BOOLEAN-VALUED LOOPS

David S. Wise
Daniel P. Friedman
Stuart C. Shapiro
Mitchell Wand

Computer Science Department
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT NO. 21
BOOLEAN-VALUED LOOPS
DAVID S. WISE
Daniel P. Friedman
Stuart C. Shapiro
Mitchell Wand
REVISED: JUNE, 1975

To appear in BIT.
Boolean-Valued Loops

David S. Wise
Daniel P. Friedman
Stuart C. Shapiro
Mitchell Wand

Abstract

A new control structure construct, the while-until, is introduced as a syntactic combination of the while statement and the repeat-until statement. Examples show that the 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 the classical looping structures. It is shown to be equivalent in power to those structures with exit when a value-returning if-then-else is allowed. As a consequence, there are flowcharts whose implementations require control structures stronger than the while-until. Implementation details are discussed and Hoare-like axioms are presented. A closing discussion on aesthetics discourages some natural generalizations, but it concludes that the basic while-until is convenient for all parties on a programming team: coder, reader, compiler, and validator.

Contents

I. Introduction
II. The value-returning while-until
III. While-until versus while
IV. While-until versus until/exit
V. Implementation and axioms
VI. Multiple terminations and multiple values
VII. Conclusion
I. Introduction

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 control structure construct, the while-until, which often 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 is an iterative control structure with two natural exits, which can be simulated less succinctly by existing control structures. A very natural interpretation of the while-until as a Boolean-valued expression will be shown to be a more powerful control structure than the while or until structures discussed by Dijkstra [5].

The existing constructs by which we are directly motivated are

\[
\text{while } \beta \text{ repeat } s
\]

and

\[
\text{repeat } s \text{ until } \beta
\]

Dijkstra presents these graphically as in Figures 1 and 2.

The syntactic construct we are proposing is

\[
\text{repeat}
\]
\[
\begin{align*}
  & s_1; \\
  & \text{while } \beta_1; \\
  & s_2; \\
  & \text{until } \beta_2; \\
  & s_3; \\
  & \text{taeper.}
\end{align*}
\]
This syntax is a generalization of the authors' earlier proposal [7]. The semicolons are separators which provide for easy expansion of $s_1$, $s_2$, and $s_3$ into sequences of statements.

The essence of the **while-until** at this level is the pair of natural exits from within the body of a loop. It is related to the Bohm and Jacopini structures [3], which might encourage us to provide more exits by allowing more occurrences of "while $\beta_1;$" or "until $\beta_j;$". We postpone these generalizations until Section VI.

To allow comparison between different programs we say that two programs are **equivalent** if, for every input, the programs execute the same (possibly infinite) sequence of elementary operations and tests. Dummy operations (e.g. CONTINUE in FORTRAN) and tests on Boolean literals (e.g. **if** true **then** ...) are not included in the sequence. Two expressions are **value-equivalent** if their evaluation programs are equivalent and they return the same value. A difference between this and other definitions of program equivalence [1,14] is that two infinite loops are not necessarily equivalent. For example, "while true do print(1)" and "while true do print(2)" are not equivalent.

Existing constructs can generate programs equivalent to those which use the **while-until** if we allow some duplication of code and value-returning conditionals. The following remarks are therefore to be distinguished from those based on conditionals as valueless statements [12].

Let true(s) be a function which performs statement $s$ and returns the value true; we define it formally later. Then,

\[
\text{repeat } s_1; \text{ while } \beta_1; s_2; \text{ until } \beta_2; s_3 \text{ taeper}
\]

is equivalent to

\[
\text{if true}(s_1) \land \beta_1 \text{ then repeat } s_2 \text{ until if } \beta_2 \text{ then true else true}(s_3) \land \text{true}(s_1) \land \neg\beta_1
\]
and also to

\[
\text{if } \text{true}(s_1) \& \beta_1 \text{ then}
\begin{align*}
\text{begin } & s_2; \\
\text{while(} & \text{if } \beta_2 \text{ then false else true}(s_3) \& \text{true}(s_1) \& \beta_1) \\
\text{repeat } & s_2 \\
\text{end.}
\end{align*}
\]

