

THE STRUCTURE THEOREM

CS 340, FALL 2004

PURPOSE

The basic idea behind structured programming is to reduce the number of entities to be considered, and thus the number of interactions.¹ Of particular interest is the question “can all programs be written in a structured manner?” This question is answered in the affirmative by the structure theorem. This document presents the structure theorem discussed in class, along with an example of the application of its constructive proof.

CONTENTS

Purpose	1
1. Definitions	1
2. Prime Programs	3
3. Structured Programs	4
4. Example	6

1. DEFINITIONS

Definition 1.1. Let P be a program. For every input x for which P halts, an output y is determined. The set of all such pairs $\langle x, y \rangle$ is a function called the *program function* of P and denoted $[P]$.

In definition 1.1 the input should include initial state and all inputs applied during the execution of P . Likewise, output should be taken to include the final state and any output generated during the execution of P .

Definition 1.2. Two programs P and Q are *execution equivalent* if and only if they perform the same computations in the same sequence.

Example 1.3. The following two C snippets are execution equivalent:

<pre>f(); while(p) { f(); }</pre>	\equiv	<pre>do { f(); } while(p);</pre>
---	----------	--

Definition 1.4. Two programs P and Q are *function equivalent* if and only if they have the same program function ($[P] = [Q]$).

Date: 30th September 2004.

¹The approach is similar to the design patterns work being done today. Design patterns represent a disciplined use of object-oriented constructs. See *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, et. al.

Example 1.5. The following two C snippets are not execution equivalent (they evaluate the predicates in the opposite order), but they are function equivalent:

```

if (p) {
    if (q) {
        f();
    }
}
≡
if (q) {
    if (p) {
        f();
    }
}

```

Example 1.6. Consider the following C code snippet for integers x and y :

```

x = y+1;
y = x-1;

```

The input to this program is the initial state $\langle x, y \rangle$. The output is the transformed state $\langle y+1, y \rangle$. There are a number of ways to write this function. For example, one can write the set of ordered pairs:

$$\{ \langle \langle x, y \rangle, \langle y+1, y \rangle \rangle \mid x, y \in \mathbb{Z} \}.$$

This is just an approximation, since not everything in \mathbb{Z} has a computer representation, but it gets the point across. Another way of writing this is to show the mapping:

$$\langle x, y \rangle \mapsto \langle y+1, y \rangle.$$

This notation is simpler, and it expresses the function of the code.


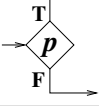

According to referential transparency, we should be able to replace the code with any other code computing the same function. For example, the following code has the same program function, and is thus functionally equivalent to the previous code snippet:

```

x = y+1;

```

Notation 1.7. Flowcharts in this document will use the following simple notation.

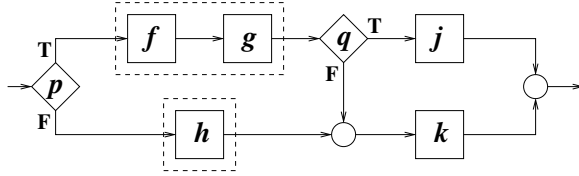
Notation	Meaning	Notes
	Function Node	Labeled with program function.
	Predicate Node	No side effects.
	Collector Node	

Function nodes perform some program function, mapping an initial state to a transformed state. Predicate nodes take an input state and compute some condition on the state, but do not make any changes to the state or write any output (no side effects). Control then flows in one direction if the predicate is true, and in the other if the predicate is false. Collector nodes simply collect two flows into one.

Definition 1.8. A *proper program* is a program with a single point of entry, a single exit, and, for each node, a path from entry to exit through the node.

Definition 1.9. A *proper subprogram* is a proper program embedded in another program.

Example 1.10. The following is a simple proper program. The dashed boxes pick out some proper subprograms.



