGES

There is one function that has been added to the implementation since the book went to print (though it's documented in the help file):

```
>>> WRITE now
(1995, 8, 4, 18, 3, 44.125996)
```

The function returns a compound consisting of (year, month, day, hour, minute, seconds) with fractional seconds to the accuracy supplied by the operating system. The month is in the range {1..12}, the day {1..31}, the hour {0..23}, the minutes {0..59}, and the seconds $0 \leq secs < 60$.

For instance, here is a function to return what day of the week it is today:

```
HOW TO RETURN today:
  PUT now IN (year, month, day,
              hour, minute, second)
  RETURN day.name[which]^"day"
which:
  RETURN (day.in.year -
                dominic) mod 7
dominic:
  PUT floor(year/100) IN century
  RETURN 7-((year+(floor(year/4))-
       century+(floor(century/4))-
                     leap) mod 7)
leap:
  IF (year mod 4 = 0 AND
      year mod 100 <> 0) OR
      year mod 400 = 0:
    RETURN 1
  RETURN 0
day.in.year:
  PUT {[2]: 28+leap} IN length
  FOR i IN {1; 3; 5; 7; 8; 10; 12}:
    PUT 31 IN length[i]
  FOR i IN {4; 6; 9; 11}:
    PUT 30 IN length[i]
  PUT day IN total
  FOR m IN {1..month-1}:
    PUT total+length[m] IN total
  RETURN total
day.name:
  RETURN split "Sun Mon Tues Wednes
                  Thurs Fri Satur"

>>> WRITE today
Friday
```

IMPLEMENTATIONS

The Unix, Macintosh, MS-DOS and Atari ST implementations are available by ftp and WWW and on floppy disk and tape from us. The implementations were also posted to comp.binaries.{atari.st; ibm-pc; mac} and comp.sources.unix (make sure you get the three patches too, for that one).

The ftp address is ftp.cwi.nl in directory /pub/abc., the WWW address is http://www.cwi.nl/~steven/abc.html

At the time of writing the files available are:

`index` for a list of all files available
`abc.intro` for an overview of ABC (also included with the implementations below)
`abcst.arc` for the Atari ST version
`abcpc.arc` for the IBM PC version
`abc.mac.sit.hqx` for the Mac version
`abc.unix.tar.Z` for the Unix version
`README` for an explanation of how to unpack the above files
`abcversions` for a description of updates as they are made.

There will be a new version of the sources made available that will be considerably easier to port. Contact `abc@cwi.nl`.

There is a new test version of ABC available. Currently only the (Unix) source is available in the archive, since it hasn't been tried on a wide range of machines yet. What would be nice is if all you Abecedarians with Unix try it out, report the problems, which we fix, and then we can distribute the versions for everybody else; the test version is called test.tar.Z in /pub/abc on ftp.cwi.nl.

Are there people with the time and responsibility to do the MS-DOS and Mac compilations when the time comes?

The new version has the following additions:
— support of ABC source 'archives'
— support of a 'central' workspace.

Both of these are explained below.

ARCHIVES

Many people have asked for a repository of useful ABC programs. The problem was that ABC didn't easily support importing programs, and certainly not in a machine-independent way. The new version does. It is done with two new flags -p (pack) for producing an archive, and -u (unpack) for importing an archive. For instance:

```
abc -w program -p > program.abc
```

produces an archive of the workspace 'program';

```
abc -w new -u < program.abc
```

will unpack it again in the workspace 'new'.

CENTRAL WORKSPACE

Very often when writing ABC programs, you use how-tos that you have already written in other workspaces and have to copy them physically across.

The central workspace is a place where you can store all the how-tos that are common across workspaces: how-tos in the workspace called 'abc' are now accessible from all other workspaces, suggestions and all.

SOURCE ARCHIVE

To go with the new archiving version of ABC is a repository of example programs. It is in the usual place (ftp.cwi.nl), and currently the programs available are:

advent: an (incomplete) adventure game
grammar: a collection of grammar analysis tools. This workspace also contains how-tos for set, sequence, tree, and graph handling.
eliza: a psychoanalysis program, and a paranoid patient

We will be adding more, and candidate programs can be sent to us for inclusion.

The directory is programming/languages/abc/examples, where there is a general README.

We also plan to add a FAQ (frequently asked questions) directory for answers that we often have to give.

FUTURE

Other future plans, given the time, include a windowing version of ABC, graphics and system extensions, and the older newsletters available by ftp. Now, where did we put that time? :-)

PUBLICATIONS

J. Zwaan, and R. Zwart, *Graphics for ABC*, CWI Report CS-R9255, CWI, Amsterdam, 1992.

This report gives the first steps towards a graphical facility for the programming language ABC. It discusses which features are to be included as primitives in a graphical extension to the language, the way pictures could be represented and gives directions towards an implementation.

L.G.L.T. Meertens, S. Pemberton, and G. van Rossum, *The ABC structure editor: Structure-based editing for the ABC programming environment.* CWI Report CS-R9256, CWI, Amsterdam, 1992

ABC is an interactive programming language where both ease of learning and ease of use stood high amongst its principle design aims. The language is embedded in a dedicated environment that includes a structure-based editor. In with the design aims, the editor had to be easy to learn, demanding a small command set, and easy to use, demanding a powerful command set and strong support for the user in composing programs, without enforcing a computer-science understanding of issues of syntax and the like. Some novel design rules have led to an interesting editor, where the user may enter and edit text either structurally or non-structurally, without having to use different 'modes'.

