

RIFL: A Language with Filtered Iterators

by

Jiasi Shen

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

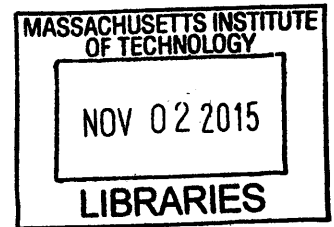
Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

ARCHIVES



© Massachusetts Institute of Technology 2015. All rights reserved.

Signature redacted

Author

Department of Electrical Engineering and Computer Science

August 24, 2015

Signature redacted

Certified by

Martin C. Rinard

Professor of Computer Science

Thesis Supervisor

Signature redacted

Accepted by

1 UU

Leslie A. Kolodziejcki

Professor of Electrical Engineering

Chair, Department Committee on Graduate Students

RIFL: A Language with Filtered Iterators

by

Jiasi Shen

Submitted to the Department of Electrical Engineering and Computer Science
on August 24, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

RIFL is a new programming language that enables developers to write only common-case code to robustly process structured inputs. RIFL eliminates the need to manually handle errors with a new control structure, filtered iterators. A filtered iterator treats inputs as collections of input units, iterates over the units, uses the program itself to filter out unanticipated units, and atomically updates program state for each unit.

Filtered iterators can greatly simplify the development of robust programs. We formally define filtered iterators in RIFL. The semantics of filtered iterators ensure that each input unit affects program execution atomically. Our benchmarks show that using filtered iterators reduces an average of 41.7% lines of code, or 58.5% conditional clauses and 33.4% unconditional computation, from fully manual implementations.

Thesis Supervisor: Martin C. Rinard
Title: Professor of Computer Science

Acknowledgments

I would like to thank Martin Rinard, my advisor, for his guidance and inspiration. I would also like to thank Deokhwan Kim for his help in improving the ways I formalize and organize ideas. I would like to thank Jeffrey Bosboom, Fan Long, Sasa Misailovic and Stelios Sidiroglou for their insightful comments. Finally, I would like to thank my parents and grandparents for their love and unconditional support.

Contents

1	Introduction	15
1.1	Basic Approach	15
1.2	Background	17
1.3	Contributions	19
2	Example	21
3	Design	25
3.1	Filtered Iterators	25
3.1.1	Iterating over input units	26
3.1.2	Filtering input units	27
3.2	RIFL Core Language	28
3.2.1	Language support for filtered iterators	28
3.2.2	Abstract syntax	30
3.2.3	Operational semantics	31
3.2.4	Properties	44
3.3	Extensions for Explicit Error Handling	45
3.4	Discussion	50
4	Experiments	53
4.1	Benchmarks	53
4.2	Experimental Setup	56
4.2.1	Independent variables	57

4.2.2	Dependent variables	58
4.3	Results	60
4.4	Discussion	63
5	Related Work	67
6	Conclusion and Future Work	69
A	Source Code for Benchmarks	71
A.1	CSV	71
A.1.1	Fully-implicit version	71
A.1.2	Protective-check version	72
A.1.3	Explicit-recovery version	73
A.1.4	Plain-loop version	74
A.2	OBJ	75
A.2.1	Fully-implicit version	75
A.2.2	Protective-check version	77
A.2.3	Explicit-recovery version	78
A.2.4	Plain-loop version	80
A.3	JSON	83
A.3.1	Fully-implicit version	83
A.3.2	Protective-check version	85
A.3.3	Explicit-recovery version	87
A.3.4	Plain-loop version	90
A.4	PNG	94
A.4.1	Fully-implicit version	94
A.4.2	Protective-check version	95
A.4.3	Explicit-recovery version	97
A.4.4	Plain-loop version	99
A.5	ZIP	101
A.5.1	Fully-implicit version	101

A.5.2	Protective-check version	103
A.5.3	Explicit-recovery version	105
A.5.4	Plain-loop version	108
A.6	RGIF	111
A.6.1	Fully-implicit version	111
A.6.2	Protective-check version	114
A.6.3	Explicit-recovery version	117
A.6.4	Plain-loop version	121
A.7	PCAP/DNS	126
A.7.1	Fully-implicit version	126
A.7.2	Protective-check version	129
A.7.3	Explicit-recovery version	133
A.7.4	Plain-loop version	137

List of Figures

2-1	Snippet of a CSV parser using conventional loops	24
2-2	Snippet of a CSV parser using filtered iterators	24
3-1	Abstract syntax	30
3-2	Semantics for simple expressions	33
3-3	Semantics for simple assignments	33
3-4	Semantics for arrays	34
3-5	Semantics for function calls	35
3-6	Semantics for basic control structures	35
3-7	Semantics for assertions	36
3-8	Semantics for filtered iterators—with good text inputs	38
3-9	Semantics for filtered iterators—with bad text inputs	39
3-10	Semantics for filtered iterators—with good binary inputs	40
3-11	Semantics for filtered iterators—with bad binary inputs	41
3-12	Semantics for input file expressions	42
3-13	Semantics for opening input files	42
3-14	Semantics for accessing input files	43
3-15	Semantics for iterators without error handling—with good text inputs	46
3-16	Semantics for iterators without error handling—with bad text inputs	47
3-17	Semantics for iterators without error handling—with good binary inputs	47
3-18	Semantics for iterators without error handling—with bad binary inputs	48
3-19	Semantics for arrays with error codes	49
3-20	Semantics for input file expressions with error codes	49

3-21	Semantics for opening input files with error codes	50
3-22	Semantics for accessing input files with error codes	51
3-23	Semantics for expressions to support error-code extensions	52
4-1	Control-flow complexity	61
4-2	Data-manipulation complexity	61
4-3	Lines of code	61
4-4	Effects of filtering and iterating	63

List of Tables

3.1	Error-code interfaces for system calls	50
4.1	Language restrictions for four versions	59
4.2	Average decrease in control-flow complexity	62
4.3	Average decrease in data-manipulation complexity	62
4.4	Average decrease in lines of code	62

Chapter 1

Introduction

Programs have bugs. Developers writing error-handling code often overlook uncommon inputs, and these unanticipated inputs can cause the programs to behave unexpectedly. This phenomenon is especially dangerous for programs that are directly exposed to potentially malicious inputs. Besides degrading the output quality, malicious inputs can also trigger undefined behavior that enables security exploits.

One approach to prevent this vulnerability is to simply reject the entire unanticipated input. Unfortunately, this approach is sub-optimal. Inputs often consist of sequences of input units. Even if some input units trigger errors, programs can often correctly process other input units that are often valuable to the users. Thus, it is desirable to discard only those *bad* input units—the input units that the programs cannot process. In fact, developers fix many of the vulnerabilities in these programs by adding input validations that allow the programs to skip the bad input units and continue on to process the remaining input [25]. Explicit programming language support for this pattern would allow for automatic error handling and would greatly simplify the development of robust programs.

1.1 Basic Approach

The Robust Input Filtering Language (RIFL) allows programmers to develop robust programs without explicit error handling. RIFL enables this feature with a new

control structure called *filtered iterators*.

Filtered iterators treat inputs as input units and automatically handle errors at the granularity of input units. Each filtered iteration is an atomic transaction that uses the program itself as a filter on whether the input unit should affect the program or not. Filtered iterators ensure that each input unit atomically affects the program's execution: If any error occurs when processing an input unit, the RIFL runtime would discard the input unit, and the program would then continue to process the remaining inputs as if this bad input unit had never existed. The RIFL interpreter supports filtered iterators by executing the iteration tentatively, detecting unanticipated errors dynamically, recovering the program states transactionally, and discarding bad input units to make forward progress.

Filtered iterations give the program a purified view on inputs: The program execution would be the same as if the program were reading some other input, specifically, the original input minus those bad input units which would generate errors if processed. This key property enables filtered iterators to simplify the development of robust programs. At present, programs are vulnerable to errors triggered by unanticipated inputs. With filtered iterators, developers may write robust programs that contain only common-case code and necessary assertions. The RIFL implementation then gracefully and automatically filters out bad input units.

From a higher level, the iterator construct clarifies the program structure by making the pattern of input units explicit to the language implementation. This explicit structure allows RIFL to handle errors automatically by filtering out bad input units. This automatic filtering strategy protects programs against all unanticipated circumstances where the programs would not originally be able to proceed.

We conducted experiments over seven benchmark applications that robustly process structured input formats. Filtered iterators eliminate the need for error-handling code, reducing an average of 41.7% lines of code, or 58.5% conditional clauses and 33.4% unconditional computation, from these programs.

Although these numbers are compelling, they may not fully capture the significant difference between programs with or without filtered iterators. Filtered iterators

eliminate the error-handling code that is harder to implement correctly than common-case code. Error detection requires developers to consider exceptional situations that appear less often and are harder to anticipate than common-case situations. Two potential problems are that (a) developers may fail to anticipate and write error-handling code for all the exceptional cases and that (b) developers may fail to write comprehensive test suites that exercise all the error-handling code. Furthermore, error recovery requires developers to maintain resources and to adjust the program logic according to exceptional situations, which are harder to reason about than common-case logic. Real-world experience also shows that error-handling code is prone to errors [25].

1.2 Background

The notion of filtered iterators combines filters, iterators and atomic transactions with input units.

Filters: A filter function extracts a subset from a collection so that each element satisfies a given predicate. Many programming languages have standard support for filters, such as the `remove-if` function in Common Lisp [33], the `select` message in Smalltalk [12], and the `filter` function in Haskell [19].

RIFL uses the program's safe execution as an implicit predicate for filtering input units.

Iterators: An iterator is a generalization of loops over collections, which separates the action performed on each object from the selection of the objects. The concept of iterators was originally proposed in CLU [21] as a control abstraction. It is now a mainstream structure which many programming languages support in various forms. For example, Smalltalk [12] supports enumeration messages to concisely express sequences of messages on collection elements. In Java [13], the framework for collections enables manipulating collections independently of the details of their representation. Python [36] also has built-in support for iterations over collections.

RIFL abstracts the input as a structured collection of input units. Filtered iterators separate the instructions for operating input units from the instructions for extracting input units. If an input unit triggers an error, the filtered iterator automatically discards the partial updates from this unit and restarts program execution from the next unit. This property allows developers to concentrate on operating common inputs, rather than on recovering from errors.

Transactions: A standard transaction is a group of actions with ACID properties: atomicity, consistency, isolation and durability. Transactions ensure the consistency in spite of concurrency and failures for database systems [11, 16, 15] and for distributed systems [20]. Centralized multiprocess systems also use transactions as an alternative abstraction to explicit synchronization [22]: Various transactional memory [17, 34, 30] implementations simplify the management of shared-memory data structures. These standard transaction implementations support multiple concurrent updates and survive unreliable environments. In contrast to these complexities, the transactions used in filtered iterators concentrate on the atomicity of input units.

Hierarchical structures further enhance the expressiveness of filtered iterators. For example, a program with two nested filtered iterators can process nesting input units with two layers. When an error occurs in an inner iteration, the inner filtered iterator atomically discards the bad inner input unit. When an error occurs in an outer iteration but outside of the inner filtered iterator, the outer filtered iterator atomically discards the bad input unit it is processing, including all the updates that the inner filtered iterator has successfully processed. Traditional nested transactions [22, 26, 14] have various designs on whether or not to discard inner commits when an outer transaction aborts.

In short, filtered iterators treat inputs as collections of input units, iterate over the units, use the programs to filter out bad units, and atomically update program states for each unit.

1.3 Contributions

This thesis makes the following contributions.

1. We present filtered iterators, a novel control structure that discards bad input units atomically and automatically.
2. We formally present RIFL, a novel programming language that supports filtered iterators and eliminates the need to write most error-handling code.
3. We describe a set of metrics to estimate the relative difficulty of program implementations and evaluate filtered iterators from this empirical viewpoint.

In this thesis, we present example RIFL programs, present the concepts of filtered iterators, formally describe the RIFL language, evaluate filtered iterators with RIFL programs that use different error-handling strategies, and discuss related work on software error recovery.

Chapter 2

Example

RIFL supports filtered iterators with “inspect” loops that handle errors implicitly. These loops are specialized for processing input files that consist of *input units*. Like conventional “while” loops, an “inspect” loop repeats executing a code block while a given condition holds. Unlike “while” loops, an “inspect” loop additionally adjusts the offset associated with a given input file during execution. In normal situations, the “inspect” loop maintains the input offset according to the boundaries of input units. The effect is that each loop iteration processes exactly one input unit. In abnormal situations, the “inspect” loop avoids visible errors by adjusting the input offset. The effect is the *atomic* property: each input unit is either successfully processed in an iteration or is completely ignored.

For text input formats, an “inspect” loop has the syntax of `inspectt`. The simplified structure of an `inspectt` loop is

```
inspectt (e, f, du) { ... }
```

which iterates through input units in a text file `f` when expression `e` evaluates to true. A delimiter `du` defines the boundaries between input units. Section 3.2.1 presents a more generalized `inspectt` syntax.

A key principle of RIFL is to encourage writing only common-case code and handling errors implicitly with filtered iterators. To illustrate this idea, we present two example code snippets that have the same functionality but differ in error-handling

techniques. Both code snippets extract and print the fields in the content lines for Comma Separated Values (CSV) files. Both snippets come from programs that parse CSV files. Section 4.1 describes the functionality of the CSV parsers in more detail.

The two snippets differ in the available language features related to error handling. They correspond to the plain-loop and the fully-implicit versions, respectively, that are defined in Section 4.2.1. The plain-loop snippet uses the system calls that return explicit error codes. Besides, this snippet may use only the conventional looping construct, `while`, but may not use filtered iterators. On the other hand, the fully-implicit snippet uses system calls that trigger errors to be handled implicitly. This snippet may use both conventional `while` loops and filtered iterators such as `inspectt`.

Figure 2-1 presents the plain-loop snippet that handles all errors explicitly. Appendix A.1.4 presents the full program. On lines 60, 68, 74, 82, and 90–92, the program identifies the boundaries between input units. On lines 60 and 86, the program validates input units. On lines 52–56 and 77–80, the program maintains an output buffer to ensure that bad input units would not produce partial outputs.

For example, if the program snippet reads the following input with `columns = 3`,

```
1,2,too much data,3
4,5,6
7,8
9,10,11,12
```

it produces the following output:

```
1,2,3
4,5,6
9,10,11
```

Figure 2-2 presents the fully-implicit snippet that uses filtered iterators, or the `inspectt` construct, to handle errors implicitly. Appendix A.1.1 presents the full program. This snippet has the same functionality as the plain-loop snippet in Figure 2-1. Unlike the plain-loop program, this fully-implicit program uses `inspectt` loops to implicitly and atomically discard any fields or content lines that violate assertions

or trigger other errors in an iteration. On line 26, the program uses the `inspectt` construct to loop through content lines in the CSV file. On line 28, the program uses the `inspectt` construct again to loop through fields in each content line. On lines 29–35, the program implicitly requires that each field is at most 10 characters long. On lines 28 and 44, the program explicitly specifies that each content line should have at least “`columns`” fields and that it processes the first “`columns`” fields on each content line. With the example input above, the `inspectt` loops in this snippet implicitly discard the field “`too much data`” which is too long and the line “`7,8`” which contains too few fields, without affecting the program state.

The fully-implicit snippet is shorter and simpler than the plain-loop snippet. This fact is consistent with the intuition that filtered iterators can simplify the implementations of robust programs.

```

51 while (!end_ec(f)) {
52     idx = 0;
53     while (idx < 11 * columns) {
54         buffer[idx] = 0;
55         idx = idx + 1;
56     }
57     idx = 0;
58     j = 0;
59     x = 0;
60     while (j < columns && x >= 0 && x != '\n') {
61         start = idx;
62         if (j > 0) {
63             buffer[idx] = ',';
64             idx = idx + 1;
65         }
66         i = 0;
67         x = read_ec(f);
68         while (x >= 0 && x != '\n' && x != ',' && i < 10) {
69             buffer[idx] = x;
70             idx = idx + 1;
71             i = i + 1;
72             x = read_ec(f);
73         }
74         if (x == '\n' || x == ',') {
75             j = j + 1;
76         } else { // skip unit
77             while (idx > start) {
78                 buffer[idx] = 0;
79                 idx = idx - 1;
80             }
81         }
82         while (x >= 0 && x != '\n' && x != ',') {
83             x = read_ec(f);
84         }
85     }
86     if (j == columns) {
87         print(buffer);
88         print('\n');
89     } // skip unit
90     while (x >= 0 && x != '\n') {
91         x = read_ec(f);
92     }
93 }

```

Figure 2-1: Snippet of a CSV parser using conventional loops

```

26 inspectt (!end(f), f, '\n') {
27     j = 0;
28     inspectt (j < columns, f, ',') {
29         field = malloc(10);
30         i = 0;
31         while (!end(f)) {
32             x = read(f);
33             field[i] = x;
34             i = i + 1;
35         }
36         if (j > 0) {
37             print(',');
38         }
39         print(field);
40         free(field);
41         j = j + 1;
42     }
43     print('\n');
44     assert(j == columns);
45 }

```

Figure 2-2: Snippet of a CSV parser using filtered iterators

Chapter 3

Design

In this chapter, we first introduce filtered iterators, a simple and powerful way to structure programs that process inputs in input units. We also formally present the core language of RIFL, a new programming language that supports handling errors implicitly with filtered iterators. Then we present optional extensions that support handling errors explicitly in RIFL. Finally, we discuss the design rationale.

3.1 Filtered Iterators

A *filtered iterator* is a new control structure that models inputs as collections of *input units*, iterates over the units, uses the programs to filter out bad units, and atomically updates program state for each unit. Filtered iterators dynamically decide whether an input unit is *good* or *bad*. Good input units allow a program to successfully execute in the way that the code defines. Bad input units trigger errors or undefined behaviors if processed. In other words, developers anticipate only good units but no bad units. Filtered iterators feed the program with good input units and discard bad units.

To illustrate the behavior of filtered iterators in detail, we next explain the two features: iterating and filtering.

3.1.1 Iterating over input units

A filtered iterator in a RIFL program automatically dissects the inputs into input units according to several parameters in the program. Each iteration may access one input unit. After each iteration, RIFL implementation automatically advances the file pointer to the start of the next input unit.

This structure of iterators abstracts away the details of identifying boundaries and encourages the program to focus on processing the contents. This abstraction also gives RIFL opportunities to automatically recover the program execution from bad input units.

RIFL enables the robust decomposition of inputs by enforcing predefined information for each input unit. There are two ways to specify the structure of input units—delimiters and length fields.

Input units with delimiters: Developers may specify delimiters that mark the ends of input units. As long as these delimiters do not collide with the input unit contents, it is always possible to isolate input units from each other.

Input units with length fields: Developers may also specify the upper-bound lengths for input units. RIFL uses these lengths to indicate where each input unit must end, similarly to delimiters. When length fields in nesting input units do not exactly add up, RIFL identifies the boundaries of input units as follows. If the lengths of the inner components exceed the length indicated by the outer unit, RIFL would treat the last component as an incomplete, bad input unit. On the other hand, if all the inner contents do not fill up the length indicated by the outer input unit, RIFL would skip the trailing bytes after executing the iteration.

The input format affects the ability of RIFL to recover programs from errors. The scope of RIFL is to handle the input formats where delimiters or length fields unambiguously indicate the ends of input units.

Delimiters are more natural in text inputs while length fields are more natural in binary inputs. The reason is that the effective contents in text inputs often take a

small set of possible byte values such as visible characters. It is easier in text inputs than in binary inputs to define delimiters that do not collide with the input unit contents. RIFL implementation supports delimiters for text inputs and length fields for binary inputs.

3.1.2 Filtering input units

The RIFL implementation detects bad input units dynamically, discards them atomically, and then resumes the program's execution. In effect, a RIFL program performs updates from good input units only, so that it is as if bad units did not exist.

Bad input units are the units that trigger detectable errors or undefined behavior during the program execution. Such situations include:

1. Internal errors such as divide-by-zero errors, integer overflows, null pointer dereferences, and out-of-bounds array accesses. These errors often come from missing input validations.
2. Errors related to external contexts such as file access failures and resource exhaustions. These errors can result from missing validations on inputs or system calls.
3. Assertion violations. Assertions are optional but helpful for enforcing subtle requirements on the input formats. For example, developers may use assertions to cause RIFL to discard certain undesirable input units that may not otherwise trigger errors during the execution.

RIFL detects and recovers programs from all these undesirable situations.

The distinction of an input unit being good or bad depends on the program state. For example, it may depend on some good input units that the program have previously processed.

3.2 RIFL Core Language

In this section, we describe the design of filtered iterators in RIFL, present the abstract syntax, present the big-step operational semantics, and discuss the properties of the RIFL core language.

3.2.1 Language support for filtered iterators

A main difference of RIFL from conventional languages is the support for filtered iterators. RIFL supports filtered iterators with a special loop construct, “inspect”. Each iteration processes one input unit as an atomic transaction, whose updates either all succeed or nothing happens.

Iterating: An “inspect” loop takes several parameters to identify input units in the input files. There are two language keywords, `inspectt` and `inspectb`, that process text and binary inputs, respectively.

The basic usage of “inspect” loops for text inputs is

```
inspectt (e, f, du, ds) { ... }
```

which iterates through input units in a text file `f` when expression `e` evaluates to true. Each loop iteration may access an input unit that consists of the contents of file `f` up to the end-of-unit delimiters specified in `du`. The loop terminates when `e` evaluates to false, when the program reads the end-of-sequence delimiters specified in `ds`, or when the program reaches the end of file `f`.

The basic usage of “inspect” loops for binary inputs is

```
inspectb (e, f, o, w, c) { ... }
```

which iterates through input units in a binary file `f` when expression `e` evaluates to true. Each loop iteration may access an input unit that consists of the contents of file `f` up to a cutoff position as specified by the parameters `o`, `w`, and `c`. The loop terminates when `e` evaluates to false, when the program reaches the end of an outer-level input unit, or when the program reaches the end of file `f`. The cutoff position for each

input unit is computed as follows. Before each loop iteration, RIFL implementation identifies the length field in file f using the offset o and the width w . It extracts the value of the length field according to the endianness that the developer specifies when opening file f . RIFL implementation then precomputes a cutoff position of the current input unit by summing up the current file offset, the value of the length field, and the extra length c .

Besides sequential input units, developers may also process complex input structures with nesting and recursion. When identifying input units in these complex structures, RIFL prioritizes the delimiters and the length fields from outer nesting levels.

Filtering: In addition to iterating, “inspect” loops also dynamically filter out bad input units. If a loop iteration triggers an error, RIFL implementation would recover the program execution by restoring all the program state except for advancing the file pointer past the bad input unit. Technically, the implementation contains the following steps.

1. Undo all updates that the program has performed when processing the current bad input unit.
2. Skip this bad input unit in f according to the parameters. Text files use the end-of-unit delimiters specified in du ; binary files use the cutoff positions computed from o , w and c .
3. Restart program execution from the original loop iteration.

For text files, “inspect” loops can precisely skip a bad input unit as long as the delimiter is intact and unambiguous. For binary files, “inspect” loops can precisely skip a bad input unit as long as the real length of this input unit corresponds to the parameters that describe its length.

```

Prog := Stmt | func  $q(x)$ {Stmt;return  $y$ };Prog
Exp :=  $n$  |  $x$  | Exp op Exp |  $a[\text{Exp}]$  | valid( $a$ ) | end( $f$ ) | pos( $f$ )
Stmt :=  $x = \text{Exp}$  |  $a = \text{malloc}(\text{Exp})$  | free( $a$ ) |  $a[\text{Exp}] = \text{Exp}$  |  $x = q(\text{Exp})$ 
      | Stmt;Stmt | if(Exp){Stmt}else{Stmt} | while(Exp){Stmt}
      | inspectt(Exp,  $f$ ,  $d_u$ ,  $d_s$ ){Stmt} | inspectb(Exp,  $f$ , Exp, Exp, Exp){Stmt}
      |  $f = \text{opent}(str)$  |  $f = \text{openb}(str)$  | seek( $f$ , Exp) |  $x = \text{read}(f)$ 
      | assert(Exp)

```

$x, y \in IVar$	$q \in \text{function names}$
$a, d_u, d_s \in AVar$	$n \in Int$
$f \in FVar$	$str \in String$

Figure 3-1: Abstract syntax

3.2.2 Abstract syntax

Figure 3-1 presents the abstract syntax of the core language. RIFL is an imperative language with integer operations, array operations, file operations, sequential composition, conditional statements, loops including filtered iterators, functions, and assertions.

RIFL adds error handling to conventional operations including arithmetic expressions, `valid` expressions which test array variables, `pos` expressions which return file pointer offsets, array accesses, `assert` statements, and file operations `seek` and `read`. RIFL also integrates error handling into control structures including sequential composition, conditional statements, loops, and function calls.

The main new constructs are the `inspectt` and `inspectb` loops which implement filtered iterators for text and binary files, respectively. To distinguish text and binary input formats, RIFL supports the `opent` and `openb` constructs that open text and binary files, respectively. To serve the process of reading inputs, RIFL also supports the `end` predicate which tests the end of the current input unit.

3.2.3 Operational semantics

Figures 3-2–3-14 present the big-step operational semantics using the following domain:

$$\begin{array}{ll}
 \textit{State} = \textit{Stack} \times \textit{Heap} \times \textit{Files} \times \textit{Disk} & \textit{Data} = \textit{Offs} \rightarrow \textit{Int} \\
 \textit{Stack} = \textit{Var} \rightarrow \textit{Value} & \textit{FDesc} = \textit{FName} \times \textit{Offs} \times \textit{SOU} \times \textit{UDesc} \\
 \textit{Heap} = \textit{Addr} \rightarrow \textit{Data} \times \textit{Size} & \textit{FName} = \textit{String} \\
 \textit{Files} = \textit{FHndl} \rightarrow \textit{FDesc} & \textit{UDesc} = \textit{Delim} \cup \textit{Cutoff} \\
 \textit{Disk} = \textit{FName} \rightarrow \textit{Data} \times \textit{Size} & \textit{Delim} = \textit{EOU} \times \textit{EOS} \times \textit{OSD} \\
 \textit{Var} = \textit{IVar} \cup \textit{FVar} \cup \textit{AVar} & \textit{Size} = \textit{Offs} = \textit{Cutoff} = \textit{Int} \\
 \textit{Value} = \textit{Int} \cup \textit{FHndl} \cup \textit{Addr} & \textit{EOU} = \textit{EOS} = \textit{OSD} = \mathcal{P}(\textit{Int})
 \end{array}$$

A state $\sigma \in \textit{State}$ contains information about the stack memory, the heap memory, the status of opened files and the disk. The stack maps variables to values, which can be integers, file handlers or memory addresses. The heap maps memory addresses to array contents. The file status maps file handlers to file descriptors. The disk maps file names to file contents.