If \text{exit} (or \text{escape} or \text{break}) were employed another equivalent form is

\[
\text{repeat}
\begin{align*}
s_1; \\
\text{if } & \neg \beta_1 \text{ then exit;}
\end{align*}
\]

\[
\begin{align*}
s_2; \\
\text{if } & \beta_2 \text{ then exit;}
\end{align*}
\]

\[
\begin{align*}
s_3 \\
\text{taeper.}
\end{align*}
\]

In those cases where Figure 3 is the desired control structure, it appears that the \text{while-untill} yields clearer, more understandable code. In the particular case when \(s_1\) and \(s_3\) are null, the \text{while-untill} yields particularly palatable code:

\[
\text{repeat while } \beta_1; s_2; \text{ until } \beta_2 \text{ taeper}
\]

whose semantic content is easily paraphrased: "While it is possible to try, keep trying until you succeed". If \(s_1\) or \(s_3\) is not null, the \text{while-untill} is an instance of the "n and a half" loop attributed to O. Dahl by Knuth [11].
The \textbf{while} and \textbf{until} loops are simply special cases of the \textbf{while}-\textbf{until}:

\begin{verbatim}
while \beta \ repeat s = def repeat
  while \beta;
  s;
  until false taeper;
repeat s until \beta = def repeat
  while true;
  s;
  until \beta taeper.
\end{verbatim}
It is obvious from the definition of equivalence that the phrases until false and while true are useless, so we shall drop them from now on.

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 a sorted table T[1:N];
low := 0;
high := N+1;
repeat
    while low < high-1;
        try := [(low+high)/2];
    until T[try]=A;
    if T[try] < A then low := try else high := try;
    repeat
```
An appropriate application for the while-until occurs whenever a loop includes an operation which requires a test prior to its execution and a test afterwards to confirm success. An example of this is copying a file up to the end-of-file mark onto an output file with no record being read unless there is enough space for it on the output file.

```plaintext
comment Copy file INPUT into file OUTPUT;
repeat
while Spaceleft(OUTPUT);
    Inbuffer(INPUT,b);
until Endoffile(INPUT);
    Outbuffer(OUTPUT,b);
    taeper.
```
II. The Value-Returning While-Until

In the spirit of languages in which all statements are expressions which return values, such as LISP [14], ALGOL 68 [18], and BLISS [20], we assign a value to be returned by the while-until which is quite natural for the author and audience of a program. The value of a while-until expression is defined to be the value of the last expression evaluated before termination. That is, the value of

\[
\text{repeat } s_1; \text{ while } \beta_1; s_2; \text{ until } \beta_2; s_3 \ \text{taeper}
\]

is false if and only if the loop terminates due to the value of \( \beta_1 \) (see Figure 4). A true value results from the affirmative evaluation of \( \beta_2 \). In this way we have associated Boolean values with the two possible exits, allowing the programmer to test the cause of termination from outside the loop.

This convention allows the while-until to behave like a single-entrance/double-exit loop, and it is this behavior which makes it so powerful. We present examples below in which the while-until is used as a condition following if (or even while). In such cases, the definition of the loop's value becomes relevant only at compile time because the run-time effect ought to be a transfer of control from within the loop directly to one of the two appropriate points in the outer structure.

Zahn [21] has presented a proposal for a single-entry/n-exit control structure which is a generalization of our proposal in the same way that a case statement is a generalization of an if-then-else. Accordingly, his structure is more powerful, but is, of
course, not omnipotent. (The beauty of his proposal lies in his view of the "situation" as a case selector.) Example algorithms which have a loop with three or more distinct actions upon exit, however, seem scarce.

Symes [17] has recently argued for a double-entrance/double-exit loop which would also generalize this embellishment of the while-until. His proposal, however, includes no viable syntax and admits flowcharts which are not even reducible in the sense of Hecht
and Ullman [6], a property which has come to be considered necessary for "structured programs".

Let us return to the previous example of the search routine and convert it to a program for table insertion by using the value of the while-until.

