Exploring The Wonder Of D-Charting

SVFIG

Aug. 22, 2020

Bill Ragsdale

```
#function out=cprint(FileIn,FirstRow,LastRow,ColSelector,FieldWidths,Trailing]
# Formatted variable column string reporter.

 # for test
 FileIn = {'12345678901234567890','abc',1234567890 ,'def','ghi';'jkl','mno',888,'pqr','stu'} ;


### As a function this section will be deleted  ###
 SelectionParams = {'none' ,      1,       4,        [5:-1:2 ] } ;
# [ input text, first row, last row, column span, Field Widths, Trail Char ]

# Characters per data field
Col_1 = 15 ; Col_2 = 8; Col_3 = 6 ; Col_4 = 8 ; Col_5 = 8 ;
Trailing = 2 # only for testing. Blanks after each field

 # Control parameters only for testing.
FirstRow       = cell2mat(SelectionParams(2))  ;
LastRow        = cell2mat(SelectionParams(3))  ;
ColSelector    = cell2mat(SelectionParams(4))  ;
FieldWidths    = [Col_1, Col_2, Col_3, Col_4, Col_5]  ;
Trailing       = blanks(Trailing) ;

 ### Program from here on ###


 #  Keep the user's parameters in the range of the text file.
 FirstRow    = max(FirstRow,1) ; # 1 or lower
 LastRow     = min(LastRow,rows(FileIn)) ; # keep in range
 ColSelector = max(ColSelector,1) ; # 1 or greater
 ColSelector = min(ColSelector,columns(FileIn)) # keep in range


# Declare variables
ActiveWidth=1 ; FieldBuffer=1; LineBuffer=1 ;

# Probably should limit row and column to actual rows and columns size.

printf('\n') ; # clean for display
# to make nested loops must be a conditional test for a num2str, just in case
for rownum = FirstRow(1):LastRow(1) # Over the specified sequence of rows
    LineBuffer = char([]) ;
    for colnum = ColSelector ;   # The vector of column numbers
     CW = FieldWidths(colnum) ;   # column width
     BW = blanks(CW)           ;  # buffer blanks, left Number or right (text)
     if  isnumeric(FileIn{rownum,colnum}) ; # separate action for numbers & text
 #        disp('have a number')
         TargetText = num2str(TextIn{rownum,colnum}) ; # grab the rxc cell contents
 #   size(TargetText)
 #   double(TargetText)
         FieldBuffer = [BW TargetText] ; # apply leading blanks
 #        double(FieldBuffer)
 #        isstring(FieldBuffer)
         FBcols = columns(FieldBuffer) ; # get the current buffer size
         FieldBuffer = FieldBuffer(FBcols-CW+1:end) ; # trim to specified width
 #        iscell(FieldBuffer)
       else  # is text
 #        disp(' ')
 #        disp('Have text')
         TargetText= (FileIn{rownum,colnum}) ;# gram the rxc cell contents
         FieldBuffer = [TargetText BW]  ; # add the trailing blanks
         FieldBuffer = [FieldBuffer(1,1:CW)] ; # trim to specified width
      endif
       #  double([LineBuffer FieldBuffer FieldSep])
#        size(LineBuffer)
       LineBuffer = [LineBuffer FieldBuffer Trailing] ; # append to buffer
    endfor # loop for next field in the current row
```

What We'll Cover

A bit of programming history.

Traditional flow charting.

D-chart symbols.

D-chart structures.

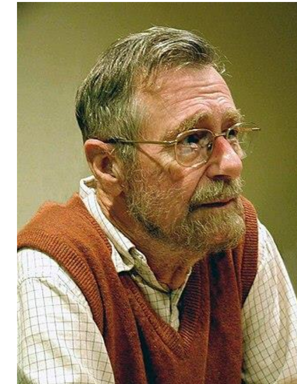
Comparing traditional and D-charting.

D-charting and Forth structures.

Some actual programming examples.

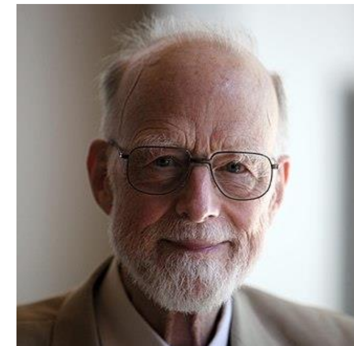
The Four Horsemen of Apocalypse

The Three Horsemen of Structured Programming



Edsger Djikstra coined the phrase “Structured Programming. His letter to CACM, "A Case Against the Goto Statement", was retitled by editor Niklaus Wirth "Go To Statement Considered Harmful“.   Turing Award 1972. Popularized parsing into RPN.
“Control complexity by layering”.

Niklaus Wirth, Turing Award 1984.
Designer of Algol and Pascal.



CAR ‘Tony’ Hoare created Quicksort and Quickselect. Coauthor of *Structured Programming* with Djikstra and Dahl. Turing Award 1970

FORTH DIMENSIONS

OCTOBER/NOVEMBER VOLUME 1, NO. 3

CONTENTS

HISTORICAL PERSPECTIVE Page 24

CONTRIBUTED MATERIAL Page 24

DTC vs ITC for FORTH Page 25
David J. Sirag

D-CHARTS Page 30
Kim Harris

FORTH vs ASSEMBLY Page 33
Richard B. Main

HIGH SPEED DISC COPY Page 34
Richard B. Main

SUBSCRIPTION OPPORTUNITY Page 35

October, 1978

D-CHARTS

Kim Harris

An alternative style of flowcharts called D-charts will be described. But first the purpose of flowcharting will be discussed as well as the shortcomings of traditional flowcharting.

A flowchart should be a tool for the design and analysis of sequential procedures which make the control flow of a procedure clear. With FORTH and other modern languages, flowcharts should be optimized for the top-down design of structured programs and should help the understanding and debugging of existing ones. An analogy may be made with a road map. This graphic representation of data makes it easy to choose an optimum route to some destination, but when driving, a sequential list of instructions is easier to use (e.g., turn right on 3rd street, left on Ave. F, go 3 blocks, etc.). Indentation of source statements to show control structures is helpful and is recommended, but a two dimensional graphic display of those control structures can be superior. A good flowchart notation should be easy to learn, convenient to use (e.g., good legibility with free-hand drawn charts), compact (minimizing off-page lines), adaptable to specialized notations, language, and personal style, and modifiable with minimum redrawing of unchanged sections.

Traditional flowcharting using ANSI standard symbols has been so unsuccessful at meeting these goals that "flowchart" has become a dirty word. This style is not structured, is at a lower level than any higher level language (e.g., no loop symbol), requires the use of symbol templates for legibility, and forces program statements to be crammed inside these symbols like captions in a cartoon.

D-charts have a simplicity and power similar to FORTH. They are the invention of Prof. Edsger W. Dijkstra, a champion of top-down design, structured programming, and clear, concise notation. They form a context-free language. D-charts are denser than ANSI flowcharts usually allowing twice as much program to be displayed per page. There are only two symbols in the basic language; however, like FORTH, extensions may be added for convenience.

Sequential statements are written in free form, one below the other, and without boxes.

```
statement
next statement
next statement
     :
```

The only "lines" in D-charts are used to show nonsequential control paths (e.g., conditional branches, loops). In a proper D-chart, no lines go up; all lines either go down or sideways. Any need for lines directed up can be (and should be) met with the loop symbols. This simplifies the reading of a D-chart since it always starts at the top of a page and ends at the bottom.

It is customary to underline the entry name (or FORTH definition name) at the top of a D-chart.

2-WAY BRANCH SYMBOL

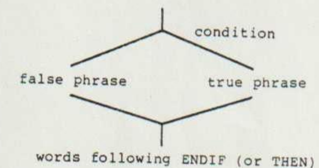
In FORTH, this structure takes the form:

```
condition IF    true phrase
          ELSE  false phrase
          THEN  .
```

Another FORTH structure which is used for conditional compilation has more mnemonic names:

```
condition IFTRUE     true phrase
          OTHERWISE  false phrase
          ENDIF      .
```

The D-chart symbol has parts for each of these elements:

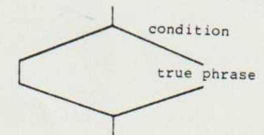


The "condition" is evaluated. If it is true, the "true phrase" is executed; otherwise, the "false phrase" is executed. The words following ENDIF (or THEN) are unconditionally executed.

If either phrase is omitted, as with

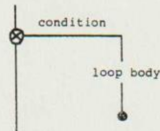
```
condition IF   true phrase   THEN
```

a vertical line is drawn as shown:



LOOP SYMBOL

The basic loop defining symbol for D-charts is properly structured.



The switch symbol:

indicates that when the switch is encountered, the "condition" (on the side line) is evaluated.

1. If the "condition" is true, then the side line path is taken; if false, then the down line is taken (and the loop is terminated).

2. If the side line is taken, all statements down to the dot are executed. The dot is the loop end symbol and indicates that control is returned to the switch.

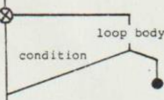
3. The "condition" is again evaluated. Its outcome might have changed during the execution of the loop statement.

   Repeat these steps starting with Step 1.