Robin Jones, Clive Maynard, Ian Stewart, *The Art of Lisp Programming*, Springer Verlag, Berlin etc, 1990, ISBN 0-387-19568-8 and ISBN 3-540-19568-8.

A textbook for anyone who has a passing acquaintance with procedural languages, such as BASIC or Pascal, but who has not met a functional language like Lisp before. In addition to providing a step-by-step introduction to Lisp, this book is unique in illustrating the use of Lisp through the development of a realistic project: the design and implementation of a Lisp-based interpreter for the language ABC.

Peter Landrock, Knud Nissen, *Kryptologi*, Abacus Publishers, Vejle, Denmark, 1990, ISBN 87-89182-24-3

An introduction to cryptography, with example programs in ABC. In Danish.

The CWI reports mentioned here are available as PostScript files by ftp from site ftp.cwi.nl in directory /pub/CWIreports/AA; the file names are the report numbers mentioned above.

READERS WRITE

We get sent a lot of email about ABC, much of which is questions about using ABC or why aspects of ABC are like they are. Here is one.

The Type Checker

Dear ABC people:

There's a problem in the type checker: when you interactively enter a table with different element types it works, but when you leave ABC and restart, ABC won't accept the types any more.

Here's an example:

```
$ abc
ABC Release 1.01.07.
>first
>>> PUT {} IN zork
>>> PUT 10 IN zork[″integer″]
>>> PUT ″ten″ IN zork[″string″]
>>> WRITE zork
{[″integer″]:10; [″string″]:″ten″}
>>> WRITE zork[″string″]
ten
>>> QUIT
```

```
$ abc
ABC Release 1.01.07.
>first
>>> WRITE zork
*** Can't reconcile the types in your location zork:
{[″integer″]:10; [″string″]:″ten″}
*** The problem is: I found type EG 0 where I
    expected ″″
>>> ?
```

What's the problem?

Nico Verwer, Utrecht University, The Netherlands.

We reply:

What you are trying to do isn't allowed in ABC: the error message is correct. The problem is that ABC failed to spot it in the first instance.

The problem is in the typechecker: it checks each immediate command separately, without looking at the contents of locations. Thus:

```
>>> PUT {} IN zork
```

No problem.

```
>>> PUT 10 IN zork[″integer″]
```

Still no problem.

```
>>> PUT ″ten″ IN zork[″string″]
```

Officially, it should complain at this point, but because it doesn't look at the contents of locations, it doesn't spot this error.

```
>>> WRITE zork
{[″integer″]: 10; [″string″]:″ten″}
>>> WRITE zork[″string″]
ten
```

The ABC system is built so that *internally* tables with mixed element types work (the system uses them itself for symbol tables, for instance), but because ABC is strongly typed, it is not allowed.

In the second session:

```
>>> WRITE zork
*** Can't reconcile the types in your location zork
{[″integer″]:10;[″string″]: ″ten″}
*** The problem is: I found type EG 0 where I expected ””
>>> ?
```

In this case it processes the types in one go and does see the problem. Similarly, if you type

```
>>> PUT {[″integer″]: 10;
         [″string″]: ″ten″} IN zork
```

as an immediate command, it would complain as well.

So the problem in this case is that ABC (the language) doesn't allow something you thought it might.

Solutions?

1) Store integers as texts, and convert:

```
>>> WRITE zork
{[″integer″]:″10″;[″string″]:″ten″
}
HOW TO RETURN value t:
  IF t=””: RETURN 0
  IF t|1=″-″: RETURN -(val (t@2))
  RETURN (val (t|#t-1))*10+dig
dig: RETURN #{″1″..t item #t}

>>> WRITE value zork[″integer″]
10
```

2) Store integers or texts as indexes into another table:

```
>>> WRITE zork
{[″integer″]: 10; [″string″]: 1}
>>> WRITE strings
{[1]: ″ten″}
```

or

```
>>> WRITE zork
{[″integer″]:″1″;[″string″]:″ten″}
>>> WRITE integers
{[″1″]: 10}
```

How could you imitate the ABC system's symbol table in ABC? You need one table for the symbols themselves, and 5 for the values of each ABC type (number, text, compound, list, table):

```
>>> WRITE value
{[″a″]: ″n1″; [″b″]: ″t1″; [″c″]:
″l1″; [″d″]: ″T1″}
```

The keys are the names of locations, and the elements are indexes into the tables for each type:

```
>>> WRITE number
{[″n1″]: 123; [″n2″]: 10; [″n3″]:
20; [″n4″]: 2.718; [″n5″]: 3.142}

>>> WRITE text
{[″t1″]: ″Hello there″; [″t2″]:
″e″; [″t3″]: ″pi″}

>>> WRITE list
{[″l1″]: {[1]: ″n2″; [2]: ″n3″}}

  >>> WRITE table
{[″T1″]: {[1]: (″t2″, ″n4″); [2]:
(″t3″, ″n5″)}}
```