```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 repeat
    try := l(low+high)/2];
while low < high-1;
until T[try] = A;
if T[try] < A then low := try
    else high := try;
taepet then Insertafter (A,T,low).
```

The following deMorgan-like duality shows that there is no loss in generality from the requirement that the while occur first:
Observation 1. The following two expressions are value equivalent:

a) \text{repeat } s_1; \text{ while } \beta_1; s_2; \text{ until } \beta_2; s_3 \text{ taeper}

b) \text{repeat } s_1; \text{ until } \beta_1; s_2; \text{ while } \beta_2; s_3 \text{ taeper}.

Therefore, we are free to use the until test before the while test within the loop. Using Observation 1, we may also eliminate occurrences of "if \text{ taeper}"

A more interesting example is the following program for HEAPSORT as specified by Knuth [10]. This program uses four while-until structures and illustrates the use of the structure as a simple statement, as expressions (which incidentally cannot iterate), and as a nested loop. It is most instructive to relate this code to the discussion by Knuth. The conceptually manageable units which he chooses are isolated very nicely in this code.
comment Heapsort of N Records with Keys;

H1: $\ell := \lceil N/2 \rceil + 1$

$r := N$

H2: repeat

until if $\ell > 1$ then repeat $\ell := \ell - 1$

$R := R_\ell$

$K := K_\ell$

while false
taeper
else repeat $R := R_r$

$K := K_r$

$R_r := R_\ell$

$r := r - 1$

while $r = 1$

$R_\ell := R$

until true
taeper;
H3: \( j := l; \)

H4: \[
\text{repeat} \quad 1 := j; \\
\quad j := 2j; \\
\text{until} \quad j > n; \\
\]

H5: \[
\text{if} \quad j < n \quad \& \quad K_j < K_{j+1} \quad \text{then} \quad j := j + 1; \\
\]

H6: \[
\text{while} \quad K < K_j; \\
\]

H7: \[
R_1 := R_j; \\
\text{taeper;} \\
\]

H8: \[
R_1 := R \\
\text{taeper.} \\
\]
The previous example demonstrates a particularly strong use of the while-until loop: as the Boolean value controlling an outer while-until loop. Whenever the control structure is so nested, either indirectly as in the heapsort program or directly indicated by the phrase "until repeat" (or "while repeat"), termination of the inner loop due to the affirmative evaluation of until $\beta_2$ (respectively false value of while $\beta_1$) will cause the outer loop to be terminated as well, returning the same value. This property generalizes to any level and allows an outer loop to be terminated because of a Boolean expression evaluated within a deeply nested inner loop.

The property just described is an aid to reading programs using the while-until. In writing such programs the outer loop is properly completed on the basis of a specification of the inner loop, and the occurrence of even more deeply nested while-until loops is ignored. For instance, the copy-file example from the previous section may be expanded into a more complete function using multiple output files. Note that the previous code remains intact, but that the property discussed in the previous paragraph does help in reading the code.
Integer function copy(file INPUT, file array OUTPUT, integer n):
begin
    comment Copy file INPUT to OUTPUT[1:n] as needed, returning the number of files used. Return n+1 if the copy is incomplete due to lack of space;
    buffer b;
    copy := 1;
    if repeat
        while copy ≤ n;
            Open(OUTPUT[copy]);
            until repeat
                while Spaceleft(OUTPUT[copy]);
                    Inbuffer(INPUT,b);
                until Endoffile(INPUT);
                Outbuffer(OUTPUT[copy],b)
            taeper
        Close(OUTPUT[copy]);
        copy := copy+1
    taeper then Close(OUTPUT[copy])
end copy.

We admit that this function will appear too concise for many adherents of structured programming. They might feel better if each while-until were specified as a Boolean valued function, as was necessary in previous work with while loops or until loops. Yet this presentation is just as structured and even more readable because of the interpretation of termination values discussed above.
III. While-Until Versus While

In this section we shall show that the value-returning while-until and if-then-else are stronger than the conventional while or until control structures when value returning conditionals are allowed. The added power is due to the ability of a while-until to signal what caused its termination, and for this value to be passed through logical expressions using the if-then-else.