This symbol tests the loop condition before executing the loop body. However, other loops test the condition at the end of the loop body (e.g., DO .. LOOP and BEGIN .. END) or in the middle of the loop body. This loop symbol may be extended for these other cases by adding a test within the loop body. Consider the FORTH loop structure

```
BEGIN   loop body   condition END   .
```

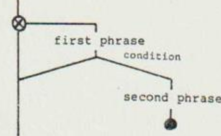
The loop body is always executed once, and is repeated as long as condition is false. The D-chart symbol for this structure would be:



A more general case is

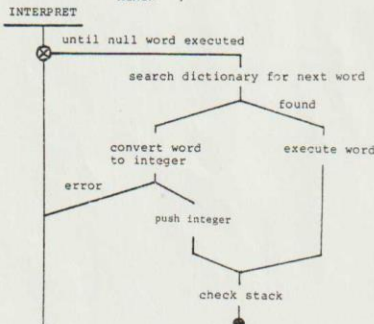
```
          BEGIN   first phrase
condition IF    second phrase
          AGAIN
```

which is explained better graphically than verbally:



Both previous symbols may be properly nested indefinitely. The following example shows how these symbols may be combined. This is the FORTH interpreter from the F.I.G. model.

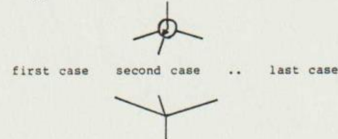
```
: INTERPRET   BEGIN    (')    IF HERE NUMBER
                              ELSE EXECUTE
                              THEN
                       ?STACK
              AGAIN    ;
```



-->

n-WAY BRANCH SYMBOL

A structured n-way branch symbol (sometimes called a CASE statement) may be defined for convenience. (It is functionally equivalent to n nested 2-way branches). One style for this symbol is:



The condition is usually an index which selects one of the cases. The rejoining of control to a single line after the cases are required by structured programming. Depending on the complexity of the cases, this symbol may be drawn differently.

D-charts are efficient and useful. They are vastly superior to traditional flowchart style.

;S KIM HARRIS

P.O. Box 8045
Austin, TX 78712
November 3, 1978

Editor, Forth Dimensions:

Thank you for your card and subsequent letter. I am sorry that I did not get back to you sooner with a copy of the source code for my FORTH system. Frankly, I was surprised that you are interested in the system, since it is rather limited in facilities and conforms with no other FORTH version in terms of names. I stopped work on the system just about the time I began to receive manuals form DECUS and the 6502 FORTH form FIG. I can see now how I would add an assembler, text editor, and random block i/o to the system, but my duties at work and at school preclude any further development of U.T. FORTH for now.

I want to especially thank you for informing me of Paul Bartholdi's visit to the University of Texas. I was able to meet with him and we had a very stimulating discussion for about an hour and a half. I was surprised to learn from him how widely FORTH is used commercially, though usually under other names. We also discussed two extensions to the language that I believe greatly enhance it: (1) syntax checking on compilation for properly balanced BEGIN..END and IF..ELSE..THEN constructs, and (2) the functions "n PARAMETERS" and "PAR]" to "PARN" that allow explicit reference to parameters on the stack. Finally, he showed me some programming examples from the FORTH manual he wrote which provide first-hand proof of the ease of programming rather sophisticated problems in FORTH. It is especially important because most people in the computer science department here respond to my presentation of FORTH with a resounding lack of interest. After all, they keep abreast of the field and if they have not heard of it....

I have been promoting FORTH among the local computer clubs and look forward to the results of FIG's micro computer efforts. Please keep in touch.

Sincerely yours,

Greg Walker

SYSTEM LANGUAGE 1

SL/1 was written by Emperical Research Group, Inc. to be exactly what it says it is, a SYSTEM language. SL/1 is a small interactive incremental compiler that generates indirect threaded code. It is a 16 bit pseudo machine for use on mini and micro computers. New definitions can be added to an already rich set of intrinsic instructions. It is this extensibility that allows any user to create the most optimum vocabulary for his individual application.

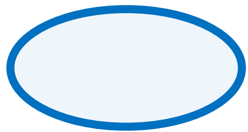
SL/1 is a virtual stack processor. Using the RPN concept for both variables and instructions makes it possible to extend stepwise programming to include stepwise debugging. SL/1 does this quite nicely. The RPN stack is also one of the most effective means of implementing top down design, bottom up coding.

SL/1 operates on a principle of threaded code. All of the elements of SL/1 (procedures, variable, compiler directives, etc.) reference the previous entry. Thus, each code indirectly "threads" the others and is in turn threaded by the code following it. Because SL/1 is a pseudo machine, portability between different processors and hardware is readily accomplished. The low level interpreter is really the P-machine. It is small (only 11 bytes are used), and fast.

One of the most powerful features of SL/1 is the fact that is uses all on-line storage media as virtual memory. In effect the user can write programs in SL/1 using the full capacity of disk storage and never be concerned with placement of information on the disk. SL/1 allows you to program machine code procedures in assembler using a high level language. This can optimize I/O or math routines.

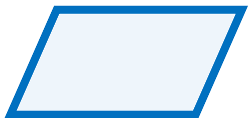
The above information was excerpted from a press release of November 3, 1978. For further information, contact Mr. Dick Jones, Emperical Research Group, Inc., 28206 144th Avenue, S.E., Kent, WA 98031. Phone (206) 631-4851.

Classical Flowchart Symbols

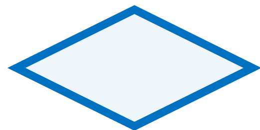


BEGIN/END

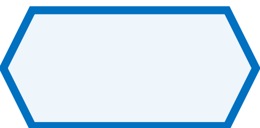
ASSIGNMENT



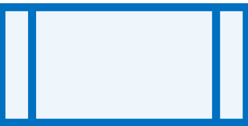
IN/OUT



DECISION



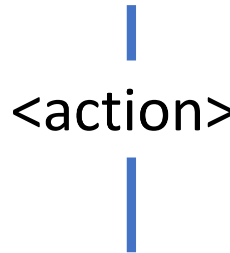
BEGIN LOOP



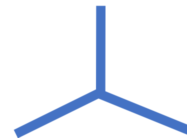
SUBROUTINE

Loop by upward arrow.

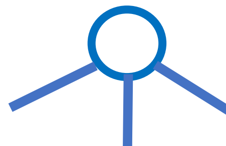
D-Chart Symbols



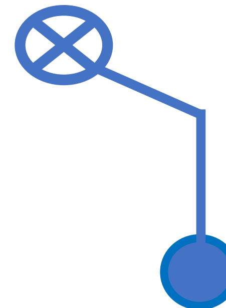
Code in Sequence (either between line segments or beside.



TEST (IF)



CASE or N-WAY



SWITCH and RETURN POINT

RETURN TO SWITCH

D-Chart Principles

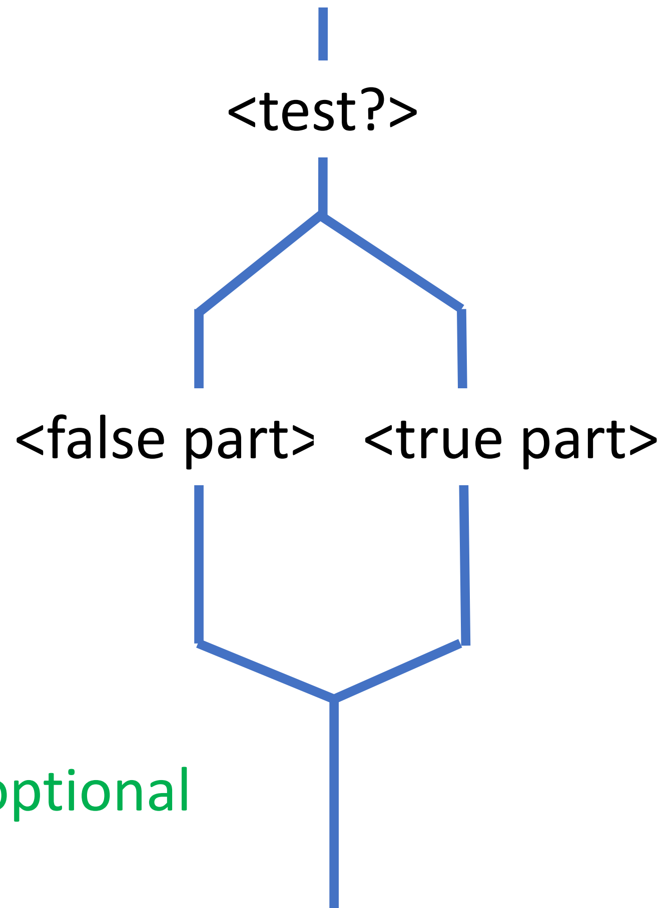
Programs start at the top and end at the bottom.

Show sequential statements vertically.

Lines show control paths.

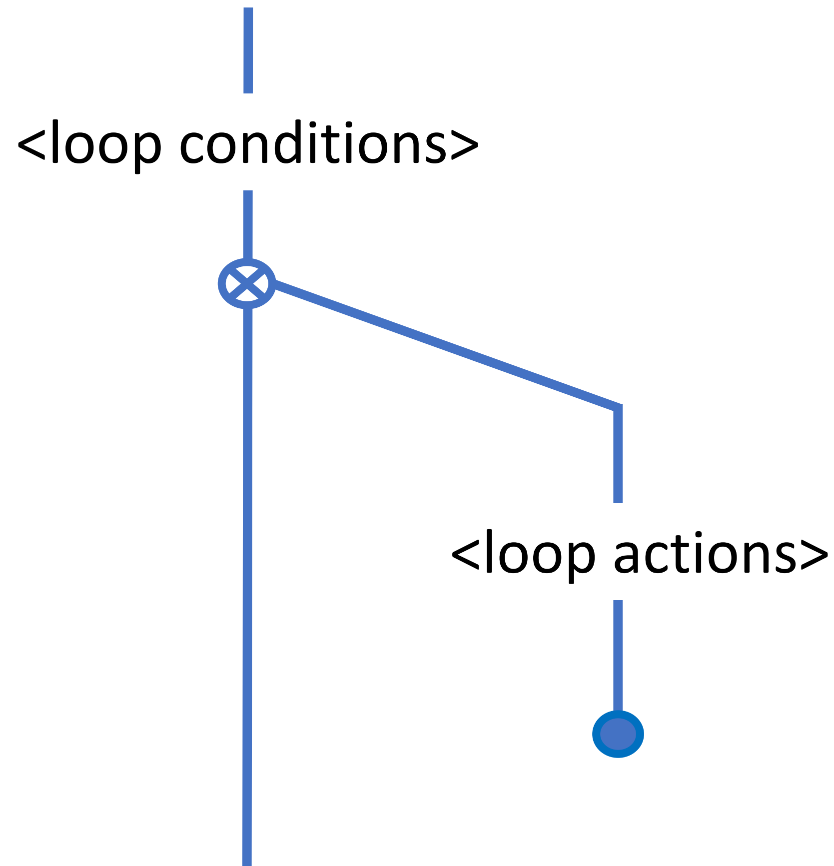
All lines go downward or to the side.