Here is a how-to to print out the contents of a location:

```
>>> FOR t IN keys value:
      WRITE t, ″: ″
      PRINT value[t]
      WRITE /
a: 123
b: ″Hello there″
c: {10; 20}
d: {[″e″]: 2.718; [″pi″]: 3.142}

HOW TO PRINT v:
  SHARE number, text, compound
  SHARE list, table
  SELECT:
    v in keys number:
      WRITE number[v]<<1
```

```
    v in keys text:
      WRITE ‘”’, text[v], ‘”’
    v in keys compound:
      PRINT COMPOUND
    v in keys list:
      PRINT LIST
    v in keys table:
      PRINT TABLE
    ELSE:
      WRITE ”?‘v‘?”
PRINT COMPOUND:
  PUT ”” IN sep
  WRITE ”(”
  FOR i IN compound[v]:
    WRITE sep
    PRINT i
    PUT ”, ” IN sep
  WRITE ”)”
PRINT LIST:
  PUT ”” IN sep
  WRITE ”{”
  FOR i IN list[v]:
    WRITE sep
    PRINT i
    PUT ”; ” IN sep
  WRITE ”}”
PRINT TABLE:
  PUT ”” IN sep
  WRITE ”{”
  FOR k, i IN table[v]:
    WRITE sep, ”[”
    PRINT k
    WRITE ”]: ”
    PRINT i
    PUT ”; ” IN sep
  WRITE ”}”
```

Here is a (simple) how-to to tell the type of a value:

```
>>> FOR t IN keys value:
      WRITE t, ”: ”
      TELL TYPE value[t]
      WRITE /
a: 0
b: ””
c: {0}
d: {[””]: 0}

HOW TO TELL TYPE v:
  SHARE number, text, compound
  SHARE list, table
  SELECT:
    v in keys number:
      WRITE ”0”
    v in keys text:
      WRITE ’””’
    v in keys compound:
      TELL COMPOUND
    v in keys list:
      TELL LIST
    v in keys table:
      TELL TABLE
    ELSE:
      WRITE ”?‘v‘?”
TELL COMPOUND:
  PUT ”” IN sep
  WRITE ”(”
  FOR u IN compound[v]:
    WRITE sep
    TELL TYPE u
    PUT ”, ” IN sep
  WRITE ”)”
TELL LIST:
  WRITE ”{”
  PUT list[v] IN c
  SELECT:
    #c > 1:TELL TYPE c item 1
    ELSE: WRITE ”?”
  WRITE ”}”
TELL TABLE:
  WRITE ”{”
  PUT table[v] IN c
  SELECT:
    #c > 1:
      PUT c item 1 IN k, i
      WRITE ”[”
      TELL TYPE k
      WRITE ”]: ”
      TELL TYPE i
    ELSE: WRITE ”[?]: ?”
  WRITE ”}”
```

Finally (to drive home the message in the ABC Programmer’s Handbook ...), here is a function to replace `PRINT` above:

```
HOW TO RETURN repr v:
  SHARE number, text, compound
  SHARE list, table
  SELECT:
    v in keys number:
      RETURN number[v]<<1
    v in keys text:
      RETURN ’”’^text[v]^’”’
```

```
    v in keys compound:
      RETURN repr.compound
    v in keys list:
      RETURN repr.list
    v in keys table:
      RETURN repr.table
    ELSE:
      RETURN "?'v'?"
repr.compound:
  RETURN "(" ^ listed.compound ^ ")"
listed.compound:
  RETURN ", " listed compound[v]
repr.list:
  RETURN "{" ^ listed.list ^ "}"
listed.list:
  RETURN "; " listed list[v]
repr.table:
  RETURN "{" ^ listed.table ^ "}"
listed.table:
  PUT "", "" IN sep, res
  FOR k, i IN table[v]:
    PUT res^sep^"['repr k']: "^
                      repr i IN res
    PUT "; " IN sep
  RETURN res

HOW TO RETURN sep listed t:
  PUT "", "" IN sep, res
  FOR i IN t:
    PUT res^s^repr i IN res
    PUT sep IN s
  RETURN res
```

(I've mixed the use of ^, ‘‘, and << for pedagogic reasons :-)

```
>>> FOR t IN keys value:
    WRITE t,": ",repr value[t] /
a: 123
b: "Hello there"
c: {10; 20}
d: {["e"]: 2.718; ["pi"]: 3.142}
```

By the way, note that although really disjoint texts have been used for the keys into the different tables (like "n1" for numbers, "t1" for texts) that's not absolutely necessary; simple integers would also work, just as long as the numbers are disjoint. All the how-to's above work without change on this set of data-structures. The advantage of the first method is when debugging: you can see immediately what the type of the value ought to have been.

GRAMMAR ANALYSIS WITH ABC

Steven Pemberton
CWI, Amsterdam

When working with grammars, I always use ABC to do it. Among the advantages are that you can do the work interactively, that you can very quickly build additional tools, and that you have the already powerful programming environment at your disposal.

What follows is a brief description of some of the tools I use, with an example. Some of what follows is also presented in the ABC Handbook, though at a somewhat more relaxed pace.

This code is available from the ABC ftp archive. For didactic reasons, what is presented here differs in detail from the code there.