The standard references, notably Bohm and Jacopini [3], do not treat value-returning conditionals although they even exist in ALGOL 60 expressions. We define the value returning if-then-else:

\[
\text{if } \beta_1 \text{ then } \beta_2 \text{ else } \beta_3
\]

as in ALGOL to evaluate \( \beta_1 \) and then one of \( \beta_2 \) or \( \beta_3 \) which becomes the value of the expression. By allowing such a structure we are adding a good deal of power, but not enough to solve the problems in Figure 5 treated below.

Observation 2. The value returning while-until and if-then-else cannot alone be used to simulate one another.

PROOF: We establish this observation by noting that the if-then-else evaluates any of its operands at most once, where a single while-until can perform a number of operations which is unbounded as interpretation is varied. On the other hand, if some while-until expression, \( \varepsilon \), is side-effect and value equivalent to a given if-then-else, then there is an equivalent \( \varepsilon' \) which evaluates with "no wasted effort" in that every evaluated operand is essential to the final value. Then \( \varepsilon' \) consists entirely of expressions of the form
repeat while $\beta_1$; until $\beta_2$ taper

since intervening statements violate "no wasted effort". Furthermore $\beta_2$ must never evaluate to false since that value would cause an improper re-evaluation of $\beta_1$. Hence, if such an $\varepsilon'$ exists its value is that of

repeat while $\beta_1'$; until true taper

which is the value of $\beta_1'$ itself. A trivial induction on length shows that $\varepsilon'$, and hence $\varepsilon$, does not exist.

To stabilize the marriage of if-then-else and while-until we also observe that no other standard Boolean operation is needed. Let us define notation for the ALGOL-like logical connectives which require evaluation of all arguments, and for the LISP-like logical connectives which are non-commutative and evaluate the second argument only if necessary. Unary negation is denoted by $\neg$.

<table>
<thead>
<tr>
<th>Binary Operation</th>
<th>ALGOL-like</th>
<th>LISP-like</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conjunction</td>
<td>$\land$</td>
<td>$&amp;$</td>
</tr>
<tr>
<td>Disjunction</td>
<td>$\lor$</td>
<td>$|$</td>
</tr>
<tr>
<td>Implication</td>
<td>$\Rightarrow$</td>
<td>$\Rightarrow$</td>
</tr>
</tbody>
</table>

Observation 3. The value returning if-then-else has more expressive power than the logical connectives.

A test using any of the connectives is value-equivalent to one using only the if-then-else operator. For instance,

$\alpha \land \beta = \text{if } \alpha \text{ then } \beta \text{ else } \beta \text{ then false else false;}

\alpha \& \beta = \text{if } \alpha \text{ then } \beta \text{ else false.}$
The rest are as easily formulated. If we try to use the standard logical connectives to simulate if α then β else γ with respect to value and side effects, we always find a case in which one of α, β, or γ is evaluated twice.

In that attempt and in consideration of the examples below, it is important to recall that equivalent code requires evaluation of expressions in precisely the same order as in the simulated program. Moreover, using extra variables to remember values and thereby to avoid a re-evaluation is not a legal operation. (LISP calls it a SETQ.) Were it allowed, as Cooper [4,11] points out, the introduction of only one memory variable (call it a program-counter) into any given program would allow its simulation by a simple while loop on a case statement. Introducing new variables, therefore, makes existing control structures omnipotent and absolutely denies their importance to the vague issue of structured programming.

As a consequence of Observations 2 and 3, we conclude that value-returning while-until and if-then-else, taken together, are the only control structures necessary for a very large class of programs. We certify their marriage with a grammar for the while-until language in which simple statements are treated as expressions:

\[
G_1 \text{ Exp } ::= \text{ if Exp then Exp else Exp |}
\]
\[
\text{repeat (Exp;)* while Exp; (Exp;)* until Exp (;Exp)* taeper|}
\]
\[
\text{repeat (Exp;)* until Exp; (Exp;)* while Exp (;Exp)* taeper|}
\]
\[
\text{elementary expression | elementary statement.}
\]

From \( G_1 \) we can define notations for special expressions which always return a constant value. We use logical operators for the