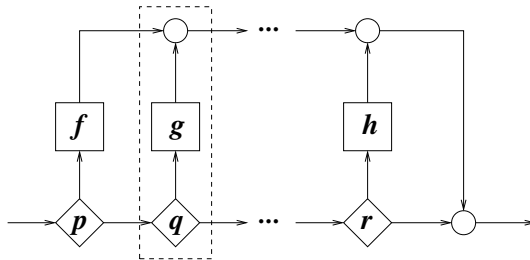
Note that there is no proper subprogram containing q other than the program itself, because either one would have to cut two entry lines, or two exit lines.

2. PRIME PROGRAMS

Definition 2.1. A *prime program* is a proper program which contains no proper subprogram of more than one node.

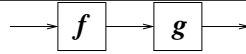
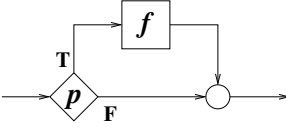
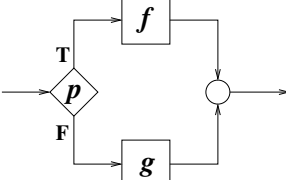
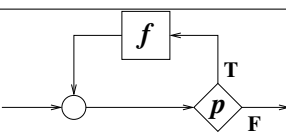
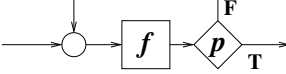
Theorem 2.2. *There is an infinite number of prime programs.*

Proof. A countably infinite number of prime programs can be exhibited as follows.



The dashed section can be repeated any number of times. This structure is essentially a case statement, and any number of cases can be present. Note that it is impossible to pick out a proper subprogram of more than one node. \square

Example 2.3. The following are several well-known prime programs.

Structure	Name	C
	sequence	$f(); g();$
	if-then	$\text{if } (p) \{ f(); \}$
	if-then-else	$\text{if } (p) \{ f(); \} \text{ else } \{ g(); \}$
	while-do	$\text{while } (p) \{ f(); \}$
	do-until	$\text{do } \{ f(); \} \text{ while } (!p);$

Axiom 2.4. [of Replacement] Any function node in a proper program may be replaced with a prime subprogram which is functionally equivalent to the original node, and the new proper program will be functionally equivalent to the original program.

The Axiom of Replacement is a consequence of referential transparency and the fact that predicate nodes do not modify the state space or generate any output.

The Axiom of Replacement allows one proper program to be built from another, iteratively. One can start with a prime program, and then iteratively replace function nodes with other prime programs. The result will be a proper program built from the set of primes used.

Definition 2.5. The set of primes used in constructing a program P is the *basis set* of P .

3. STRUCTURED PROGRAMS

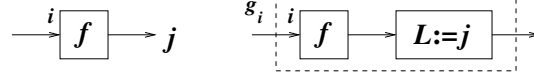
Definition 3.1. A program constructed from a fixed basis set of primes is a *structured program*.

Clearly any proper program can be built from primes, but is it possible to restrict the basis set to some fixed set in all cases? That is, can any proper program be built from some (small) set of primes? This question is answered in the affirmative by the structure theorem.

Theorem 3.2. [Structure Theorem] Any proper program is function equivalent to a structured program with basis set {sequence, if-then-else, and while-do}, composed of function and predicate nodes from the original program, and modifications to and tests on one additional counter, L , not present in the original program.

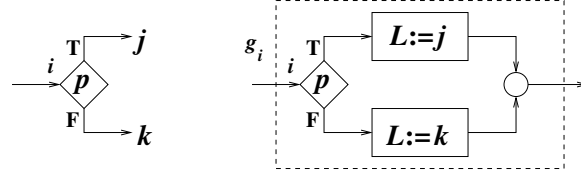
Proof. The proof of the structure theorem is constructive; it explains how to construct a new program from the required basis set such that the resulting program is functionally equivalent to the original program.