A file descriptor $fd \in \textit{FDesc}$ describes the current status of reading an input file, including the file name, the current offset into the file, the starting offset of the current input unit, and an input unit descriptor. An input unit descriptor $ud \in \textit{UDesc}$ describes the delimiters in use for text files and the cutoff positions for binary files.

For text files, a delimiter definition $dlim \in \textit{Delim}$ describes three sets of delimiters that identify the boundaries between input units: $dlim = \langle eou, eos, osd \rangle$ where $eou \in \textit{EOU}$ is the set of end-of-unit delimiters, $eos \in \textit{EOS}$ is the set of end-of-sequence delimiters, and $osd \in \textit{OSD}$ is the set of outside delimiters that serve nested input units. The next delimiter in the input file, whether it is one in $eou \cup eos \cup osd$ or the end of the file, marks the end of the current input unit. The set of outside delimiters osd updates at runtime as follows. For single-layer `inspectt` loops and the outermost `inspectt` loops in nested structures, $osd = \emptyset$. For inner `inspectt`

loops, *osd* includes all delimiters in $eou \cup eos$ for all the outer **inspectt** layers, except for those that also appear in $eou \cup eos$ of the current layer. This exception is useful for input formats that reuse delimiters across layers, such as JavaScript Object Notation (JSON). However, the developer should be careful about reusing delimiters across the hierarchy. Reuse makes the meanings of delimiters ambiguous, which may cause the program to misinterpret the input structures in face of delimiter corruptions.

For binary files, a cutoff position $cut \in Cutoff$ describes where the current input unit ends. This value also updates at runtime according to nesting **inspectb** layers.

The relation $\langle e, \sigma \rangle \Downarrow_e \mu$ denotes that evaluating the expression e in state σ yields the result $\mu \in Int \cup \{err\}$. A result $\mu \in Int$ indicates that the evaluation is successful and that the numerical result is μ . A result $\mu = err$ indicates that the evaluation fails, which would then trigger an error in the surrounding statement.

The relation $\langle s, \sigma \rangle \Downarrow_s \xi$ denotes that executing the statement s in the state σ yields the output configuration $\xi \in State \times \{ok, bad\}$. An output configuration $\xi = \langle \sigma', ok \rangle$ indicates that the program execution is successful and that the resulting state is σ' . An output configuration $\xi = \langle \sigma', bad \rangle$ indicates that the program execution triggers an error and that the latest reasonable program state is σ' . In this case, RIFL implementation would report the error to the surrounding “inspect” environment which would resolve the problem.

Basic operations

Figures 3-2–3-7 present some basic operations. Figure 3-2 presents the semantics for simple expressions. Arithmetic errors and invalid array reads trigger errors in the surrounding statement (*iop-bad*, *ard-null*, *ard-out*). The **valid** predicate tests whether an array variable is not null (*avald-t*, *avald-t*). Figure 3-3 presents the semantics for simple assignments. When assigning a bad expression to a variable, rule (*vwr-bad*) treats the statement as a no-op and reports the error. Figure 3-4 presents the semantics for arrays. A successful **malloc** statement allocates a space of the specified size in the heap, initializes all the elements to 0, and sets the array variable to the heap address (*malloc-ok*). A successful **free** statement deallocates

$$\begin{array}{c}
\frac{}{\langle n, \sigma \rangle \Downarrow_e n} \quad (\text{int}) \\
\\
\frac{}{\langle x, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \sigma_S(x)} \quad (\text{ivar}) \\
\\
\frac{\langle e_1, \sigma \rangle \Downarrow_e u_1 \quad \langle e_2, \sigma \rangle \Downarrow_e u_2 \quad u_1 \text{ op } u_2 = v}{\langle e_1 \text{ op } e_2, \sigma \rangle \Downarrow_e v} \quad (\text{iop-ok}) \\
\\
\frac{\langle e_1, \sigma \rangle \Downarrow_e u_1 \quad \langle e_2, \sigma \rangle \Downarrow_e u_2 \quad u_1 \text{ op } u_2 = \perp}{\langle e_1 \text{ op } e_2, \sigma \rangle \Downarrow_e \text{err}} \quad (\text{iop-bad}) \\
\\
\frac{\sigma_S(a) = \text{null}}{\langle a[e], \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err}} \quad (\text{ard-null}) \\
\\
\frac{\sigma_S(a) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n \rangle \quad u < 0 \vee u \geq n}{\langle a[e], \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err}} \quad (\text{ard-out}) \\
\\
\frac{\sigma_S(a) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n \rangle \quad 0 \leq u < n}{\langle a[e], \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \gamma(u)} \quad (\text{ard-ok}) \\
\\
\frac{\sigma_S(a) \neq \text{null}}{\langle \text{valid}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true}} \quad (\text{avaliid-t}) \\
\\
\frac{\sigma_S(a) = \text{null}}{\langle \text{valid}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{false}} \quad (\text{avaliid-f})
\end{array}$$

Figure 3-2: Semantics for simple expressions

$$\begin{array}{c}
\frac{\langle e, \sigma \rangle \Downarrow_e \text{err}}{\langle x = e, \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{vwr-bad}) \\
\\
\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u}{\langle x = e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto u], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{vwr-ok})
\end{array}$$

Figure 3-3: Semantics for simple assignments

$$\begin{array}{c}
\frac{\langle e, \sigma \rangle \Downarrow_e \text{err}}{\langle a = \text{malloc}(e), \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{malloc-bad}) \\
\frac{\langle e, \sigma \rangle \Downarrow_e u \quad u \leq 0}{\langle a = \text{malloc}(e), \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{malloc-neg}) \\
\frac{\langle e, \sigma \rangle \Downarrow_e u \quad u > 0 \quad \text{heap allocate}(u) = \perp}{\langle a = \text{malloc}(e), \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{malloc-ovf}) \\
\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad u > 0 \quad \text{heap allocate}(u) = \text{addr}}{\langle \langle \sigma_S[a \mapsto \text{addr}], \sigma_H[\text{addr} \mapsto \langle [0 \mapsto 0, 1 \mapsto 0, \dots, u-1 \mapsto 0], u \rangle], \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{malloc-ok}) \\
\frac{\sigma_S(a) = \text{null}}{\langle \text{free}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{free-null}) \\
\frac{\sigma_S(a) \neq \text{null}}{\langle \text{free}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[a \mapsto \text{null}], \sigma_H[\sigma_S(a) \mapsto \perp], \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{free-ok}) \\
\frac{\sigma_S(a) = \text{null}}{\langle a[e_1] = e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{awr-null}) \\
\frac{\sigma_S(a) \neq \text{null} \quad \langle e_1, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err} \vee \langle e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err}}{\langle a[e_1] = e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{awr-bad}) \\
\frac{\sigma_S(a) \neq \text{null} \quad \langle e_1, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n \rangle \quad u < 0 \vee u \geq n}{\langle a[e_1] = e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{awr-out}) \\
\frac{\sigma_S(a) \neq \text{null} \quad \langle e_1, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_1 \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n \rangle \quad 0 \leq u_1 < n \quad \langle e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_2}{\langle a[e_1] = e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H[\sigma_S(a) \mapsto \langle \gamma[u_1 \mapsto u_2], n \rangle], \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{awr-ok})
\end{array}$$

Figure 3-4: Semantics for arrays

$$\begin{array}{c}
\frac{\langle e, \sigma \rangle \Downarrow_e \text{err}}{\langle x = q(e), \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{fn-arg}) \\
\frac{\text{stack allocate}(fr(q)) = \perp}{\langle x = q(e), \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{fn-ovf}) \\
\frac{\text{stack allocate}(fr(q)) \neq \perp \quad \langle e, \sigma \rangle \Downarrow_e u \quad \langle body(q), \langle [arg(q) \mapsto u], \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{bad} \rangle}}{\langle x = q(e), \sigma \rangle \Downarrow_s \langle \sigma_S, \sigma_H, \sigma'_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{fn-body}) \\
\frac{\text{stack allocate}(fr(q)) \neq \perp \quad \langle e, \sigma \rangle \Downarrow_e u \quad \langle body(q), \langle [arg(q) \mapsto u], \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle}}{\langle x = q(e), \sigma \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto \sigma'_S(ret(q))], \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle} \quad (\text{fn-prgr})
\end{array}$$

Figure 3-5: Semantics for function calls

$$\begin{array}{c}
\frac{\langle s_1, \sigma \rangle \Downarrow_s \langle \sigma', \text{bad} \rangle}{\langle s_1; s_2, \sigma \rangle \Downarrow_s \langle \sigma', \text{bad} \rangle} \quad (\text{seq-bad}) \\
\frac{\langle s_1, \sigma \rangle \Downarrow_s \langle \sigma', \text{ok} \rangle \quad \langle s_2, \sigma' \rangle \Downarrow_s \xi}{\langle s_1; s_2, \sigma \rangle \Downarrow_s \xi} \quad (\text{seq-prgr}) \\
\frac{\langle e, \sigma \rangle \Downarrow_e \text{err}}{\langle \text{if}(e)\{s_1\}\text{else}\{s_2\}, \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{if-bad}) \\
\frac{\langle e, \sigma \rangle \Downarrow_e \text{true} \quad \langle s_1, \sigma \rangle \Downarrow_s \xi}{\langle \text{if}(e)\{s_1\}\text{else}\{s_2\}, \sigma \rangle \Downarrow_s \xi} \quad (\text{if-t}) \\
\frac{\langle e, \sigma \rangle \Downarrow_e \text{false} \quad \langle s_2, \sigma \rangle \Downarrow_s \xi}{\langle \text{if}(e)\{s_1\}\text{else}\{s_2\}, \sigma \rangle \Downarrow_s \xi} \quad (\text{if-f}) \\
\frac{\langle e, \sigma \rangle \Downarrow_e \text{err}}{\langle \text{while}(e)\{s\}, \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{while-bad}) \\
\frac{\langle e, \sigma \rangle \Downarrow_e \text{false}}{\langle \text{while}(e)\{s\}, \sigma \rangle \Downarrow_s \langle \sigma, \text{ok} \rangle} \quad (\text{while-end}) \\
\frac{\langle e, \sigma \rangle \Downarrow_e \text{true} \quad \langle s, \sigma \rangle \Downarrow_s \langle \sigma', \text{bad} \rangle}{\langle \text{while}(e)\{s\}, \sigma \rangle \Downarrow_s \langle \sigma', \text{bad} \rangle} \quad (\text{while-body}) \\
\frac{\langle e, \sigma \rangle \Downarrow_e \text{true} \quad \langle s, \sigma \rangle \Downarrow_s \langle \sigma', \text{ok} \rangle \quad \langle \text{while}(e)\{s\}, \sigma' \rangle \Downarrow_s \xi}{\langle \text{while}(e)\{s\}, \sigma \rangle \Downarrow_s \xi} \quad (\text{while-prgr})
\end{array}$$

Figure 3-6: Semantics for basic control structures

$$\frac{\langle e, \sigma \rangle \Downarrow_e \text{err}}{\langle \text{assert}(e), \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{assert-bad})$$

$$\frac{\langle e, \sigma \rangle \Downarrow_e \text{true}}{\langle \text{assert}(e), \sigma \rangle \Downarrow_s \langle \sigma, \text{ok} \rangle} \quad (\text{assert-t})$$

$$\frac{\langle e, \sigma \rangle \Downarrow_e \text{false}}{\langle \text{assert}(e), \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{assert-f})$$

Figure 3-7: Semantics for assertions

the space from the heap and resets the array variable to null (free-ok). A successful assignment to an array element changes the specified element in the heap (awr-ok). On bad expressions, invalid `malloc` parameters, heap overflows, null array accesses, or out-of-bounds array writes, rules (malloc-bad, awr-bad, malloc-neg, malloc-ovf, free-null, awr-null, awr-out) treat the statement as a no-op and reports the error. Figure 3-5 presents the semantics for function calls using the following helper functions:

For each function definition `func q(x){s;return y}`,

let $arg(q) \triangleq x$, $body(q) \triangleq s$, $ret(q) \triangleq y$, $fr(q) \triangleq$ size of q 's stack frame.

A successful function call updates the global states and assigns the return value to the receiving variable (fn-prgr). If the argument uses a bad expression or if the stack overflows, rules (fn-arg, fn-ovf) treat the function call as a no-op and report the error. If an error occurs inside the function call, rule (fn-body) updates only the file descriptors and then reports the error. Figure 3-6 presents the semantics for basic control structures. When an error occurs, the program stops executing and reports the error (seq-bad, while-body). If an “inspect” loop surrounds these statements, then this “inspect” loop would discard the updates in the current iteration and would restart with the remaining input. Figure 3-7 presents the semantics for assertions. True assertions are no-ops (assert-t); false assertions or bad expressions generate errors (assert-f, assert-bad).

Filtered iterators

Figures 3-8–3-11 present the semantics for filtered iterators. An “inspect” loop automatically maintains the file descriptor and other program states, using delimiters for text files and length fields for binary files as follows.

Text input formats: Figures 3-8 and 3-9 present the semantics for filtered iterators for text input files, using the following helper functions:

$$\Omega(a) \triangleq \{j \in \text{Int} \mid \exists i \in \text{Int}, \sigma_H(\sigma_S(a)) = \langle \gamma, n \rangle, \gamma(i) = j\} \quad (a \in \text{AVar}, a \neq \text{null})$$

returns the set of elements in array a .

$$\Lambda'(l') \triangleq \underset{k \geq l'}{\operatorname{argmin}} \{k = n' \vee \gamma'(k) \in \text{eou}' \cup \text{eos}' \cup \text{osd}'\} \quad (l' = 0, 1, \dots, n')$$

returns the offset of the upcoming delimiter from offset l' .

An `inspectt` loop updates the starting offset of current input unit, updates the delimiters in use, and advances the offset according to the boundaries of input units (`inspt-prgr`). The loop terminates if the predicate evaluates to false (`inspt-end`) or if the program reaches the end of the unit sequence (`inspt-eos`, `inspt-osd`). Situations that end a sequence include reaching one of the explicit d_s delimiters, reaching a delimiter from outer `inspectt` layers, and reaching the end of the file. An `inspectt` loop handles a bad input unit by advancing the offset past the bad unit, restoring all other program states, and recovering the execution (`inspt-dsc-eou`, `inspt-dsc-eos`, `inspt-dsc-osd`). The delimiter arrays and the loop predicate should be valid (`inspt-null`, `inspt-bad`). The two sets of delimiters $\Omega(d_u)$ and $\Omega(d_s)$ should not intersect (`inspt-dupl`).

Binary input formats: Figures 3-10 and 3-11 present the semantics for filtered iterators for binary input files, using the following helper function:

$$\frac{\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{false}}{\langle \text{inspecttt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{inspt-end})$$

$$\frac{\begin{array}{l} \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \\ \Omega(d_u) \cap \Omega(d_s) = \emptyset \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle \\ \text{dlm} = \langle \Omega(d_u), \Omega(d_s), (\text{eou} \cup \text{eos} \cup \text{osd}) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{dlm} \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle \text{str}, l', \text{sou}', \langle \text{eou}', \text{eos}', \text{osd}' \rangle \rangle \\ \sigma'_D(\text{str}) = \langle \gamma', n' \rangle \quad \Lambda'(l') \geq n' \vee \gamma'(\Lambda'(l')) \in \text{osd}' \end{array}}{\langle \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle \text{str}, \Lambda'(l'), \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle], \sigma'_D \rangle, \text{ok} \rangle} \quad (\text{inspt-osd})$$

$$\frac{\begin{array}{l} \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \\ \Omega(d_u) \cap \Omega(d_s) = \emptyset \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle \\ \text{dlm} = \langle \Omega(d_u), \Omega(d_s), (\text{eou} \cup \text{eos} \cup \text{osd}) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{dlm} \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle \text{str}, l', \text{sou}', \langle \text{eou}', \text{eos}', \text{osd}' \rangle \rangle \\ \sigma'_D(\text{str}) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in \text{eos}' \end{array}}{\langle \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle \text{str}, \Lambda'(l') + 1, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle], \sigma'_D \rangle, \text{ok} \rangle} \quad (\text{inspt-eos})$$

$$\frac{\begin{array}{l} \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \\ \Omega(d_u) \cap \Omega(d_s) = \emptyset \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle \\ \text{dlm} = \langle \Omega(d_u), \Omega(d_s), (\text{eou} \cup \text{eos} \cup \text{osd}) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{dlm} \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle \text{str}, l', \text{sou}', \langle \text{eou}', \text{eos}', \text{osd}' \rangle \rangle \\ \sigma'_D(\text{str}) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in \text{eou}' \\ \langle \text{inspecttt}(e, f, d_u, d_s)\{s\}, \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle \text{str}, \Lambda'(l') + 1, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle], \sigma'_D \rangle \rangle \Downarrow_s \xi \end{array}}{\langle \text{inspecttt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \xi} \quad (\text{inspt-prgr})$$

Figure 3-8: Semantics for filtered iterators—with good text inputs

$$\frac{\sigma_S(d_u) = \text{null} \vee \sigma_S(d_s) = \text{null}}{\langle \text{inspecttt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{inspt-null})$$

$$\frac{\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err}}{\langle \text{inspecttt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{inspt-bad})$$

$$\frac{\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \Omega(d_u) \cap \Omega(d_s) \neq \emptyset}{\langle \text{inspecttt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{inspt-dupl})$$

$$\frac{\begin{array}{l} \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \\ \Omega(d_u) \cap \Omega(d_s) = \emptyset \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle \\ \text{dlm} = \langle \Omega(d_u), \Omega(d_s), (\text{eou} \cup \text{eos} \cup \text{osd}) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{dlm} \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{bad} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle \text{str}, l', \text{sou}', \langle \text{eou}', \text{eos}', \text{osd}' \rangle \rangle \\ \sigma'_D(\text{str}) = \langle \gamma', n' \rangle \quad \Lambda'(l') \geq n' \vee \gamma'(\Lambda'(l')) \in \text{osd}' \end{array}}{\langle \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, \Lambda'(l'), \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad (\text{inspt-dsc-osd})$$

$$\frac{\begin{array}{l} \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \\ \Omega(d_u) \cap \Omega(d_s) = \emptyset \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle \\ \text{dlm} = \langle \Omega(d_u), \Omega(d_s), (\text{eou} \cup \text{eos} \cup \text{osd}) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{dlm} \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{bad} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle \text{str}, l', \text{sou}', \langle \text{eou}', \text{eos}', \text{osd}' \rangle \rangle \\ \sigma'_D(\text{str}) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in \text{eos}' \end{array}}{\langle \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, \Lambda'(l') + 1, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad (\text{inspt-dsc-eos})$$

$$\frac{\begin{array}{l} \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \\ \Omega(d_u) \cap \Omega(d_s) = \emptyset \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle \\ \text{dlm} = \langle \Omega(d_u), \Omega(d_s), (\text{eou} \cup \text{eos} \cup \text{osd}) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{dlm} \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{bad} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle \text{str}, l', \text{sou}', \langle \text{eou}', \text{eos}', \text{osd}' \rangle \rangle \\ \sigma'_D(\text{str}) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in \text{eou}' \\ \langle \text{inspecttt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \\ \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, \Lambda'(l') + 1, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle], \sigma_D \rangle \rangle \Downarrow_s \xi \end{array}}{\langle \text{inspecttt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \xi} \quad (\text{inspt-dsc-eou})$$

Figure 3-9: Semantics for filtered iterators—with bad text inputs

$$\frac{\langle e, \sigma \rangle \Downarrow_e \text{false}}{\langle \text{inspectb}(e, f, o, w, c)\{s\}, \sigma \rangle \Downarrow_s \langle \sigma, \text{ok} \rangle} \quad (\text{inspb-end})$$

$$\frac{\begin{array}{l} \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_o \\ \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_w \quad \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_c \\ \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{cut} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u_o \geq 0 \quad u_w \geq 0 \\ \text{parseint}(l + u_o, u_w) = v \quad \text{cut} = l \vee u_o = u_w = u_c = 0 \vee v < 0 \end{array}}{\langle \text{inspectb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, \text{cut}, \text{sou}, \text{cut} \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad (\text{inspb-eos})$$

$$\frac{\begin{array}{l} \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_o \\ \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_w \quad \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_c \\ \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{cut} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u_o \geq 0 \quad u_w \geq 0 \\ \text{parseint}(l + u_o, u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l + u_o + u_w + v + u_c = \text{cut}' \quad l < \text{cut}' \leq \text{cut} \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{cut}' \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \langle \text{inspectb}(e, f, o, w, c)\{s\}, \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle \text{str}, \text{cut}', \text{sou}, \text{cut} \rangle], \sigma'_D \rangle \rangle \Downarrow_s \xi \end{array}}{\langle \text{inspectb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \xi} \quad (\text{inspb-prgr})$$

Figure 3-10: Semantics for filtered iterators—with good binary inputs

$$\text{parseint}(l, u) \triangleq \begin{cases} 0, & \text{if } u = 0 \\ \text{integer value decoded from} \\ \text{bytes } \gamma(l), \dots, \gamma(l + u - 1), & \text{if } u = 1, 2, \dots, n - l + 1 \end{cases}$$

$$(l = 0, 1, \dots, n)$$

returns the integer value decoded from the u bytes starting from offset l .

An `inspectb` loop updates the starting offset of current input unit, updates the cutoff position, and advances the offset according to the boundaries of input units (`inspb-prgr`). The loop terminates if the predicate evaluates to false (`inspb-end`) or if the program reaches the end of the unit sequence (`inspb-eos`). Situations that end a sequence include reaching the cutoff position of the surrounding `inspectb` layer, reaching a zero-length input unit, or reaching a negative length field. An `inspectb` loop handles a bad input unit by advancing the offset past the bad unit, restoring all other program states, and recovering the execution (`inspb-dsc-eou`, `inspb-dsc-eos`).

$$\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err} \vee \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err} \vee \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err} \vee \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err}}{\langle \text{inspectb}(e, f, o, w, c)\{s\}, \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{inspb-bad})$$

$$\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_o \quad \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_w \quad \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_c \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{cut} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u_o < 0 \vee u_w < 0 \vee u_c < 0}{\langle \text{inspectb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{inspb-neg})$$

$$\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_o \quad \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_w \quad \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_c \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{cut} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u_o \geq 0 \quad u_w \geq 0 \quad \text{parseint}(l + u_o, u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l + u_o + u_w + v + u_c = \text{cut}' \quad \text{cut}' > \text{cut}}{\langle \text{inspectb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, \text{cut}, \text{sou}, \text{cut} \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad (\text{inspb-dsc-eos})$$

$$\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_o \quad \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_w \quad \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_c \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{cut} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u_o \geq 0 \quad u_w \geq 0 \quad \text{parseint}(l + u_o, u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l + u_o + u_w + v + u_c = \text{cut}' \quad l < \text{cut}' \leq \text{cut} \quad \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{cut}' \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \sigma', \text{bad} \rangle}{\langle \text{inspectb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, \text{cut}', \text{sou}, \text{cut} \rangle], \sigma_D \rangle \rangle \Downarrow_s \xi} \quad (\text{inspb-dsc-eou})$$

Figure 3-11: Semantics for filtered iterators—with bad binary inputs

$$\frac{\sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \quad \sigma_D(str) = \langle \gamma, n \rangle \quad l = \Lambda(l)}{\langle \mathbf{end}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \mathbf{true}} \quad (\text{end-t})$$

$$\frac{\sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \quad \sigma_D(str) = \langle \gamma, n \rangle \quad l < \Lambda(l)}{\langle \mathbf{end}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \mathbf{false}} \quad (\text{end-f})$$

$$\frac{\sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle}{\langle \mathbf{pos}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e l} \quad (\text{pos})$$

Figure 3-12: Semantics for input file expressions

$$\frac{\text{file } str \text{ does not exist}}{\langle f = \mathbf{opent}(str), \sigma \rangle \Downarrow_s \langle \sigma, \mathbf{bad} \rangle} \quad (\text{opent-bad})$$

$$\frac{\text{file } str \text{ exists}}{\langle f = \mathbf{opent}(str), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[f \mapsto \mathbf{hndl}], \sigma_H, \sigma_F[\mathbf{hndl} \mapsto \langle str, 0, 0, \langle \emptyset, \emptyset, \emptyset \rangle] \rangle, \sigma_D \rangle, \mathbf{ok} \rangle} \quad (\text{opent-ok})$$

$$\frac{\text{file } str \text{ does not exist}}{\langle f = \mathbf{openb}(str), \sigma \rangle \Downarrow_s \langle \sigma, \mathbf{bad} \rangle} \quad (\text{openb-bad})$$

$$\frac{\text{file } str \text{ exists} \quad \sigma_D(str) = \langle \gamma, n \rangle}{\langle f = \mathbf{openb}(str), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[f \mapsto \mathbf{hndl}], \sigma_H, \sigma_F[\mathbf{hndl} \mapsto \langle str, 0, 0, n \rangle] \rangle, \sigma_D \rangle, \mathbf{ok} \rangle} \quad (\text{openb-ok})$$

Figure 3-13: Semantics for opening input files

The parameters that identify the length field should be nonnegative (inspb-neg). The loop predicate and parameters should be valid (inspb-bad).

Explicit file operations

Figures 3-12–3-14 present explicit file operations using the following helper function:

$$\Lambda(l) \triangleq \begin{cases} \operatorname{argmin}_{k \geq l} \{k = n \vee \gamma(k) \in \mathit{eou} \cup \mathit{eos} \cup \mathit{osd}\}, & \text{if } ud = \langle \mathit{eou}, \mathit{eos}, \mathit{osd} \rangle \in \mathit{Delim} \\ \mathit{cut}, & \text{if } ud = \mathit{cut} \in \mathit{Cutoff} \end{cases}$$

$$(l = 0, 1, \dots, n)$$

returns the offset of the upcoming delimiter from offset l for text inputs.

and returns the upcoming cutoff position for binary inputs.

$$\begin{array}{c}
\frac{\langle e, \sigma \rangle \Downarrow_e \text{err}}{\langle \text{seek}(f, e), \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{sk-bad}) \\
\\
\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad u < \text{sou}}{\langle \text{seek}(f, e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{sk-l}) \\
\\
\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad \text{sou} \leq u \leq \Lambda(l)}{\langle \text{seek}(f, e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, u, \text{sou}, \text{ud} \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad (\text{sk-ok}) \\
\\
\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u > \Lambda(l)}{\langle \text{seek}(f, e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{sk-r}) \\
\\
\frac{\sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad l = \Lambda(l)}{\langle x = \text{read}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{frd-out}) \\
\\
\frac{\sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad l < \Lambda(l)}{\langle x = \text{read}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto \gamma(l)], \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l + 1, \text{ud} \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad (\text{frd-ok})
\end{array}$$

Figure 3-14: Semantics for accessing input files

Figure 3-12 presents the semantics for input file expressions. The `end` predicate uses the file size and the input unit descriptors from all the nested “inspect” layers to test whether the input file offset is at the end of the current input unit (`end-t`, `end-f`). The `pos` function returns the current offset of the file descriptor (`pos`). Figure 3-13 presents the semantics for opening input files. A successful `opent` statement sets the file variable to a fresh text file handler (`opent-ok`). A successful `openb` statement sets the file variable to a fresh binary file handler (`openb-ok`). Figure 3-14 presents the semantics for accessing input files. A successful `seek` statement sets the file offset to the specified position (`sk-ok`). A successful `read` operation assigns the current input byte to the receiving variable and advances the offset in the file descriptor (`frd-ok`). The new position for `seek` must be inside the current input unit (`sk-l`, `sk-r`). Likewise, the developer should not invoke `read` at a delimiter in a text file, at a cutoff position in a binary file, or at the end of the file (`frd-out`). Instead, they should use the `end` predicate to test before reading. To read multiple bytes into an array, the developer

may write a loop that reads multiple times. While another operation that is dedicated for this purpose would be expressive, the rules for this operation are complicated and do not add much insight beyond handling array bounds and input unit boundaries.