Example Of A Test

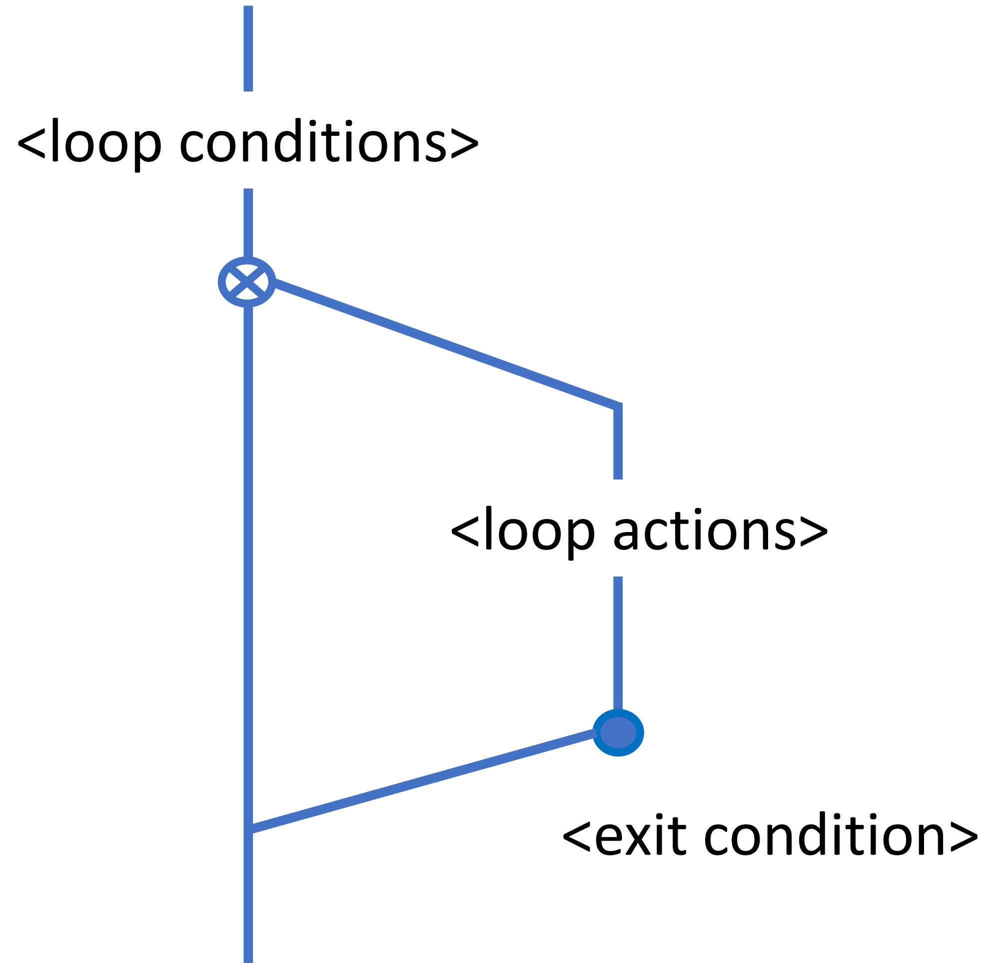


<false part> is optional

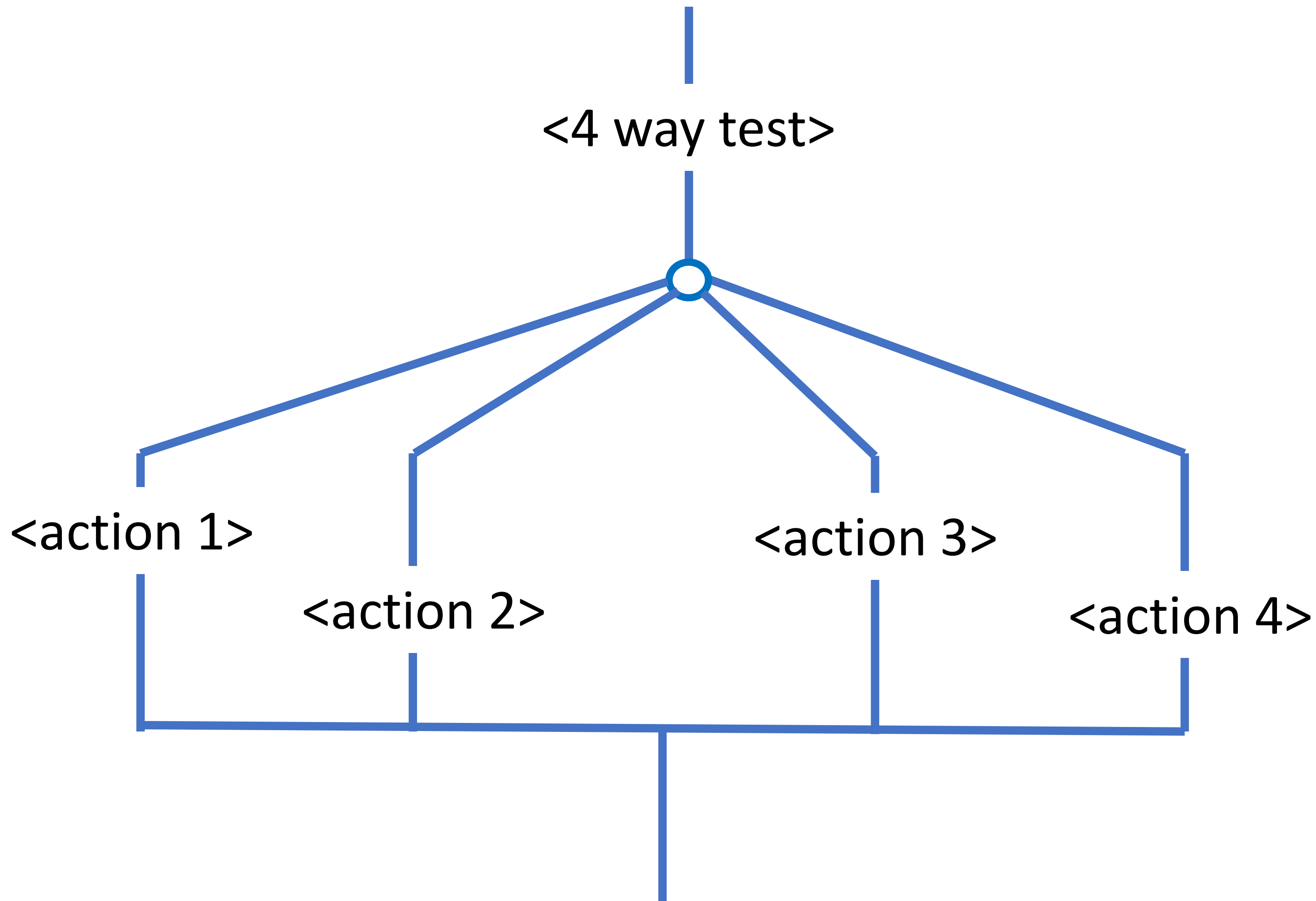
Example Of A Loop



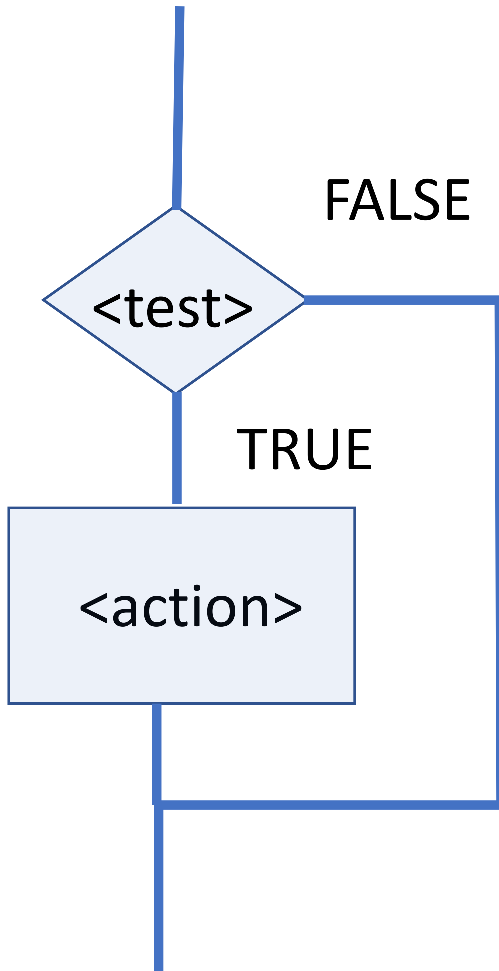
Example Of A Loop with a Concluding Test (BEGIN-UNTIL)