appropriate if-then-else expression according to Observation 3.

\[
\begin{align*}
\text{true}(e) &= \text{def } \text{if } e \text{ then true else true.} \\
\text{false}(e) &= \text{def } \neg\text{true}(e).
\end{align*}
\]

Sequences of expressions can then be reduced to a single constant valued expression:

\[
\text{true}(e_1; e_2; \ldots; e_n) = \text{def} \\
\quad \text{true}(e_1) \& \text{true}(e_2) \& \ldots \& \text{true}(e_n)
\]

with false(...sequence...) defined similarly. This notation and Observation 1 lead us to a much more simplified grammar equivalent to \( G_1 \).

\[
G_2 \\
\text{Exp ::= if Exp then Exp else Exp |} \\
\quad \text{repeat while Exp until Exp taeper |} \\
\quad \text{elementary statement |} \\
\quad \text{elementary expression}
\]

**Lemma 1.** Any expression in the language of \( G_1 \), where elementary expressions can include expressions on the logical connectives and parenthesizations, is value-equivalent to one in the language of \( G_2 \) and conversely.

**Proof:** The logical connectives become conditionals according to Observation 3. We need only add that the sequence can be subsumed into the termination tests:
\[
\text{repeat} \quad s_1; s_2; \ldots; s_l; \quad \text{while} \quad \text{true}(s_1; s_2; \ldots; s_l)
\]
\[
\text{while} \quad \beta_1; \quad \text{while} \quad \text{true}(s_1; s_2; \ldots; s_l) \quad \& \quad \beta_1 \& \quad \text{true}(s_{l+1}; \ldots; s_j)
\]
\[
\text{until} \quad \beta_2; \quad \text{until} \quad \beta_2 \# \quad \text{false}(s_{j+1}; \ldots; s_k)
\]
\[
\text{taeper} \quad \text{taeper.}
\]

The converse is trivial. ■

With this notation we are prepared for the first main result:

Theorem 1. There exist flowcharts not implementable with while loops which can be implemented with the language of \(C_2\).
PROOF: We give an example in the language of $G_2$ from Ashcroft and Manna [1] whose flowchart appears as Figure 5. Their proof shows that this example cannot be translated into while loops without adding extra variables because of arbitrary inner looping within the control of an outer loop. Because the if-then-else cannot effect such loops, it is clear that our solution is due to the power of the while-until rather than to any added power of if-then-else in returning a value.
The program is

\begin{verbatim}
if repeat
    repeat while P; h taeper;
    while Q;
    h;
    repeat while Q; g taeper;
until \textit{upP};
\end{verbatim}
\textit{up}
\begin{verbatim}
    g
    taeper then g else h;
\end{verbatim}
\textit{x}.

Another problem from Peterson, Kasami, and Tokura [16] is solved with the \textit{while-until} in our earlier paper [7]. We can apply Lemma 1 to such solutions to eliminate all logicals, parentheses, and semi-colons, but at the expense of clarity.
IV. While-Until Versus Until/Exit

There are several existing notations for providing alternative exits from loops. Bochmann [2] and Evans [6] mention two, and we have discussed some already. From a theoretical point of view, the best understood [12] is a while or until structure used in conjunction with an exit or escape statement. We select the until/exit structure for analysis and define its language with the grammar $G_3$.

$$G_3 :\begin{align*}
E &::= \text{repeat (S;)* S until E} \mid \\
&\quad \text{if E then E else E} \mid \\
&\quad \text{elementary expression}
\end{align*}

S :::= E \mid \text{CS} \mid \text{exit} \mid \text{elementary statement}

CS :::= \text{if E then (S;)* S else (S;)* S fi} \mid \\
&\quad \text{if E then (S;)* S fi}
$$

Note that $G_3$ provides both value-returning conditionals which must have an else and conditional statements bracketed by if-fi.

The iterative structure of $G_3$ is considered a Boolean expression whose value indicates whether control left the loop because of execution of an exit statement or via "normal" termination. For concreteness, let us say that an until/exit takes the value true on normal termination and false on execution of an exit. The next result establishes such constructions as equivalent to the while-unti. A similar result holds for while/exit structures.

**Theorem 2.** To every until/exit statement (expression) from $G_3$ there is an equivalent (value-equivalent) while-unti expression in the language of $G_1$ and vice-versa.
PROOF: The while-until is expressible in terms of the until/exit via Lemma 1 and the equivalence

\[