3.2.4 Properties

The semantics of RIFL ensure that scanning programs process each input unit *atomically*. We consider properties of input units identified by “inspect” loops only.

To characterize this property in detail, we first define scanning programs. A program is *scanning* if it (a) does not contain `seek` or `pos` instructions and (b) is written to visit any location in any input file at most once when there are no errors. Note that, on errors, the RIFL implementation may cause a scanning program to visit some input location multiple times. For example, if a scanning program nests two “inspect” loops for two different files, it is possible that an error in the inner “inspect” loop can cause RIFL implementation to make the program execution to revisit some locations in the file that the outer “inspect” loop processes.

Conceptually, RIFL ensures that a scanning program either (a) aborts with no output, or (b) succeeds as if the program were executed with only the good parts of the inputs that exclude bad input units. To illustrate this property more precisely, we define more terms as follows. A program *completely accepts* an input unit if the program execution successfully processes this input unit. A program *completely rejects* an input unit if the program execution discards this input unit without affecting the program state. A program *selectively accepts* an input unit if the program (a) completely rejects all the bad input units that nest inside and (b) completely accepts all other regions that allow successful execution. A *simple* input unit is an input unit that no other input units nest inside. A *composite* input unit is an input unit that contains inner input units. A scanning RIFL program behaves as follows.

1. The program either completely accepts or completely rejects any simple input unit.
2. The program either selectively accepts or completely rejects any composite input

unit.

3. If all input units are good, the program completely accepts all the input units.
4. The program always completely rejects bad input units.

Note, however, that the program may or may not accept good input units. For example, a good input unit may nest inside a bad composite input unit that the program later completely rejects.

3.3 Extensions for Explicit Error Handling

For evaluation purposes, we extend RIFL to support handling errors explicitly. In general, however, we recommend developers to use only the core language features that handle errors implicitly.

We extend RIFL to support “lookat” loops. A “lookat” loop iterates through the inputs as an “inspect” loop does, except that the “lookat” loop does not attempt to detect or recover from errors. Figures 3-15 and 3-16 present the semantics for “lookat” loops for text input files, using the same helper functions as Figures 3-8 and 3-9. Figures 3-17 and 3-18 present the semantics for “lookat” iterators for binary input files, using the same helper functions as Figures 3-10 and 3-11.

We also extend RIFL to support an additional set of system calls for file and memory operations. These extensions resemble the existing system calls when there are no errors, but return explicit error codes instead of triggering RIFL’s error recovery procedure in other situations. Table 3.1 lists the mappings from core-language system calls to their error-code interfaces. The “Core-language interface” column lists the system call constructs in the core language. The “Error-code extension” column lists the system call constructs to support explicit error handling with error codes. Each row lists the two versions of a system call. Figures 3-19–3-22 present the semantics for these additional system calls.

These extensions make it possible for a file variable to become null. For this reason, we also extend RIFL to support testing whether a file variable is not null

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{false}}{\langle \text{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{lookt-end})$$

$$\frac{\begin{array}{l} \sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \\ \Omega(d_u) \cap \Omega(d_s) = \emptyset \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle \\ \text{dlm} = \langle \Omega(d_u), \Omega(d_s), (\text{eou} \cup \text{eos} \cup \text{osd}) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{dlm} \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle \text{str}, l', \text{sou}', \langle \text{eou}', \text{eos}', \text{osd}' \rangle \rangle \\ \sigma'_D(\text{str}) = \langle \gamma', n' \rangle \quad \Lambda'(l') \geq n' \vee \gamma'(\Lambda'(l')) \in \text{osd}' \end{array}}{\langle \text{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle \text{str}, \Lambda'(l'), \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle], \sigma'_D \rangle, \text{ok} \rangle} \quad (\text{lookt-osd})$$

$$\frac{\begin{array}{l} \sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \\ \Omega(d_u) \cap \Omega(d_s) = \emptyset \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle \\ \text{dlm} = \langle \Omega(d_u), \Omega(d_s), (\text{eou} \cup \text{eos} \cup \text{osd}) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{dlm} \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle \text{str}, l', \text{sou}', \langle \text{eou}', \text{eos}', \text{osd}' \rangle \rangle \\ \sigma'_D(\text{str}) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in \text{eos}' \end{array}}{\langle \text{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle \text{str}, \Lambda'(l') + 1, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle], \sigma'_D \rangle, \text{ok} \rangle} \quad (\text{lookt-eos})$$

$$\frac{\begin{array}{l} \sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \\ \Omega(d_u) \cap \Omega(d_s) = \emptyset \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle \\ \text{dlm} = \langle \Omega(d_u), \Omega(d_s), (\text{eou} \cup \text{eos} \cup \text{osd}) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{dlm} \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle \text{str}, l', \text{sou}', \langle \text{eou}', \text{eos}', \text{osd}' \rangle \rangle \\ \sigma'_D(\text{str}) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in \text{eou}' \\ \langle \text{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle \text{str}, \Lambda'(l') + 1, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle], \sigma'_D \rangle \rangle \Downarrow_s \xi \end{array}}{\langle \text{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \xi} \quad (\text{lookt-prgr})$$

Figure 3-15: Semantics for iterators without error handling—with good text inputs

$$\begin{array}{c}
\frac{\sigma_S(f) = \text{null} \vee \sigma_S(d_u) = \text{null} \vee \sigma_S(d_s) = \text{null}}{\langle \text{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{lookt-null}) \\
\\
\frac{\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err}}{\langle \text{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{lookt-bad}) \\
\\
\frac{\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \Omega(d_u) \cap \Omega(d_s) \neq \emptyset}{\langle \text{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{lookt-dupl}) \\
\\
\frac{\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \Omega(d_u) \cap \Omega(d_s) = \emptyset \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \langle \text{eou}, \text{eos}, \text{osd} \rangle \rangle \quad \text{dlm} = \langle \Omega(d_u), \Omega(d_s), (\text{eou} \cup \text{eos} \cup \text{osd}) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \quad \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{dlm} \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \sigma', \text{bad} \rangle}{\langle \text{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \sigma', \text{bad} \rangle} \quad (\text{lookt-abort})
\end{array}$$

Figure 3-16: Semantics for iterators without error handling—with bad text inputs

$$\begin{array}{c}
\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{false}}{\langle \text{lookatb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{lookb-end}) \\
\\
\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_o \quad \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_w \quad \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_c \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{cut} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u_o \geq 0 \quad u_w \geq 0 \quad \text{parseint}(l + u_o, u_w) = v \quad \text{cut} = l \vee u_o = u_w = u_c = 0 \vee v < 0}{\langle \text{lookatb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, \text{cut}, \text{sou}, \text{cut} \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad (\text{lookb-eos}) \\
\\
\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_o \quad \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_w \quad \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_c \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{cut} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u_o \geq 0 \quad u_w \geq 0 \quad \text{parseint}(l + u_o, u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l + u_o + u_w + v + u_c = \text{cut}' \quad l < \text{cut}' \leq \text{cut} \quad \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{cut}' \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle}{\langle \text{lookatb}(e, f, o, w, c)\{s\}, \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle \text{str}, \text{cut}', \text{sou}, \text{cut}' \rangle], \sigma'_D \rangle \rangle \Downarrow_s \xi} \quad (\text{lookb-prgr})
\end{array}$$

Figure 3-17: Semantics for iterators without error handling—with good binary inputs

$$\frac{\sigma_S(f) = \text{null}}{\langle \text{lookatb}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{lookb-null})$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \begin{array}{l} \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err} \vee \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err} \\ \vee \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err} \vee \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err} \end{array}}{\langle \text{lookatb}(e, f, o, w, c)\{s\}, \sigma \rangle \Downarrow_s \langle \sigma, \text{bad} \rangle} \quad (\text{lookb-bad})$$

$$\frac{\begin{array}{l} \sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_o \\ \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_w \quad \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_c \\ \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{cut} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u_o < 0 \vee u_w < 0 \vee u_c < 0 \end{array}}{\langle \text{lookatb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{lookb-neg})$$

$$\frac{\begin{array}{l} \sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_o \\ \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_w \quad \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_c \\ \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{cut} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u_o \geq 0 \quad u_w \geq 0 \\ \text{parseint}(l + u_o, u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l + u_o + u_w + v + u_c = \text{cut}' \quad \text{cut}' > \text{cut} \end{array}}{\langle \text{lookatb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad (\text{lookb-large})$$

$$\frac{\begin{array}{l} \sigma_S(F) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true} \quad \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_o \\ \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_w \quad \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u_c \\ \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{cut} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u_o \geq 0 \quad u_w \geq 0 \\ \text{parseint}(l + u_o, u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l + u_o + u_w + v + u_c = \text{cut}' \quad l < \text{cut}' \leq \text{cut} \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l, l, \text{cut}' \rangle], \sigma_D \rangle \rangle \Downarrow_s \langle \sigma', \text{bad} \rangle \end{array}}{\langle \text{lookatb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \sigma', \text{bad} \rangle} \quad (\text{lookb-abort})$$

Figure 3-18: Semantics for iterators without error handling—with bad binary inputs

$$\begin{array}{c}
\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err}}{\langle a = \text{malloc_ec}(e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[a \mapsto \text{null}], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{malloc-ec-bad}) \\
\\
\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad u \leq 0}{\langle a = \text{malloc_ec}(e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[a \mapsto \text{null}], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{malloc-ec-neg}) \\
\\
\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad u > 0 \quad \text{heap allocate}(u) = \perp}{\langle a = \text{malloc_ec}(e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[a \mapsto \text{null}], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{malloc-ec-ovf}) \\
\\
\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad u > 0 \quad \text{heap allocate}(u) = \text{addr}}{\langle \langle \sigma_S[a \mapsto \text{addr}], \sigma_H[\text{addr} \mapsto \langle [0 \mapsto 0, 1 \mapsto 0, \dots, u-1 \mapsto 0], u \rangle], \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{malloc-ec-ok}) \\
\\
\frac{\sigma_S(a) = \text{null}}{\langle x = \text{free_ec}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{free-ec-null}) \\
\\
\frac{\sigma_S(a) \neq \text{null}}{\langle \langle \sigma_S[x \mapsto 0, a \mapsto \text{null}], \sigma_H[\sigma_S(a) \mapsto \perp], \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{free-ec-ok})
\end{array}$$

Figure 3-19: Semantics for arrays with error codes

$$\begin{array}{c}
\frac{\sigma_S(f) = \text{null}}{\langle \text{end_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e -1} \quad (\text{end-ec-null}) \\
\\
\frac{\sigma_S(f) \neq \text{null} \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad l = \Lambda(l)}{\langle \text{end_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true}} \quad (\text{end-ec-t}) \\
\\
\frac{\sigma_S(f) \neq \text{null} \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad l < \Lambda(l)}{\langle \text{end_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{false}} \quad (\text{end-ec-f}) \\
\\
\frac{\sigma_S(f) = \text{null}}{\langle \text{pos_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e -1} \quad (\text{pos-ec-null}) \\
\\
\frac{\sigma_S(f) \neq \text{null} \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle}{\langle \text{pos_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e l} \quad (\text{pos-ec-ok})
\end{array}$$

Figure 3-20: Semantics for input file expressions with error codes

file str does not exist	
$\langle f = \text{opent_ec}(str), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[f \mapsto \text{null}], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle$	(opent-ec-bad)
file str exists	
$\langle f = \text{opent_ec}(str), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[f \mapsto \text{hdl}], \sigma_H, \sigma_F[\text{hdl} \mapsto \langle str, 0, 0, \langle \emptyset, \emptyset, \emptyset \rangle] \rangle, \sigma_D \rangle, \text{ok} \rangle$	(opent-ec-ok)
file str does not exist	
$\langle f = \text{openb_ec}(str), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[f \mapsto \text{null}], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle$	(openb-ec-bad)
file str exists $\sigma_D(str) = \langle \gamma, n \rangle$	
$\langle f = \text{openb_ec}(str), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[f \mapsto \text{hdl}], \sigma_H, \sigma_F[\text{hdl} \mapsto \langle str, 0, 0, n \rangle] \rangle, \sigma_D \rangle, \text{ok} \rangle$	(openb-ec-ok)

Figure 3-21: Semantics for opening input files with error codes

Table 3.1: Error-code interfaces for system calls

Core-language interface	Error-code extension
$a = \text{malloc}(e)$	$a = \text{malloc_ec}(e)$
$\text{free}(a)$	$x = \text{free_ec}(a)$
$\text{end}(f)$	$\text{end_ec}(f)$
$\text{pos}(f)$	$\text{pos_ec}(f)$
$f = \text{opent}(str)$	$f = \text{opent_ec}(str)$
$f = \text{openb}(str)$	$f = \text{openb_ec}(str)$
$\text{seek}(f, e)$	$x = \text{seek_ec}(f, e)$
$x = \text{read}(f)$	$x = \text{read_ec}(f)$

with the valid predicate. Figure 3-23 presents the semantics for this feature. We also need to extend the semantics for core-language constructs. Specifically, the rules for `inspectt`, `inspectb`, `end`, `pos`, `seek`, and `read` should all additionally consider the situation where $\sigma_S(f) = \text{null}$.

3.4 Discussion

RIFL filtered iterators process each input unit atomically: They completely discard the bad input units that programs cannot process. After discarding, they resume the programs' execution. There are two major aspects for alternative error-recovery

$$\frac{\sigma_S(f) = \text{null}}{\langle x = \text{seek_ec}(f, e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{sk-ec-null})$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{err}}{\langle x = \text{seek_ec}(f, e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{sk-ec-bad})$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad u < \text{sou}}{\langle x = \text{seek_ec}(f, e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{sk-ec-l})$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad \text{sou} \leq u \leq \Lambda(l)}{\langle x = \text{seek_ec}(f, e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto u], \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, u, \text{sou}, \text{ud} \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad (\text{sk-ec-ok})$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e u \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad u > \Lambda(l)}{\langle x = \text{seek_ec}(f, e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{sk-ec-r})$$

$$\frac{\sigma_S(f) = \text{null}}{\langle x = \text{read_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{frd-ec-null})$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad l = \Lambda(l)}{\langle x = \text{read_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad (\text{frd-ec-out})$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_F(\sigma_S(f)) = \langle \text{str}, l, \text{sou}, \text{ud} \rangle \quad \sigma_D(\text{str}) = \langle \gamma, n \rangle \quad l < \Lambda(l)}{\langle x = \text{read_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto \gamma(l)], \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle \text{str}, l+1, \text{ud} \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad (\text{frd-ec-ok})$$

Figure 3-22: Semantics for accessing input files with error codes

$$\frac{\sigma_S(f) \neq \text{null}}{\langle \text{valid}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{true}} \quad (\text{fvalid-t})$$

$$\frac{\sigma_S(f) = \text{null}}{\langle \text{valid}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_e \text{false}} \quad (\text{fvalid-f})$$

Figure 3-23: Semantics for expressions to support error-code extensions

strategies.

Some programs may accept partial outputs and partial updates. This partial strategy may be desirable if

1. The developers are confident that partial updates would only affect the program execution in certain expected ways.
2. Consumer programs are designed to correctly process partial outputs from these programs.

This alternative strategy has better performance and is easier to implement manually than the atomic strategy. For example, there is no need to manually maintain the output quality by buffering partial outputs for loops and recursive functions. However, developers using this strategy also need to be more cautious and ensure that bad input units will not trigger unexpected behavior or other vulnerabilities. The atomic recovery strategy in filtered iterators is convenient for programs that desire good output quality and wish to completely discard bad updates.

Some programs may wish to exit on whatever errors they encounter. This fail-fast strategy is reasonable for situations where the integrity of the inputs is paramount, or where the inputs become worthless whenever there are errors. For example, some data compression algorithms that encode information across entire files may demand that the entire compressed files are intact. On the other hand, the tolerating strategy in filtered iterators is convenient for programs that wish to process as many input units as possible. For example, a network packet analyzer that helps diagnose network problems should not abort on malformed network packets. As another example, a file archiving application should not refuse to extract all the user files when only one of them is corrupted.

Chapter 4

Experiments

We evaluate the effectiveness of filtered iterators by comparing versions of the same RIFL programs that differ only in strategies of handling inputs.

4.1 Benchmarks

We implemented seven benchmark applications that exhibit the pattern of input units. Each benchmark parses a structured input format and outputs some nontrivial information for each input unit. There are three text formats: Comma Separated Values (CSV), a three-dimensional geometry definition file format called OBJ, and JavaScript Object Notation (JSON).

CSV: CSV files store tables. A CSV file contains a title line followed by content lines. Each line contains fields that are separated by commas.

The benchmark program reads a CSV file, expecting that each field is at most 10 characters long, and that each content line has the same number of fields as the title line. The program outputs the fields and lines that do not trigger errors. In effect, the program discards fields that are longer than 10 characters, trailing fields that would otherwise make a line contain too many fields, and content lines that do not have enough fields.

OBJ: OBJ files express the shapes of objects. An OBJ file contains lines that

each defines a vertex or a face. Each vertex definition contains the character 'v' followed by three space-delimited decimal numbers that represent a three-dimensional position. Each face definition contains the character 'f' followed by at least three space-delimited positive integers that each refer back to a vertex. Specifically, an integer i refers to the i -th vertex that the preceding inputs define.

The benchmark program reads an OBJ file, expecting that each decimal number is at most 10 characters long. The program also checks to ensure that each integer in a face definition is at most the number of existing vertices. The program outputs the definitions of vertices and faces that do not trigger errors. In effect, the program discards the malformed numbers, the vertices that do not contain exactly three dimensions, and the faces that do not have at least three valid vertex references.

JSON: JSON files describe object attributes. A JSON file contains a unit which we define as follows. A token is a string that does not contain commas, colons, brackets, or braces. A key-value pair contains a token, a colon, and a unit. An object is at least one key-value pair surrounded by braces. An array is at least one unit surrounded by brackets. A unit can be a token, an object or an array. Commas separate the key-value pairs inside objects and the units inside arrays. The benchmark program reads a JSON file, expecting that there are at most 10 tokens, and that each token is at most 20 characters long. The program outputs the contents of inputs that do not trigger errors. In effect, the program discards tokens that are longer than 20 characters, malformed key-value pairs in objects, malformed units in arrays, and units that would otherwise make the program store more than 10 tokens.

The other four formats are binary: Portable Network Graphics (PNG), an archive file format called ZIP, a customized resilient Graphics Interchange Format (GIF), and Domain Name System (DNS) packets in a network packet capture format called PCAP. We use RGIF to denote the customized resilient GIF format. We use PCAP/DNS to

denote the format of DNS packets inside PCAP files.

PNG: PNG files store images. A PNG file contains a magic string followed by chunks. Each chunk contains a 4-byte nonnegative length field, a 4-byte type field, “length” bytes of data, and a 4-byte cyclic redundancy code (CRC).

The benchmark program reads a PNG file, expecting that each chunk is small enough to fit in the heap memory. When a chunk length is too large, the program flushes the input until the end of the chunk and keeps parsing. The program outputs the data contents of all chunks with type “IDAT” that do not trigger errors. The program rejects PNG files with malformed headers. The program discards trailing inputs after seeing a chunk with a negative length.

ZIP: ZIP files archive user files. A ZIP file contains a portion of archive data, a central directory, and an end of central directory (EOCD) record. The archive portion contains the archived user files. The central directory contains a list of records. Each record in the central directory describes metadata of a user file and the starting offset of this user file in the archive portion. The EOCD record is at the end of the ZIP file and describes the starting offset of the central directory.

The benchmark program reads a ZIP file, iterating over the records in the central directory. The program expects that each file name and the contents for each archived user file are both short enough to fit in the heap memory. When a file name is too long, the program flushes the input until the end of the record and keeps parsing. When the contents of an archived user file is too large, the program stops parsing the current record and continues with the next record. The program outputs (a) the file names as listed the central directory and (b) the file names and the contents in archived user files. The program rejects ZIP files with malformed EOCD records. The program discards trailing inputs after seeing a record with a negative file-name length.

RGIF: RGIF files store animated images. RGIF is based on the GIF format. An GIF file contains header information followed by blocks. Each block starts with

one or two bytes that describe the block type, such as image data, metadata, or other extensions. Each image may contain several consecutive blocks. The RGIF format differs from GIF only in that it adds two bytes in front of each image to describe the total length of all the blocks for the image.

The benchmark program reads an RGIF file and converts it to the GIF format. When an image is corrupted, the program discards the image and keeps parsing. The program rejects RGIF files with malformed headers. The program discards trailing inputs after seeing an image with negative length.

PCAP/DNS: PCAP files store network packets. A PCAP file contains a header and capture items. Each capture item contains 12 bytes of metadata, a 4-byte length field, and a network packet of “length” bytes. The PCAP/DNS benchmark considers only DNS packets. Specifically, each network packet of interest contains an Ethernet header, an Internet Protocol version 4 (IPv4) header, a User Datagram Protocol (UDP) header, the DNS packet, and some trailing Ethernet bytes.

The benchmark program reads a PCAP file and extracts DNS packets. For each valid DNS packet, the program prints the source Internet Protocol (IP) address, the destination IP address, the DNS identification number, and the DNS questions and answers. When a network packet is malformed, the program discards the capture item and keeps parsing. The program rejects PCAP files with malformed headers. The program discards trailing inputs after seeing a capture item with negative length.

4.2 Experimental Setup

This section describes the independent variables and dependent variables for our experiments.

4.2.1 Independent variables

For each benchmark application, we built four different versions that have the same functionality. Each version uses one of the following strategies to handle inputs.

1. RIFL’s automatic error-handling strategy, or the *fully-implicit* version. This version uses “inspect” loops to automatically extract input units, to detect bad units, and to recover from bad units.
2. RIFL’s automatic error-recovery strategy with manual error detection, or the *protective-check* version. This version identifies bad inputs with (ideally) exhaustive error checks and assertions, so that the execution triggers only assertion failures but no other runtime errors. On assertion failures, “inspect” loops automatically recover the program from bad input units. One way to implement this version is to add assertions to explicitly exclude bad input units that trigger errors in the fully-implicit version.
3. RIFL’s iterator structure but with fully manual error handling, or the *explicit-recovery* version. This version uses “lookat” loops instead of “inspect” loops to iterate over input units. These iterators still automatically extract input units, but do not recover from errors. One way to implement this version is to add error recovery procedures to the protective-check version.
4. Only conventional language constructs, or the *plain-loop* version. There are two major design choices that developers can make to iterate over the inputs using conventional language constructs. One choice is to use plain loops that both extract and process input units. This manual approach is more flexible in file-pointer movements, but would also require more effort to implement correctly. For example, programs for input units with delimiters need to explicitly check the delimiters for each byte it reads. Programs for input units with length fields also need to be careful about bad input units whose contents are inconsistent with their lengths. The other choice is to preload each input unit into a buffer and then process the buffer. This structure is natural for input units with

length fields, but also works for those with delimiters. This method avoids the runtime overhead that RIFL interpreter imposes on each read operation. Developers may also focus on the contents of each input unit: on errors, the file pointers are already at the desired locations. A disadvantage of this structure is the need to manually maintain input buffers, such as deciding their sizes and cleaning up.

Ideally, all versions of the same benchmark should produce the same outputs on all inputs. In practice, it is sometimes inappropriate to require exactly the same outputs. An example situation is when an input file significantly differs from the desired input format and a program wishes to reject the file completely. The fully-implicit version may naturally express this functionality with assertions outside “inspect” loops. When an input file violates these assertions, the program terminates with an error and with no output. On the other hand, the explicit-recovery and plain-loop versions handle all errors manually. If we strictly enforce the same outputs, these versions need to ensure that the program produces no output on rejections. It is more natural to reject the file using `exit` statements that immediately terminates with an error code and possibly with some partial outputs. We allow such optimizations in our experiments.

Table 4.1 summarizes the language restrictions for the four versions of each application. The “Loops” column and the “System calls” column contain the available constructs for loops and system calls, respectively. The “Fully-implicit” row, the “Protective-check” row, the “Explicit-recovery” row, and the “Plain-loop” row correspond to the four versions with the same names.

Appendix A presents the source code for all our benchmark programs.

4.2.2 Dependent variables

We measure the difficulty of program implementations from mainly two aspects: the *control flow* and the *data manipulation*. The complexity of these two aspects roughly correspond to the difficulty of error detection and error recovery.

Table 4.1: Language restrictions for four versions

	Loops	System calls
Fully-implicit	<code>inspectt</code> , <code>inspectb</code> , <code>lookatt</code> , <code>lookatb</code> , <code>while</code>	<code>assert</code> , <code>malloc</code> , <code>free</code> , <code>end</code> , <code>pos</code> , <code>opent</code> , <code>openb</code> , <code>seek</code> , <code>read</code>
Protective-check	<code>inspectt</code> , <code>inspectb</code> , <code>lookatt</code> , <code>lookatb</code> , <code>while</code>	<code>assert</code> , <code>malloc_ec</code> , <code>free_ec</code> , <code>end_ec</code> , <code>pos_ec</code> , <code>opent_ec</code> , <code>openb_ec</code> , <code>seek_ec</code> , <code>read_ec</code>
Explicit-recovery	<code>lookatt</code> , <code>lookatb</code> , <code>while</code>	<code>malloc_ec</code> , <code>free_ec</code> , <code>end_ec</code> , <code>pos_ec</code> , <code>opent_ec</code> , <code>openb_ec</code> , <code>seek_ec</code> , <code>read_ec</code>
Plain-loop	<code>while</code>	<code>malloc_ec</code> , <code>free_ec</code> , <code>end_ec</code> , <code>pos_ec</code> , <code>opent_ec</code> , <code>openb_ec</code> , <code>seek_ec</code> , <code>read_ec</code>

Control-flow complexity: To estimate the difficulty of error detection, we observe the number of conditional clauses in programs. We collect this number by counting the number of `if` statements, `assert` statements, logical conjunctions, and logical disjunctions. This number is roughly the number of situations that developers need to consider, regardless of their coding style.