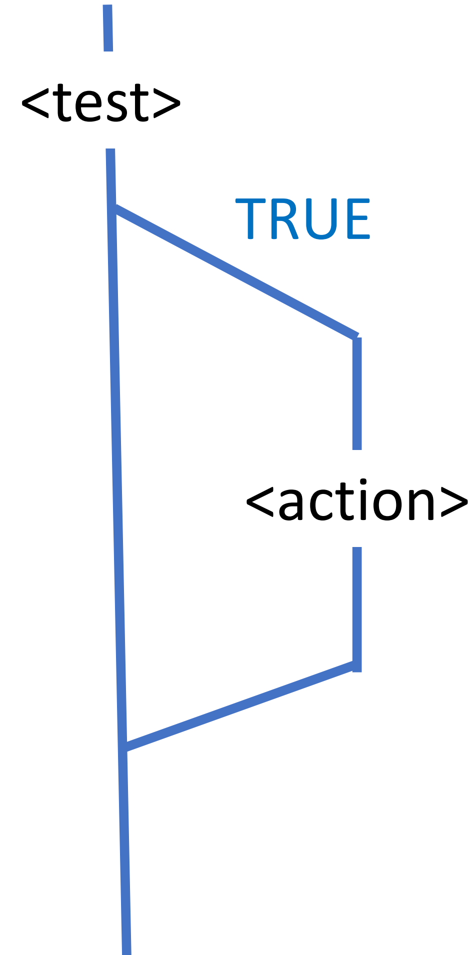


Example Of A Case Statement

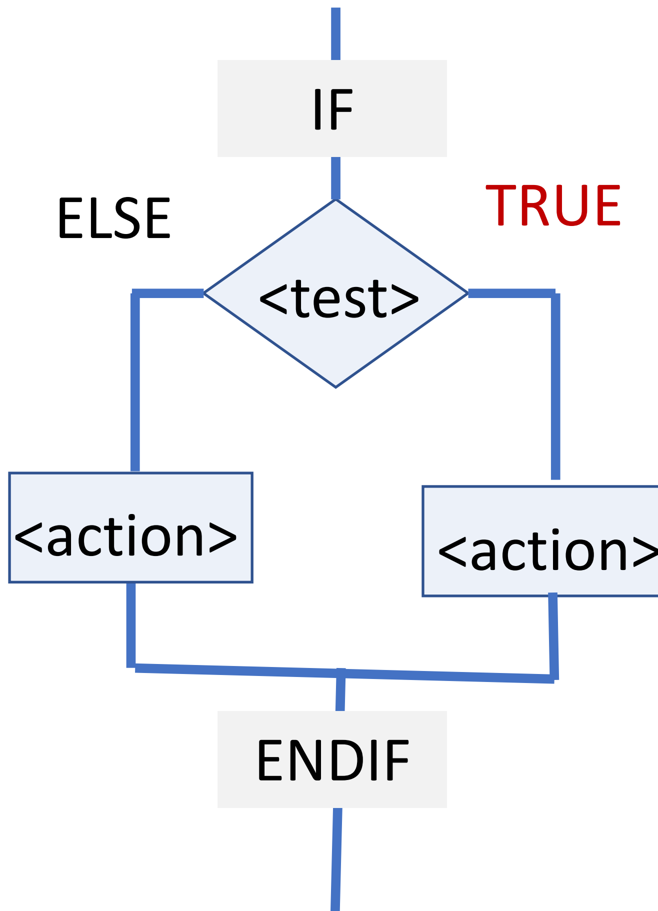


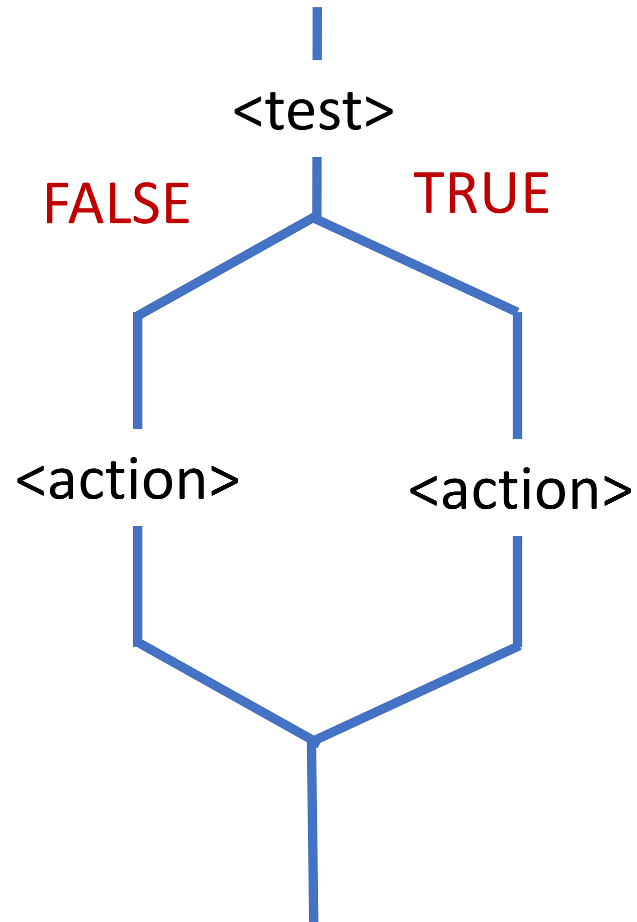
One-Way Logical Test



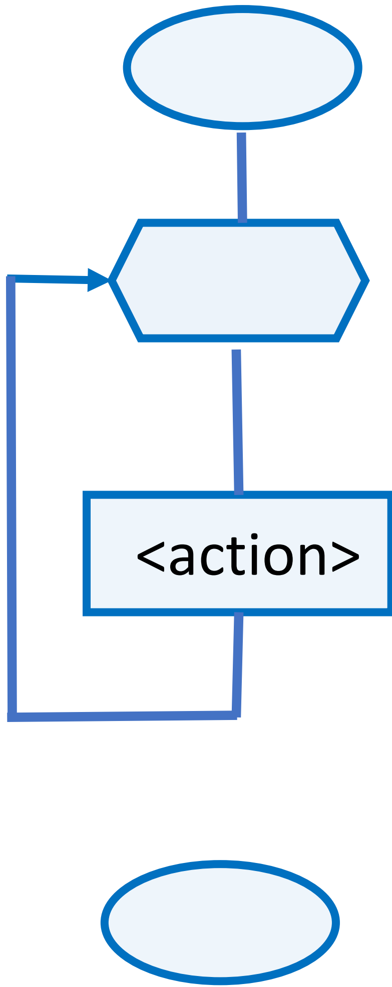


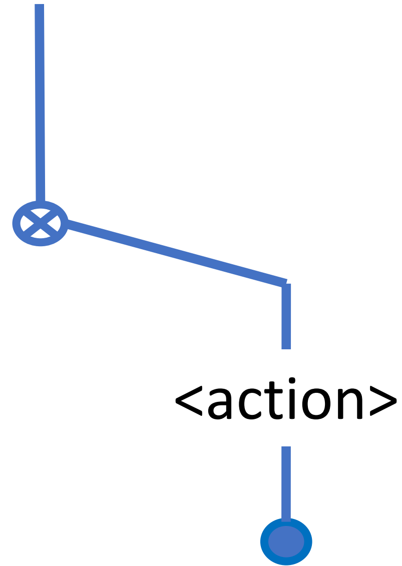
Two Way Logical Test



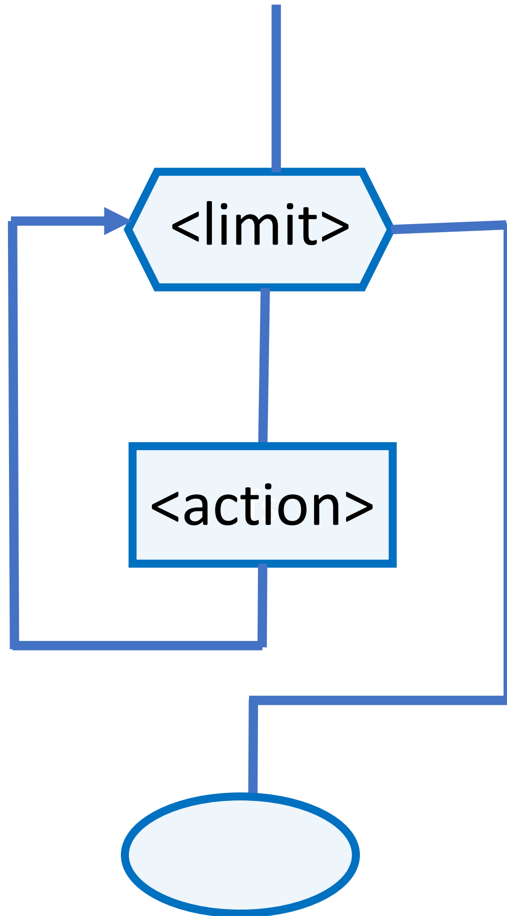


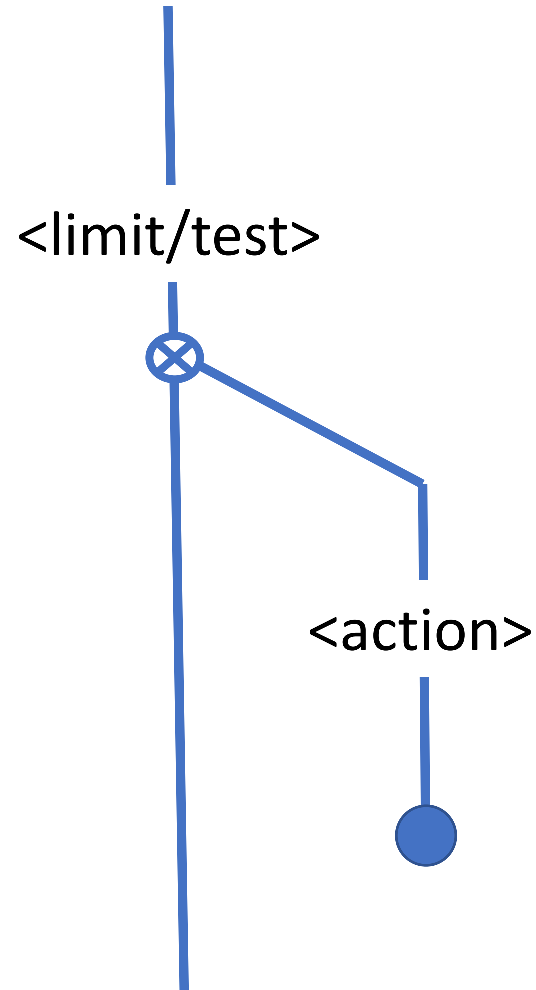
Infinite Loop



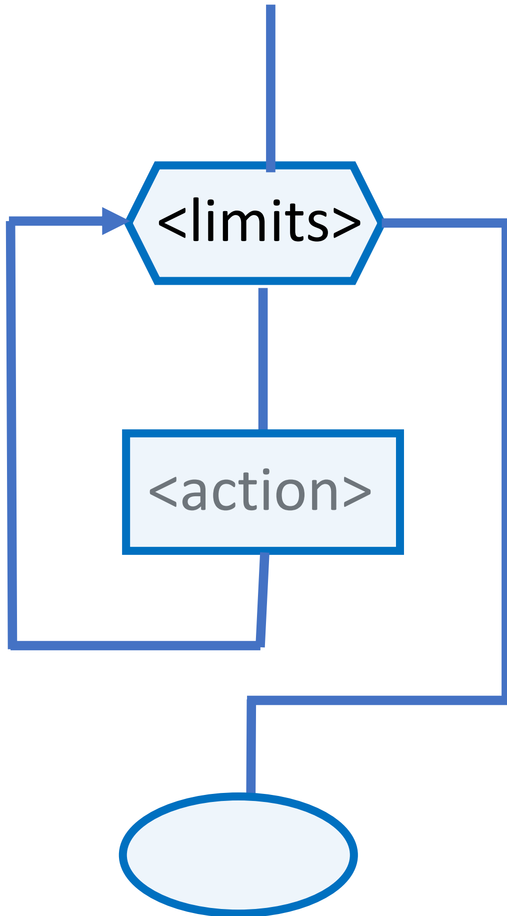


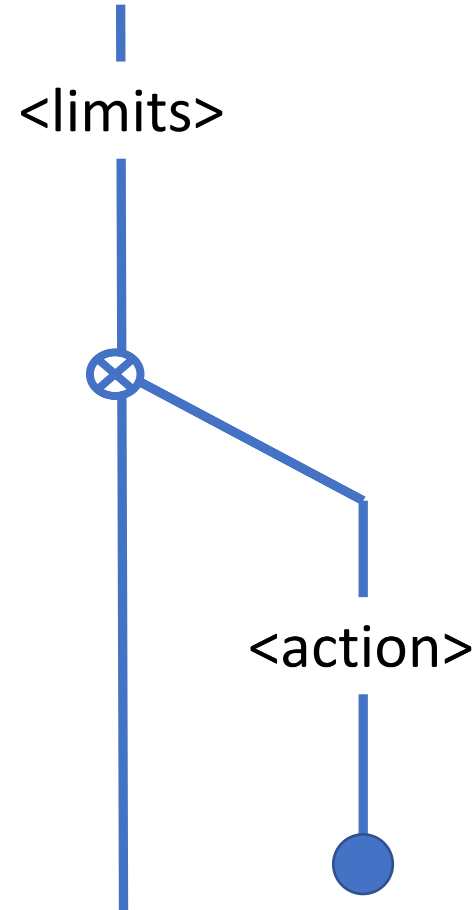
Indefinite Loop





Finite Loop





Pros and Cons of D-Charts

PRO

- D-Charts have more information per page.