\text{repeat while } \beta_1; \text{ until } \beta_2 \text{ taesper} = \text{repeat if } \beta_1 \text{ then exit } f_1 \text{ until } \beta_2.
\]

For the converse, let us define two functions, \(\phi\) and \(\phi'\). \(\phi\) takes a string derivable from \(E\) in \(G_3\) and converts it into an equivalent while-until expression derivable from \(G_1\). \(\phi'\) takes a string derivable from \(S\) in \(G_3\) and converts it to an expression derivable from \(G_1\) which will evaluate to false if and only if \(w\) terminated via an exit. Sequences of statements are then simulated by a conjunction of their \(\phi'\) images by the \& operator giving an expression with identical requirements on its value. Since \(G_3\) is unambiguous, we may follow it:

\[
\phi(\text{repeat } s_1; \ldots ; s_n \text{ until } \beta) = \text{repeat while } \phi'(s_1) \& \phi'(s_2) \& \ldots \& \phi'(s_n); \text{ until } \phi(\beta) \text{ taesper}.
\]

\[
\phi(\text{if } \alpha \text{ then } \beta \text{ else } \gamma) = \text{if } \phi(\alpha) \text{ then } \phi(\beta) \text{ else } \phi(\gamma).
\]

\[
\phi(\text{elementary expression}) = \text{elementary expression}.
\]

\[
\phi'(\text{Expression}) = \text{if } \phi(\text{Expression}) \text{ then true else true}.
\]

\[
\phi'(\text{if } \beta \text{ then } s_1; \ldots ; s_n \text{ else } t_1; \ldots ; t_m \text{ f1}) = \text{if } \phi(\beta) \text{ then } \phi'(s_1) \& \ldots \& \phi'(s_n) \& \text{else } \phi'(t_1) \& \ldots \& \phi'(t_m).
\]

\[
\phi'(\text{if } \beta \text{ then } s_1; \ldots ; s_n \text{ f1}) = \text{if } \phi(\beta) \text{ then } \phi'(s_1) \& \ldots \& \phi'(s_n) \text{ else true}.
\]

\[
\phi'(\text{exit}) = \text{false}.
\]

\[
\phi'(\text{elementary statement}) = \text{true(elementary statement)}.
\]

The correctness of the construction is a trivial induction on length, using the sequential operation of ";"; in the repeat expres-
sion, $s_1$, $s_2$, etc. are executed in turn until some $f'(s_i)$ turns false, which happens when and only when an exit occurs.

A `leave` statement within a nested loop has been defined in BLISS [11] so as to be able to terminate outer loops. Let us define the statement `exit-1` where 1 indicates the number of loops to be terminated. The simple exit discussed above becomes `exit-1` when
repeat/exit structures are used purely as statements (rather than as expressions which might control an outer loop).

**Theorem 3.** (A weakening of Theorem 6 of [12]) For every $n \geq 1$ there exists a program using the language of $G_3$ with exit-$i$ for $i \leq n$ which cannot be simulated by the same language when $i < n$.

The while-until falls at the same position in Kosaraju's hierarchy of control structures [12] as the until/exit.

**Corollary.** There exists a flowchart for which the language of $G_3$ is insufficient.
Independently of Kosaraju [12] we arrived at Figure 6 to prove the corollary. (The program may be started at any edge with the same result because of symmetry.) Figure 7 illustrates more simply the basic limitation on a while-until. If we view a loop from outside (top-down), only one of two Boolean values can be returned in spite of side effects. If one of three distinct values is required, as in Figure 7b, the while-until is insufficient.
V. Implementation and Axioms

Since we have approached the while-until from a programmer's point of view, issues of compilation are secondary. Yet the elegance of the while-until extends to this topic, and the observations are straightforward. The value definition requires no extra computation at the transfer of control from the loop, since the value of the last Boolean evaluated is in some register at that time. It is not difficult to optimize that final jump to exit several levels directly. We have built a compiler which restructures the loop and optimizes such jumps. Indeed, the jump pattern of Knuth's hand-polished MIX code for heapsort [10] is compilable. (Optimization on register allocation and predefined input criteria would naturally follow this jump optimization.)

In the appendix we present a LISP definition of a function to interpret the while-until, which we have found useful in learning its ways. We invite the reader to tinker.