Data-manipulation complexity: To estimate the difficulty of error recovery, we observe the number of lines of code for unconditional computation. We collect this number by counting the number of statements or control constructs except for `if` and `assert`. The difference in this number across different versions of the same benchmark indicates the lines of additional computation needed to handle errors, such as for maintaining the atomicity of program states or maintaining file pointers. Inaccuracies may happen when each version tailors its control flow according to its input-handling strategy. Our experiments allow such optimizations.

For generality, we also measure the number of lines of code for the programs. This measurement considers all lines in the source code, including the lines that contain

no statements.

4.3 Results

Figures 4-1–4-3 present the measurements for all versions of our benchmarks. The vertical axes in the three figures represent control complexity, data complexity, and lines of code, respectively. Each bar represents a benchmark. Inside each bar, different colors represent the differences between the four versions. Specifically, the blue portion represents the complexity measurement of the fully-implicit version. The green portion represents the amount by which the protective-check version increases beyond the fully-implicit version. The red portion represents the amount by which the explicit-recovery version increases further beyond the protective-check version. The purple portion represents the amount by which the plain-loop version increases further beyond the explicit-recovery version. Each number represents the amount of the corresponding portion.

The heights of the purple, red, and green portions indicate the benefits of iterators without automatic error handling, of automatic error recovery without automatic error detection, and of automatic error detection, respectively.

We compare these differences and summarize the results in Tables 4.2–4.4. These three tables present the average percentage decrease in control complexity, data complexity, and lines of code from each version. The “Text formats” columns contain averages over benchmarks CSV, OBJ, and JSON. The “Binary formats” columns contain averages over benchmarks PNG, ZIP, RGIF and PCAP/DNS. The “Overall” columns contain averages over all seven benchmarks. The “Iterator” rows represent the average percentage decrease in complexity from plain-loop versions to explicit-recovery versions. The “Automatic recovery” rows represent the average percentage decrease in complexity from explicit-recovery versions to protective-check versions. The “Automatic detection” rows represent the average percentage decrease in complexity from protective-check versions to fully-implicit versions. The “Overall” rows represent the average percentage decrease in complexity from plain-loop versions to

Conditional clauses

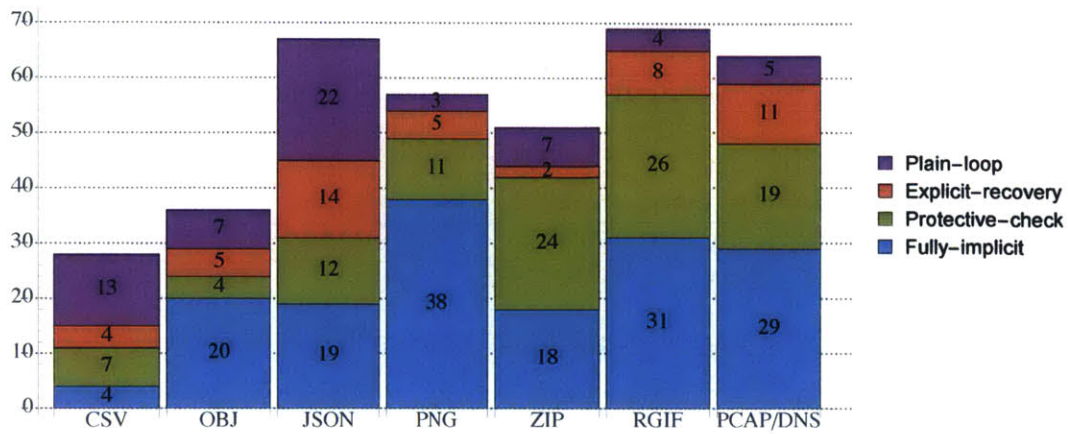


Figure 4-1: Control-flow complexity

Unconditional computation

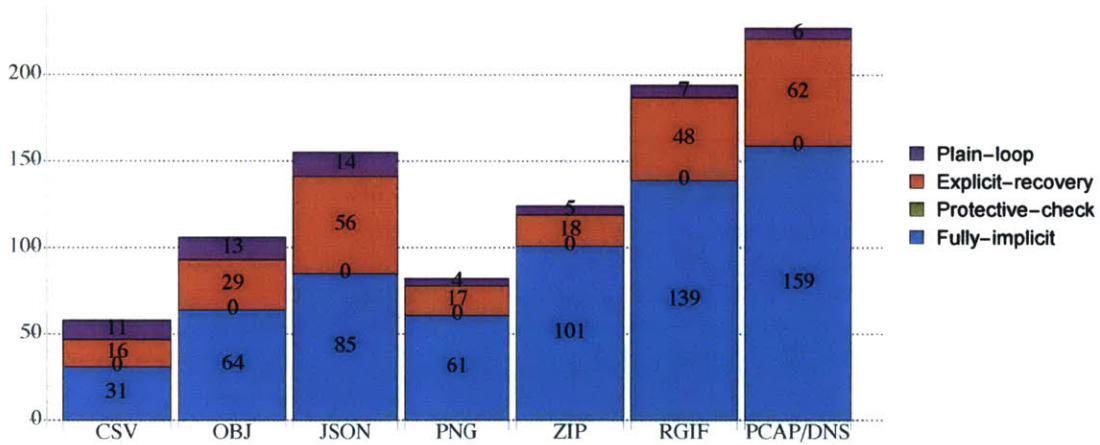


Figure 4-2: Data-manipulation complexity

Lines of code

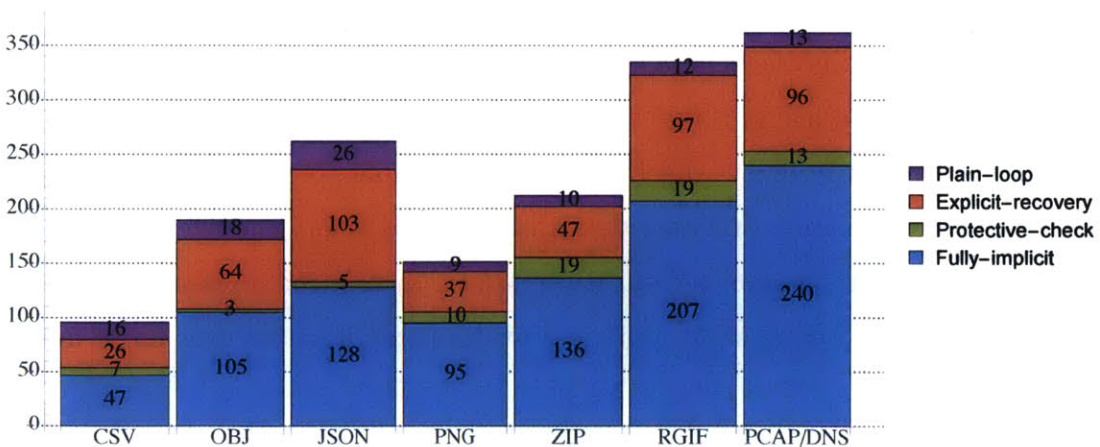


Figure 4-3: Lines of code

Table 4.2: Average decrease in control-flow complexity

	Text formats	Binary formats	Overall
Iterator	32.9%	8.1%	18.8%
Automatic recovery	25.0%	11.2%	17.1%
Automatic detection	39.7%	41.2%	40.5%
Overall	67.3%	51.9%	58.5%

Table 4.3: Average decrease in data-manipulation complexity

	Text formats	Binary formats	Overall
Iterator	13.4%	3.8%	7.9%
Automatic recovery	35.0%	22.7%	27.9%
Automatic detection	0.0%	0.0%	0.0%
Overall	43.8%	25.6%	33.4%

Table 4.4: Average decrease in lines of code

	Text formats	Binary formats	Overall
Iterator	12.0%	4.5%	7.7%
Automatic recovery	37.8%	26.7%	31.5%
Automatic detection	6.5%	8.8%	7.8%
Overall	49.0%	36.2%	41.7%

fully-implicit versions.

To observe the effects of filtered iterators from both control and data aspects, we visualize these numbers in Figure 4-4. The vertical axis represents control complexity. The horizontal axis represents data complexity. Each data point represents a version of a benchmark. Each line connects three versions of one benchmark: the fully-implicit, the explicit-recovery, and the plain-loop versions from lower left to upper right. We mark text input formats with dashed lines and binary ones with solid lines. On each line, the two segments from lower left to upper right represent the extra work for manually filtering input units and for manually iterating over input units, respectively.

The relative position of each point from the origin point indicates the effort needed to implement the program from scratch. The large or small slope from the origin point

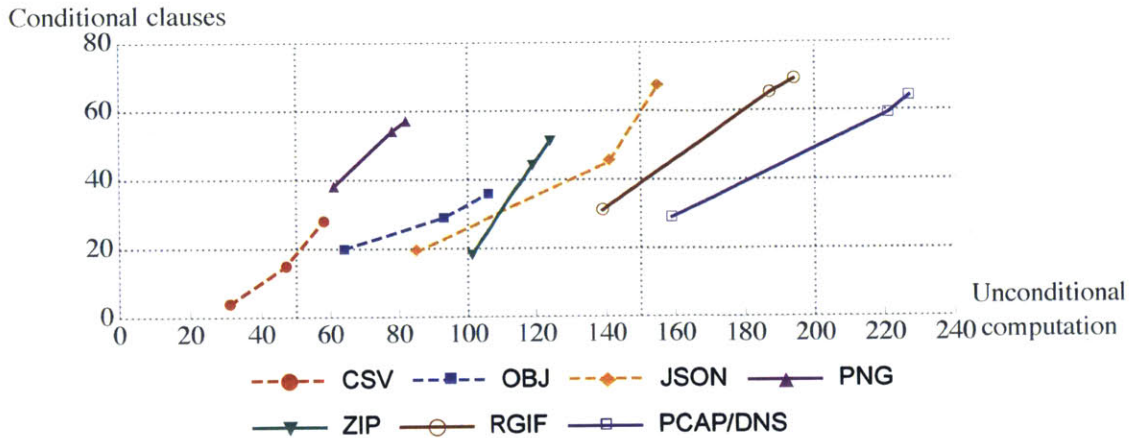


Figure 4-4: Effects of filtering and iterating

to a data point indicates whether the program is control-centric or data-centric. With the same or similar slopes, as for the fully-implicit JSON parser and the fully-implicit RGIF parser, a longer distance indicates a higher implementation complexity. With different slopes, comparisons should be made case by case.

4.4 Discussion

Filtered iterators reduce an average of 58.5% conditional clauses and 33.4% unconditional computation from the plain-loop versions. An alternative viewpoint is that filtered iterators reduce an average of 41.7% lines of code from the plain-loop versions. Specifically,

Iterators: The use of iterators without automatic error handling reduces an average of 32.9% conditional clauses and 13.4% unconditional computation for text formats. This benefit is less significant but still exists for binary formats, reducing 8.1% and 3.8%, respectively. We attribute these improvements to the elimination of the need to manually check the boundaries when reading input units.

Automatic recovery: The use of automatic error recovery without automatic error detection further reduces an overall average of 17.1% conditional clauses and 27.9%

unconditional computation. We attribute these improvements to the elimination of the need to manually maintain program states and file pointers for bad input units. Inappropriate maintenance may lead to unexpected partial updates and partial outputs.

Automatic detection: The use of automatic error detection further reduces an overall average of 40.5% conditional clauses. We attribute this improvement to the elimination of the need to manually consider error situations. For example, the protective checks may ensure that a `read_ec` operation succeeds, that a `malloc_ec` operation returns a valid array, or that a global array is large enough. Missing such checks in a conventional programming language can lead to program crashes. RIFL filtered iterators implicitly capture and handle all these errors.

All benchmarks except for OBJ show that the extra code for filtering input units is slightly shorter but more control-centric than the basic, fully implicit code. This observation indicates that such error-handling code is nontrivial to implement and tends to be more critical than the common-case code. This trend is consistent with the real world experience that error-handling code tends to be harder to implement correctly.

Explicit error detection in the OBJ benchmark incurs relatively few extra conditional clauses. This small difference is because there are many ways an OBJ file can format badly but without causing the parser to crash. The fully implicit OBJ parser already checks many conditions to enforce the input format, while the explicit versions add only a few more.

Explicit error recovery in the ZIP benchmark introduces relatively little extra code for data manipulation. This small difference is because the ZIP parser in our benchmark does little computation except for extracting fields from the inputs. Thus, there is little need to maintain external states across input units.

Explicit error handling in the JSON benchmark significantly increases both control complexity and data complexity. This significant amount of error-handling code is because JSON's recursive structure requires the explicit-recovery version to maintain

global updates similarly to a stack.

Another observation about the JSON benchmark is that the extra code to manually iterate over input units is heavily control-centric with little extra computation. Such code is critical but appeared easier to implement than the error-handling code. This observation is consistent with our understanding that the implementation complexity comes from both conditional clauses and unconditional computation.

Chapter 5

Related Work

Researchers have proposed ways to recover software from errors. Many of them are external to the language definition.

Transactional recovery techniques such as backward recovery [4] and forward recovery [18] recover programs from transient errors. Rx [27] rollbacks and replays programs in new configurations to survive failures. These techniques are unlikely to solve deterministic errors. Filtered iterators capture all detectable errors and recovery from them by discarding bad inputs and restarting loop iterations.

Input filtering systems [37, 6, 3, 5, 35] generate vulnerability signatures from known attacks and drop malicious inputs accordingly. These techniques are unsound—they cannot block all attacks. SIFT [24] is sound, but only prevents integer overflow errors. Input rectification [29, 23] prevents program failures by modifying bad inputs to good ones using prior knowledge about benign inputs. RIFL implementation dynamically captures and recovers from all detectable vulnerabilities with neither false positives nor false negatives. RIFL also uses programs themselves as filters, without external knowledge.

Failure-oblivious techniques [28, 25] purposefully change the program semantics to survive inputs that the program would otherwise be unable to process. Error virtualization [31, 32] recovers program execution by turning function executions into transactions and mapping faults into return values used by the application code. The effects of these techniques may not be clear to the users. RIFL, on the other hand,

handles errors with formally-defined filtered iterators that provide the atomicity property of input units.

Researchers have also proposed language-based techniques to error recovery.

Recovery blocks [1] and N-version programming [2] tolerate software faults using multiple implementations of the same component. This methodology adds significant software development costs. Exception handling [7] improves program structures by separating common-case and error-handling code. This structure requires developers to anticipate the exceptional events and maintain states accordingly. In contrast to these methodologies, RIFL aims at eliminating the need for error-handling code. RIFL's recovery strategy primarily focuses on programs that process inputs in input units. This strategy enables RIFL to encapsulate local errors with the atomicity property.

Bristlecone [8, 10, 9] ensures error-free execution using decoupled transactional tasks according to high-level task specifications. RIFL and Bristlecone both indicate that discarding some of the computation from the conceptual level has the potential in helping programs survive failures. RIFL explicitly supports discarding inputs in combination to computation, using formal semantics and providing the atomicity property. We also conduct quantitative experiments and gain insight into how language-based error-handling techniques affect the development of robust programs. Our benchmarks show that RIFL filtered iterators can help eliminate a significant amount of control complexity and data complexity from programs that process input units.

Chapter 6

Conclusion and Future Work

RIFL is a language that encourages developers to write only common-case code. RIFL integrates error-detection and error-recovery strategies into the language with filtered iterators, utilizing the patterns of input units. Preliminary results show that using filtered iterators significantly reduces both the number of conditional clauses and the amount of unconditional computation from fully manual implementations.

Further research directions include performance, generalization and consistency.

Performance: While RIFL can simplify the development of robust programs, it trades off performance for robustness. Dynamic error detection and transactional error recovery both have intrinsic performance issues. One possible direction to reduce these overheads is to eliminate dynamic checks based on static analysis techniques, so that a program may perform external updates only when the input unit is guaranteed to be good.

Generalization: The notion of filtered iterators may generalize beyond input files, such as for iterating over elements of any collections. If the program fails in an iteration, it would undo updates and skip the bad element in the collection. This generalization would support a wider range of programs that prefer loading the input file into data structures before performing critical computation.

Consistency: An immediate challenge with this generalization is consistency. For example, to process input formats that start with a field describing the number of input units that follow, developers may desire that this field changes consistently when discarding data structure elements. It is generally difficult to manipulate program states with strong consistency guarantees. Further research may explore this goal by expressing more information about the inputs with the language.

Appendix A

Source Code for Benchmarks

This appendix presents the source code for all of our benchmarks. Section 4.1 describes the seven input formats and applications. Section 4.2.1 describes the four implementations for each application.

A.1 CSV

A.1.1 Fully-implicit version

```
1  main {
2    f = opent("../inputs/csv/newlines.csv");
3
4    // titles
5    columns = 0;
6    inspectt (1, f, ',', '\n') {
7        title = malloc(10);
8        i = 0;
9        while (!end(f)) {
10           x = read(f);
11           title[i] = x;
12           i = i + 1;
13        }
14        if (columns > 0) {
15           print(' ');
16        }
17        print(title);
18        free(title);
19        columns = columns + 1;
20    }
21    print(' \n');
22    print(' \n');
23
24    // contents
25    assert(columns > 0);
26    inspectt (!end(f), f, '\n') {
27        j = 0;
28        inspectt (j < columns, f, ',') {
29           field = malloc(10);
30           i = 0;
```

```

31     while (!end(f)) {
32         x = read(f);
33         field[i] = x;
34         i = i + 1;
35     }
36     if (j > 0) {
37         print(',');
38     }
39     print(field);
40     free(field);
41     j = j + 1;
42 }
43 print('\n');
44 assert(j == columns);
45 }
46 return 0;
47 }

```

A.1.2 Protective-check version

```

1 main {
2     f = opent_ec("../inputs/csv/newlines.csv");
3     assert(valid(f));
4
5     // titles
6     columns = 0;
7     inspectt (1, f, ',', '\n') {
8         title = malloc_ec(10);
9         assert(valid(title));
10        i = 0;
11        while (!end_ec(f)) {
12            assert(i < 10);
13            x = read_ec(f);
14            assert(x >= 0);
15            title[i] = x;
16            i = i + 1;
17        }
18        if (columns > 0) {
19            print(',');
20        }
21        print(title);
22        x = free_ec(title);
23        columns = columns + 1;
24    }
25    print('\n');
26    print('\n');
27
28    // contents
29    assert(columns > 0);
30    inspectt (!end_ec(f), f, '\n') {
31        j = 0;
32        inspectt (j < columns, f, ',', '\n') {
33            field = malloc_ec(10);
34            assert(valid(field));
35            i = 0;
36            while (!end_ec(f)) {
37                assert(i < 10);
38                x = read_ec(f);
39                assert(x >= 0);
40                field[i] = x;
41                i = i + 1;
42            }
43            if (j > 0) {
44                print(',');
45            }
46            print(field);
47            x = free_ec(field);
48            j = j + 1;
49        }
50        print('\n');
51        assert(j == columns);
52    }
53    return 0;

```


54 }

A.1.3 Explicit-recovery version

```
1 main {
2   f = opent_ec("../inputs/csv/newlines.csv");
3   if (!valid(f)) {
4     exit(1);
5   }
6
7   // titles
8   columns = 0;
9   lookatt (1, f, ',', '\n') {
10    title = malloc_ec(10);
11    if (valid(title)) {
12      bad = 0;
13      i = 0;
14      while (!end_ec(f) && i < 10) {
15        x = read_ec(f);
16        if (x < 0) {
17          bad = 1;
18        }
19        title[i] = x;
20        i = i + 1;
21      }
22      if (end_ec(f) && !bad) {
23        if (columns > 0) {
24          print(',');
25        }
26        print(title);
27        columns = columns + 1;
28      } // skip unit
29      x = free_ec(title);
30    } // skip unit
31  }
32  print('\n');
33  print('\n');
34
35  // contents
36  if (columns <= 0) {
37    exit(1);
38  }
39  buffer = malloc_ec(11 * columns);
40  if (!valid(buffer)) {
41    exit(1);
42  }
43  lookatt (!end_ec(f), f, '\n') {
44    idx = 0;
45    while (idx < 11 * columns) {
46      buffer[idx] = 0;
47      idx = idx + 1;
48    }
49    idx = 0;
50    j = 0;
51    lookatt (j < columns, f, ',') {
52      start = idx;
53      if (j > 0) {
54        buffer[idx] = ',';
55        idx = idx + 1;
56      }
57      i = 0;
58      while (!end_ec(f) && x >= 0 && i < 10) {
59        x = read_ec(f);
60        buffer[idx] = x;
61        idx = idx + 1;
62        i = i + 1;
63      }
64      if (end_ec(f) && x >= 0) {
65        j = j + 1;
66      } else { // skip unit
67        while (idx > start) {
68          buffer[idx] = 0;
69          idx = idx - 1;

```

```

70     }
71   }
72 }
73   if (j == columns) {
74     print(buffer);
75     print('\n');
76   } // skip unit
77 }
78 x = free_ec(buffer);
79 return 0;
80 }

```

A.1.4 Plain-loop version

```

1  main {
2    f = opent_ec("../inputs/csv/newlines.csv");
3    if (!valid(f)) {
4      exit(1);
5    }
6
7    // titles
8    columns = 0;
9    finish = 0;
10   while (!finish) {
11     title = malloc_ec(10);
12     if (valid(title)) {
13       bad = 0;
14       i = 1;
15       x = read_ec(f);
16       while (!bad && x != ',' && x != '\n' && i < 10) {
17         if (x < 0) {
18           bad = 1;
19         }
20         title[i] = x;
21         i = i + 1;
22         x = read_ec(f);
23       }
24       if (!bad && (x == ',' || x == '\n')) {
25         if (columns > 0) {
26           print(',');
27         }
28         print(title);
29         columns = columns + 1;
30       } // skip unit
31       while (x >= 0 && x != ',' && x != '\n') {
32         x = read_ec(f);
33       }
34       dummy = free_ec(title);
35     } // skip unit
36     if (x < 0 || x == '\n') {
37       finish = 1;
38     }
39   }
40   print('\n');
41   print('\n');
42
43   // contents
44   if (columns <= 0) {
45     exit(1);
46   }
47   buffer = malloc_ec(11 * columns);
48   if (!valid(buffer)) {
49     exit(1);
50   }
51   while (!end_ec(f)) {
52     idx = 0;
53     while (idx < 11 * columns) {
54       buffer[idx] = 0;
55       idx = idx + 1;
56     }
57     idx = 0;
58     j = 0;
59     x = 0;

```

```

60     while (j < columns && x >= 0 && x != '\n') {
61         start = idx;
62         if (j > 0) {
63             buffer[idx] = ',';
64             idx = idx + 1;
65         }
66         i = 0;
67         x = read_ec(f);
68         while (x >= 0 && x != '\n' && x != ',' && i < 10) {
69             buffer[idx] = x;
70             idx = idx + 1;
71             i = i + 1;
72             x = read_ec(f);
73         }
74         if (x == '\n' || x == ',') {
75             j = j + 1;
76         } else { // skip unit
77             while (idx > start) {
78                 buffer[idx] = 0;
79                 idx = idx - 1;
80             }
81         }
82         while (x >= 0 && x != '\n' && x != ',') {
83             x = read_ec(f);
84         }
85     }
86     if (j == columns) {
87         print(buffer);
88         print('\n');
89     } // skip unit
90     while (x >= 0 && x != '\n') {
91         x = read_ec(f);
92     }
93 }
94 x = free_ec(buffer);
95 return 0;
96 }

```

A.2 OBJ

A.2.1 Fully-implicit version

```

1  f = opent("../inputs/obj/icosahedron-2.obj");
2  num = malloc(10);
3  dim = 0;
4
5  func cleannum(dummy) {
6      i = 0;
7      while (i < 10) {
8          num[i] = 0;
9          i = i + 1;
10     }
11     return 0;
12 }
13
14 func readfloat(dummy) {
15     dummy = cleannum(0);
16
17     dot = 0;
18     i = 0;
19     x = 0;
20     sign = 1;
21     while (!end(f)) {
22         c = read(f);
23         if (c == ',') {
24             assert(dot == 0);
25             dot = 1;
26         } else {
27             if (c == '-') {

```

```

28     assert(i == 0);
29     sign = -1;
30 } else {
31     assert(c >= '0' && c <= '9');
32     x = x * 10;
33     x = x + (c - '0');
34     assert(x >= 0);
35 }
36 }
37 num[i] = c;
38 i = i + 1;
39 }
40 return x * sign;
41 }
42
43 func readidx(n) {
44     dummy = cleannum(0);
45
46     idx = 0;
47     i = 0;
48     while (!end(f)) {
49         c = read(f);
50         assert(c >= '0' && c <= '9');
51         idx = idx * 10;
52         idx = idx + (c - '0');
53         assert(idx >= 0);
54         num[i] = c;
55         i = i + 1;
56     }
57
58     assert(idx >= 1 && idx <= n);
59     return idx;
60 }
61
62 main {
63     n = 0;
64     m = 0;
65     inspectt (!end(f), f, '\n') {
66         c = read(f);
67         assert(c == 'v' || c == 'f');
68         space = read(f);
69         assert(space == ' ');
70         if (c == 'v') {
71             print('v');
72             dim = 0;
73             inspectt (1, f, ' ') {
74                 assert(!end(f));
75                 x = readfloat(0);
76                 print(' ');
77                 print(num);
78                 dim = dim + 1;
79             }
80             assert(dim == 3);
81             print('\n');
82             n = n + 1;
83         } else {
84             print('f');
85             dim = 0;
86             inspectt (1, f, ' ') {
87                 assert(!end(f));
88                 idx = readidx(n);
89                 print(' ');
90                 print(num);
91                 dim = dim + 1;
92             }
93             assert(dim >= 3);
94             print('\n');
95             m = m + 1;
96         }
97     }
98
99     print(n);
100    print('\n');
101    print(m);
102    print('\n');
103