- Quick to draw for rapid revision.
- No need for a template.
- Context free for any language.

CON

- Will confuse those unfamiliar.
- Not suitable for publication.

Let’s Look At Forth Code Fragments

Two Way Logical Test

. . .

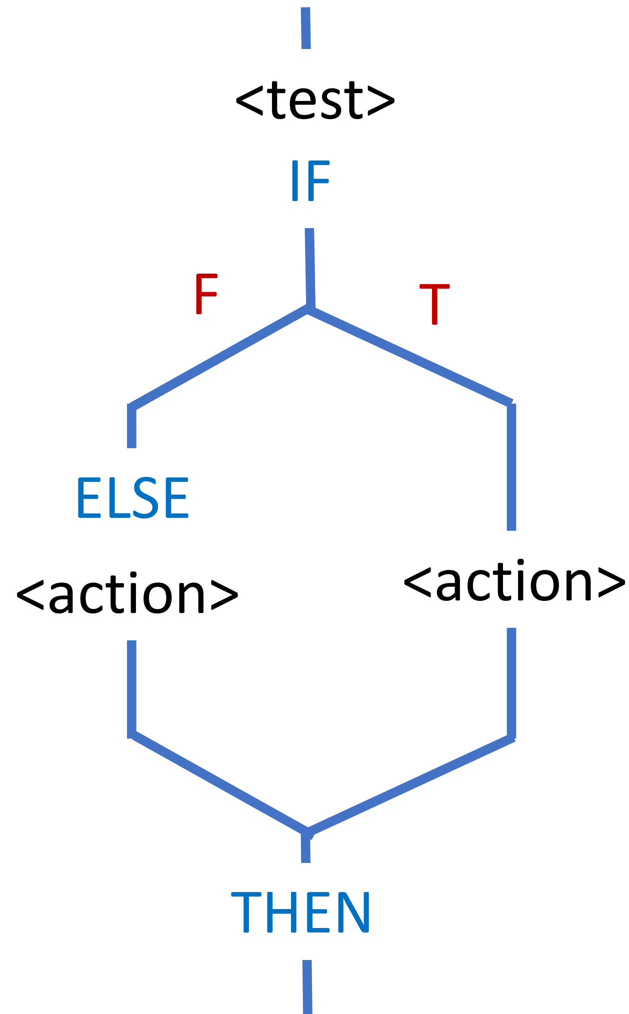
<test>

IF <action>

ELSE <action>

THEN

. . .



Infinite Loop

. . .

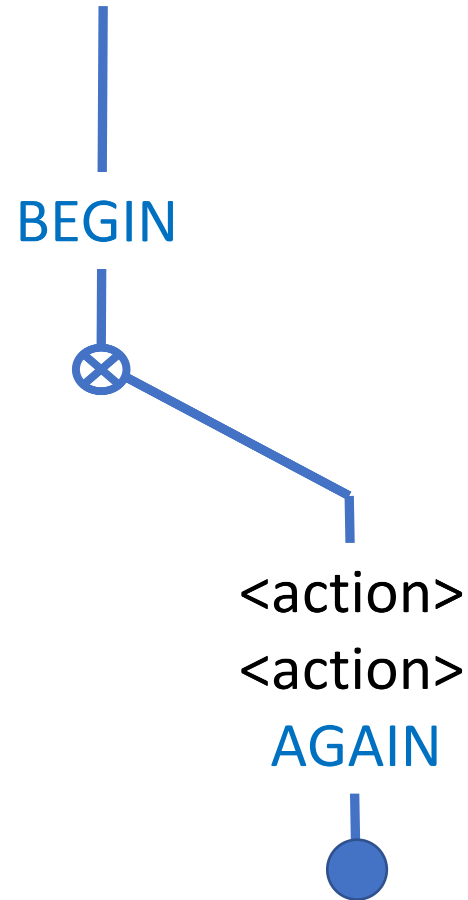
BEGIN

<action>

<action>

AGAIN

. . .



Definite Loop

. . .

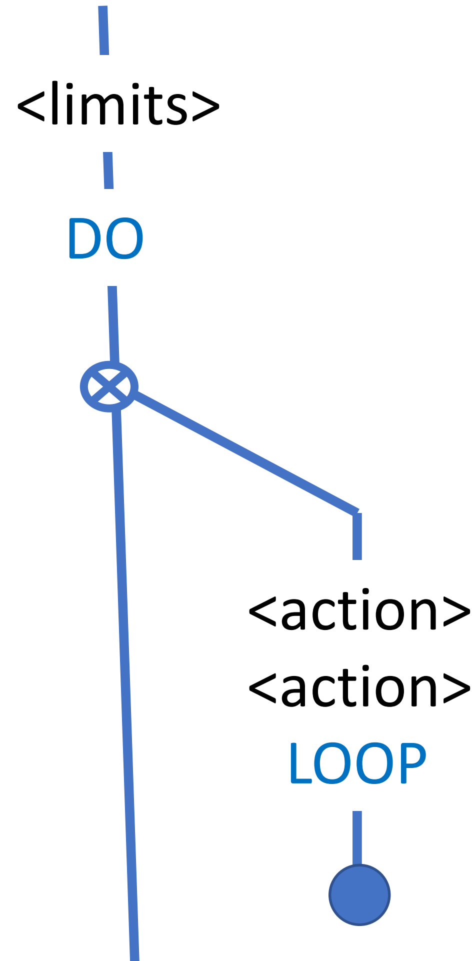
<limits>

DO <action>

<action>

LOOP

. . .



Indefinite Loop

. . .