GRAMMARS

The representation that I use is more or less a direct transcription of what a grammar is. I use a table whose keys are texts (i.e. strings) representing the nonterminals of the language, and whose items are sets of alternatives. Each alternative is a sequence of texts, representing terminals and nonterminals. So here is a how-to that displays a grammar in this form:

```
HOW TO DISPLAY grammar:
  FOR name IN keys grammar:
    WRITE "'name': " /
    FOR alt IN grammar[name]:
      WRITE "    "
      FOR symbol IN alt:
        WRITE symbol, " "
      WRITE /
```

and as example:

```
>>> DISPLAY sentence
ADJ:
  EMPTY
  clever
  shy
BOY:
  John
  Kevin
EMPTY:
```

```
GIRL:
  Mary
  Susan
OBJ:
  SUBJ
SENT:
  SUBJ loves OBJ
SUBJ:
  ADJ BOY
  ADJ GIRL
```

You can generate a random phrase from a grammar with the following:

```
HOW TO GENERATE sym FROM grammar:
  SELECT:
    sym in keys grammar: \Nonterminal
      FOR new IN choice grammar[sym]:
        GENERATE new FROM grammar
    ELSE: \Terminal symbol
      WRITE sym, " "

>>> GENERATE "SENT" FROM sentence
Susan loves clever John
```

SETS

Here are some necessary functions on sets. Set union:

```
HOW TO RETURN set1 with set2:
  FOR x IN set2:
    IF x not.in set1:
      INSERT x IN set1
  RETURN set1
```

Set difference:

```
HOW TO RETURN set1 less set2:
  FOR x IN set2:
    IF x in set1:
      REMOVE x FROM set1
  RETURN set1
```

Here is a function that collects all symbols used in the rules of a grammar:

```
HOW TO RETURN used grammar:
  PUT {} IN all
  FOR rule IN grammar:
    FOR alt IN rule:
      FOR sym IN alt:
        IF sym not.in all:
          INSERT sym IN all
  RETURN all
>>> WRITE used sentence
{"ADJ"; "BOY"; "EMPTY"; "GIRL";
"John"; "Kevin"; "Mary"; "OBJ";
"SUBJ"; "Susan"; "clever"; "loves";
"shy"}
```

The terminals of the grammar are all the symbols less the nonterminals:

```
>>> WRITE (used sentence)less keys
                                sentence
{"John"; "Kevin"; "Mary"; "Susan";
"clever"; "loves"; "shy"}
```

and the unused nonterminals (such as the root symbol) are the nonterminals less the used symbols:

```
>>> WRITE (keys sentence)less used
                              sentence
{"SENT"}
```

For neater output, "listed" converts a set to a text:

```
HOW TO RETURN listed set:
  PUT "" IN line
  FOR element IN set:
    PUT line^"'element' " IN line
  RETURN line

>>> WRITE listed ((used
      sentence)less keys sentence)
John Kevin Mary Susan clever loves
shy
```

A useful set is the set of nonterminals that can generate empty. This is generated by repeatedly doing a pass over the rules that we don't know yet can generate empty, until we find no more:

```
HOW TO RETURN empties grammar:
  PUT keys grammar IN to.do
  PUT {} IN empties
  WHILE SOME name IN to.do HAS
                              empty:
    INSERT name IN empties
    REMOVE name FROM to.do
  RETURN empties
empty:
  REPORT SOME alt IN grammar[name]
                       HAS empty.alt
empty.alt:
  REPORT EACH sym IN alt HAS sym in empties

>>> WRITE listed empties sentence
ADJ EMPTY
```

RELATIONS

Relations between symbols of the grammar are the essential element of the grammar tools. A relation is represented as a table whose keys are symbols, and whose items are sets of symbols.

For instance, if symbol *b* follows symbol *a* in some rule, `"b"` will be in the set for `follows["a"]`, so you can say, for instance:

```
IF "b" in follows["a"]: ....
```

Relations are sparse (i.e. a symbol is not in the keys of the relation if the set of elements is empty), so we use the following to access a relation:

```
HOW TO RETURN relation for k:
  \relation[k] for sparse relations
  IF k in keys relation:
    RETURN relation[k]
  RETURN {}
```

To add an element to a relation, we use this:

```
HOW TO ADD x TO relation FOR thing:
  IF thing not.in keys relation:
    \First time
    PUT {} IN relation[thing]
  IF x not.in relation[thing]:
    INSERT x IN relation[thing]
```

though you may prefer

```
HOW TO ADD x TO relation FOR thing:
  PUT (relation for thing) with {x}
                 IN relation[thing]
```

For instance:

```
>>> ADD "b" TO follows FOR "a"
```

We'll display a relation with:

```
HOW TO SHOW relation:
  FOR k IN keys relation:
    WRITE "‘k‘: ", listed
                      relation[k]/
```

Here are some general functions on relations. The inverse:

```
HOW TO RETURN inverse relation:
  PUT {} IN inv
  FOR k IN keys relation:
    FOR x IN relation[k]:
      ADD k TO inv FOR x
  RETURN inv
```