```

```

104     return 0;
105 }

```

A.2.2 Protective-check version

```

1  f = opent_ec("../inputs/obj/icosahedron-2.obj");
2  num = malloc_ec(10);
3  dim = 0;
4
5  func cleannum(dummy) {
6      i = 0;
7      while (i < 10) {
8          num[i] = 0;
9          i = i + 1;
10     }
11     return 0;
12 }
13
14 func readfloat(dummy) {
15     dummy = cleannum(0);
16
17     dot = 0;
18     i = 0;
19     x = 0;
20     sign = 1;
21     while (!end_ec(f)) {
22         assert(i < 10);
23         c = read_ec(f);
24         if (c == '.') {
25             assert(dot == 0);
26             dot = 1;
27         } else {
28             if (c == '-') {
29                 assert(i == 0);
30                 sign = -1;
31             } else {
32                 assert(c >= '0' && c <= '9');
33                 x = x * 10;
34                 x = x + (c - '0');
35                 assert(x >= 0);
36             }
37         }
38         num[i] = c;
39         i = i + 1;
40     }
41     return x * sign;
42 }
43
44 func readidx(n) {
45     dummy = cleannum(0);
46
47     idx = 0;
48     i = 0;
49     while (!end_ec(f)) {
50         assert(i < 10);
51         c = read_ec(f);
52         assert(c >= '0' && c <= '9');
53         idx = idx * 10;
54         idx = idx + (c - '0');
55         assert(idx >= 0);
56         num[i] = c;
57         i = i + 1;
58     }
59
60     assert(idx >= 1 && idx <= n);
61     return idx;
62 }
63
64 main {
65     assert(valid(f) && valid(num));
66     n = 0;
67     m = 0;
68     inspectt (!end(f), f, '\n') {

```

```

69     c = read_ec(f);
70     assert(c == 'v' || c == 'f');
71     space = read_ec(f);
72     assert(space == ' ');
73     if (c == 'v') {
74         print('v');
75         dim = 0;
76         inspectt (1, f, ' ') {
77             assert(!end(f));
78             x = readfloat(0);
79             print(' ');
80             print(num);
81             dim = dim + 1;
82         }
83         assert(dim == 3);
84         print('\n');
85         n = n + 1;
86     } else {
87         print('f');
88         dim = 0;
89         inspectt (1, f, ' ') {
90             assert(!end(f));
91             idx = readidx(n);
92             print(' ');
93             print(num);
94             dim = dim + 1;
95         }
96         assert(dim >= 3);
97         print('\n');
98         m = m + 1;
99     }
100 }
101
102 print(n);
103 print('\n');
104 print(m);
105 print('\n');
106
107 return 0;
108 }

```

A.2.3 Explicit-recovery version

```

1 f = opent_ec("../inputs/obj/icosahedron-2.obj");
2 num = malloc_ec(100);
3 numidx = 0;
4 dim = 0;
5
6 bad = 0;
7
8 func cleannum(dummy) {
9     i = 0;
10    while (i < 100) {
11        num[i] = 0;
12        i = i + 1;
13    }
14    numidx = 0;
15    return 0;
16 }
17
18 func putnum(c) {
19     if (numidx >= 100) {
20         bad = 1;
21     } else {
22         num[numidx] = c;
23         numidx = numidx + 1;
24     }
25     return 0;
26 }
27
28 func revertnum(start) {
29     while (numidx > start) {
30         numidx = numidx - 1;

```

```

31     num[numidx] = 0;
32 }
33 return 0;
34 }
35
36 func readfloat(dummy) {
37     dummy = putnum(' ');
38
39     dot = 0;
40     i = 0;
41     x = 0;
42     sign = 1;
43     while (!end_ec(f)) {
44         c = read_ec(f);
45         if (i >= 10 || numidx >= 100) {
46             bad = 1;
47         } else {
48             if (c == ',.') {
49                 if (dot != 0) {
50                     bad = 1;
51                 }
52                 dot = 1;
53             } else {
54                 if (c == '-') {
55                     if (i != 0) {
56                         bad = 1;
57                     }
58                     sign = -1;
59                 } else {
60                     if (c >= '0' && c <= '9') {
61                         x = x * 10;
62                         x = x + (c - '0');
63                         if (x < 0) {
64                             bad = 1;
65                         }
66                     } else {
67                         bad = 1;
68                     }
69                 }
70             }
71             dummy = putnum(c);
72             i = i + 1;
73         }
74     }
75     return x * sign;
76 }
77
78 func readidx(n) {
79     dummy = putnum(' ');
80
81     idx = 0;
82     i = 0;
83     while (!end_ec(f)) {
84         c = read_ec(f);
85         if (i >= 10 || numidx >= 100) {
86             bad = 1;
87         } else {
88             if (c >= '0' && c <= '9') {
89                 idx = idx * 10;
90                 idx = idx + (c - '0');
91                 if (idx < 0) {
92                     bad = 1;
93                 }
94             } else {
95                 bad = 1;
96             }
97         }
98         dummy = putnum(c);
99         i = i + 1;
100     }
101
102     if (!(idx >= 1 && idx <= n)) {
103         bad = 1;
104     }
105     return idx;
106 }

```

```

107
108 main {
109     if (!(valid(f) && valid(num))) {
110         exit(1);
111     }
112     n = 0;
113     m = 0;
114     lookatt (!end_ec(f), f, '\n') {
115         c = read_ec(f);
116         if (c == 'v' || c == 'f') {
117             space = read_ec(f);
118             if (space == ' ') {
119                 if (c == 'v') {
120                     dim = 0;
121                     dummy = cleannum(0);
122                     lookatt (1, f, ' ') {
123                         bad = 0;
124                         if (!end_ec(f)) {
125                             start = numidx;
126                             x = readfloat(0);
127                             if (!bad) {
128                                 dim = dim + 1;
129                             } else { // discard bad updates
130                                 dummy = revertnum(start);
131                             }
132                         }
133                     }
134                     if (dim == 3) {
135                         print('v');
136                         print(num);
137                         print('\n');
138                         n = n + 1;
139                     } // skip unit
140                 } else {
141                     dim = 0;
142                     dummy = cleannum(0);
143                     lookatt (1, f, ' ') {
144                         bad = 0;
145                         if (!end_ec(f)) {
146                             start = numidx;
147                             idx = readidx(n);
148                             if (!bad) {
149                                 dim = dim + 1;
150                             } else { // discard bad updates
151                                 dummy = revertnum(start);
152                             }
153                         }
154                     }
155                     if (dim >= 3) {
156                         print('f');
157                         print(num);
158                         print('\n');
159                         m = m + 1;
160                     } // skip unit
161                 }
162             } // skip unit
163         } // skip unit
164     }
165
166     print(n);
167     print('\n');
168     print(m);
169     print('\n');
170
171     return 0;
172 }

```

A.2.4 Plain-loop version

```

1 f = opent_ec("../inputs/obj/icosahedron-2.obj");
2 num = malloc_ec(100);
3 numidx = 0;
4 dim = 0;

```



```

5
6 bad = 0;
7 endlime = 0;
8
9 func cleannum(dummy) {
10     i = 0;
11     while (i < 100) {
12         num[i] = 0;
13         i = i + 1;
14     }
15     numidx = 0;
16     return 0;
17 }
18
19 func putnum(c) {
20     if (numidx >= 100) {
21         bad = 1;
22     } else {
23         num[numidx] = c;
24         numidx = numidx + 1;
25     }
26     return 0;
27 }
28
29 func revertnum(start) {
30     while (numidx > start) {
31         numidx = numidx - 1;
32         num[numidx] = 0;
33     }
34     return 0;
35 }
36
37 func readfloat(dummy) {
38     dummy = putnum(' ');
39
40     dot = 0;
41     i = 0;
42     x = 0;
43     sign = 1;
44     c = read_ec(f);
45     while (c >= 0 && c != '\n' && c != ' ') {
46         if (i >= 10 || numidx >= 100) {
47             bad = 1;
48         } else {
49             if (c == '.') {
50                 if (dot != 0) {
51                     bad = 1;
52                 }
53                 dot = 1;
54             } else {
55                 if (c == '-') {
56                     if (i != 0) {
57                         bad = 1;
58                     }
59                     sign = -1;
60                 } else {
61                     if (c >= '0' && c <= '9') {
62                         x = x * 10;
63                         x = x + (c - '0');
64                         if (x < 0) {
65                             bad = 1;
66                         }
67                     } else {
68                         bad = 1;
69                     }
70                 }
71             }
72             dummy = putnum(c);
73             i = i + 1;
74         }
75         c = read_ec(f);
76     }
77
78     if (c != ' ') {
79         endlime = 1;
80     }

```

```

81     if (i <= 0) {
82         bad = 1;
83     }
84     return x * sign;
85 }
86
87 func readidx(n) {
88     dummy = putnum(' ');
89
90     idx = 0;
91     i = 0;
92     c = read_ec(f);
93     while (c >= 0 && c != '\n' && c != ' ') {
94         if (i >= 10 || numidx >= 100) {
95             bad = 1;
96         } else {
97             if (c >= '0' && c <= '9') {
98                 idx = idx * 10;
99                 idx = idx + (c - '0');
100                 if (idx < 0) {
101                     bad = 1;
102                 }
103             } else {
104                 bad = 1;
105             }
106         }
107         dummy = putnum(c);
108         i = i + 1;
109         c = read_ec(f);
110     }
111
112     if (c != ' ') {
113         newline = 1;
114     }
115     if (!(i > 0 && idx >= 1 && idx <= n)) {
116         bad = 1;
117     }
118     return idx;
119 }
120
121 main {
122     if (!(valid(f) && valid(num))) {
123         exit(1);
124     }
125     n = 0;
126     m = 0;
127     while (!end_ec(f)) {
128         c = read_ec(f);
129         if (c == 'v' || c == 'f') {
130             space = read_ec(f);
131             if (space == ' ') {
132                 if (c == 'v') {
133                     dim = 0;
134                     dummy = cleannum(0);
135                     newline = 0;
136                     while (!newline) {
137                         bad = 0;
138                         start = numidx;
139                         x = readfloat(0);
140                         if (!bad) {
141                             dim = dim + 1;
142                         } else { // discard bad updates
143                             dummy = revertnum(start);
144                         }
145                     }
146                     if (dim == 3) {
147                         print('v');
148                         print(num);
149                         print('\n');
150                         n = n + 1;
151                     } // skip unit
152                 } else {
153                     dim = 0;
154                     dummy = cleannum(0);
155                     newline = 0;
156                     while (!newline) {

```

```

157         bad = 0;
158         start = numidx;
159         idx = readidx(n);
160         if (!bad) {
161             dim = dim + 1;
162         } else { // discard bad updates
163             dummy = revertnum(start);
164         }
165     }
166     if (dim >= 3) {
167         print('f');
168         print(num);
169         print('\n');
170         m = m + 1;
171     } // skip unit
172 }
173 dummy = seek_ec(f, pos_ec(f) - 1);
174 c = read_ec(f);
175 } else { // skip unit
176     c = space;
177 }
178 // skip unit
179 while (c >= 0 && c != '\n') {
180     c = read_ec(f);
181 }
182 }
183
184 print(n);
185 print('\n');
186 print(m);
187 print('\n');
188
189 return 0;
190 }

```

A.3 JSON

A.3.1 Fully-implicit version

```

1  tokenstart = malloc(10);
2  tokenlen = malloc(10);
3  tokentype = malloc(10);
4  tokencount = 0;
5
6  func DFS(f, level) {
7      first = read(f);
8      // remove leading whitespace
9      while (first == ' ' || first == '\n') {
10         first = read(f);
11     }
12     assert (first != ',' && first != ':' && first != '}' && first != ']');
13     if (first == '{') { // object
14         objlen = 0;
15         inspectt (1, f, ',', ')') {
16             c = read(f);
17             key = malloc(20);
18             while (c == ' ' || c == '\n') {
19                 c = read(f);
20             }
21             // parse key token
22             tokenstart[tokencount] = pos(f);
23             i = 0;
24             while (c != ':') {
25                 assert(c != '[' && c != '{');
26                 if (c == '\n') {
27                     key[i] = ' ';
28                 } else {
29                     key[i] = c;
30                 }

```

```

31     i = i + 1;
32     c = read(f);
33 }
34 // remove trailing whitespace
35 while (i > 0 && (key[i-1] == ' ' || key[i-1] == '\n')) {
36     key[i-1] = 0;
37     i = i - 1;
38 }
39 // maintain global arrays
40 tokenlen[tokencount] = i;
41 tokentype[tokencount] = 'k';
42 tokencount = tokencount + 1;
43 // print key token
44 j = 0;
45 print('\n');
46 while (j < level) {
47     print(' ');
48     print(' ');
49     j = j + 1;
50 }
51 print(key);
52 free(key);
53 print(':');
54 print(' ');
55 // parse value
56 vallen = DFS(f, level + 1);
57 assert (vallen > 0);
58 objlen = objlen + vallen + i;
59 }
60 return objlen;
61 } else {
62     if (first == '[') { // array
63         arrlen = 0;
64         print('\n');
65         // parse array elements
66         inspectt (1, f, ',', ',') {
67             i = 0;
68             while (i < level) {
69                 print(' ');
70                 print(' ');
71                 i = i + 1;
72             }
73             print('-');
74             print(' ');
75             elemelen = DFS(f, level + 1);
76             assert (elemelen > 0);
77             print('\n');
78             arrlen = arrlen + elemelen;
79         }
80         return arrlen;
81     } else { // single token
82         word = malloc(20);
83         // parse value token
84         tokenstart[tokencount] = pos(f);
85         word[0] = first;
86         i = 1;
87         while (!end(f)) {
88             c = read(f);
89             assert(c != ':');
90             if (c == '\n') {
91                 word[i] = ' ';
92             } else {
93                 word[i] = c;
94             }
95             i = i + 1;
96         }
97         // remove trailing whitespace
98         while (i > 0 && (word[i-1] == ' ' || word[i-1] == '\n')) {
99             word[i-1] = 0;
100            i = i - 1;
101        }
102        // maintain global arrays
103        tokenlen[tokencount] = i;
104        tokentype[tokencount] = 'v';
105        tokencount = tokencount + 1;
106        // print value token

```

```

107     print(word);
108     free(word);
109     return i;
110 }
111 }
112 }
113
114 main {
115     f = open("../inputs/json/widget.json");
116     total = DFS(f, 0);
117     print('\n');
118
119     print(tokencount);
120     print('\n');
121     print(tokenstart);
122     print('\n');
123     print(tokenlen);
124     print('\n');
125     print(tokentype);
126     print('\n');
127     return total;
128 }

```

A.3.2 Protective-check version

```

1  tokenstart = malloc_ec(10);
2  tokenlen = malloc_ec(10);
3  tokentype = malloc_ec(10);
4  tokencount = 0;
5
6  func DFS(f, level) {
7      first = read_ec(f);
8      // remove leading whitespace
9      while (first == ' ' || first == '\n') {
10         first = read_ec(f);
11     }
12     assert (first >= 0 && first != ',' && first != ':' && first != '}') && first
13         != ']');
14     if (first == '{') { // object
15         objlen = 0;
16         inspect (tokencount < 9, f, ',', ',') {
17             c = read_ec(f);
18             key = malloc_ec(20);
19             assert (valid(key));
20             while (c == ' ' || c == '\n') {
21                 c = read_ec(f);
22             }
23             // parse key token
24             tokenstart[tokencount] = pos_ec(f);
25             assert (tokenstart[tokencount] >= 0);
26             i = 0;
27             while (c != ':') {
28                 assert (i < 20 && c >= 0 && c != '[' && c != '{');
29                 if (c == '\n') {
30                     key[i] = ' ';
31                 } else {
32                     key[i] = c;
33                 }
34                 i = i + 1;
35                 c = read_ec(f);
36             }
37             // remove trailing whitespace
38             while (i > 0 && (key[i-1] == ' ' || key[i-1] == '\n')) {
39                 key[i-1] = 0;
40                 i = i - 1;
41             }
42             // maintain global arrays
43             tokenlen[tokencount] = i;
44             tokentype[tokencount] = 'k';
45             tokencount = tokencount + 1;
46             // print key token
47             j = 0;
48             print('\n');

```

```

48     while (j < level) {
49         print(' ');
50         print(' ');
51         j = j + 1;
52     }
53     print(key);
54     dummy = free_ec(key);
55     print(':');
56     print(' ');
57     // parse value
58     vallen = DFS(f, level + 1);
59     assert (vallen > 0);
60     objlen = objlen + vallen + i;
61 }
62 return objlen;
63 } else {
64     if (first == '[') { // array
65         arrlen = 0;
66         print('\n');
67         // parse array elements
68         inspectt (1, f, ',', ' ')] {
69             i = 0;
70             while (i < level) {
71                 print(' ');
72                 print(' ');
73                 i = i + 1;
74             }
75             print('-');
76             print(' ');
77             elemflen = DFS(f, level + 1);
78             assert (elemflen > 0);
79             print('\n');
80             arrlen = arrlen + elemflen;
81         }
82         return arrlen;
83     } else { // single token
84         word = malloc_ec(20);
85         assert(valid(word));
86         // parse value token
87         tokenstart[tokencount] = pos_ec(f);
88         word[0] = first;
89         i = 1;
90         while (!end_ec(f)) {
91             assert(i < 20);
92             c = read_ec(f);
93             assert(c >= 0 && c != ':');
94             if (c == '\n') {
95                 word[i] = ' ';
96             } else {
97                 word[i] = c;
98             }
99             i = i + 1;
100         }
101         // remove trailing whitespace
102         while (i > 0 && (word[i-1] == ' ' || word[i-1] == '\n')) {
103             word[i-1] = 0;
104             i = i - 1;
105         }
106         // maintain global arrays
107         tokenlen[tokencount] = i;
108         tokentype[tokencount] = 'v';
109         tokencount = tokencount + 1;
110         // print value token
111         print(word);
112         dummy = free_ec(word);
113         return i;
114     }
115 }
116 }
117
118 main {
119     f = opent_ec("../inputs/json/widget.json");
120     assert(valid(f) && valid(tokenstart) && valid(tokenlen) && valid(tokentype))
121     ;
122     total = DFS(f, 0);
123     print('\n');

```

```

123
124     print(tokencount);
125     print('\n');
126     print(tokenstart);
127     print('\n');
128     print(tokenlen);
129     print('\n');
130     print(tokentype);
131     print('\n');
132     return total;
133 }

```

A.3.3 Explicit-recovery version

```

1  tokenstart = malloc_ec(10);
2  tokenlen = malloc_ec(10);
3  tokentype = malloc_ec(10);
4  tokencount = 0;
5
6  stack = malloc_ec(150);
7  stackidx = 0;
8  stackbad = 0;
9
10 func putstack(c) {
11     if (stackidx >= 150) {
12         stackbad = 1;
13     } else {
14         stack[stackidx] = c;
15         stackidx = stackidx + 1;
16     }
17     return 0;
18 }
19
20 func revertstack(start) {
21     while (stackidx > start) {
22         if (stackidx < 150) {
23             stack[stackidx] = 0;
24         }
25         stackidx = stackidx - 1;
26     }
27     if (stackidx < 150) {
28         stackbad = 0;
29     }
30     return 0;
31 }
32
33 func reverttoken(start) {
34     while (tokencount > start) {
35         if (tokencount < 10) {
36             tokenstart[tokencount] = 0;
37             tokenlen[tokencount] = 0;
38             tokentype[tokencount] = 0;
39         }
40         tokencount = tokencount - 1;
41     }
42     return 0;
43 }
44
45 func DFS(f, level) {
46     first = read_ec(f);
47     // remove leading whitespace
48     while (first == ' ' || first == '\n') {
49         first = read_ec(f);
50     }
51     if (!(first >= 0 && first != ',' && first != ':' && first != '}' && first !=
52         ']')) {
53         return -1;
54     }
55     if (first == '{') { // object
56         objlen = 0;
57         lookatt (tokencount < 9, f, ',', '}', '}') {
58             bad = 0;
59             c = read_ec(f);

```

```

59 key = malloc_ec(20);
60 if (valid(key)) {
61     while (c == ' ' || c == '\n') {
62         c = read_ec(f);
63     }
64     // parse key token
65     position = pos_ec(f);
66     if (position < 0) {
67         bad = 1;
68     }
69     i = 0;
70     while (!bad && c != ':') {
71         if (!(i < 20 && c >= 0 && c != '[' && c != '{')) {
72             bad = 1;
73         }
74         if (c == '\n') {
75             key[i] = ' ';
76         } else {
77             key[i] = c;
78         }
79         i = i + 1;
80         c = read_ec(f);
81     }
82     if (!bad) {
83         stackok = stackidx;
84         tokenok = tokencount;
85         // remove trailing whitespace
86         while (i > 0 && (key[i-1] == ' ' || key[i-1] == '\n')) {
87             key[i-1] = 0;
88             i = i - 1;
89         }
90         // maintain global arrays
91         tokenstart[tokencount] = position;
92         tokenlen[tokencount] = i;
93         tokentype[tokencount] = 'k';
94         tokencount = tokencount + 1;
95         // print key token
96         j = 0;
97         dummy = putstack('\n');
98         while (j < level) {
99             dummy = putstack(' ');
100            dummy = putstack(' ');
101            j = j + 1;
102        }
103        j = 0;
104        while (j < i) {
105            x = key[j];
106            dummy = putstack(x);
107            j = j + 1;
108        }
109        dummy = putstack(':');
110        dummy = putstack(' ');
111        // parse value
112        vallen = DFS(f, level + 1);
113        if (!stackbad && vallen > 0) {
114            objlen = objlen + vallen + i;
115        } else { // discard bad updates
116            dummy = revertstack(stackok);
117            dummy = reverttoken(tokenok);
118        }
119        } // skip unit
120        dummy = free_ec(key);
121    } // skip unit
122 }
123 return objlen;
124 } else {
125     if (first == '[') { // array
126         arrlen = 0;
127         dummy = putstack('\n');
128         // parse array elements
129         lookatt (1, f, ',', ')') {
130             stackok = stackidx;
131             tokenok = tokencount;
132             i = 0;
133             while (i < level) {
134                 dummy = putstack(' ');

```



```

135     dummy = putstack(' ');
136     i = i + 1;
137 }
138 dummy = putstack('-');
139 dummy = putstack(' ');
140 elemflen = DFS(f, level + 1);
141 dummy = putstack('\n');
142 if (!stackbad && elemflen > 0) {
143     arrlen = arrlen + elemflen;
144 } else { // discard bad updates
145     dummy = revertstack(stackok);
146     dummy = reverttoken(tokenok);
147 }
148 }
149 return arrlen;
150 } else { // single token
151     word = malloc_ec(20);
152     bad = 0;
153     if (valid(word)) {
154         // parse value token
155         position = pos_ec(f);
156         if (position < 0) {
157             bad = 1;
158         }
159         word[0] = first;
160         i = 1;
161         while (!bad && !end_ec(f)) {
162             if (i >= 20) {
163                 bad = 1;
164             } else {
165                 c = read_ec(f);
166                 if (!(c >= 0 && c != ',')) {
167                     bad = 1;
168                 } else {
169                     if (c == '\n') {
170                         word[i] = ' ';
171                     } else {
172                         word[i] = c;
173                     }
174                 }
175                 i = i + 1;
176             }
177         }
178         if (!bad) {
179             stackok = stackidx;
180             // remove trailing whitespace
181             while (i > 0 && (word[i-1] == ' ' || word[i-1] == '\n')) {
182                 word[i-1] = 0;
183                 i = i - 1;
184             }
185             // maintain global arrays
186             tokenstart[tokencount] = position;
187             tokenlen[tokencount] = i;
188             tokentype[tokencount] = 'v';
189             tokencount = tokencount + 1;
190             // print value token
191             j = 0;
192             while (j < i) {
193                 x = word[j];
194                 dummy = putstack(x);
195                 j = j + 1;
196             }
197             if (stackbad) { // discard bad updates
198                 dummy = revertstack(stackok);
199                 dummy = reverttoken(tokencount - 1);
200                 i = 0;
201             }
202         } else { // skip unit
203             i = 0;
204         }
205         dummy = free_ec(word);
206         return i;
207     } else { // skip unit
208         return -1;
209     }
210 }

```

```

211     }
212 }
213
214 main {
215     f = opent_ec("../inputs/json/widget.json");
216     if (!(valid(f) && valid(tokenstart) && valid(tokenlen) && valid(tokentype)
                && valid(stack))) {
217         exit(1);
218     }
219     total = DFS(f, 0);
220     if (total < 0) {
221         exit(1);
222     } else {
223         print(stack);
224         print('\n');
225
226         print(tokencount);
227         print('\n');
228         print(tokenstart);
229         print('\n');
230         print(tokenlen);
231         print('\n');
232         print(tokentype);
233         print('\n');
234     }
235     return total;
236 }

```

A.3.4 Plain-loop version

```

1  tokenstart = malloc_ec(10);
2  tokenlen = malloc_ec(10);
3  tokentype = malloc_ec(10);
4  tokencount = 0;
5
6  stack = malloc_ec(150);
7  stackidx = 0;
8  stackbad = 0;
9
10 func putstack(c) {
11     if (stackidx >= 150) {
12         stackbad = 1;
13     } else {
14         stack[stackidx] = c;
15         stackidx = stackidx + 1;
16     }
17     return 0;
18 }
19
20 func revertstack(start) {
21     while (stackidx > start) {
22         if (stackidx < 150) {
23             stack[stackidx] = 0;
24         }
25         stackidx = stackidx - 1;
26     }
27     if (stackidx < 150) {
28         stackbad = 0;
29     }
30     return 0;
31 }
32
33 func reverttoken(start) {
34     while (tokencount > start) {
35         if (tokencount < 10) {
36             tokenstart[tokencount] = 0;
37             tokenlen[tokencount] = 0;
38             tokentype[tokencount] = 0;
39         }
40         tokencount = tokencount - 1;
41     }
42     return 0;
43 }