BEGIN <test>

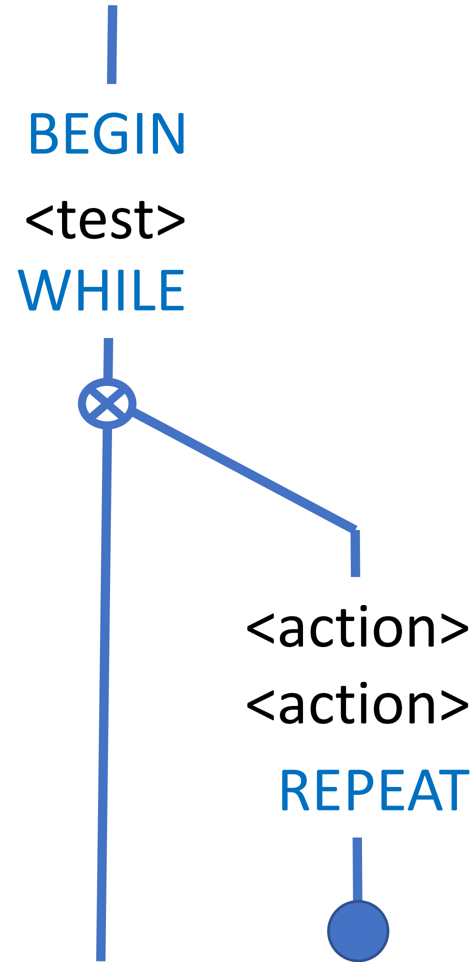
WHILE

<action>

<action>

REPEAT

. . .



Another Indefinite Loop

. . .

BEGIN

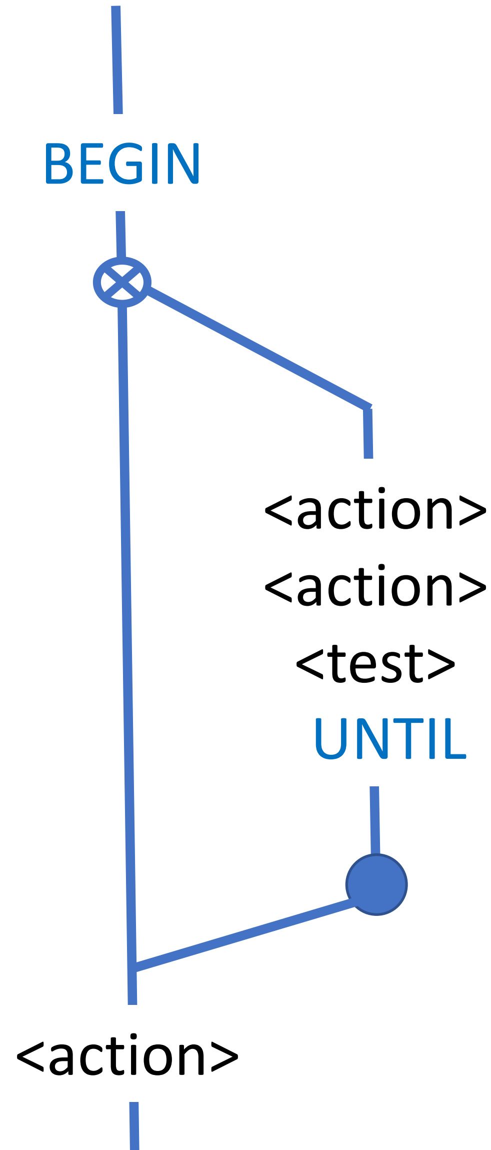
<action>

<action>

<test>

UNTIL

. . .



Three Examples

Absolute Value Conversion

and

String Formatting

and

Twos-Complement Math

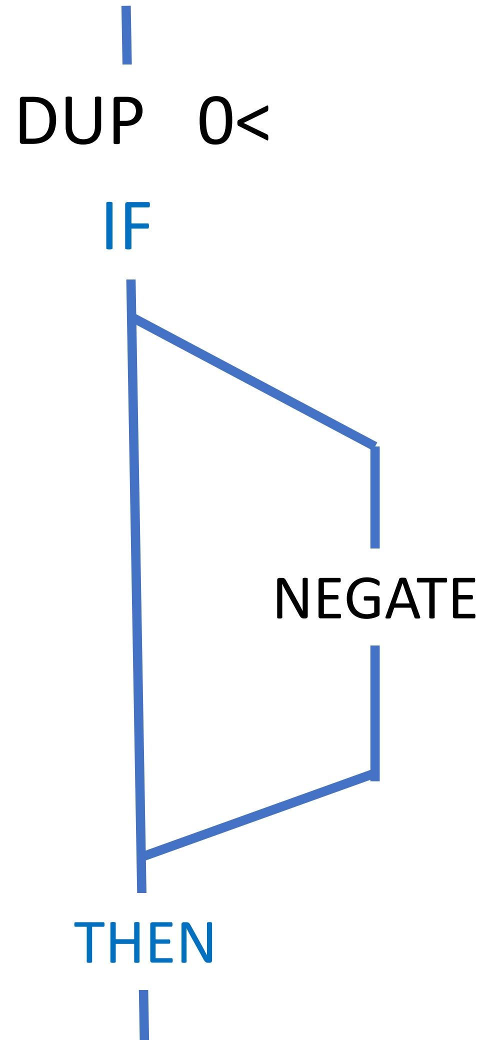
Absolutue Value Conversion

. . .

DUP 0<

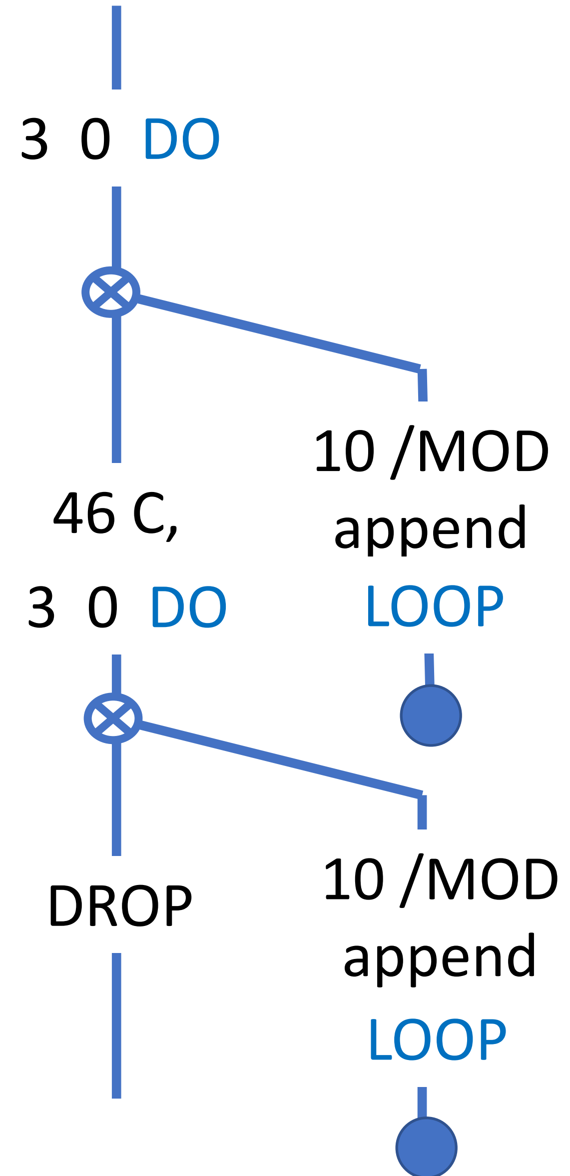
IF NEGATE THEN

. . .