The product of two relations (a P c iff a R1 b and b R2 c):

```
HOW TO RETURN r1 prod r2:
  PUT {} IN prod
  FOR c IN keys r2:
    FOR b IN r2[c]:
      IF b in keys r1:
        FOR a IN r1[b]:
          ADD a TO prod FOR c
  RETURN prod
```

The closure:

```
HOW TO RETURN closure r:
  FOR i IN keys r:
    FOR j IN keys r:
      IF i in r[j]:
        PUT r[i] with r[j] IN r[j]
  RETURN r
```

To make a relation reflexive, we use the following. Since relations are sparse, we also have to pass the set of symbols that it must be reflexive over:

```
HOW TO RETURN symbols reflexive r:
  FOR sym IN symbols:
    ADD sym TO r FOR sym
  RETURN r
```

SOME EXAMPLES OF RELATIONS

To collect the *direct* followers for each symbol, we walk along each alternative, collecting adjacent symbols. There is one catch: in a rule like:

```
SENT: the ADJ PERSON
```

"the" and "ADJ" are adjacent, but if "ADJ" can generate empty, then so are "the" and "PERSON":

```
HOW TO RETURN followers grammar:
  PUT {} IN foll
  PUT empties grammar IN empty
  FOR rule IN grammar:
    FOR alt IN rule:
      TREAT ALT
  RETURN foll
TREAT ALT:
  FOR i IN {1..#alt-1}:
    PUT alt item i IN this
    TREAT PART
TREAT PART:
  FOR j IN {i+1..#alt}:
    PUT alt item j IN next
    ADD next TO foll FOR this
    IF next not.in empty: QUIT
```

```
>>> SHOW followers sentence
```

```
ADJ: BOY GIRL
SUBJ: loves
loves: OBJ
```

To collect the direct starter symbols of each rule, you also have to deal with symbols that produce empty:

```
HOW TO RETURN heads grammar:
  PUT {} IN heads
  PUT empties grammar IN empty
  FOR name IN keys grammar:
    FOR alt IN grammar[name]:
      TREAT ALT
    RETURN heads
TREAT ALT:
  FOR i IN {1..#alt}:
    PUT alt item i IN head
    ADD head TO heads FOR name
    IF head not.in empty: QUIT
```

```
>>> SHOW heads sentence
ADJ: EMPTY clever shy
BOY: John Kevin
GIRL: Mary Susan
OBJ: SUBJ
SENT: SUBJ
SUBJ: ADJ BOY GIRL
```

Similarly for the direct enders:

```
HOW TO RETURN tails grammar:
  PUT {} IN tails
  PUT empties grammar IN empty
  FOR name IN keys grammar:
    FOR alt IN grammar[name]:
      TREAT ALT
  RETURN tails
TREAT ALT:
  FOR i' IN {-#alt..-1}:
    PUT -i' IN i
    PUT alt item i IN tail
    ADD tail TO tails FOR name
    IF tail not.in empty: QUIT
```

The closure of the head relation represents all symbols that can start a rule, either directly or indirectly:

```
>>> SHOW closure heads sentence
ADJ: EMPTY clever shy
BOY: John Kevin
GIRL: Mary Susan
OBJ: ADJ BOY EMPTY GIRL John Kevin
     Mary SUBJ Susan clever shy
SENT: ADJ BOY EMPTY GIRL John Kevin
      Mary SUBJ Susan clever shy
SUBJ: ADJ BOY EMPTY GIRL John
      Kevin Mary Susan clever shy
```

Symbol *b* may follow symbol *a* in a phrase if *b* follows *a* in an alternative, or if *B* follows *A* in an alternative and *b* is in *heads*(B)* and *a* is in *tails*(A)*. This is expressed as the product:

head . follow . inverse(tail*).*

Now we have enough to define a command for a grammar g, that prints for each symbol in each alternative what may follow that symbol at that point:

```
HOW TO SHOW LOCAL FOLLOWERS g:
  PUT (used g) with keys g IN symbols
  PUT symbols reflexive
     (closure heads g) IN head.star
  PUT symbols reflexive
     (closure tails g) IN tail.star
  PUT followers g IN follow
  PUT (head.star prod follow) prod
              (inverse tail.star) IN
                        deep.follow
  FOR parent IN keys g:
    FOR alt IN g[parent]:
      TREAT ALT
ANNOUNCE ALT:
  WRITE "'parent': ", listed alt /
TREAT ALT:
  ANNOUNCE ALT
  FOR i IN {1..#alt}:
    TREAT SYM
TREAT SYM:
  PUT alt item i IN sym
  WRITE " 'sym': "
  WRITE listed local.follow /
local.follow:
  PUT {} IN foll
  FOR j IN {i+1..#alt}:
    PUT alt item j IN next
    PUT foll with (head.star for
                       next) IN foll
    IF next not.in empty:
      RETURN foll
  RETURN foll with (deep.follow for
                              parent)
```

This prints each alternative separately, followed by each symbol of the alternative indented one to a line followed by the symbols that can follow it at that point.