```

```

44
45 func DFS(f, level) {
46     first = read_ec(f);
47     // remove leading whitespace
48     while (first == ' ' || first == '\n') {
49         first = read_ec(f);
50     }
51     if (!(first >= 0 && first != ',' && first != ':' && first != '}' && first !=
        ']')) {
52         return -1;
53     }
54     if (first == '{') { // object
55         objlen = 0;
56         finish = 0;
57         while (tokencount < 9 && !finish) {
58             bad = 0;
59             c = read_ec(f);
60             key = malloc_ec(20);
61             if (valid(key)) {
62                 while (c == ' ' || c == '\n') {
63                     c = read_ec(f);
64                 }
65                 // parse key token
66                 position = pos_ec(f);
67                 if (position < 0) {
68                     bad = 1;
69                 }
70                 i = 0;
71                 while (!(bad && c != ':' && c != ',' && c != '}' && c != ']')) {
72                     if (!(i < 20 && c >= 0 && c != '[' && c != '{')) {
73                         bad = 1;
74                     }
75                     if (c == '\n') {
76                         key[i] = ' ';
77                     } else {
78                         key[i] = c;
79                     }
80                     i = i + 1;
81                     c = read_ec(f);
82                 }
83                 if (!bad) {
84                     stackok = stackidx;
85                     tokenok = tokencount;
86                     // remove trailing whitespace
87                     while (i > 0 && (key[i-1] == ' ' || key[i-1] == '\n')) {
88                         key[i-1] = 0;
89                         i = i - 1;
90                     }
91                     // maintain global arrays
92                     tokenstart[tokencount] = position;
93                     tokenlen[tokencount] = i;
94                     tokentype[tokencount] = 'k';
95                     tokencount = tokencount + 1;
96                     // print key token
97                     j = 0;
98                     dummy = putstack('\n');
99                     while (j < level) {
100                         dummy = putstack(' ');
101                         dummy = putstack(' ');
102                         j = j + 1;
103                     }
104                     j = 0;
105                     while (j < i) {
106                         x = key[j];
107                         dummy = putstack(x);
108                         j = j + 1;
109                     }
110                     dummy = putstack(':');
111                     dummy = putstack(' ');
112                     // parse value
113                     vallen = DFS(f, level + 1);
114                     if (!stackbad && vallen > 0) {
115                         objlen = objlen + vallen + i;
116                     } else { // discard bad updates
117                         dummy = revertstack(stackok);
118                         dummy = reverttoken(tokenok);

```

```

119     }
120   } // skip unit
121   dummy = free_ec(key);
122   } // skip unit
123   while (c >= 0 && c != ',' && c != '}' && c != ']') { // end of unit
124     c = read_ec(f);
125   }
126   if (c < 0 || c == '}') { // end of sequence
127     finish = 1;
128   }
129   if (c == ']') { // outside delimiters
130     finish = 1;
131     dummy = seek_ec(f, pos_ec(f) - 1);
132   }
133 }
134 return objlen;
135 } else {
136   if (first == '[') { // array
137     arrlen = 0;
138     dummy = putstack('\n');
139     // parse array elements
140     while (!finish) {
141       stackok = stackidx;
142       tokenok = tokencount;
143       i = 0;
144       while (i < level) {
145         dummy = putstack(' ');
146         dummy = putstack(' ');
147         i = i + 1;
148       }
149       dummy = putstack('-');
150       dummy = putstack(' ');
151       elemflen = DFS(f, level + 1);
152       dummy = putstack('\n');
153       if (!stackbad && elemflen > 0) {
154         arrlen = arrlen + elemflen;
155       } else { // discard bad updates
156         dummy = revertstack(stackok);
157         dummy = reverttoken(tokenok);
158       }
159       c = read_ec(f);
160       while (c >= 0 && c != ',' && c != '}' && c != ']') { // end of unit
161         c = read_ec(f);
162       }
163       if (c < 0 || c == '}') { // end of sequence
164         finish = 1;
165       }
166       if (c == ']') { // outside delimiters
167         finish = 1;
168         dummy = seek_ec(f, pos_ec(f) - 1);
169       }
170     }
171     return arrlen;
172   } else { // single token
173     word = malloc_ec(20);
174     bad = 0;
175     if (valid(word)) {
176       // parse value token
177       position = pos_ec(f);
178       if (position < 0) {
179         bad = 1;
180       }
181       word[0] = first;
182       i = 1;
183       c = read_ec(f);
184       while (!bad && c >= 0 && c != ',' && c != '}' && c != ']') {
185         if (i >= 20) {
186           bad = 1;
187         } else {
188           if (!(c >= 0 && c != ':')) {
189             bad = 1;
190           } else {
191             if (c == '\n') {
192               word[i] = ' ';
193             } else {
194               word[i] = c;

```

```

195     }
196     }
197     i = i + 1;
198     c = read_ec(f);
199 }
200 }
201 if (c == ',' || c == ']' || c == '}') {
202     dummy = seek_ec(f, pos_ec(f) - 1);
203 }
204 if (!bad) {
205     stackok = stackidx;
206     // remove trailing whitespace
207     while (i > 0 && (word[i-1] == ' ' || word[i-1] == '\n')) {
208         word[i-1] = 0;
209         i = i - 1;
210     }
211     // maintain global arrays
212     tokenstart[tokencount] = position;
213     tokenlen[tokencount] = i;
214     tokentype[tokencount] = 'v';
215     tokencount = tokencount + 1;
216     // print value token
217     j = 0;
218     while (j < i) {
219         x = word[j];
220         dummy = putstack(x);
221         j = j + 1;
222     }
223     if (stackbad) { // discard bad updates
224         dummy = revertstack(stackok);
225         dummy = reverttoken(tokencount - 1);
226         i = 0;
227     }
228 } else { // skip unit
229     i = 0;
230 }
231 dummy = free_ec(word);
232 return i;
233 } else { // skip unit
234     return -1;
235 }
236 }
237 }
238 }
239
240 main {
241     f = opent_ec("../inputs/json/widget.json");
242     if (!(valid(f) && valid(tokenstart) && valid(tokenlen) && valid(tokentype)
243         && valid(stack))) {
244         exit(1);
245     }
246     total = DFS(f, 0);
247     if (total < 0) {
248         exit(1);
249     } else {
250         print(stack);
251         print('\n');
252         print(tokencount);
253         print('\n');
254         print(tokenstart);
255         print('\n');
256         print(tokenlen);
257         print('\n');
258         print(tokentype);
259         print('\n');
260     }
261     return total;
262 }

```

A.4 PNG

A.4.1 Fully-implicit version

```
1 func readintbytes (f, n) {
2   x = 0;
3   i = 0;
4   while (i < n) {
5     byte = read(f);
6     x = x << 8;
7     x = x | byte;
8     i = i + 1;
9   }
10  return x;
11 }
12
13 main {
14   f = openb("../inputs/png0/oi4n0g16-2.png", 0);
15
16   magic = malloc(8);
17   i = 0;
18   while (i < 8) {
19     x = read(f);
20     magic[i] = x;
21     i = i + 1;
22   }
23   assert(magic[0] == 137 && magic[1] == 'P' && magic[2] == 'N' && magic[3] ==
        'G' && magic[4] == 13 && magic[5] == 10 && magic[6] == 26 && magic[7] ==
        10);
24   free(magic);
25
26   n = readintbytes(f, 4);
27   assert(n == 13);
28
29   ihdr = malloc(4);
30   i = 0;
31   while (i < 4) {
32     x = read(f);
33     ihdr[i] = x;
34     i = i + 1;
35   }
36   assert(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr[2] == 'D' && ihdr[3] == 'R')
        ;
37   free(ihdr);
38
39   w = readintbytes(f, 4);
40   h = readintbytes(f, 4);
41   assert(w > 0 && h > 0);
42
43   depth = read(f);
44   color = read(f);
45   compression = read(f);
46   filter = read(f);
47   interlace = read(f);
48   assert((depth == 1 || depth == 2 || depth == 4 || depth == 8 || depth == 16)
        && color == 0 && compression == 0 && filter == 0 && (interlace == 0 ||
        interlace == 1));
49
50   crc = readintbytes(f, 4);
51   // check CRC
52
53   ndata = 0;
54   finish = 0;
55   type = malloc(4);
56   inspectb (!finish, f, 0, 4, 8) {
57     length = readintbytes(f, 4);
58
59     i = 0;
60     while (i < 4) {
61       x = read(f);
62       type[i] = x;
63       i = i + 1;
```

```

64     }
65     // check type legal
66
67     if (type[0] == 'I' && type[1] == 'E' && type[2] == 'N' && type[3] == 'D')
68     { //
69         IEND
70         a = read(f);
71         b = read(f);
72         c = read(f);
73         d = read(f);
74         assert(length == 0 && a == 174 && b == 66 && c == 96 && d == 130);
75         finish = 1;
76     } else {
77         bytes = malloc(length);
78         i = 0;
79         while (i < length) {
80             x = read(f);
81             bytes[i] = x;
82             i = i + 1;
83         }
84         crc = readintbytes(f, 4);
85         // check CRC
86         if (type[0] == 'I' && type[1] == 'D' && type[2] == 'A' && type[3] == 'T') { //
87             IDAT
88             ndata = ndata + 1;
89             // decompress
90             // decode
91             print(bytes);
92         }
93         free(bytes);
94     }
95 }

```

A.4.2 Protective-check version

```

1 func readintbytes (f, n) {
2     x = 0;
3     i = 0;
4     while (i < n) {
5         byte = read_ec(f);
6         assert(byte >= 0);
7         x = x << 8;
8         x = x | byte;
9         i = i + 1;
10    }
11    return x;
12 }
13
14 main {
15     f = openb_ec("../inputs/png0/oi4n0g16-2.png", 0);
16
17     magic = malloc_ec(8);
18     assert(valid(f) && valid(magic));
19     i = 0;
20     while (i < 8) {
21         x = read_ec(f);
22         assert(x >= 0);
23         magic[i] = x;
24         i = i + 1;
25     }
26     assert(magic[0] == 137 && magic[1] == 'P' && magic[2] == 'N' && magic[3] ==
27            'G' && magic[4] == 13 && magic[5] == 10 && magic[6] == 26 && magic[7] ==
28            10);
29     x = free_ec(magic);
30
31     n = readintbytes(f, 4);
32     assert(n == 13);
33
34     ihdr = malloc_ec(4);

```

```

33  assert(valid(ihdr));
34  i = 0;
35  while (i < 4) {
36      x = read_ec(f);
37      assert(x >= 0);
38      ihdr[i] = x;
39      i = i + 1;
40  }
41  assert(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr[2] == 'D' && ihdr[3] == 'R')
42      ;
43  x = free_ec(ihdr);
44
45  w = readintbytes(f, 4);
46  h = readintbytes(f, 4);
47  assert(w > 0 && h > 0);
48
49  depth = read_ec(f);
50  color = read_ec(f);
51  compression = read_ec(f);
52  filter = read_ec(f);
53  interlace = read_ec(f);
54  assert((depth == 1 || depth == 2 || depth == 4 || depth == 8 || depth == 16)
55      && color == 0 && compression == 0 && filter == 0 && (interlace == 0 ||
56      interlace == 1));
57
58  crc = readintbytes(f, 4);
59  // check CRC
60
61  ndata = 0;
62  finish = 0;
63  type = malloc_ec(4);
64  assert(valid(type));
65  inspectb (!finish, f, 0, 4, 8) {
66      length = readintbytes(f, 4);
67
68      i = 0;
69      while (i < 4) {
70          x = read_ec(f);
71          assert(x >= 0);
72          type[i] = x;
73          i = i + 1;
74      }
75      // check type legal
76
77      if (type[0] == 'I' && type[1] == 'E' && type[2] == 'N' && type[3] == 'D')
78          { //
79              IEND
80              a = read_ec(f);
81              b = read_ec(f);
82              c = read_ec(f);
83              d = read_ec(f);
84              assert(length == 0 && a == 174 && b == 66 && c == 96 && d == 130);
85              finish = 1;
86          } else {
87              assert(length > 0);
88              bytes = malloc_ec(length);
89              assert(valid(bytes));
90              i = 0;
91              while (i < length) {
92                  x = read_ec(f);
93                  assert(x >= 0);
94                  bytes[i] = x;
95                  i = i + 1;
96              }
97              crc = readintbytes(f, 4);
98              // check CRC
99              if (type[0] == 'I' && type[1] == 'D' && type[2] == 'A' && type[3] == 'T'
100                  ) { //
101                  IDAT
102                  ndata = ndata + 1;
103                  // decompress
104                  // decode
105                  print(bytes);
106              }
107              x = free_ec(bytes);
108          }
109      }
110  }
111  }

```



```

102     }
103     x = free_ec(type);
104     return 0;
105 }

```

A.4.3 Explicit-recovery version

```

1  bad = 0;
2
3  func readintbytes (f, n) {
4      x = 0;
5      i = 0;
6      while (i < n && !bad) {
7          byte = read_ec(f);
8          if (byte >= 0) {
9              x = x << 8;
10             x = x | byte;
11             i = i + 1;
12         } else {
13             bad = 1;
14             return -1;
15         }
16     }
17     return x;
18 }
19
20 main {
21     f = openb_ec("../inputs/png0/oi4n0g16-2.png", 0);
22
23     magic = malloc_ec(8);
24     if (!valid(f) || !valid(magic)) {
25         exit(1);
26     }
27     i = 0;
28     while (i < 8) {
29         x = read_ec(f);
30         if (x < 0) {
31             exit(1);
32         }
33         magic[i] = x;
34         i = i + 1;
35     }
36     if (!(magic[0] == 137 && magic[1] == 'P' && magic[2] == 'N' && magic[3] == 'G'
37         && magic[4] == 13 && magic[5] == 10 && magic[6] == 26 && magic[7] == 10)) {
38         exit(1);
39     }
40     x = free_ec(magic);
41
42     n = readintbytes(f, 4);
43     if (bad || n != 13) {
44         exit(1);
45     }
46     ihdr = malloc_ec(4);
47     if (!valid(ihdr)) {
48         exit(1);
49     }
50     i = 0;
51     while (i < 4) {
52         x = read_ec(f);
53         if (x < 0) {
54             exit(1);
55         }
56         ihdr[i] = x;
57         i = i + 1;
58     }
59     if (!(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr[2] == 'D' && ihdr[3] == 'R'))
60     {
61         exit(1);
62     }
63     x = free_ec(ihdr);

```

```

64 w = readintbytes(f, 4);
65 h = readintbytes(f, 4);
66 if (bad || w <= 0 || h <= 0) {
67     exit(1);
68 }
69
70 depth = read_ec(f);
71 color = read_ec(f);
72 compression = read_ec(f);
73 filter = read_ec(f);
74 interlace = read_ec(f);
75 if (!(depth == 1 || depth == 2 || depth == 4 || depth == 8 || depth == 16)
    || color != 0 || compression != 0 || filter != 0 || !(interlace == 0 ||
    interlace == 1)) {
76     exit(1);
77 }
78
79 crc = readintbytes(f, 4);
80 if (bad) {
81     exit(1);
82 }
83 // check CRC
84
85 ndata = 0;
86 finish = 0;
87 type = malloc_ec(4);
88 if (!valid(type)) {
89     exit(1);
90 }
91 lookatb (!finish, f, 0, 4, 8) {
92     bad = 0;
93     length = readintbytes(f, 4);
94
95     i = 0;
96     while (i < 4) {
97         x = read_ec(f);
98         if (x < 0) {
99             bad = 1;
100         }
101         type[i] = x;
102         i = i + 1;
103     }
104     // check type legal
105
106     if (type[0] == 'I' && type[1] == 'E' && type[2] == 'N' && type[3] == 'D')
        { //
            IEND
107         a = read_ec(f);
108         b = read_ec(f);
109         c = read_ec(f);
110         d = read_ec(f);
111         if (length == 0 && a == 174 && b == 66 && c == 96 && d == 130) { // good
112             finish = 1;
113         } // skip unit
114     } else {
115         if (length > 0) { // good
116             bytes = malloc_ec(length);
117             if (valid(bytes)) {
118                 i = 0;
119                 while (i < length) {
120                     x = read_ec(f);
121                     if (x < 0) {
122                         bad = 1;
123                     }
124                     bytes[i] = x;
125                     i = i + 1;
126                 }
127                 crc = readintbytes(f, 4);
128                 // check CRC
129                 if (!bad && type[0] == 'I' && type[1] == 'D' && type[2] == 'A' &&
                    type[3] == 'T') { //
                    IDAT
130                     ndata = ndata + 1;
131                     // decompress
132                     // decode
133                     print(bytes);

```

```

134         } // skip unit
135         x = free_ec(bytes);
136     } // skip unit
137 } // skip unit
138 }
139 }
140 x = free_ec(type);
141 return 0;
142 }

```

A.4.4 Plain-loop version

```

1  bad = 0;
2
3  func readintbytes (f, n) {
4      x = 0;
5      i = 0;
6      while (i < n && !bad) {
7          byte = read_ec(f);
8          if (byte >= 0) {
9              x = x << 8;
10             x = x | byte;
11             i = i + 1;
12         } else {
13             bad = 1;
14             return -1;
15         }
16     }
17     return x;
18 }
19
20 main {
21     f = openb_ec("../inputs/png0/oi4n0g16-2.png", 0);
22
23     magic = malloc_ec(8);
24     if (!valid(f) || !valid(magic)) {
25         exit(1);
26     }
27     i = 0;
28     while (i < 8) {
29         x = read_ec(f);
30         if (x < 0) {
31             exit(1);
32         }
33         magic[i] = x;
34         i = i + 1;
35     }
36     if (!(magic[0] == 137 && magic[1] == 'P' && magic[2] == 'N' && magic[3] == '
37         G' && magic[4] == 13 && magic[5] == 10 && magic[6] == 26 && magic[7] ==
38         10)) {
39         exit(1);
40     }
41     x = free_ec(magic);
42
43     n = readintbytes(f, 4);
44     if (bad || n != 13) {
45         exit(1);
46     }
47
48     ihdr = malloc_ec(4);
49     if (!valid(ihdr)) {
50         exit(1);
51     }
52     i = 0;
53     while (i < 4) {
54         x = read_ec(f);
55         if (x < 0) {
56             exit(1);
57         }
58         ihdr[i] = x;
59         i = i + 1;
60     }

```

```

59  if (!(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr[2] == 'D' && ihdr[3] == 'R'))
60      {
61          exit(1);
62      }
63  x = free_ec(ihdr);
64  w = readintbytes(f, 4);
65  h = readintbytes(f, 4);
66  if (bad || w <= 0 || h <= 0) {
67      exit(1);
68  }
69
70  depth = read_ec(f);
71  color = read_ec(f);
72  compression = read_ec(f);
73  filter = read_ec(f);
74  interlace = read_ec(f);
75  if (!(depth == 1 || depth == 2 || depth == 4 || depth == 8 || depth == 16)
      || color != 0 || compression != 0 || filter != 0 || !(interlace == 0 ||
      interlace == 1)) {
76      exit(1);
77  }
78
79  crc = readintbytes(f, 4);
80  if (bad) {
81      exit(1);
82  }
83  // check CRC
84
85  ndata = 0;
86  finish = 0;
87  type = malloc_ec(4);
88  if (!valid(type)) {
89      exit(1);
90  }
91  while (!finish) {
92      bad = 0;
93      length = readintbytes(f, 4);
94      eou = pos_ec(f) + length + 8;
95      if (!bad && length >= 0) {
96          i = 0;
97          while (i < 4) {
98              x = read_ec(f);
99              if (x < 0) {
100                 bad = 1;
101             }
102             type[i] = x;
103             i = i + 1;
104         }
105         // check type legal
106
107         if (type[0] == 'I' && type[1] == 'E' && type[2] == 'N' && type[3] == 'D'
            ') { //
            IEND
108             a = read_ec(f);
109             b = read_ec(f);
110             c = read_ec(f);
111             d = read_ec(f);
112             if (length == 0 && a == 174 && b == 66 && c == 96 && d == 130) { //
                good
113                 finish = 1;
114             } // skip unit
115         } else { // other types
116             if (length > 0) { // good
117                 bytes = malloc_ec(length);
118                 if (valid(bytes)) {
119                     i = 0;
120                     while (i < length) {
121                         x = read_ec(f);
122                         if (x < 0) {
123                             bad = 1;
124                         }
125                         bytes[i] = x;
126                         i = i + 1;
127                     }
128                     crc = readintbytes(f, 4);

```

```

129         // check CRC
130         if (!bad && type[0] == 'I' && type[1] == 'D' && type[2] == 'A' &&
            type[3] == 'T') { //
            IDAT
131             ndata = ndata + 1;
132             // decompress
133             // decode
134             print(bytes);
135         } // skip unit
136         x = free_ec(bytes);
137     } // skip unit
138 } // skip unit
139 }
140 // go to next unit according to length
141 x = seek_ec(f, eou);
142 if (x < 0) { // give up
143     finish = 1;
144 }
145 } else { // give up
146     finish = 1;
147 }
148 }
149 x = free_ec(type);
150 return 0;
151 }

```

A.5 ZIP

A.5.1 Fully-implicit version

```

1 func readintbytesl (f, n) { // little endian
2     x = read(f);
3     if (n == 1) {
4         return x;
5     } else {
6         y = readintbytesl(f, n-1);
7         return (y << 8) + x;
8     }
9 }
10
11 main {
12     fdir = openb("../inputs/zip/stuff-1.zip", 1);
13     fent = openb("../inputs/zip/stuff-1.zip", 1);
14
15     eocd = size(fdir) - 4;
16     found = 0;
17     while (!found) {
18         seek(fdir, eocd);
19         dir_sig = readintbytesl(fdir, 4);
20         if (dir_sig == 101010256) { // 0x06054b50
21             found = 1;
22         } else {
23             eocd = eocd - 1;
24         }
25     }
26     assert(found);
27     diskid = readintbytesl(fdir, 2);
28     ndisk = readintbytesl(fdir, 2);
29     dirid = readintbytesl(fdir, 2);
30     ndir = readintbytesl(fdir, 2);
31
32     dir_size = readintbytesl(fdir, 4);
33     dir_start = readintbytesl(fdir, 4);
34     eocd_comm_size = readintbytesl(fdir, 2);
35     if (eocd_comm_size > 0) {
36         eocd_comm = malloc(eocd_comm_size);
37         i = 0;
38         while (i < eocd_comm_size) {
39             b = read(fdir);

```

```

40     eocd_comm[i] = b;
41     i = i + 1;
42 }
43 }
44
45 seek(fdir, dir_start);
46 inspectb (pos(fdir) < eocd, fdir, 28, 2, 40) {
47     dir_sig = readintbytesl(fdir, 4);
48     dir_ver_made = readintbytesl(fdir, 2);
49     dir_ver_extr = readintbytesl(fdir, 2);
50     dir_flag = readintbytesl(fdir, 2);
51     dir_comp = readintbytesl(fdir, 2);
52     dir_modif = readintbytesl(fdir, 4);
53     dir_crc = readintbytesl(fdir, 4);
54     dir_ent_size = readintbytesl(fdir, 4);
55     dir_ent_size_uncomp = readintbytesl(fdir, 4);
56     dir_name_size = readintbytesl(fdir, 2);
57     dir_extr_size = readintbytesl(fdir, 2);
58     dir_comm_size = readintbytesl(fdir, 2);
59     dir_diskid = readintbytesl(fdir, 2);
60     dir_attr = malloc(6);
61     i = 0;
62     while (i < 6) {
63         b = read(fdir);
64         dir_attr[i] = b;
65         i = i + 1;
66     }
67     dir_ent_start = readintbytesl(fdir, 4);
68     assert(dir_sig == 33639248 && dir_extr_size == 24 && dir_comm_size == 0);
69     dir_name = malloc(dir_name_size);
70     i = 0;
71     while (i < dir_name_size) {
72         b = read(fdir);
73         dir_name[i] = b;
74         i = i + 1;
75     }
76     print(dir_name);
77     print('\n');
78     dir_extr = malloc(dir_extr_size);
79     i = 0;
80     while (i < dir_extr_size) {
81         b = read(fdir);
82         dir_extr[i] = b;
83         i = i + 1;
84     }
85
86     seek(fent, dir_ent_start);
87
88     ent_sig = readintbytesl(fent, 4);
89     ent_ver_extr = readintbytesl(fent, 2);
90     ent_flag = readintbytesl(fent, 2);
91     ent_comp = readintbytesl(fent, 2);
92     ent_modif = readintbytesl(fent, 4);
93     ent_crc = readintbytesl(fent, 4);
94     ent_ent_size = readintbytesl(fent, 4);
95     ent_ent_size_uncomp = readintbytesl(fent, 4);
96     ent_name_size = readintbytesl(fent, 2);
97     ent_extr_size = readintbytesl(fent, 2);
98
99     assert(ent_sig == 67324752 && ent_ver_extr == dir_ver_extr && ent_flag ==
100            dir_flag && ent_comp == dir_comp && ent_modif == dir_modif &&
101            ent_name_size == dir_name_size);
102     ent_name = malloc(ent_name_size);
103     i = 0;
104     while (i < ent_name_size) {
105         b = read(fent);
106         ent_name[i] = b;
107         i = i + 1;
108     }
109     print(ent_name);
110     print('\n');
111     if (ent_extr_size > 0) {
112         ent_extr = malloc(ent_extr_size);
113         i = 0;
114         while (i < ent_extr_size) {
115             b = read(fent);

```

```

114     ent_extr[i] = b;
115     i = i + 1;
116 }
117 }
118 assert(ent_flag & 8 == 0 && ent_crc == dir_crc && ent_ent_size ==
    dir_ent_size && ent_ent_size_uncomp == dir_ent_size_uncomp);
119
120 raw_data = malloc(ent_ent_size);
121 i = 0;
122 while (i < ent_ent_size) {
123     b = read(fent);
124     raw_data[i] = b;
125     i = i + 1;
126 }
127 // decompress
128 // check crc
129 print(raw_data);
130 print('\n');
131 free(ent_name);
132 free(raw_data);
133 }
134 seek(fdir, eocd);
135 return 0;
136 }

```

A.5.2 Protective-check version

```

1 func readintbytesl (f, n) { // little endian
2     x = read_ec(f);
3     assert(x >= 0);
4     if (n == 1) {
5         return x;
6     } else {
7         y = readintbytesl(f, n-1);
8         return (y << 8) + x;
9     }
10 }
11
12 main {
13     fdir = openb_ec("../inputs/zip/stuff-1.zip", 1);
14     fent = openb_ec("../inputs/zip/stuff-1.zip", 1);
15     assert(valid(fdir) && valid(fent));
16
17     eocd = size_ec(fdir) - 4;
18     found = 0;
19     while (!found) {
20         x = seek_ec(fdir, eocd);
21         assert(x >= 0);
22         dir_sig = readintbytesl(fdir, 4);
23         if (dir_sig == 101010256) { // 0x06054b50
24             found = 1;
25         } else {
26             eocd = eocd - 1;
27         }
28     }
29     assert(found);
30     diskid = readintbytesl(fdir, 2);
31     ndisk = readintbytesl(fdir, 2);
32     dirid = readintbytesl(fdir, 2);
33     ndir = readintbytesl(fdir, 2);
34
35     dir_size = readintbytesl(fdir, 4);
36     dir_start = readintbytesl(fdir, 4);
37     eocd_comm_size = readintbytesl(fdir, 2);
38     if (eocd_comm_size > 0) {
39         eocd_comm = malloc_ec(eocd_comm_size);
40         assert(valid(eocd_comm));
41         i = 0;
42         while (i < eocd_comm_size) {
43             b = read_ec(fdir);
44             assert(b >= 0);
45             eocd_comm[i] = b;
46             i = i + 1;