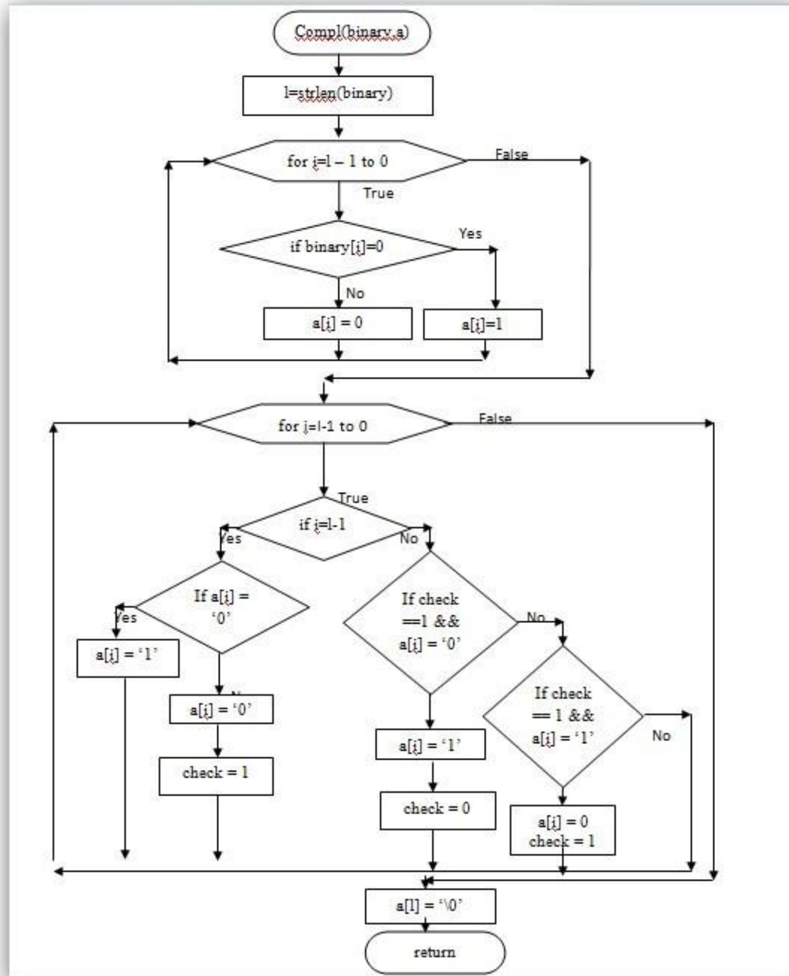


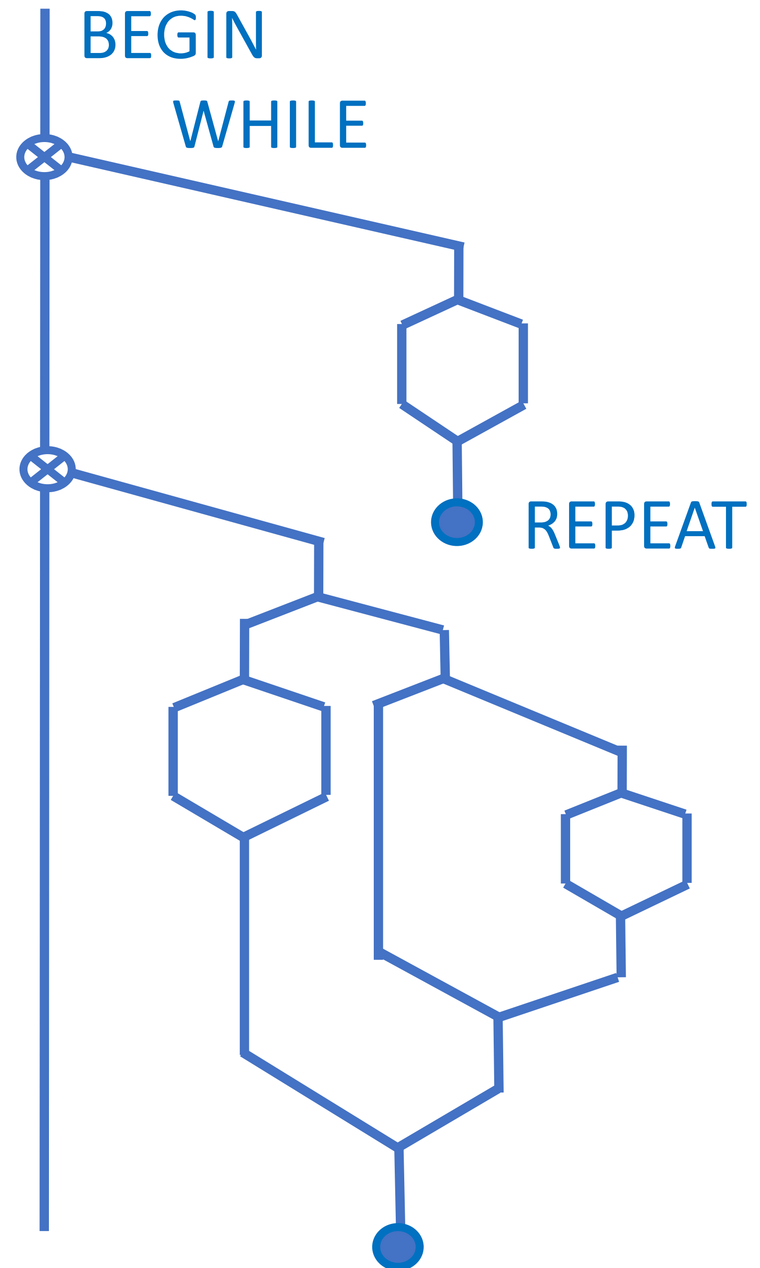
Number Conversion into: ‘nnn.nnn’

. . .
 3 0 (3 digits)
DO 10 /MOD append
    LOOP
46   C, (ascii dot)
3 0
DO 10 /MOD append
    LOOP
DROP   . . .



Twos Complement Arithmetic





Create A Report From A CSV File

Input CSV file
Setup control params
Convert all rows
And display

This is the whole program in four lines.

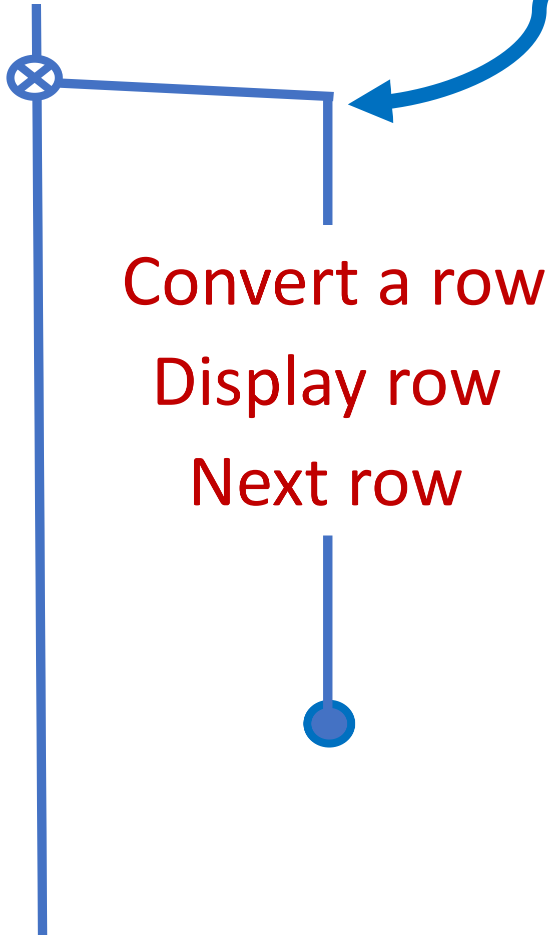
Create a report from a CSV file

Input CSV file
Setup control params
Convert all rows
And display

This is the whole program in four lines.

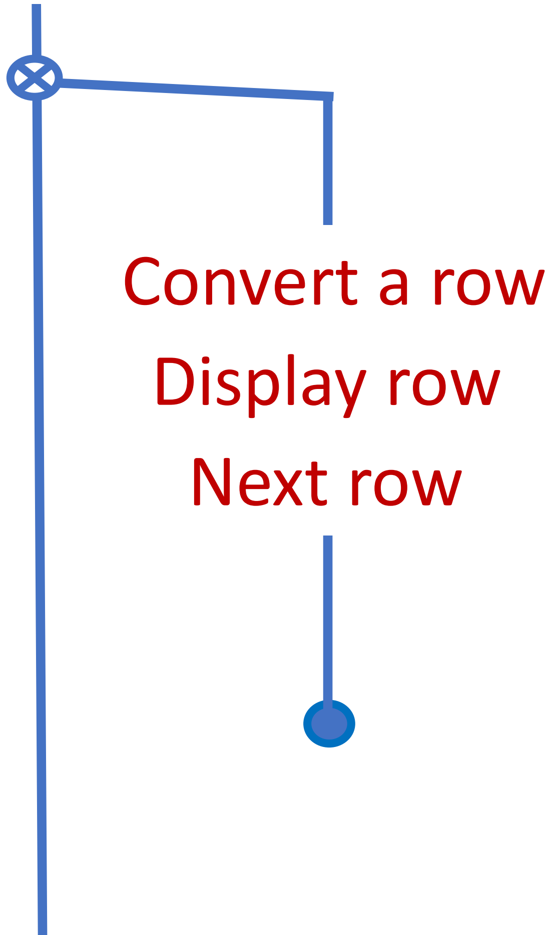
So how do we convert all rows for output?

Input CSV file
Setup control params
Over all rows



Add a loop over all the rows in the file

Input CSV file
Setup control params
Over all rows



Add a loop over all the rows in the file

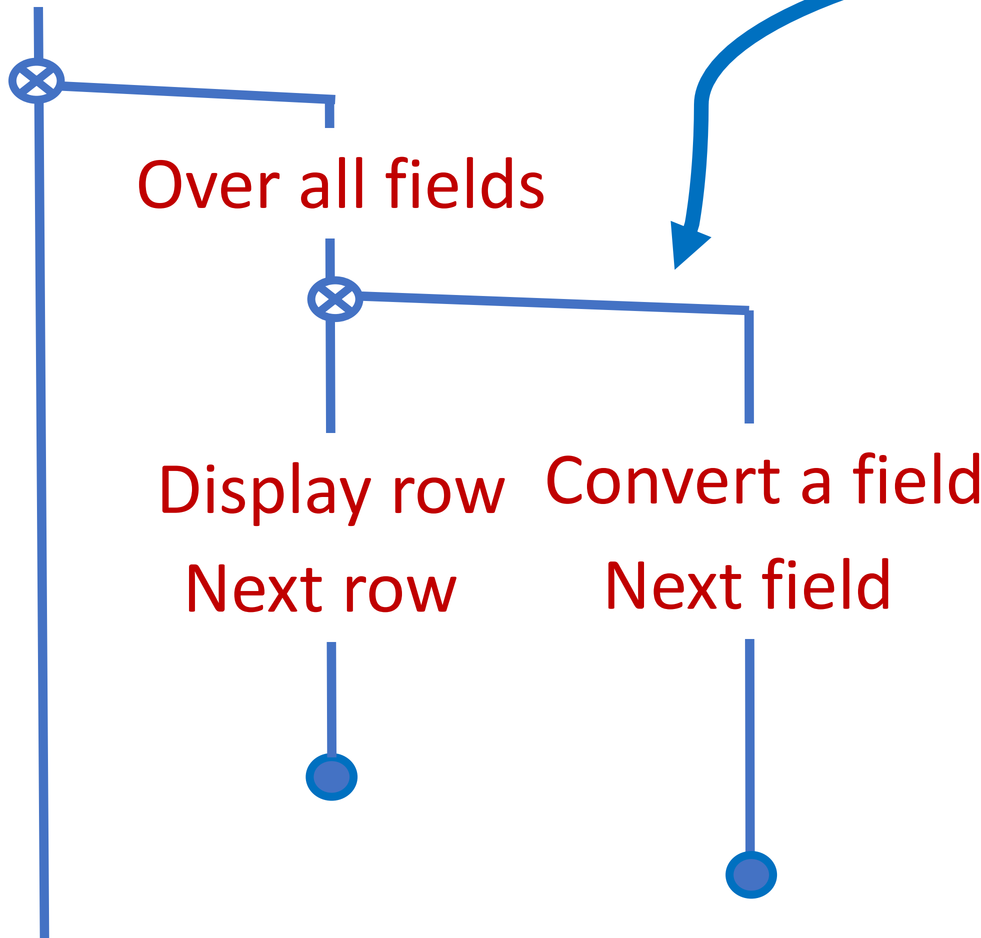
So how do we convert one row?