Another task is required in the effort to integrate the while-until into the programming culture. The wide acceptance of Hoare's axiomatic approach to programming [9] obliges us to construct axioms for the while-until. We present axioms for these two structures in Table 8. Following Manna [13], we introduce a new Boolean-valued register, val, to retain the value of the last (Boolean) expression evaluated. Note the simplicity of the consequent for the while-until. Total correctness rules [13] are also available.
VI. Multiple Terminations and Multiple Values

In this section we observe the natural extensions to the \texttt{while-until} which satisfy demands for returning multiple values and allowing multiple termination conditions. We argue that these features should be avoided because they muddle the simple elegance of the loop structure and offer power in excess of the spirit of structured programming.

At the first introduction of the \texttt{while-until} above we suppressed multiple occurrences of the phrases "\texttt{while }\beta_i;\texttt{ }" or "\texttt{until }\beta_j;\texttt{ }". If we now allow them, but restrict that all \	exttt{while}'s precede all \	exttt{until}'s we get a structure of the form:

\[
\begin{align*}
\text{repeat} & \quad ((s;)*(\texttt{while }\beta;)*)* \\
\texttt{while }\beta_1; & \quad ((s;)*(\texttt{until }\beta;)*)* \\
\texttt{until }\beta_2; & \quad (s)* \\
\text{tepep}. & \\
\end{align*}
\]

A construction technique similar to that used in proving Lemma 1 establishes that this control structure is equivalent to the \texttt{while-until}. Therefore (due more to the value returning \texttt{if-then-else} than the \texttt{while-until}) Jacopini's $\Omega_n$ charts [3] are indeed expressible using the \texttt{while-until}. If only side-effect equivalent code is needed, the \texttt{while} and \texttt{until} phrases may be mixed freely since "\texttt{while }\beta;" is equivalent to "\texttt{until }\beta;" under that requirement. If value-equivalence is needed, however, mixing \texttt{while}'s and \texttt{until}'s muddies the picture: while Observation 1 still holds, the proof
of Theorem 2 fails. We are dealing with a new animal, but one so subtly different that its merit is doubtful.

It is surprisingly easy to provide for one of many (rather than just two) values to be returned from a while-until. For example, BLISS [20] allows many distinct values to be interpreted as true or false when taken as Booleans. Under such a scheme, the satisfaction of an until (or failure of a while) could terminate the loop with one of several true (false) values which could be distinguished in the outer context. Even LISP would allow many non-false values to be returned on termination at an until line.

But Observation 1 will still hold only if the set of values is partitioned into two sets—true and false—and the negation map has the property that $\neg \neg x = x$. Then the strength of the while-until is enhanced tremendously and the corollary is invalidated since we can return more than two values. For instance, if false is canonically the number zero then the value of $j$ in

$$\text{repeat until } l+\text{repeat until } l+\text{repeat until } j \ldots$$

can effect an exit-$i$ for $0 \leq i \leq 3$ by setting $j = -i$.

Such enhancements to the value-returning power of the while-until run counter to the aesthetics which motivate this paper. When the while-until is restricted to its original form it is obvious from its value what was the last line executed within the loop; even if the control structure is nested, the flow is easy to follow. The restriction makes a program easier to read, easier to compile optimally, easier to prove, and easier to modify. It is consistent with other constraints imposed on programmers which
restrict his repertoire but improve his product. It is notable that this property is not even shared by the value returning if-then-else, accounting for the lack of popularity that control structure has endured. (See the consequents of Table 8.) Provisions for multiple termination points or multiple non-\texttt{false} values returned through \texttt{while} or \texttt{until} lines destroy this elegance. We do not advocate them and leave their formal study to others.
VII. Conclusion

It is not easy to expect a new control structure for simple loops to be willingly adopted after twenty years of programming language development. Yet the **while-until** offers more than raw expressive power. Its termination conditions and values are obvious and it becomes very easy for the coder to adopt. Anyone who reads code can readily follow its flow of control because of its linear internal structure and because of the unique origin of each termination value. The compiler writer is asked to provide translation and optimization for an expression which is similar to control structures already extant and which involves readily available values. If the program must be validated or "proved" the same properties admit an inductive analysis.

