Computer Science Department  
Indiana University  
Lindley Hall 101  
Bloomington, Indiana 47401  

Key Words and Phrases:  
structured programming, control structures,  
looping, while statement, until statement,  
exit statement, flowchart, ease of programming.  

CR Categories: 4.4.2, 4.4.9, 5.5.2  

TECHNICAL REPORT No. 10  

A CASE FOR WHILE-UNTIL  

DANIEL P. FRIEDMAN  
STUART C. SHAPIRO  

APRIL 5, 1974
Abstract

A new control structure construct, the *while-until*, is introduced as a syntactic combination of the *while* and the *until*. Examples are shown indicating that use of the *while-until* can lead to structured programs that are conceptually more manageable than those attainable without it. The *while-until* statement is then extended to a value returning expression which is shown to be more powerful than either the *while* or the *until*.

A major suggestion of structured programming is to employ looping control structures in order to break the program down into conceptually manageable units. The purpose of this paper is to propose an additional looping control structure construct (the *while-until*) that, in certain instances, yields program loops that are closer to the conceptual organization of the segment than is possible with the existing constructs. The *while-until* as a statement will be shown to be equivalent to the existing looping control structures. The *while-until* as a value (Bool-
ean) returning expression will be shown to be a more powerful control structure than the while or until structures discussed by Dijkstra [1].

The existing constructs that we are concerned with are

```plaintext
while β repeat s
```

and

```plaintext
repeat s until β.
```

Dijkstra [1] presents these graphically as in Figures 1 and 2.

![Fig. 1 while β repeat s](image)

![Fig. 2 repeat s until β](image)

The syntactic construct we are proposing is

```plaintext
while β₁ repeat s until β₂
```

which is presented graphically in Figure 3.
The while-until does not involve nesting, but is some other combination [2] of the features of the while and the until loops. The while-until may be replaced by the until or the while as the only looping structure since

\[
\text{while } \beta_1 \text{ repeat } s \text{ until } \beta_2
\]

is equivalent to

\[
\text{if } \beta_1 \text{ then repeat } s \text{ until if}(\beta_2 \text{ then true else } \neg \beta_1)
\]
and also to

if $\beta_1$ then begin s; while if($\beta_2$ then false else $\beta_1$) repeat s end

If escape (or break or exit) were employed, another equivalent form is

A: while $\beta_1$ repeat
   begin s; if $\beta_2$ then escape A end

end A

In those cases where Figure 3 is the desired control structure it appears that the while-until yields clearer, more understandable code than any of the above alternatives. We can paraphrase the semantic content of the while-until as: "while it is possible to try, keep trying until you succeed." The while and the until loops can be defined in terms of the while-until in the following way:

while $\beta$ repeat s $=_{\text{def}}$ while $\beta$ repeat s until false

and

repeat s until $\beta$ $=_{\text{def}}$ while true repeat s until $\beta$

The while-until is a natural control structure for searching, since every search terminates either by finding the desired element or by determining that it is not present. As an example, we show its use for a binary search:
comment Find item A in table T[1:N];
low := 0;
high := N + 1;
while low < high - 1 repeat
  try := (low + high) / 2;
  if T[try] < A then low := try else high := try
until T[try] = A;

An appropriate application for the while-until occurs whenever a loop includes two operations, one of which requiring a test prior to its execution and the other requiring a test which can only be performed after its execution. An example of this is: copy a file up to and including the end-of-file mark onto an output file, however, nothing may be written on the output file unless there is enough space for a record.

comment Copy file INPUT onto the file OUTPUT;
while Spaceleft(OUTPUT) repeat
  Inbuffer(INPUT, b);
  Outbuffer(OUTPUT, b)
until Eof(INPUT);

In languages in which statements are expressions having values, for example LISP [3], ALGOL 68 [4] and BLISS [5], the while-until can be assigned a value in an especially useful way. We define the value of the while-until expression to be the value of the last evaluated
Boolean. That is, the value of

\[
\text{while } \beta_1 \text{ repeat } s \text{ until } \beta_2
\]

is false if and only if the loop terminates due to the evaluation of \(\beta_1\) (see Figure 4). A non-Boolean value could be returned on certain termination conditions (e.g., \text{exit} in BLISS or predicates in LISP).

![Fig. 4 value returning while-until](image-url)
There are two ways in which the loop may be terminated: the programmer will want to ascertain which of the two Booleans caused termination. This is precisely the information provided by the value of the while-until.

Using the value of the while-until, we may easily incorporate the above search routine into an insertion.

```plaintext
comment T[1:M] is a table containing \( N < M \) active elements. Insert \( A \) in \( T \) if it is not already present;
low := 0;
high := N + 1;
if (while low < high - 1 repeat
  try := (low + high) / 2
  if T[try] < A then low := try else high := try
until T[try] = A)
then Insertafter(A,T,low);
```

The previously presented copy routine can be incorporated into an algorithm that uses up to \( N \) output files, depending on the length of the input file:

```plaintext
comment Place one copy of file INPUT onto OUTPUT[1:N] as needed;
i := 1;
while i ≤ N repeat
  Open(OUTPUT[i])
  if (if while Spaceleft(Output [i]) repeat
```


Inbuffer(INPUT, b);
Outbuffer(OUTPUT[i], b)
until Eof(INPUT))
then Close(OUTPUT[i]); i := i + 1
until Eof(INPUT);

Earlier, we showed that the while-until statement is definable in terms of just the while or just the until. This is not true, however, for the while-until expression. Peterson, Kasami, and Tokura [6], p. 506, have shown that "There exist flowcharts that cannot be translated into [if and until] programs with single-level exits, even if node splitting is allowed." Their example of such a flowchart is shown in Figure 5.

The following program using value-returning while-until and if expression is a translation of this flowchart.
s; if A then begin a₁;

if (while (while B repeat b₁
     until ¬(while ¬C repeat c₂ until true))
repeat c₁
until ¬(while (while D repeat d₁
     until ¬(while ¬A repeat a₂ until true))
repeat a₁ until true))
then d₂ else b₂
end
else begin a₂;

if (while (while D repeat d₁
     until ¬(while ¬A repeat a₂ until true))
repeat a₁
until ¬(while (while B repeat b₁
     until ¬(while ¬B repeat c₂ until true))
repeat c₁ until true))
then b₂ else d₂
end
Fig. 5 Flowchart from Peterson, Kasami, and Tokura [6].
Similarly, Ashcroft and Manna [7] have exhibited a flow-chart, shown in Figure 6, which cannot be translated into an if and while program. The following while-until program, due to M. Wand and D. Wise, is a translation of this flowchart.

if (while (if (while P repeat h until false) then true else Q)

repeat h

until ~ (while (if (while Q repeat g until false) then true else P)

repeat g until true))

then g else h

We have introduced the while-until as an additional control structure for structured programming. We have demonstrated cases in which use of the while-until results in more readable programs and allows programmers to program closer to the way they think. Although the while-until statement can be defined in terms of the while or the until, we have shown that the while-until expression is a stronger control structure than either.
Fig. 6 Flowchart from Ashcroft and Manna [7].
References