- (1) Number the function and predicate nodes sequentially. Number the first node reached on the entry line 1. Number the exit line 0. Number the remaining nodes 2, 3, ..., n .
- (2) For each function node i , create sequence prime g_i :



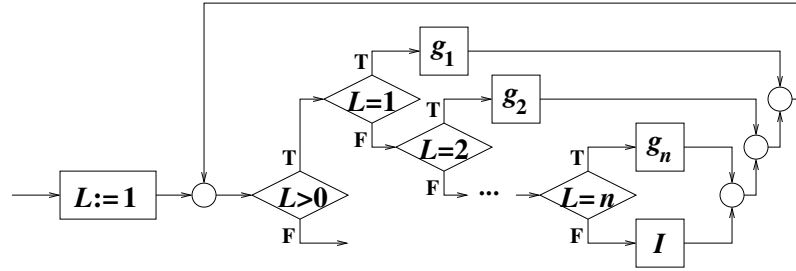
After executing the function node f , control passes on to node j or the exit line $j = 0$. This is replaced by setting the counter L to j .

- (3) For each predicate node i , create if-then-else prime g_i :



The predicate p is computed. If true, then control passes to node j or the exit line $j = 0$. This is replaced by setting the counter L to j . If false, then control passes to node k or the exit line $k = 0$. This is replaced by setting the counter L to k .

- (4) Combine the primes g_i into an initialized, while-do program. This is shown below:



The new program consists of a cascade of if-then-else primes embedded in a while-do. The entire assembly is a sequence. Thus the new program is built from the basis set {sequence, if-then-else, while-do}, as required.

It remains to show that the new program computes the same function as the original program. Since L is not in the original program, this can be done by showing that the new program executes the function and predicate nodes of the original program in the same order as the original program. This can be done inductively.

The original program starts by executing node one. The new program first sets L to one and then checks to see if L is greater than zero. Since it is, the loop body is executed and the test causes g_1 to be executed. This executes the original function or predicate node one from the original program, then sets L to the appropriate next node or exit line 0.

Assume that the new program executes the first k nodes (for some $k \geq 1$) of the original program in the correct order, and has set L to the appropriate next node or exit line 0. If $L = 0$, then the program terminates. If $L \neq 0$ then it must be the case that $L > 0$, and the loop body is run. This executes function g_L , which performs node L of the original program, and then sets L to the appropriate next node. Thus the new program executes the first $k + 1$ steps correctly.

The new program thus executes all $k > 1$ steps correctly or terminates, as appropriate, and computes the same program function as the original program. The two programs are function equivalent.

□

4. EXAMPLE

Consider the following C source code.

```

while(1) {
    advance_marker();
HERE:
    a=read();
    if (a==INT_ON_READ) {
        goto HERE;
    }
    if (a==EOF) {
        break;
    }
    if (a&0x01) {
        goto LOW_SET;
    }
HIGH_SET:
    a=a&0xFE;
LOW_SET:
    rc=send_with_parity(a);
    if (rc < 0) {
        pause(3000);
        goto HERE;
    } else {
        clear_send_buffer();
    }
}

```

To structure this code, first number the nodes as required by the structure theorem. The above code fragment can be expressed as the flowchart in fig. 4.1, with the first node numbered one, the exit line numbered zero, and the other function and predicate nodes numbered arbitrarily.

Each of the function and predicate nodes is used to construct a new sequence or if-then-else prime program. These are shown in fig. 4.2.

One can next make a list of the values of L which are only referenced once: 2, 4, 5, 6, 7, 9, and 10. The corresponding primes can be substituted for their references, giving the three structured constructs in fig. 4.3.

With this new set of structured constructs $L:=8$ only appears once, and it can be replaced with the corresponding structured construct, leading to the further compressed structures in fig. 4.4. Since the predicates do not have side effects and are disjoint rules, their evaluation is order-independent, and the first two predicates have been swapped so the “exit” rule is checked first. Further, the `advance_marker()` function node has been duplicated to eliminate the reference to $L:=1$.

The large construct for $L:=3$ can be expressed as a loop. Duplicating the `a=read()` node allows elimination of the counter L , giving the final, restructured flowchart in fig. 4.5. In order to re-write as a while loop, the sense of the predicate `a==EOF` has been reversed to `a!=EOF`.