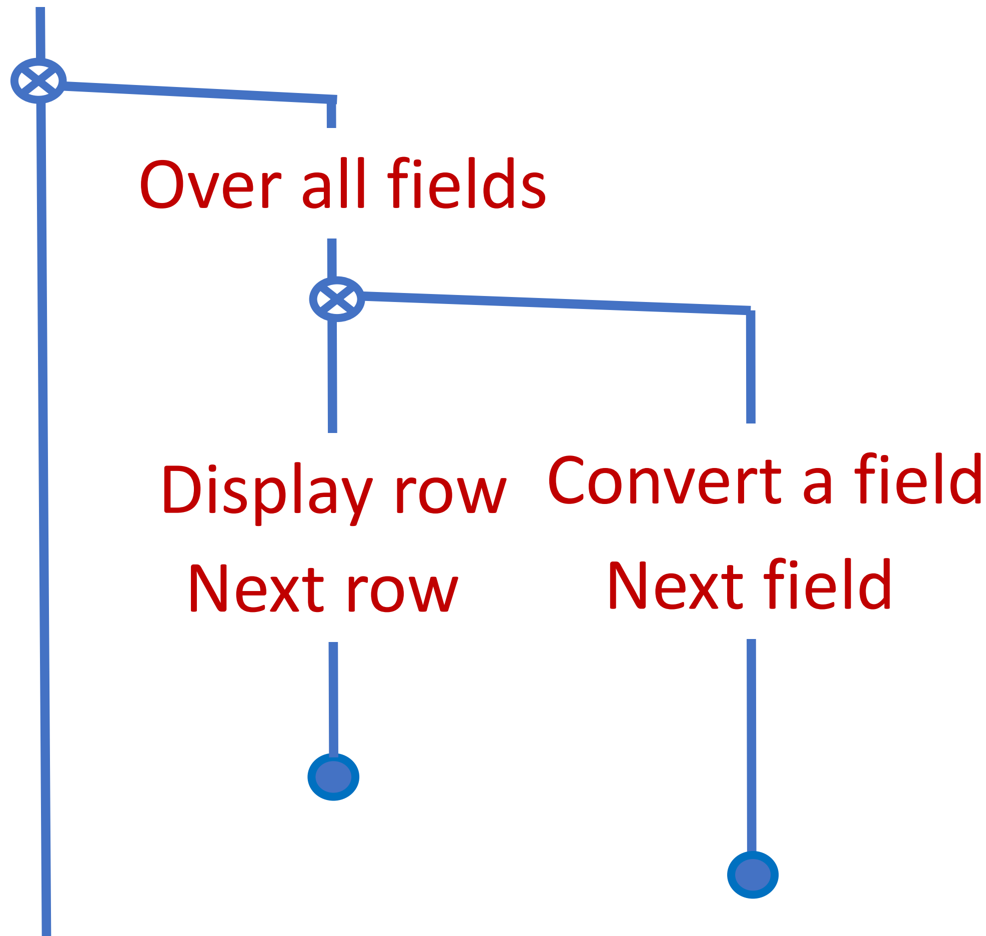
Input CSV file

Setup control params

Over all rows

Add a loop over all fields in a row.





Add a loop over all fields in a row.

So how do we convert one field?

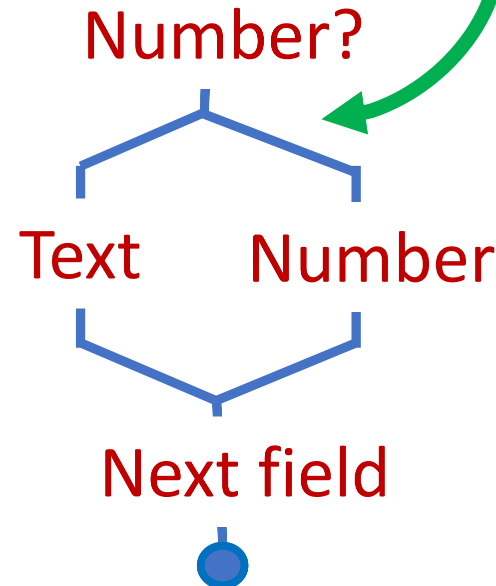
Input CSV file
Setup control params
Over all rows

Over all fields

Display row
Next row

Add a test for input data type.

Right justify for numbers; left justify for text.



```
:  report
read-file    setup-parameters
BEGIN  another-row?
     WHILE   (rows remain)
        BEGIN  another-field?
           WHILE  (while fields remain) a-number?
                IF   process-number
               ELSE  process-text  THEN
               next-field
           REPEAT
     display-row    next-row
   REPEAT
;
```

```
#function out=cprint(FileIn,FirstRow,LastRow,ColSelector,FieldWidths,Trailing]
# Formatted variable column string reporter.

 # for test
 FileIn = {'12345678901234567890','abc',1234567890 ,'def','ghi';'jkl','mno',888,'pqr','stu'} ;


### As a function this section will be deleted  ###
 SelectionParams = {'none' ,       1,        4,       [5:-1:2 ] } ;
# [ input text, first row, last row, column span, Field Widths, Trail Char ]

# Characters per data field
Col_1 = 15 ; Col_2 = 8; Col_3 = 6 ; Col_4 = 8 ; Col_5 = 8 ;
Trailing = 2; # only for testing. Blanks after each field

 # Control parameters only for testing.
FirstRow       = cell2mat(SelectionParams(2))  ;
LastRow        = cell2mat(SelectionParams(3))  ;
ColSelector    = cell2mat(SelectionParams(4))  ;
FieldWidths    = [Col_1, Col_2, Col_3, Col_4, Col_5]  ;
Trailing       = blanks(Trailing) ;

 ### Program from here on ###


 #  Keep the user's parameters in the range of the text file.
 FirstRow    = max(FirstRow,1) ; # 1 or lower
 LastRow     = min(LastRow,rows(FileIn)) ; # keep in range
 ColSelector = max(ColSelector,1) ; # 1 or greater
 ColSelector = min(ColSelector,columns(FileIn)) # keep in range


# Declare variables
ActiveWidth=  ; FieldBuffer=1; LineBuffer=1 ;

# Probably should limit row and column to actual rows and columns size.

printf('\n') ; # clean for display
# to make nested loops must be a conditional test for a num2str, just in case
for rownum = FirstRow(1):LastRow(1) # Over the specified sequence of rows
    LineBuffer = char([]) ;
    for colnum = ColSelector ;  # The vector of column numbers
     CW = FieldWidths(colnum) ; # column width
     BW = blanks(CW)          ; # buffer blanks, left Number or right (text)
     if  isnumeric(FileIn{rownum,colnum}) ; # separate action for numbers & text
 #       disp('have a number')
        TargetText = num2str(TextIn{rownum,colnum}) ; # grab the rxc cell contents
 #   size(TargetText)
 #   double(TargetText)
        FieldBuffer = [BW TargetText] ; # apply leading blanks
 #       double(FieldBuffer)
 #       isstring(FieldBuffer)
        FBcols = columns(FieldBuffer) ; # get the current buffer size
        FieldBuffer = FieldBuffer(FBcols-CW+1:end) ; # trim to specified width
 #       iscell(FieldBuffer)
       else # is text
 #       disp(' ')
 #       disp('Have text')
        TargetText= (FileIn{rownum,colnum}) ;# gram the rxc cell contents
        FieldBuffer = [TargetText BW]  ; # add the trailing blanks
        FieldBuffer = [FieldBuffer(1,1:CW)] ; # trim to specified width
      endif
       #  double([LineBuffer FieldBuffer FieldSep])
#       size(LineBuffer)
       LineBuffer = [LineBuffer FieldBuffer Trailing] ; # append to buffer
    endfor # loop for next field in the current row
```

On paper this is beautiful.

How can you get a running program?

IMMEDIATELY.

The answer?

Start with the pseudocode as printout

Stepwise, refine the pseudocode into code, with only one change at a time.

Express each line of pseudocode as a print statement.

You now have a functioning program even if it is only a shell.

```
:  report  cr
." Input CSV file"  cr
." Setup control params"  cr
." Convert all rows"  cr
." And display"
;
```

How can we make it do more?

Add a word ‘fake-load’ with CSV data in a text buffer & ‘display-buffer’ to display it.

And again, you always have a functioning program or are just one step away.

```
: report cr
  fake-load
  ." Setup control params" cr
  ." Convert all rows" cr
  display-buffer
;
```

Now. . .How to convert the rows?

Create code to translate fields of the CSV buffer into the desired output and display.

And you still have a functioning program or are just one step away.

```
: report cr
  fake-load
  ." Setup control params" cr
  simple-field-conversion
  display-buffer
;
```

Continue refining stepwise, by converting pseudocode into actual code and testing.

Summary

Learning D-Charts is like learning another programming language.

Only much, much simpler.

D-Charts augment writing pseudocode.

Plan on using and discarding many sheets of paper ***before*** going to your computer.

You can capture a whole program, or just a part, on a single sheet of paper.