```

```

47     }
48 }
49
50 x = seek_ec(fdir, dir_start);
51 assert(x >= 0);
52 inspectb (pos_ec(fdir) < eocd && pos_ec(fdir) >= 0, fdir, 28, 2, 40) {
53     dir_sig = readintbytesl(fdir, 4);
54     dir_ver_made = readintbytesl(fdir, 2);
55     dir_ver_extr = readintbytesl(fdir, 2);
56     dir_flag = readintbytesl(fdir, 2);
57     dir_comp = readintbytesl(fdir, 2);
58     dir_modif = readintbytesl(fdir, 4);
59     dir_crc = readintbytesl(fdir, 4);
60     dir_ent_size = readintbytesl(fdir, 4);
61     dir_ent_size_uncomp = readintbytesl(fdir, 4);
62     dir_name_size = readintbytesl(fdir, 2);
63     dir_extr_size = readintbytesl(fdir, 2);
64     dir_comm_size = readintbytesl(fdir, 2);
65     dir_diskid = readintbytesl(fdir, 2);
66     dir_attr = malloc_ec(6);
67     assert(valid(dir_attr));
68     i = 0;
69     while (i < 6) {
70         b = read_ec(fdir);
71         assert(b >= 0);
72         dir_attr[i] = b;
73         i = i + 1;
74     }
75     dir_ent_start = readintbytesl(fdir, 4);
76     assert(dir_sig == 33639248 && dir_extr_size == 24 && dir_comm_size == 0 &&
77           dir_name_size > 0);
78     dir_name = malloc_ec(dir_name_size);
79     assert(valid(dir_name));
80     i = 0;
81     while (i < dir_name_size) {
82         b = read_ec(fdir);
83         assert(b >= 0);
84         dir_name[i] = b;
85         i = i + 1;
86     }
87     print(dir_name);
88     print('\n');
89     dir_extr = malloc_ec(dir_extr_size);
90     assert(valid(dir_extr));
91     i = 0;
92     while (i < dir_extr_size) {
93         b = read_ec(fdir);
94         assert(b >= 0);
95         dir_extr[i] = b;
96         i = i + 1;
97     }
98     x = seek_ec(fent, dir_ent_start);
99     assert(x >= 0);
100
101     ent_sig = readintbytesl(fent, 4);
102     ent_ver_extr = readintbytesl(fent, 2);
103     ent_flag = readintbytesl(fent, 2);
104     ent_comp = readintbytesl(fent, 2);
105     ent_modif = readintbytesl(fent, 4);
106     ent_crc = readintbytesl(fent, 4);
107     ent_ent_size = readintbytesl(fent, 4);
108     ent_ent_size_uncomp = readintbytesl(fent, 4);
109     ent_name_size = readintbytesl(fent, 2);
110     ent_extr_size = readintbytesl(fent, 2);
111
112     assert(ent_sig == 67324752 && ent_ver_extr == dir_ver_extr && ent_flag ==
113           dir_flag && ent_comp == dir_comp && ent_modif == dir_modif &&
114           ent_name_size == dir_name_size && ent_name_size > 0);
115     ent_name = malloc_ec(ent_name_size);
116     assert(valid(ent_name));
117     i = 0;
118     while (i < ent_name_size) {
119         b = read_ec(fent);
120         assert(b >= 0);
121         ent_name[i] = b;

```



```

120     i = i + 1;
121 }
122 print(ent_name);
123 print('\n');
124 if (ent_extr_size > 0) {
125     ent_extr = malloc_ec(ent_extr_size);
126     assert(valid(ent_extr));
127     i = 0;
128     while (i < ent_extr_size) {
129         b = read_ec(fent);
130         assert(b >= 0);
131         ent_extr[i] = b;
132         i = i + 1;
133     }
134 }
135 assert(ent_flag & 8 == 0 && ent_crc == dir_crc && ent_ent_size ==
    dir_ent_size && ent_ent_size_uncomp == dir_ent_size_uncomp &&
    ent_ent_size > 0);

136 raw_data = malloc_ec(ent_ent_size);
137 assert(valid(raw_data));
138 i = 0;
139 while (i < ent_ent_size) {
140     b = read_ec(fent);
141     assert(b >= 0);
142     raw_data[i] = b;
143     i = i + 1;
144 }
145 // decompress
146 // check crc
147 print(raw_data);
148 print('\n');
149 x = free_ec(ent_name);
150 x = free_ec(raw_data);
151 }
152 x = seek_ec(fdir, eocd);
153 return 0;
154 }
155 }

```

A.5.3 Explicit-recovery version

```

1 bad = 0;
2
3 func readintbytes1 (f, n) { // little endian
4     x = read_ec(f);
5     if (x >= 0) {
6         if (n == 1) {
7             return x;
8         } else {
9             y = readintbytes1(f, n-1);
10            if (bad) {
11                return -1;
12            }
13            return (y << 8) + x;
14        }
15    } else {
16        bad = 1;
17        return -1;
18    }
19 }
20
21 main {
22     fdir = openb_ec("../inputs/zip/stuff-1.zip", 1);
23     fent = openb_ec("../inputs/zip/stuff-1.zip", 1);
24     if (!valid(fdir) || !valid(fent)) {
25         exit(1);
26     }
27
28     eocd = size_ec(fdir) - 4;
29     found = 0;
30     while (!found) {
31         x = seek_ec(fdir, eocd);
32         if (x < 0) {

```

```

33     exit(1);
34 }
35 dir_sig = readintbytesl(fdir, 4);
36 if (dir_sig == 101010256) { // 0x06054b50
37     found = 1;
38 } else {
39     eocd = eocd - 1;
40 }
41 }
42 if (!found) {
43     exit(1);
44 }
45 diskid = readintbytesl(fdir, 2);
46 ndisk = readintbytesl(fdir, 2);
47 dirid = readintbytesl(fdir, 2);
48 ndir = readintbytesl(fdir, 2);
49
50 dir_size = readintbytesl(fdir, 4);
51 dir_start = readintbytesl(fdir, 4);
52 eocd_comm_size = readintbytesl(fdir, 2);
53 if (eocd_comm_size > 0) {
54     eocd_comm = malloc_ec(eocd_comm_size);
55     if (!valid(eocd_comm)) {
56         exit(1);
57     }
58     i = 0;
59     while (i < eocd_comm_size) {
60         b = read_ec(fdir);
61         if (b < 0) {
62             exit(1);
63         }
64         eocd_comm[i] = b;
65         i = i + 1;
66     }
67 }
68
69 x = seek_ec(fdir, dir_start);
70 if (x < 0) {
71     exit(1);
72 }
73 lookatb (pos_ec(fdir) < eocd && pos_ec(fdir) >= 0, fdir, 28, 2, 40) {
74     bad = 0;
75     dir_sig = readintbytesl(fdir, 4);
76     dir_ver_made = readintbytesl(fdir, 2);
77     dir_ver_extr = readintbytesl(fdir, 2);
78     dir_flag = readintbytesl(fdir, 2);
79     dir_comp = readintbytesl(fdir, 2);
80     dir_modif = readintbytesl(fdir, 4);
81     dir_crc = readintbytesl(fdir, 4);
82     dir_ent_size = readintbytesl(fdir, 4);
83     dir_ent_size_uncomp = readintbytesl(fdir, 4);
84     dir_name_size = readintbytesl(fdir, 2);
85     dir_extr_size = readintbytesl(fdir, 2);
86     dir_comm_size = readintbytesl(fdir, 2);
87     dir_diskid = readintbytesl(fdir, 2);
88     dir_attr = malloc_ec(6);
89     if (valid(dir_attr)) {
90         i = 0;
91         while (i < 6) {
92             b = read_ec(fdir);
93             if (b < 0) {
94                 bad = 1;
95             }
96             dir_attr[i] = b;
97             i = i + 1;
98         }
99     dir_ent_start = readintbytesl(fdir, 4);
100     if (dir_sig == 33639248 && dir_extr_size == 24 && dir_comm_size == 0 &&
101         dir_name_size > 0) {
102         dir_name = malloc_ec(dir_name_size);
103         if (valid(dir_name)) {
104             i = 0;
105             while (i < dir_name_size) {
106                 b = read_ec(fdir);
107                 if (b < 0) {
108                     bad = 1;

```

```

108     }
109     dir_name[i] = b;
110     i = i + 1;
111 }
112 dir_extr = malloc_ec(dir_extr_size);
113 if (valid(dir_extr)) {
114     i = 0;
115     while (i < dir_extr_size) {
116         b = read_ec(fdir);
117         if (b < 0) {
118             bad = 1;
119         }
120         dir_extr[i] = b;
121         i = i + 1;
122     }
123
124     x = seek_ec(fent, dir_ent_start);
125     if (x >= 0) {
126         ent_sig = readintbytesl(fent, 4);
127         ent_ver_extr = readintbytesl(fent, 2);
128         ent_flag = readintbytesl(fent, 2);
129         ent_comp = readintbytesl(fent, 2);
130         ent_modif = readintbytesl(fent, 4);
131         ent_crc = readintbytesl(fent, 4);
132         ent_ent_size = readintbytesl(fent, 4);
133         ent_ent_size_uncomp = readintbytesl(fent, 4);
134         ent_name_size = readintbytesl(fent, 2);
135         ent_extr_size = readintbytesl(fent, 2);
136
137         print(dir_name);
138         print('\n');
139         if (ent_sig == 67324752 && ent_ver_extr == dir_ver_extr &&
140             ent_flag == dir_flag && ent_comp == dir_comp && ent_modif ==
141             dir_modif && ent_name_size == dir_name_size &&
142             ent_name_size > 0) {
143             ent_name = malloc_ec(ent_name_size);
144             if (valid(ent_name)) {
145                 i = 0;
146                 while (i < ent_name_size) {
147                     b = read_ec(fent);
148                     if (b < 0) {
149                         bad = 1;
150                     }
151                     ent_name[i] = b;
152                     i = i + 1;
153                 }
154                 if (ent_extr_size > 0) {
155                     ent_extr = malloc_ec(ent_extr_size);
156                     if (valid(ent_extr)) {
157                         i = 0;
158                         while (i < ent_extr_size) {
159                             b = read_ec(fent);
160                             if (b < 0) {
161                                 bad = 1;
162                             }
163                             ent_extr[i] = b;
164                             i = i + 1;
165                         }
166                     } else {
167                         bad = 1;
168                     }
169                 }
170                 if (ent_flag & 8 == 0 && ent_crc == dir_crc && ent_ent_size
171                     == dir_ent_size && ent_ent_size_uncomp ==
172                     dir_ent_size_uncomp && ent_ent_size > 0) {
173                     raw_data = malloc_ec(ent_ent_size);
174                     if (valid(raw_data)) {
175                         i = 0;
176                         while (i < ent_ent_size) {
177                             b = read_ec(fent);
178                             if (b < 0) {
179                                 bad = 1;
180                             }
181                             raw_data[i] = b;
182                             i = i + 1;

```

```

179     }
180     // decompress
181     // check crc
182     if (!bad) {
183         print(ent_name);
184         print('\n');
185         print(raw_data);
186         print('\n');
187     } // skip unit
188     x = free_ec(ent_name);
189     x = free_ec(raw_data);
190 } // skip unit
191 } // skip unit
192 } // skip unit
193 } // skip unit
194 } // skip unit
195 } // skip unit
196 } // skip unit
197 } // skip unit
198 } // skip unit
199 }
200 x = seek_ec(fdir, eocd);
201 return 0;
202 }

```

A.5.4 Plain-loop version

```

1  bad = 0;
2
3  func readintbytesl (f, n) { // little endian
4      x = read_ec(f);
5      if (x >= 0) {
6          if (n == 1) {
7              return x;
8          } else {
9              y = readintbytesl(f, n-1);
10             if (bad) {
11                 return -1;
12             }
13             return (y << 8) + x;
14         }
15     } else {
16         bad = 1;
17         return -1;
18     }
19 }
20
21 main {
22     fdir = openb_ec("../inputs/zip/stuff-1.zip", 1);
23     fent = openb_ec("../inputs/zip/stuff-1.zip", 1);
24     if (!valid(fdir) || !valid(fent)) {
25         exit(1);
26     }
27
28     eocd = size_ec(fdir) - 4;
29     found = 0;
30     while (!found) {
31         x = seek_ec(fdir, eocd);
32         if (x < 0) {
33             exit(1);
34         }
35         dir_sig = readintbytesl(fdir, 4);
36         if (dir_sig == 101010256) { // 0x06054b50
37             found = 1;
38         } else {
39             eocd = eocd - 1;
40         }
41     }
42     if (!found) {
43         exit(1);
44     }
45     diskid = readintbytesl(fdir, 2);
46     ndisk = readintbytesl(fdir, 2);

```

```

47  dirid = readintbytesl(fdir, 2);
48  ndir = readintbytesl(fdir, 2);
49
50  dir_size = readintbytesl(fdir, 4);
51  dir_start = readintbytesl(fdir, 4);
52  eocd_comm_size = readintbytesl(fdir, 2);
53  if (eocd_comm_size > 0) {
54      eocd_comm = malloc_ec(eocd_comm_size);
55      if (!valid(eocd_comm)) {
56          exit(1);
57      }
58      i = 0;
59      while (i < eocd_comm_size) {
60          b = read_ec(fdir);
61          if (b < 0) {
62              exit(1);
63          }
64          eocd_comm[i] = b;
65          i = i + 1;
66      }
67  }
68
69  x = seek_ec(fdir, dir_start);
70  if (x < 0) {
71      exit(1);
72  }
73  finish = 0;
74  while (!finish && pos_ec(fdir) < eocd && pos_ec(fdir) >= 0) {
75      bad = 0;
76      dir_sig = readintbytesl(fdir, 4);
77      dir_ver_made = readintbytesl(fdir, 2);
78      dir_ver_extr = readintbytesl(fdir, 2);
79      dir_flag = readintbytesl(fdir, 2);
80      dir_comp = readintbytesl(fdir, 2);
81      dir_modif = readintbytesl(fdir, 4);
82      dir_crc = readintbytesl(fdir, 4);
83      dir_ent_size = readintbytesl(fdir, 4);
84      dir_ent_size_uncomp = readintbytesl(fdir, 4);
85      dir_name_size = readintbytesl(fdir, 2);
86      eou = pos_ec(fdir) + dir_name_size + 40;
87      if (!bad && dir_name_size >= 0) {
88          dir_extr_size = readintbytesl(fdir, 2);
89          dir_comm_size = readintbytesl(fdir, 2);
90          dir_diskid = readintbytesl(fdir, 2);
91          dir_attr = malloc_ec(6);
92          if (!bad && valid(dir_attr)) {
93              i = 0;
94              while (i < 6) {
95                  b = read_ec(fdir);
96                  if (b < 0) {
97                      bad = 1;
98                  }
99                  dir_attr[i] = b;
100                 i = i + 1;
101             }
102             dir_ent_start = readintbytesl(fdir, 4);
103             if (!bad && dir_sig == 33639248 && dir_extr_size == 24 &&
104                 dir_comm_size == 0 && dir_name_size > 0) {
105                 dir_name = malloc_ec(dir_name_size);
106                 if (valid(dir_name)) {
107                     i = 0;
108                     while (i < dir_name_size) {
109                         b = read_ec(fdir);
110                         if (b < 0) {
111                             bad = 1;
112                         }
113                         dir_name[i] = b;
114                         i = i + 1;
115                     }
116                     dir_extr = malloc_ec(dir_extr_size);
117                     if (!bad && valid(dir_extr)) {
118                         i = 0;
119                         while (i < dir_extr_size) {
120                             b = read_ec(fdir);
121                             if (b < 0) {
122                                 bad = 1;

```

```

122     }
123     dir_extr[i] = b;
124     i = i + 1;
125 }
126
127 x = seek_ec(fent, dir_ent_start);
128 if (x >= 0) {
129     ent_sig = readintbytesl(fent, 4);
130     ent_ver_extr = readintbytesl(fent, 2);
131     ent_flag = readintbytesl(fent, 2);
132     ent_comp = readintbytesl(fent, 2);
133     ent_modif = readintbytesl(fent, 4);
134     ent_crc = readintbytesl(fent, 4);
135     ent_ent_size = readintbytesl(fent, 4);
136     ent_ent_size_uncomp = readintbytesl(fent, 4);
137     ent_name_size = readintbytesl(fent, 2);
138     ent_extr_size = readintbytesl(fent, 2);
139
140     print(dir_name);
141     print('\n');
142     if (ent_sig == 67324752 && ent_ver_extr == dir_ver_extr &&
143         ent_flag == dir_flag && ent_comp == dir_comp && ent_modif
144         == dir_modif && ent_name_size == dir_name_size &&
145         ent_name_size > 0) {
146         ent_name = malloc_ec(ent_name_size);
147         if (valid(ent_name)) {
148             i = 0;
149             while (i < ent_name_size) {
150                 b = read_ec(fent);
151                 if (b < 0) {
152                     bad = 1;
153                 }
154                 ent_name[i] = b;
155                 i = i + 1;
156             }
157             if (ent_extr_size > 0) {
158                 ent_extr = malloc_ec(ent_extr_size);
159                 if (valid(ent_extr)) {
160                     i = 0;
161                     while (i < ent_extr_size) {
162                         b = read_ec(fent);
163                         if (b < 0) {
164                             bad = 1;
165                         }
166                         ent_extr[i] = b;
167                         i = i + 1;
168                     }
169                 } else {
170                     bad = 1;
171                 }
172             }
173             if (ent_flag & 8 == 0 && ent_crc == dir_crc &&
174                 ent_ent_size == dir_ent_size && ent_ent_size_uncomp ==
175                 dir_ent_size_uncomp && ent_ent_size > 0) {
176                 raw_data = malloc_ec(ent_ent_size);
177                 if (valid(raw_data)) {
178                     i = 0;
179                     while (i < ent_ent_size) {
180                         b = read_ec(fent);
181                         if (b < 0) {
182                             bad = 1;
183                         }
184                         raw_data[i] = b;
185                         i = i + 1;
186                     }
187                 }
188                 // decompress
189                 // check crc
190                 if (!bad) {
191                     print(ent_name);
192                     print('\n');
193                     print(raw_data);
194                     print('\n');
195                 } // skip unit
196                 x = free_ec(ent_name);
197                 x = free_ec(raw_data);

```

```

193         } // skip unit
194     } // skip unit
195     } // skip unit
196     } // skip unit
197     } // skip unit
198     } // skip unit
199     } // skip unit
200     } // skip unit
201 } // skip unit
202 x = seek_ec(fdir, eou);
203 if (x < 0) { // give up
204     finish = 1;
205 }
206 } else { // give up
207     finish = 1;
208 }
209 }
210 x = seek_ec(fdir, eocd);
211 return 0;
212 }

```

A.6 RGIF

A.6.1 Fully-implicit version

```

1  buffer = malloc(7500);
2  number = 0;
3
4  func readintbytes1 (f, n) { // little endian
5      x = read(f);
6      if (n == 1) {
7          return x;
8      } else {
9          y = readintbytes1(f, n-1);
10         return (y << 8) + x;
11     }
12 }
13
14 func readblocks (f, echo) {
15     i = 0;
16     stop = 0;
17     lookatb (!stop, f, 0, 1, 0) {
18         len = read(f);
19         if (echo) {
20             print(len);
21         }
22         if (len == 0) {
23             stop = 1;
24         } else {
25             while (!end(f)) {
26                 x = read(f);
27                 if (echo) {
28                     print(x);
29                 }
30                 buffer[i] = x;
31                 i = i + 1;
32             }
33         }
34     }
35     assert(stop);
36     return i;
37 }
38
39 func printintbytes1 (n) {
40     b = number & 255;
41     print(b);
42     if (n == 1) {
43         return 0;
44     } else {

```

```

45     number = number >> 8;
46     dummy = printintbytes1(n - 1);
47     return 0;
48 }
49 }
50
51 main {
52     f = openb("../inputs/rgif/welcome2-block.rgif", 1);
53
54     // header block
55     header = malloc(6);
56     i = 0;
57     while (i < 6) {
58         x = read(f);
59         print(x);
60         header[i] = x;
61         i = i + 1;
62     }
63     assert(header[0] == 'G' && header[1] == 'I' && header[2] == 'F' && header[3]
64            == '8' && (header[4] == '9' || header[4] == '7') && header[5] == 'a');
65     free(header);
66
67     // logical screen descriptor
68     canvasw = readintbytes1(f, 2);
69     number = canvasw;
70     canvash = readintbytes1(f, 2);
71     number = canvash;
72     dummy = printintbytes1(2);
73     x = read(f);
74     print(x);
75     gflag = (x & 128) >> 7;
76     bpp = (x & 112) >> 4;
77     gsort = (x & 8) >> 3;
78     background = read(f);
79     print(background);
80     aspect = read(f);
81     print(aspect);
82     assert(gflag && x & 7 == bpp);
83
84     // global color table
85     nglobal = 2 << bpp;
86     gcolors = malloc(nglobal);
87     i = 0;
88     while (i < nglobal) {
89         x = readintbytes1(f, 3);
90         number = x;
91         dummy = printintbytes1(3);
92         gcolors[i] = x;
93         i = i + 1;
94     }
95
96     trailer = 0;
97     inspectb (!trailer, f, 0, 2, 1) {
98         length = readintbytes1(f, 2);
99         x = read(f);
100        if (x == 59) { // trailer block
101            print(x);
102            trailer = 1;
103        } else {
104            label = read(f);
105            if (x == 33 && (label == 255 || label == 254)) {
106                if (label == 255) { // application extension
107                    idlen = read(f);
108                    id = malloc(idlen);
109                    i = 0;
110                    while (i < idlen) {
111                        x = read(f);
112                        id[i] = x;
113                        i = i + 1;
114                    }
115                    free(id);
116                    applen = readblocks(f, 0);
117                    app = malloc(applen);
118                    i = 0;
119                    while (i < applen) {

```



```

120         app[i] = buffer[i];
121         i = i + 1;
122     }
123 } else { // comment extension
124     commentlen = readblocks(f, 0);
125     comments = malloc(commentlen);
126     i = 0;
127     while (i < commentlen) {
128         comments[i] = buffer[i];
129         i = i + 1;
130     }
131     free(comments);
132 }
133 } else {
134     if (x == 33 && label == 249) { // graphics control extension
135         print(x);
136         print(label);
137         blocksize = read(f);
138         print(blocksize);
139         ctrltext = read(f);
140         print(ctrltext);
141         delay = readintbytesl(f, 2);
142         number = delay;
143         dummy = printintbytesl(2);
144         transparent = read(f);
145         print(transparent);
146         terminator = read(f);
147         print(terminator);
148         assert(terminator == 0);
149     } else {
150         assert(x != 33);
151         seek(f, pos(f)-2);
152     }
153
154     // image descriptor
155     x = read(f);
156     assert(x == 44);
157     print(x);
158     left = readintbytesl(f, 2);
159     number = left;
160     dummy = printintbytesl(2);
161     top = readintbytesl(f, 2);
162     number = top;
163     dummy = printintbytesl(2);
164     width = readintbytesl(f, 2);
165     number = width;
166     dummy = printintbytesl(2);
167     height = readintbytesl(f, 2);
168     number = height;
169     dummy = printintbytesl(2);
170     assert(left == 0 && top == 0 && width == canvasw && height == canvash)
171     ;
172     x = read(f);
173     print(x);
174     lflag = (x & 128) >> 7;
175     linterlace = (x & 64) >> 6;
176     lsort = (x & 32) >> 5;
177     lctsize = x & 7;
178
179     // local color table
180     if (lflag) {
181         nlocal = 2 << bpp;
182         lcolors = malloc(nlocal);
183         i = 0;
184         while (i < nlocal) {
185             x = readintbytesl(f, 3);
186             number = x;
187             dummy = printintbytesl(3);
188             lcolors[i] = x;
189             i = i + 1;
190         }
191     }
192
193     // data sub-blocks
194     lzw = read(f);
195     print(lzw);

```

```

195         datalen = readblocks(f, 1);
196         // decompress
197         // decode
198     }
199 }
200 }
201 free(buffer);
202 if (!trailer) {
203     x = 59;
204     print(x);
205 }
206 return 0;
207 }

```

A.6.2 Protective-check version

```

1  buffer = malloc_ec(7500);
2  number = 0;
3
4  func readintbytes1 (f, n) { // little endian
5      x = read_ec(f);
6      assert(x >= 0);
7      if (n == 1) {
8          return x;
9      } else {
10         y = readintbytes1(f, n-1);
11         return (y << 8) + x;
12     }
13 }
14
15 func readblocks (f, echo) {
16     i = 0;
17     stop = 0;
18     lookatb (!stop, f, 0, 1, 0) {
19         len = read_ec(f);
20         assert(len >= 0);
21         if (echo) {
22             print(len);
23         }
24         if (len == 0) {
25             stop = 1;
26         } else {
27             while (!end_ec(f)) {
28                 x = read_ec(f);
29                 assert(x >= 0 && i < 7500);
30                 if (echo) {
31                     print(x);
32                 }
33                 buffer[i] = x;
34                 i = i + 1;
35             }
36         }
37     }
38     assert(stop);
39     return i;
40 }
41
42 func printintbytes1 (n) {
43     b = number & 255;
44     print(b);
45     if (n == 1) {
46         return 0;
47     } else {
48         number = number >> 8;
49         dummy = printintbytes1(n - 1);
50         return 0;
51     }
52 }
53
54 main {
55     f = openb_ec("../inputs/rgif/welcome2-block.rgif", 1);
56     assert(valid(buffer) && valid(f));
57 }