For example:

```
>>> SHOW LOCAL FOLLOWERS sentence
ADJ: EMPTY
  EMPTY: BOY GIRL John Kevin Mary
         Susan
ADJ: clever
  clever: BOY GIRL John Kevin Mary
          Susan
ADJ: shy
  shy: BOY GIRL John Kevin Mary
       Susan
BOY: John
  John: loves
BOY: Kevin
  Kevin: loves
EMPTY:
GIRL: Mary
  Mary: loves
GIRL: Susan
  Susan: loves
OBJ: SUBJ
  SUBJ:
SENT: SUBJ loves OBJ
  SUBJ: loves
  loves: ADJ BOY EMPTY GIRL John
         Kevin Mary OBJ SUBJ Susan
         clever shy
  OBJ:
SUBJ: ADJ BOY
  ADJ: BOY John Kevin
  BOY: loves
SUBJ: ADJ GIRL
  ADJ: GIRL Mary Susan
  GIRL: loves
```

CONCLUSIONS

What has been presented here is a set of grammar tools that let you play with grammars and analyse them very easily, and interactively. Some major tools that are missing are a LL-1 checker, that lets you check that a grammar is parsable with an LL-1 parser, and a parser. These will be the subjects of later articles.

EIGHT QUEENS, MORE OR LESS

Steven Pemberton
CWI, Amsterdam

I was asked what an Eight Queens program would look like in ABC. The problem is, can you put eight queens on an 8 by 8 chess board board in such a way that they cannot take each other. A queen can take by moving vertically, horizontally or diagonally.

My first version was a fairly traditional version — for instance see E.W. Dijkstra's version in the book *Structured Programming* — although of course, I immediately generalised it to any number of queens[1]:

```
HOW TO QUEENS n:
  DISPLAY n filled {}
```

The command `DISPLAY` just displays the board, which is represented as a table whose keys are the row numbers (1 to n), and whose items are the column number of the piece for that row:

```
HOW TO DISPLAY board:
  IF board = {}:
    WRITE "No solution" /
  FOR p IN board:
    WRITE "# "^^(p-1), "O "
    WRITE "# "^^(#board - p) /
```

The work of finding a solution for the problem is done by the function `filled`. If the board is already of size *n* then an answer has already been found. Otherwise, a piece is added for the next row: each possible piece is tried, and if one is found that is 'safe' — it can't be taken by any of the other queens on the board — it is placed on the board (by `board'`), and `filled` is called to fill the next row. If the result of that call is successful, then that result is returned, otherwise the next piece for the current row is tried. If no piece is found, then the empty board is returned, to indicate failure.

The check for safeness uses properties of row and column numbers to see if a position is being attacked.

1. (Actually the question then arises: *Why eight? Why not twelve?* I think that the answer is that eight is the smallest value that gives an interesting result).

The function uses backtracking and the fact that ABC functions can't produce side-effects. See my article *Backtracking in B* in the *ABC Newsletter 5* for more details about backtracking in ABC.

```
HOW TO RETURN n filled board:
  IF #board = n: RETURN board
  FOR p IN {1..n}:
    IF safe:
      PUT n filled board' IN new
      IF new <> {}: RETURN new
  RETURN {}
safe: REPORT col AND left AND right
col: REPORT p not.in board
left: REPORT NO r IN keys board HAS
            r+board[r] = #board+1+p
right: REPORT NO r IN keys board
        HAS r-board[r] = #board+1-p
board':
  PUT p IN board[#board+1]
  RETURN board
```

```
>>> QUEENS 8
O # # # # # # #
# # # # O # # #
# # # # # # # O
# # # # # O # #
# # O # # # # #
# # # # # # O #
# O # # # # # #
# # # O # # # #
```

A More Direct Version

However, I also decided to try a more direct solution, by simulating what you would do by hand more closely. Here the board is again a table with the row number as key, but with a list of numbers as items. These numbers represent which positions on the current row are not being attacked. So you start out with a full board, which must then be emptied:

```
HOW TO QUEENS' n:
  PUT {} IN board
  FOR i IN {1..n}:
    PUT {1..n} IN board[i]
    DISPLAY' 1 emptied board

HOW TO DISPLAY' board:
  IF board = {}:
    WRITE "No solution" /
  FOR row IN board:
    CHECK #row = 1
    PUT min row IN p
    WRITE "# "^^(p-1), "O "
    WRITE "# "^^(#board - p) /
```

The work is done by the function `emptied`. You no longer have to check if a piece is safe or not: for each row you have just the list of safe pieces available. When you choose a piece, then all the positions that are attacked by this piece are then removed from the board, and the next row is tried.

```
HOW TO RETURN n emptied board:
  IF n > #board: RETURN board
  FOR p IN board[n]:
    PUT (n+1) emptied board' IN new
    IF new <> {}: RETURN new
  RETURN {}
board':
  PUT {p} IN board[n]
  FOR row IN {n+1..#board}:
    PUT board[row] less
          {p; p+(row-n); p-(row-n)}
                    IN board[row]
  RETURN board
```

This uses a small function to return the difference between two lists:

```
HOW TO RETURN l1 less l2:
  FOR i IN l2:
    IF i in l1:
      REMOVE i FROM l1
  RETURN l1
```

```
>>> QUEENS' 8
O # # # # # # #
# # # # O # # #
# # # # # # # O
# # # # # O # #
# # O # # # # #
# # # # # # O #
# O # # # # # #
# # # O # # # #
```

```
>>> QUEENS' 11
0 # # # # # # # # # #
# # 0 # # # # # # # #
# # # # 0 # # # # # #
# # # # # # 0 # # # #
# # # # # # # # 0 # #
# # # # # # # # # # 0
# 0 # # # # # # # # #
# # # 0 # # # # # # #
# # # # # 0 # # # # #
# # # # # # # 0 # # #
# # # # # # # # # 0 #
```

As it turns out, this version runs faster than the first version. The reason for this is that you have to search much less to find a candidate piece for a row, and when you reach a row where all positions are attacked, you know it immediately, without having to check each position separately.