C source code can be read directly from the structured flowchart.

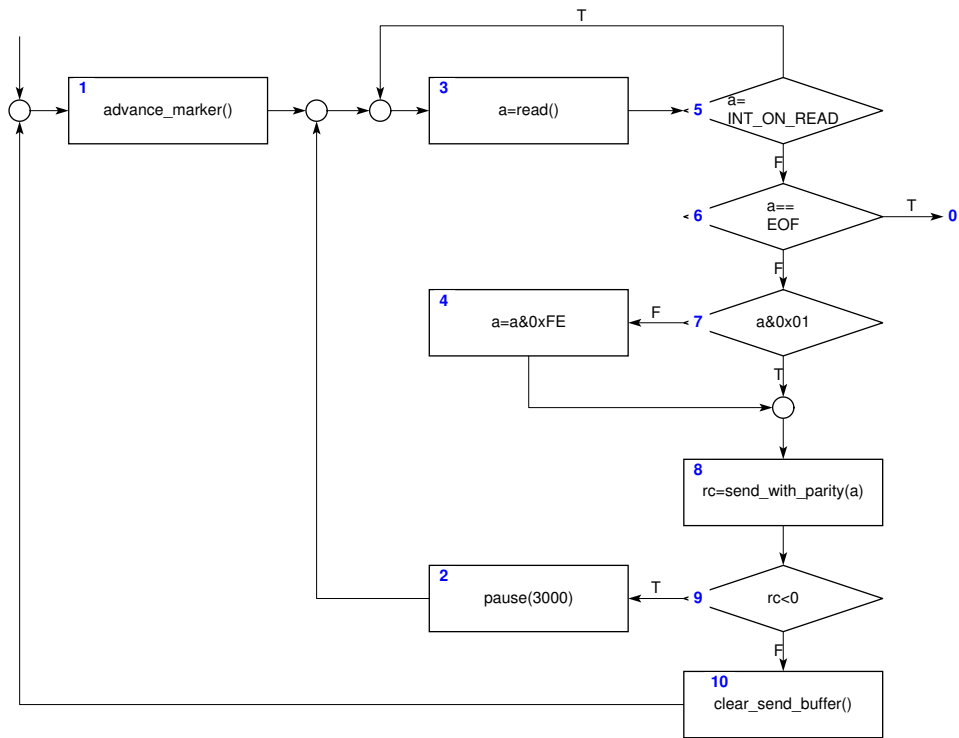


FIGURE 4.1. The unstructured flowchart

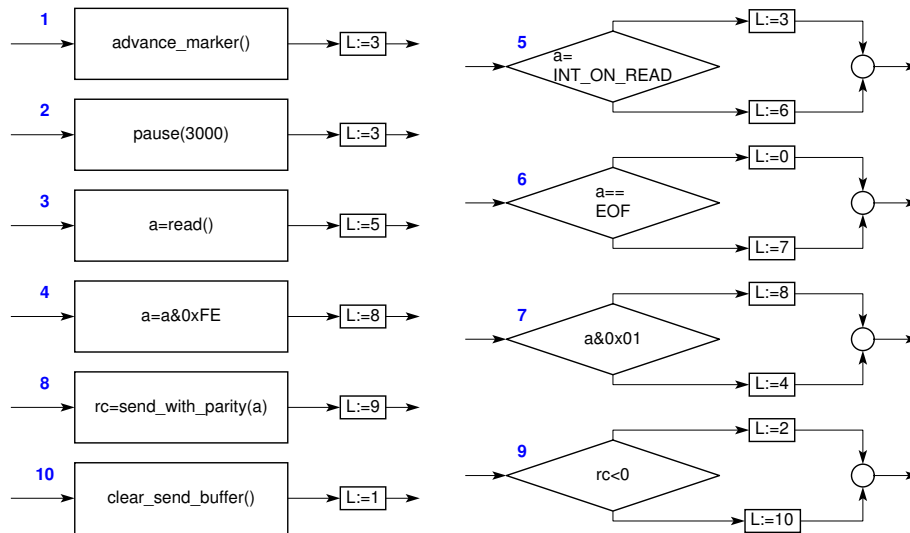


FIGURE 4.2. Primes for program

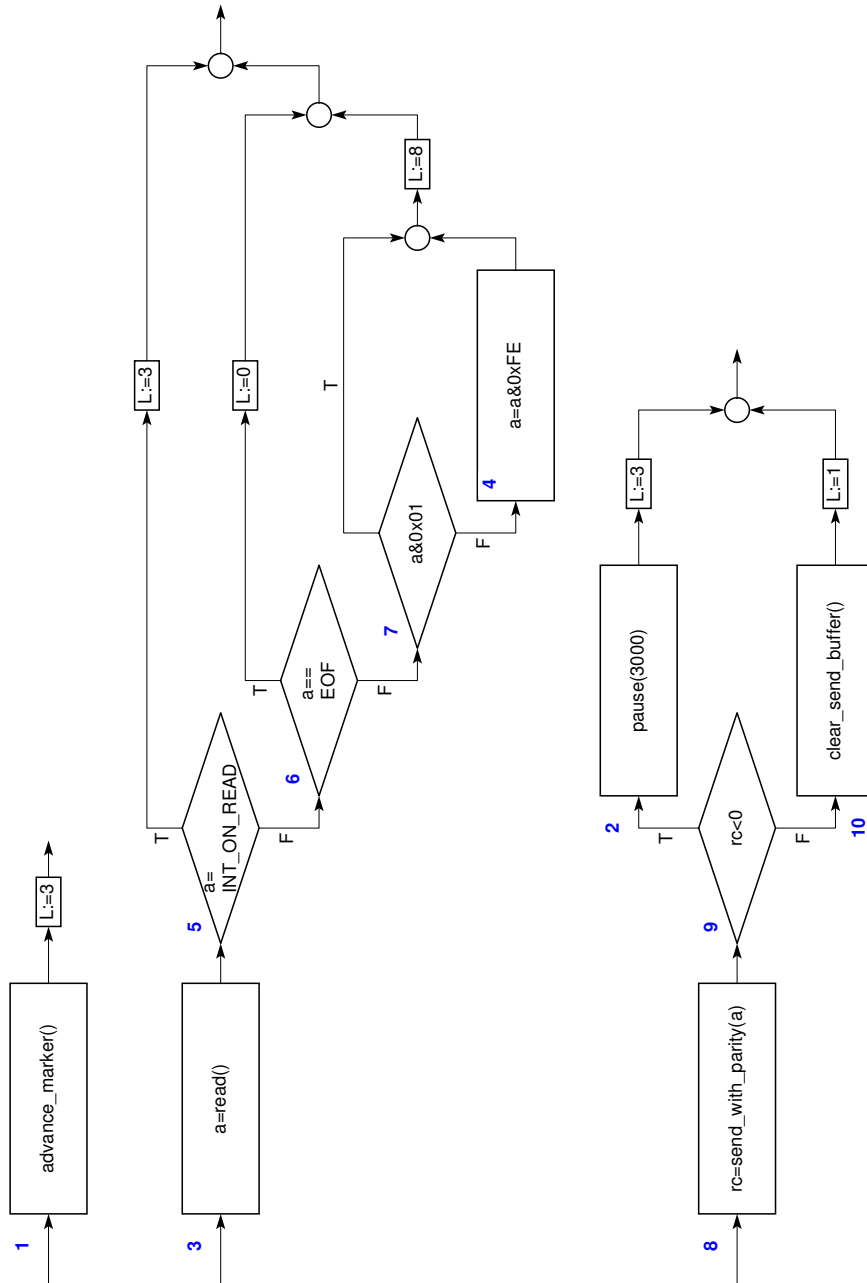