While this new control structure is not omnipotent, it does extend the repertoire of easy-to-use programming structures. Although it is side-effect equivalent to **repeat/exit** structures (and also value equivalent when such statements are taken as expressions), we feel that any value assignment to the **repeat/exit** cannot be as transparent as the value assignment to the **while-until**. Moreover, multiple **exit** commands within a single loop amount to sins on the level of GO TO abuses. Restrictions on number and position of termination tests are very natural in the **while-until**.

The **while-until** expression allows top-down programmers, as well as bottom-up programmers, to code the way they think. It is much more palatable than other control structures of similar strength. Finally, it has proved itself as a communicable control structure in classes at Indiana University to the extent that students com-
plain about its absence when introduced to the same block-structured languages which originally motivated its creation.

This neat, nested factorization of a program serves admirably well to keep the individual building blocks intellectually manageable, to explain the program to an audience and to oneself, to raise the level of confidence in the program, and to conduct informal, and even formal proofs of correctness [19]

and to whet us for the discovery of new algorithms by polishing and compacting our understanding of those already known.
Appendix

The following two functions define a LISP-based interpreter for the while-until. The use of ILLEGAL is optional.

(DE REPEAT (BODI A) (PROG (REST)
  (COND ((ILLEGAL BODI) (ERROR)) ))
TOP (SETQ REST BODI)
NEXT (COND
  ((EQUAL (CAR REST) (QUOTE UNTIL)) (COND
    ((EVAL (CADR REST) A) (RETURN T))
    (T (SETQ REST (CDR REST))) ) )
  ((EQUAL (CAR REST) (QUOTE WHILE)) (COND
    ((EVAL (CADR REST) A) (SETQ REST (CDR REST)))
    (T (RETURN NIL)) ) )
  (COND
    ((SETQ REST (CDR REST)) (GO NEXT))
    (T (GO TOP)) ) ))

(DE ILLEGAL (BODI) (PROG (WSEEN USEEN)
  (COND ((NULL BODI) (RETURN T)))))
NEXT (COND
  ((EQUAL (CAR BODI) (QUOTE WHILE)) (COND
    (WSEEN (RETURN T))
    ((NULL (CDR BODI)) (RETURN T))
    ((SETQ WSEEN T) (SETQ BODI (CDR BODI))) ) )
  ((EQUAL (CAR BODI) (QUOTE UNTIL)) (COND
    (USEEN (RETURN T))
    ((NULL (CDR BODI)) (RETURN T))
    (SETQ USEEN T) (SETQ BODI
      (CDR BODI))) ) )
  (COND ((SETQ BODI (CDR BODI)) (GO NEXT))
    (T (RETURN (OR (NOT WSEEN) (NOT USEEN)))) ) ))

Note that REPEAT is a fexpr and ILLEGAL is an expr.

An example of the use of while-until follows:

(DE MEMBER (A L) (REPEAT
  WHILE L
  UNTIL (EQUAL A (CAR L))
    (SETQ L (CDR L)) ))
References


2. Bochmann, G.V. Multiple exits from a loop without the GOTO. *Comm. ACM* 16, 7 (July, 1973), 443-444.


Figure 1. while $\beta$ repeat $s$

Figure 2. repeat $s$ until $\beta$
Figure 3. repeat $s_1$; while $\beta_1$; $s_2$; until $\beta_2$; $s_3$ taaper
Figure 4. Value returning while-until illustrated as a standard flowchart and a proposal for structured flowcharts [15]
Figure 5. Flowchart from Ashcroft and Manna [1]
Figure 6. A flowchart beyond the power of the while-until
Figure 7a.

Figure 7b. Loops which result from node-splitting Figure 6, indicating the values to be returned.
{P} \ s_1(Q) \text{ and } \{R\} \ s_2(S) \text{ and } \{T\} \ s_3(P) \\
and \{Q\} \ \beta_1(\text{val} \land R \lor \neg\text{val} \land A) \\
and \{S\} \ \beta_2(\text{val} \land B \lor \neg\text{val} \land T) \\

{P} \text{ repeat } s_1; \text{ while } \beta_1; \ s_2; \text{ until } \beta_2; \ s_3 \text{ taeper} \\
{\text{val} \land B \lor \neg\text{val} \land A}
\{P\} \beta_1 \{\text{val} \land Q \lor \neg \text{val} \land R\}
and \{Q\} \beta_2 \{\text{val} \land S \lor \neg \text{val} \land T\}
and \{R\} \beta_3 \{\text{val} \land U \lor \neg \text{val} \land V\}

\{P\} \begin{cases} 
\text{if } \beta_1 \text{ then } \beta_2 \text{ else } \beta_3 \\
\{\text{val} \land (S \lor U) \lor \neg \text{val} \land (T \lor V)\}
\end{cases}

Table 8. Axioms for Boolean valued while-until and if-then-else. The Boolean variable val appears only where explicitly mentioned.