A GENERAL-PURPOSE DATABASE

Steven Pemberton
CWI, Amsterdam

Someone came to the database experts in our department and said that the mailing lists she had to administer were getting unmanageable. They had originally been prepared by different people, so they were all in a different format, using different conventions, and in different files. She needed a program to help her manage them, and she sketched the facilities she needed:

> The database would be quite small, a couple of thousand records or so. Each record would have a number of standard fields: name, institute, department, address, city, country, email address, and so on. There should also be a 'code' field where she could say which mailing lists this address belonged to, such as ABC, the Operating System list she managed, and so on.
>
> She should then be able to look up entries, and above all make selections which could then be printed off as labels. She should be able to find out how large a selection was. She should also be able to print all records out for a card index on her desk.
>
> To aid searching, certain fields such as country, should be constrained so that only unique values are used; not United Kingdom in one case, and Great Britain in another.

At this point, I got called in on the discussions, and after rejecting some possibilities (the standard database package is only available on one computer that is not accessible for everyone who needs to access the mailing lists), without further ado, three of us sat down at a workstation, and in an afternoon wrote the program in ABC.

Data Representation

A first decision we had to take was how we were going to represent the database.

It was obvious from the specification that we didn't know exactly how many fields there were going to be, nor what they were, and furthermore that it was likely to change, so we needed to be as flexible as possible.

Therefore we decided to have a structure defining the allowable field names, and each record would then be a table from these field names to the field value. Each record would be regarded as having an entry for each field, though if it were empty, it wouldn't be physically there. This meant that if we added a new field-name to the defining structure, all records effectively got an empty field with that name.

It also meant that if a field-name was later deleted, all those fields in the database became no longer accessible, they apparently disappeared, though they were physically still there, so that if later the field name was reinstated, the values would reappear.

So to define the allowable field names:

```
PUT split "Name Institute Dept
  Address City Postcode Land Code"
   IN field.names
```

An entry in the database might then look like:

```
{["Name"]: "Jane Smith";
 ["Institute"]: "Univ. of Life";
 ["Land"]: "Erewhon"}
```

Missing fields are considered present but empty.

Now, given this format, we can immediately write a how-to to show a record:

```
HOW TO SHOW RECORD record:
  SHARE field.names
  FOR name IN field.names:
    IF name IN keys record:
      WRITE name, ": "
      WRITE record[name]/
```

and one to read a record:

```
HOW TO GET record:
  SHARE field.names
  PUT {} IN record
  FOR name IN field.names:
    WRITE name, ": "
    READ field RAW
    IF field <> "":
      PUT field IN record[name]
```

The whole database is just a set of records (with no implied ordering) So to display the whole database, record by record, we can use:

```
HOW TO SHOW db:
  FOR r IN db:
    SHOW RECORD r
    WRITE /
```

To add a record to the database:

```
HOW TO ADD TO db:
  GET record
  IF record <> {}
    INSERT record IN db
```

Selection

A basic action you want to do with a database is select records on the basis of certain criteria. To do this, we shall write some functions that given a database and a set of selection criteria, deliver a database that is a sub-set of the original one.

The represention we shall use for the criteria is just a record: for each record in the database if the record matches in the required way with the criteria-record, then it will form a part of the result:

```
HOW TO RETURN db equals criteria:
  PUT {} IN result
  FOR record IN db:
    IF matches:
      INSERT record IN result
  RETURN result

matches:
  REPORT EACH name IN keys criteria
                    HAS field.match
field.match:
  REPORT name in keys record AND
      record[name] = criteria[name]
```

This version gives an exact match, so if we say

```
SHOW db matches {["Land"]: "UK"}
```

then we'll get all records with the Land field equal to UK. If we want a partial match, we can use:

```
HOW TO RETURN db contains criteria:
  PUT {} IN result
  FOR record IN db:
    IF matches:
      INSERT record IN result
  RETURN result

matches:
  REPORT EACH name IN keys criteria
                    HAS field.match
field.match:
  REPORT name IN keys record AND
      lower record[name] includes
      lower criteria[r]
```

and `includes` reports whether the one text includes the other:

```
HOW TO REPORT t includes s:
  REPORT SOME i IN {1..#t-#s+1}
                   HAS t@i|#s = s
```

The how-to `contains` matches any record where the field contains the relevant criteria, ignoring case. So:

```
SHOW db contains {["Land"]: "UK"}
```

would match for instance UK and Ukraine.

Note that because these functions take a database as parameter, we can chain them:

```
SHOW (db equals {["Land"]: "UK"})
     contains {["City"]: "York"}
```

which selects all records that contain the city York, in the country UK. Also note that

```
SHOW db contains {[”Land”]: ””}
```

would show all entries with a Land field.