FIGURE 4.3. Structured constructs

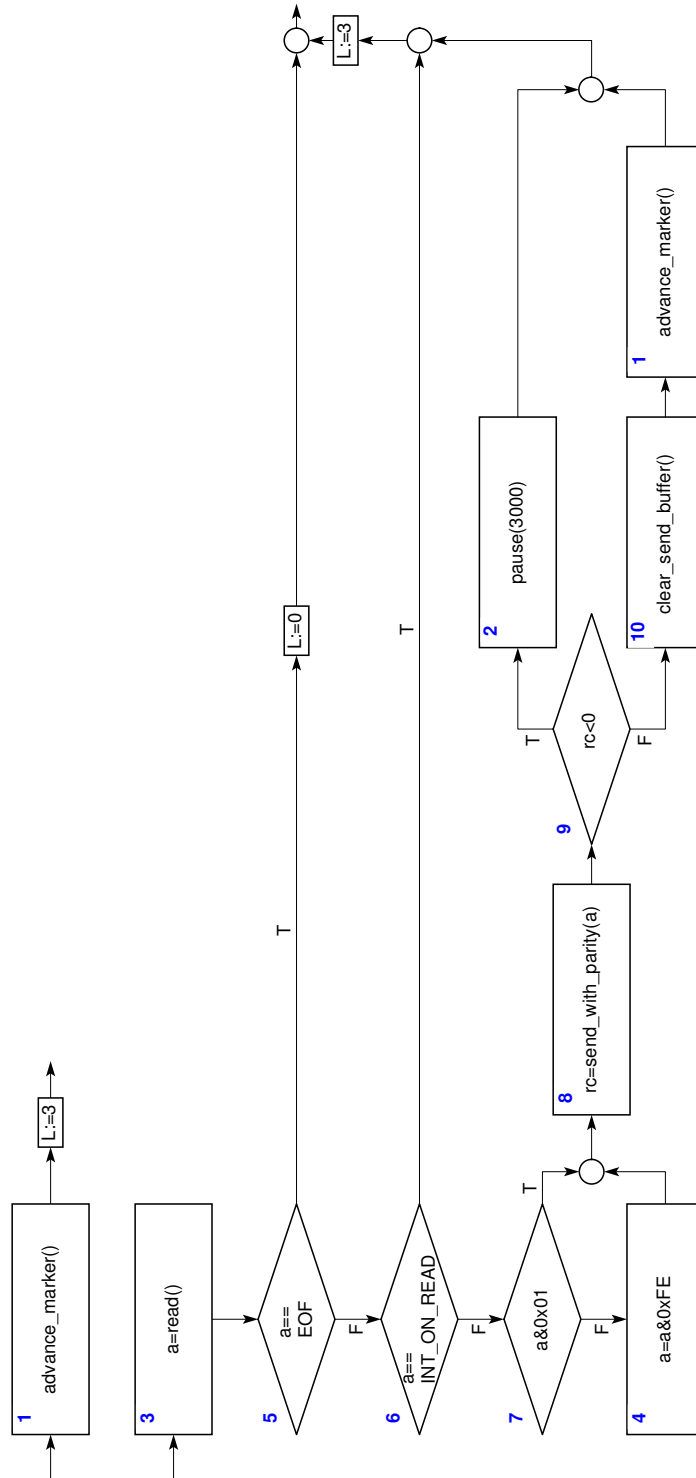


FIGURE 4.4. Compressed structured constructs

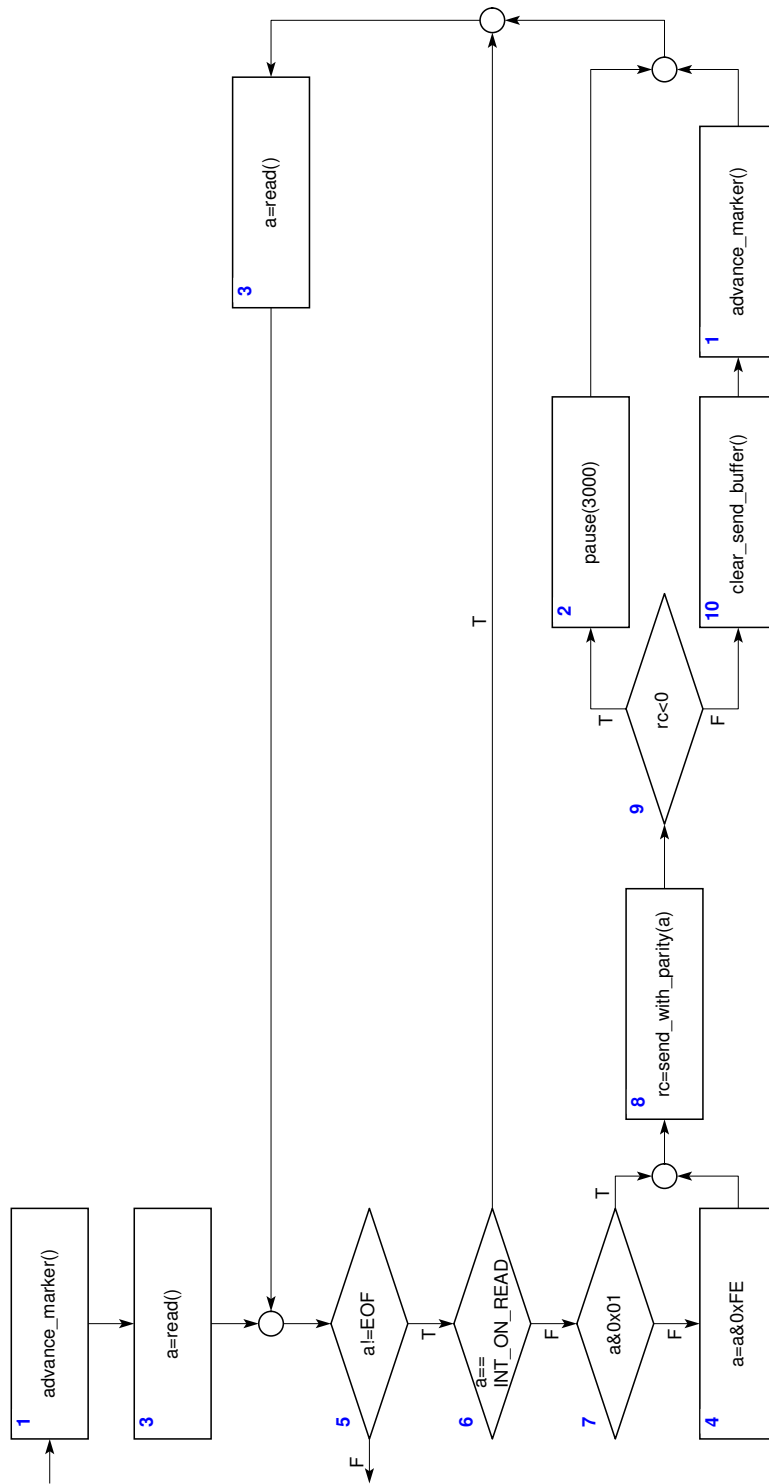


FIGURE 4.5. Restructured flowchart

```
advance_marker();
a = read();
while (a != EOF) {
    if (a != INT_ON_READ) {
        if (!(a&0x01)) {
            a = a&0xFE;
        }
        rc=send_with_parity();
        if (rc < 0) {
            pause(3000);
        } else {
            clear_send_buffer();
            advance_marker();
        }
    }
    a = read();
}
```

The very observant will have suspected that the code contains an error; it does. The test `a&0x01` clearly checks that the low-order bit is set. The intent is to clear it by `a=a&0xFE`, but the test is incorrect. This led to a bug in the original code which was quite hard to diagnose due to its unstructured nature.