Formatting

Each country has a different way of formatting its addresses, so we need a flexible method of specifying address formats.

What we are going to use here is a table of land names to formats, where each format is a single text. For instance:

```
>>> WRITE format["NL"]
Name / Dept / Institute / Address /
Postcode _ _ City / _
>>> WRITE format["UK"]
Name / Dept / Institute / Address /
City _ Postcode / UK / _
```

The text is a number of words. A "/" represents a new line, a "_" represents a space. Other words are either field names, in which case the corresponding entry in the record is substituted, or literal words. Completely empty lines are not output, but lines containing spaces are (so the last part of the format above ensures a blank line between records). Here's how we output a set of records:

```
HOW TO FORMAT records WITH formats:
  SHARE field.names, format
  FOR r IN records:
    PUT "" IN out
    FORMAT RECORD
chosen.format:
  SELECT:
    "Land" in keys r AND
        r["Land"] in keys format:
      RETURN format[r["Land"]]
    ELSE:
      RETURN format["default"]
FORMAT RECORD:
  FOR word IN split chosen.format:
    SELECT:
      word in field.names:
        IF word in keys r:
          PUT out^r[word] IN out
      word = "/":
        IF out <> "": WRITE out/
        PUT "" IN out
      word = "_":
        PUT out^" " IN out
      ELSE:
        PUT out^word IN out
    IF out <> "": WRITE out/
```

This same code then lets us get an overview of a set of entries, just by using another set of formats. For instance:

```
>>> WRITE brief
{["default"]: "Name , _ City , _ Land"}
```

Changing

Just as we had a method of inputting entries, we also need to supply a way of changing them. Here we do more or less the same as with input, except we display the field name and entry before asking for input. If the user types a newline, the entry is unchanged; if the user types other data that then replaces the old. We then need a way of deleting an entry: we do this by saying that if the user types one or more spaces as input, it has the effect of deleting the entry:

```
HOW TO RETURN modified record:
  SHARE field.names
  PUT record IN new
  FOR field IN field.names:
    WRITE field, ": "
    IF field in keys record:
      WRITE record[field], " "
    READ answer RAW
    SELECT:
      answer = "": PASS
      stripped answer = "":
        IF field in keys new:
          DELETE new[field]
      ELSE:
        PUT answer IN new[field]
  RETURN new

HOW TO REPLACE old WITH new IN db:
  IF old <> new:
    IF old IN db: REMOVE old FROM db
    IF new <> {}: INSERT new IN db
```

Note that we could now alter `ADD TO db` to:

```
HOW TO ADD TO db:
  REPLACE {} WITH modified {} IN db
```

which would have the same effect.

Putting it all together

Now that we've got a number of useful how-tos, we can put them together with a driving program. In our final version we had a parser with an extensive query language. For now here is a simple version.

There is a concept of the *current selection*. Initially the selection is the whole database. The selection command has the form "*key* = *value*" or "*key* ~ *value*" for exact or approximate matches (for instance "Land = UK"). Subsequent selection commands act on the current selection. The command *all* selects the whole database again.

You use *show* to show the current selection, *brief* to summarize the current selection, *format* to format the selection, *change* to modify records in the selection. Finally *help* gives a help message.

```
HOW TO DATABASE:
  SHARE db, field.names
  PUT db IN selection
  WRITE #db, "entries"/
  GET COMMAND
  WHILE command <> "quit":
    IF stripped command <> "":
      OBEY
    GET COMMAND
GET COMMAND:
  WRITE "> "
  READ command RAW
  PUT lower command IN command
OBEY:
  PUT split command IN words
  SELECT:
    command = "new":
      ADD TO db
    #words = 3: \Selection command
      PUT obey.select IN selection
    command = "all": \Select all
      PUT db IN selection
    command = "show":
      SHOW selection
    command = "brief":
      FORMAT selection WITH brief
    command = "format":
      FORMAT selection with formats
    command = "change":
      FOR r IN selection:
        PUT modified r IN new
        REPLACE r WITH new IN db
        REPLACE r WITH new IN selection
    command = "help": GIVE HELP
    ELSE:
      WRITE "Not recognised"/
      WRITE "Use 'help' for help" /
obey.select:
  SELECT:
    words[1] not.in field.names:
      WRITE "No such field name" /
      WRITE "Ignored" /
      RETURN selection
    words[2] = "=":
      PUT selection matches
        {[words[1]]: words[3]} IN s
    words[2] = "~":
      PUT selection contains
        {[words[1]]: words[3]} IN s
    ELSE:
      WRITE "Operator ", words[2]
      WRITE "unrecognised"/
      WRITE "Ignored" /
      RETURN selection
  SELECT:
    #s = 0:
      WRITE "No matches" /
      WRITE "Ignored" /
      RETURN selection
    #s = 1: WRITE "1 entry"/
    ELSE: WRITE #s, "entries"/
  RETURN s
```

You could easily add an undo command, by keeping a copy of the current selection every time it gets changed. For instance, replace

```
PUT obey.select IN selection
```

with

```
PUT selection IN undo
PUT obey.select IN selection
```

Then the undo command only has to swap the values of the copy and the current selection:

```
PUT undo, selection IN selection, undo
```

Initially, `undo` should be empty.