```

```

58 // header block
59 header = malloc_ec(6);
60 assert(valid(header));
61 i = 0;
62 while (i < 6) {
63     x = read_ec(f);
64     assert(x >= 0);
65     print(x);
66     header[i] = x;
67     i = i + 1;
68 }
69 assert(header[0] == 'G' && header[1] == 'I' && header[2] == 'F' && header[3]
    == '8' && (header[4] == '9' || header[4] == '7') && header[5] == 'a');
70 x = free_ec(header);
71
72 // logical screen descriptor
73 canvasw = readintbytesl(f, 2);
74 number = canvasw;
75 dummy = printintbytesl(2);
76 canvash = readintbytesl(f, 2);
77 number = canvash;
78 dummy = printintbytesl(2);
79 x = read_ec(f);
80 assert(x >= 0);
81 print(x);
82 gflag = (x & 128) >> 7;
83 bpp = (x & 112) >> 4;
84 gsort = (x & 8) >> 3;
85 background = read_ec(f);
86 print(background);
87 aspect = read_ec(f);
88 print(aspect);
89 assert(gflag && x & 7 == bpp && background >= 0 && aspect >= 0);
90
91 // global color table
92 nglobal = 2 << bpp;
93 gcolors = malloc_ec(nglobal);
94 assert(valid(gcolors));
95 i = 0;
96 while (i < nglobal) {
97     x = readintbytesl(f, 3);
98     number = x;
99     dummy = printintbytesl(3);
100    gcolors[i] = x;
101    i = i + 1;
102 }
103
104 trailer = 0;
105 inspectb (!trailer, f, 0, 2, 1) {
106     length = readintbytesl(f, 2);
107     x = read_ec(f);
108     assert(x >= 0);
109     if (x == 59) { // trailer block
110         print(x);
111         trailer = 1;
112     } else {
113         label = read_ec(f);
114         assert(label >= 0);
115         if (x == 33 && (label == 255 || label == 254)) {
116             if (label == 255) { // application extension
117                 idlen = read_ec(f);
118                 assert(idlen >= 0);
119                 id = malloc_ec(idlen);
120                 assert(valid(id));
121                 i = 0;
122                 while (i < idlen) {
123                     x = read_ec(f);
124                     assert(x >= 0);
125                     id[i] = x;
126                     i = i + 1;
127                 }
128                 x = free_ec(id);
129                 applen = readblocks(f, 0);
130                 app = malloc_ec(applen);
131                 assert(valid(app));
132                 i = 0;

```

```

133     while (i < applen) {
134         app[i] = buffer[i];
135         i = i + 1;
136     }
137 } else { // comment extension
138     commentlen = readblocks(f, 0);
139     comments = malloc_ec(commentlen);
140     assert(valid(comments));
141     i = 0;
142     while (i < commentlen) {
143         comments[i] = buffer[i];
144         i = i + 1;
145     }
146     x = free_ec(comments);
147 }
148 } else {
149     if (x == 33 && label == 249) { // graphics control extension
150         print(x);
151         print(label);
152         blocksize = read_ec(f);
153         print(blocksize);
154         ctrltext = read_ec(f);
155         print(ctrltext);
156         delay = readintbytes1(f, 2);
157         number = delay;
158         dummy = printintbytes1(2);
159         transparent = read_ec(f);
160         print(transparent);
161         terminator = read_ec(f);
162         print(terminator);
163         assert(blocksize >= 0 && ctrltext >= 0 && transparent >= 0 &&
164                terminator == 0);
165     } else {
166         assert(x != 33);
167         x = seek_ec(f, pos_ec(f)-2);
168         assert(x >= 0);
169     }
170     // image descriptor
171     x = read_ec(f);
172     assert(x == 44);
173     print(x);
174     left = readintbytes1(f, 2);
175     number = left;
176     dummy = printintbytes1(2);
177     top = readintbytes1(f, 2);
178     number = top;
179     dummy = printintbytes1(2);
180     width = readintbytes1(f, 2);
181     number = width;
182     dummy = printintbytes1(2);
183     height = readintbytes1(f, 2);
184     number = height;
185     dummy = printintbytes1(2);
186     assert(left == 0 && top == 0 && width == canvasw && height == canvash)
187     ;
188     x = read_ec(f);
189     assert(x >= 0);
190     print(x);
191     lflag = (x & 128) >> 7;
192     linterlace = (x & 64) >> 6;
193     lsort = (x & 32) >> 5;
194     lctsize = x & 7;
195     // local color table
196     if (lflag) {
197         nlocal = 2 << bpp;
198         lcolors = malloc_ec(nlocal);
199         assert(valid(lcolors));
200         i = 0;
201         while (i < nlocal) {
202             x = readintbytes1(f, 3);
203             number = x;
204             dummy = printintbytes1(3);
205             lcolors[i] = x;
206             i = i + 1;

```

```

207     }
208     }
209
210     // data sub-blocks
211     lzw = read_ec(f);
212     assert(lzw >= 0);
213     print(lzw);
214     datalen = readblocks(f, 1);
215     // decompress
216     // decode
217 }
218 }
219 }
220 x = free_ec(buffer);
221 if (!trailer) {
222     x = 59;
223     print(x);
224 }
225 return 0;
226 }

```

A.6.3 Explicit-recovery version

```

1  buffer = malloc_ec(7500);
2  number = 0;
3  prt = 1;
4
5  bad = 0;
6  out = malloc_ec(7500);
7  outidx = 0;
8
9  func readintbytes1 (f, n) { // little endian
10     x = read_ec(f);
11     if (x < 0) {
12         bad = 1;
13         return -1;
14     }
15     if (n == 1) {
16         return x;
17     } else {
18         y = readintbytes1(f, n-1);
19         if (bad) {
20             return -1;
21         }
22         return (y << 8) + x;
23     }
24 }
25
26 func readblocks (f, echo) {
27     i = 0;
28     stop = 0;
29     lookatb (!stop, f, 0, 1, 0) {
30         len = read_ec(f);
31         if (len >= 0) {
32             if (echo) {
33                 dummy = write(len);
34             }
35             if (len == 0) {
36                 stop = 1;
37             } else {
38                 while (!end_ec(f) && !bad) {
39                     x = read_ec(f);
40                     if (x >= 0 && i < 7500) {
41                         if (echo) {
42                             dummy = write(x);
43                         }
44                         buffer[i] = x;
45                         i = i + 1;
46                     } else {
47                         bad = 1;
48                     }
49                 }
50             }

```

```

51     } else {
52         bad = 1;
53     }
54 }
55 if (!stop) {
56     bad = 1;
57 }
58 return i;
59 }
60
61 func write (x) {
62     if (outidx < 7500) {
63         out[outidx] = x;
64         outidx = outidx + 1;
65         return 0;
66     } else {
67         bad = 1;
68         return -1;
69     }
70 }
71
72 func writeintbytes1 (n) {
73     b = number & 255;
74     if (prt) {
75         print(b);
76     } else {
77         dummy = write(b);
78     }
79     if (n == 1) {
80         return 0;
81     } else {
82         number = number >> 8;
83         dummy = writeintbytes1(n - 1);
84         return 0;
85     }
86 }
87
88 main {
89     f = openb_ec("../inputs/rgif/welcome2-block.rgif", 1);
90     if (!valid(buffer) || !valid(out) || !valid(f)) {
91         exit(1);
92     }
93
94     // header block
95     header = malloc_ec(6);
96     if (!valid(header)) {
97         exit(1);
98     }
99     i = 0;
100    while (i < 6) {
101        x = read_ec(f);
102        if (x < 0) {
103            exit(1);
104        }
105        print(x);
106        header[i] = x;
107        i = i + 1;
108    }
109    if (!(header[0] == 'G' && header[1] == 'I' && header[2] == 'F' && header[3]
        == '8' && (header[4] == '9' || header[4] == '7') && header[5] == 'a')) {
110        exit(1);
111    }
112    x = free_ec(header);
113
114    // logical screen descriptor
115    canvasw = readintbytes1(f, 2);
116    number = canvasw;
117    dummy = writeintbytes1(2);
118    canvash = readintbytes1(f, 2);
119    number = canvash;
120    dummy = writeintbytes1(2);
121    x = read_ec(f);
122    if (bad || x < 0) {
123        exit(1);
124    }
125    print(x);

```

```

126 gflag = (x & 128) >> 7;
127 bpp = (x & 112) >> 4;
128 gsort = (x & 8) >> 3;
129 background = read_ec(f);
130 print(background);
131 aspect = read_ec(f);
132 print(aspect);
133 if (!gflag || x & 7 != bpp || background < 0 || aspect < 0) {
134     exit(1);
135 }
136
137 // global color table
138 nglobal = 2 << bpp;
139 gcolors = malloc_ec(nglobal);
140 if (!valid(gcolors)) {
141     exit(1);
142 }
143 i = 0;
144 while (i < nglobal) {
145     x = readintbytes1(f, 3);
146     if (bad) {
147         exit(1);
148     }
149     number = x;
150     dummy = writeintbytes1(3);
151     gcolors[i] = x;
152     i = i + 1;
153 }
154
155 prt = 0;
156 trailer = 0;
157 lookatb(!trailer, f, 0, 2, 1) {
158     bad = 0;
159     outidx = 0;
160     length = readintbytes1(f, 2);
161     x = read_ec(f);
162     if (x >= 0) {
163         if (x == 59) { // trailer block
164             dummy = write(x);
165             trailer = 1;
166         } else {
167             label = read_ec(f);
168             if (label < 0) {
169                 bad = 1;
170             }
171             if (x == 33 && (label == 255 || label == 254)) {
172                 if (label == 255) { // appliation extension
173                     idlen = read_ec(f);
174                     if (idlen < 0) {
175                         bad = 1;
176                     }
177                     id = malloc_ec(idlen);
178                     if (valid(id)) {
179                         i = 0;
180                         while (i < idlen) {
181                             x = read_ec(f);
182                             if (x < 0) {
183                                 bad = 1;
184                             }
185                             id[i] = x;
186                             i = i + 1;
187                         }
188                         x = free_ec(id);
189                         applen = readblocks(f, 0);
190                         app = malloc_ec(applen);
191                         if (valid(app)) {
192                             i = 0;
193                             while (i < applen) {
194                                 app[i] = buffer[i];
195                                 i = i + 1;
196                             }
197                         } else {
198                             bad = 1;
199                         }
200                     } else {
201                         bad = 1;

```

```

202     }
203 } else { // comment extension
204     commentlen = readblocks(f, 0);
205     comments = malloc_ec(commentlen);
206     if (valid(comments)) {
207         i = 0;
208         while (i < commentlen) {
209             comments[i] = buffer[i];
210             i = i + 1;
211         }
212         x = free_ec(comments);
213     } else {
214         bad = 1;
215     }
216 }
217 } else {
218     if (x == 33 && label == 249) { // graphics control extension
219         dummy = write(x);
220         dummy = write(label);
221         blocksize = read_ec(f);
222         dummy = write(blocksize);
223         ctrleft = read_ec(f);
224         dummy = write(ctrleft);
225         delay = readintbytesl(f, 2);
226         number = delay;
227         dummy = writeintbytesl(2);
228         transparent = read_ec(f);
229         dummy = write(transparent);
230         terminator = read_ec(f);
231         dummy = write(terminator);
232         if (blocksize < 0 || ctrleft < 0 || transparent < 0 || terminator
233             != 0) {
234             bad = 1;
235         }
236     } else {
237         if (x == 33) {
238             bad = 1;
239         }
240         x = seek_ec(f, pos_ec(f)-2);
241         if (x < 0) {
242             bad = 1;
243         }
244     }
245     // image descriptor
246     x = read_ec(f);
247     if (x != 44) {
248         bad = 1;
249     }
250     dummy = write(x);
251     left = readintbytesl(f, 2);
252     number = left;
253     dummy = writeintbytesl(2);
254     top = readintbytesl(f, 2);
255     number = top;
256     dummy = writeintbytesl(2);
257     width = readintbytesl(f, 2);
258     number = width;
259     dummy = writeintbytesl(2);
260     height = readintbytesl(f, 2);
261     number = height;
262     dummy = writeintbytesl(2);
263     if (left != 0 || top != 0 || width != canvasw || height != canvash)
264     {
265         bad = 1;
266     }
267     x = read_ec(f);
268     if (x < 0) {
269         bad = 1;
270     }
271     dummy = write(x);
272     lflag = (x & 128) >> 7;
273     linterlace = (x & 64) >> 6;
274     lsort = (x & 32) >> 5;
275     lctsize = x & 7;

```



```

276         // local color table
277         if (lflag) {
278             nlocal = 2 << bpp;
279             lcolors = malloc_ec(nlocal);
280             if (valid(lcolors)) {
281                 i = 0;
282                 while (i < nlocal) {
283                     x = readintbytes1(f, 3);
284                     number = x;
285                     dummy = writeintbytes1(3);
286                     lcolors[i] = x;
287                     i = i + 1;
288                 }
289             } else {
290                 bad = 1;
291             }
292         }
293
294         // data sub-blocks
295         lzw = read_ec(f);
296         if (lzw < 0) {
297             bad = 1;
298         }
299         dummy = write(lzw);
300         datalen = readblocks(f, 1);
301         // decompress
302         // decode
303     }
304 }
305 } else {
306     bad = 1;
307 }
308 if (!bad) {
309     i = 0;
310     while (i < outidx) {
311         x = out[i];
312         print(x);
313         i = i + 1;
314     }
315 } // skip unit
316 }
317 x = free_ec(buffer);
318 if (!trailer) {
319     x = 59;
320     print(x);
321 }
322 return 0;
323 }

```

A.6.4 Plain-loop version

```

1  buffer = malloc_ec(7500);
2  number = 0;
3  prt = 1;
4
5  bad = 0;
6  out = malloc_ec(7500);
7  outidx = 0;
8
9  func readintbytes1 (f, n) { // little endian
10     x = read_ec(f);
11     if (x < 0) {
12         bad = 1;
13         return -1;
14     }
15     if (n == 1) {
16         return x;
17     } else {
18         y = readintbytes1(f, n-1);
19         if (bad) {
20             return -1;
21         }
22         return (y << 8) + x;

```

```

23     }
24 }
25
26 func readblocks (f, echo) {
27     i = 0;
28     stop = 0;
29     while (!stop) {
30         len = read_ec(f);
31         if (len >= 0) {
32             if (echo) {
33                 dummy = write(len);
34             }
35             if (len == 0) {
36                 stop = 1;
37             } else {
38                 j = 0;
39                 while (j < len) {
40                     x = read_ec(f);
41                     if (x >= 0 && i < 7500) {
42                         if (echo) {
43                             dummy = write(x);
44                         }
45                         buffer[i] = x;
46                         i = i + 1;
47                     } else {
48                         bad = 1;
49                     }
50                     j = j + 1;
51                 }
52             }
53         } else {
54             bad = 1;
55         }
56     }
57     return i;
58 }
59
60 func write (x) {
61     if (outidx < 7500) {
62         out[outidx] = x;
63         outidx = outidx + 1;
64         return 0;
65     } else {
66         bad = 1;
67         return -1;
68     }
69 }
70
71 func writeintbytes1 (n) {
72     b = number & 255;
73     if (prt) {
74         print(b);
75     } else {
76         dummy = write(b);
77     }
78     if (n == 1) {
79         return 0;
80     } else {
81         number = number >> 8;
82         dummy = writeintbytes1(n - 1);
83         return 0;
84     }
85 }
86
87 main {
88     f = openb_ec("../inputs/rgif/welcome2-block.rgif", 1);
89     if (!valid(buffer) || !valid(out) || !valid(f)) {
90         exit(1);
91     }
92
93     // header block
94     header = malloc_ec(6);
95     if (!valid(header)) {
96         exit(1);
97     }
98     i = 0;

```

```

99     while (i < 6) {
100         x = read_ec(f);
101         if (x < 0) {
102             exit(1);
103         }
104         print(x);
105         header[i] = x;
106         i = i + 1;
107     }
108     if (!(header[0] == 'G' && header[1] == 'I' && header[2] == 'F' && header[3]
109         == '8' && (header[4] == '9' || header[4] == '7') && header[5] == 'a')) {
110         exit(1);
111     }
112     x = free_ec(header);
113     // logical screen descriptor
114     canvasw = readintbytesl(f, 2);
115     number = canvasw;
116     dummy = writeintbytesl(2);
117     canvash = readintbytesl(f, 2);
118     number = canvash;
119     dummy = writeintbytesl(2);
120     x = read_ec(f);
121     if (bad || x < 0) {
122         exit(1);
123     }
124     print(x);
125     gflag = (x & 128) >> 7;
126     bpp = (x & 112) >> 4;
127     gsort = (x & 8) >> 3;
128     background = read_ec(f);
129     print(background);
130     aspect = read_ec(f);
131     print(aspect);
132     if (!gflag || x & 7 != bpp || background < 0 || aspect < 0) {
133         exit(1);
134     }
135
136     // global color table
137     nglobal = 2 << bpp;
138     gcolors = malloc_ec(nglobal);
139     if (!valid(gcolors)) {
140         exit(1);
141     }
142     i = 0;
143     while (i < nglobal) {
144         x = readintbytesl(f, 3);
145         if (bad) {
146             exit(1);
147         }
148         number = x;
149         dummy = writeintbytesl(3);
150         gcolors[i] = x;
151         i = i + 1;
152     }
153
154     prt = 0;
155     trailer = 0;
156     finish = 0;
157     while (!trailer && !finish) {
158         bad = 0;
159         outidx = 0;
160         length = readintbytesl(f, 2);
161         if (length >= 0) {
162             eou = pos_ec(f) + length + 1;
163             x = read_ec(f);
164             if (x >= 0) {
165                 if (x == 59) { // trailer block
166                     dummy = write(x);
167                     trailer = 1;
168                 } else {
169                     label = read_ec(f);
170                     if (label < 0) {
171                         bad = 1;
172                     }
173                     if (x == 33 && (label == 255 || label == 254)) {

```

```

174     if (label == 255) { // appliation extension
175         idlen = read_ec(f);
176         if (idlen < 0) {
177             bad = 1;
178         }
179         id = malloc_ec(idlen);
180         if (valid(id)) {
181             i = 0;
182             while (i < idlen) {
183                 x = read_ec(f);
184                 if (x < 0) {
185                     bad = 1;
186                 }
187                 id[i] = x;
188                 i = i + 1;
189             }
190             x = free_ec(id);
191             applen = readblocks(f, 0);
192             app = malloc_ec(applen);
193             if (valid(app)) {
194                 i = 0;
195                 while (i < applen) {
196                     app[i] = buffer[i];
197                     i = i + 1;
198                 }
199             } else {
200                 bad = 1;
201             }
202         } else {
203             bad = 1;
204         }
205     } else { // comment extension
206         commentlen = readblocks(f, 0);
207         comments = malloc_ec(commentlen);
208         if (valid(comments)) {
209             i = 0;
210             while (i < commentlen) {
211                 comments[i] = buffer[i];
212                 i = i + 1;
213             }
214             x = free_ec(comments);
215         } else {
216             bad = 1;
217         }
218     }
219 } else {
220     if (x == 33 && label == 249) { // graphics control extension
221         dummy = write(x);
222         dummy = write(label);
223         blocksize = read_ec(f);
224         dummy = write(blocksize);
225         ctrlext = read_ec(f);
226         dummy = write(ctrlext);
227         delay = readintbytes1(f, 2);
228         number = delay;
229         dummy = writeintbytes1(2);
230         transparent = read_ec(f);
231         dummy = write(transparent);
232         terminator = read_ec(f);
233         dummy = write(terminator);
234         if (blocksize < 0 || ctrlext < 0 || transparent < 0 ||
235             terminator != 0) {
236             bad = 1;
237         }
238     } else {
239         if (x == 33) {
240             bad = 1;
241         }
242         x = seek_ec(f, pos_ec(f)-2);
243         if (x < 0) {
244             bad = 1;
245         }
246     }
247     // image descriptor
248     x = read_ec(f);

```

```

249     if (x != 44) {
250         bad = 1;
251     }
252     dummy = write(x);
253     left = readintbytes1(f, 2);
254     number = left;
255     dummy = writeintbytes1(2);
256     top = readintbytes1(f, 2);
257     number = top;
258     dummy = writeintbytes1(2);
259     width = readintbytes1(f, 2);
260     number = width;
261     dummy = writeintbytes1(2);
262     height = readintbytes1(f, 2);
263     number = height;
264     dummy = writeintbytes1(2);
265     if (left != 0 || top != 0 || width != canvasw || height != canvash
        ) {
266         bad = 1;
267     }
268     x = read_ec(f);
269     if (x < 0) {
270         bad = 1;
271     }
272     dummy = write(x);
273     lflag = (x & 128) >> 7;
274     linterlace = (x & 64) >> 6;
275     lsort = (x & 32) >> 5;
276     lctsize = x & 7;
277
278     // local color table
279     if (lflag) {
280         nlocal = 2 << bpp;
281         lcolors = malloc_ec(nlocal);
282         if (valid(lcolors)) {
283             i = 0;
284             while (i < nlocal) {
285                 x = readintbytes1(f, 3);
286                 number = x;
287                 dummy = writeintbytes1(3);
288                 lcolors[i] = x;
289                 i = i + 1;
290             }
291         } else {
292             bad = 1;
293         }
294     }
295
296     // data sub-blocks
297     lzw = read_ec(f);
298     if (lzw < 0) {
299         bad = 1;
300     }
301     dummy = write(lzw);
302     datalen = readblocks(f, 1);
303     // decompress
304     // decode
305 }
306 }
307 } else {
308     bad = 1;
309 }
310 if (pos_ec(f) > eou || pos_ec(f) < 0) {
311     bad = 1;
312 }
313 x = seek_ec(f, eou);
314 if (x < 0) { // give up
315     finish = 1;
316 }
317 if (!bad && !finish) {
318     i = 0;
319     while (i < outidx) {
320         x = out[i];
321         print(x);
322         i = i + 1;
323     }

```

```

324     } // skip unit
325   } else { // give up
326     finish = 1;
327   }
328 }
329 x = free_ec(buffer);
330 if (!trailer) {
331   x = 59;
332   print(x);
333 }
334 return 0;
335 }

```

A.7 PCAP/DNS

A.7.1 Fully-implicit version

```

1  dst_mac = malloc(6);
2  src_mac = malloc(6);
3  data = malloc(1);
4
5  func readintbytesl (f, n) { // little endian
6    x = read(f);
7    if (n == 1) {
8      return x;
9    } else {
10     y = readintbytesl(f, n-1);
11     return (y << 8) + x;
12   }
13 }
14
15 func readintbytesb (f, n) { // big endian
16   x = 0;
17   i = 0;
18   while (i < n) {
19     byte = read(f);
20     x = x << 8;
21     x = x | byte;
22     i = i + 1;
23   }
24   return x;
25 }
26
27 func parse_ethernet (f, dummy) {
28   i = 0;
29   while (i < 6) {
30     x = read(f);
31     dst_mac[i] = x;
32     i = i + 1;
33   }
34   i = 0;
35   while (i < 6) {
36     x = read(f);
37     src_mac[i] = x;
38     i = i + 1;
39   }
40
41   ether_type = readintbytesl(f, 2);
42   assert(ether_type == 8); // IPv4
43
44   dummy = parse_ipv4(f, 0);
45   return 0;
46 }
47
48 func check_sum (f, len) {
49   start = pos(f);
50   assert(len > 0 && len % 2 == 0);
51   sum = 0;
52   i = 0;

```

```

53     while (i < len / 2) {
54         x = readintbytesb(f, 2);
55         sum = sum + x;
56         i = i + 1;
57     }
58     hi = (sum >> 16) & 65535;
59     lo = sum & 65535;
60     assert(hi + lo == 65535);
61     seek(f, start);
62     return 0;
63 }
64
65 func print_ip (addr) {
66     b = (addr >> 24) & 255;
67     print(b);
68     print('.');
69     b = (addr >> 16) & 255;
70     print(b);
71     print('.');
72     b = (addr >> 8) & 255;
73     print(b);
74     print('.');
75     b = addr & 255;
76     print(b);
77     return 0;
78 }
79
80 func parse_ipv4 (f, dummy) {
81     dummy = check_sum(f, 20); // no options
82
83     x = read(f);
84     version = (x & 240) >> 4;
85     hdr_size = x & 15;
86     x = read(f);
87     total = readintbytesb(f, 2);
88
89     id = readintbytesb(f, 2);
90     fragment = readintbytesb(f, 2);
91     dont_frag = (fragment & 16384) >> 14;
92     more_frag = (fragment & 8192) >> 13;
93     offs_frag = fragment & 8191;
94
95     ttl = read(f);
96     ip_type = read(f);
97     checksum = readintbytesb(f, 2);
98
99     src_ip = readintbytesb(f, 4);
100    dst_ip = readintbytesb(f, 4);
101
102    assert(version == 4 && hdr_size == 5 && total >= hdr_size * 4 && fragment &
        32768 == 0 && more_frag == 0 && offs_frag == 0 && ip_type == 17); //
        IPv4, no options, UDP
103
104    print('\n');
105    dummy = print_ip(src_ip);
106    print(' ');
107    dummy = print_ip(dst_ip);
108    print(' ');
109    dummy = parse_udp(f, total - hdr_size * 4);
110
111    return 0;
112 }
113
114 func parse_udp (f, pkt_size) {
115     src_port = readintbytesb(f, 2);
116     dst_port = readintbytesb(f, 2);
117     udp_size = readintbytesb(f, 2);
118     checksum = readintbytesb(f, 2);
119
120     assert(pkt_size <= udp_size && pkt_size > 8);
121     if (src_port == 53 || dst_port == 53) { // DNS
122         dummy = parse_dns(f, pkt_size - 8);
123     } else { // others
124         if (valid(data)) {
125             free(data);
126         }

```

```

127     data_size = pkt_size - 8;
128     data = malloc(data_size);
129     i = 0;
130     while (i < data_size) {
131         x = read(f);
132         data[i] = x;
133         i = i + 1;
134     }
135     print('\n');
136 }
137 return 0;
138 }
139
140 func parse_dns (f, data_size) {
141     id = readintbytesb(f, 2);
142     print(id);
143     print('\n');
144     flags = readintbytesb(f, 2);
145     n_q = readintbytesb(f, 2);
146     n_ans = readintbytesb(f, 2);
147     n_auth = readintbytesb(f, 2);
148     n_add = readintbytesb(f, 2);
149
150     // questions
151     i = 0;
152     while (i < n_q) {
153         stop = 0;
154         first = 1;
155         while (!stop) {
156             len = read(f);
157             if (len == 0) {
158                 stop = 1;
159             } else {
160                 if (first) {
161                     first = 0;
162                 } else {
163                     print('.');
164                 }
165                 j = 0;
166                 while (j < len) {
167                     x = read(f);
168                     print(x);
169                     j = j + 1;
170                 }
171             }
172         }
173         print('\n');
174         type_q = readintbytesb(f, 2);
175         class_q = readintbytesb(f, 2);
176         i = i + 1;
177     }
178
179     // answers
180     i = 0;
181     while (i < n_ans) {
182         ptr = readintbytesb(f, 2);
183         assert(ptr == 49164);
184         type_ans = readintbytesb(f, 2);
185         class_ans = readintbytesb(f, 2);
186         ttl = readintbytesb(f, 4);
187         len = readintbytesb(f, 2);
188         if (type_ans == 1) {
189             assert(len == 4);
190             addr = readintbytesb(f, 4);
191             dummy = print_ip(addr);
192         } else {
193             assert(type_ans == 5);
194             j = 0;
195             while (j < len) {
196                 x = read(f);
197                 print(x);
198                 j = j + 1;
199             }
200         }
201         print('\n');
202         i = i + 1;

```



```

203     }
204
205     // authority
206     // additional
207     return 0;
208 }
209
210 main {
211     f = openb("../inputs/pcap/bad.pcap", 1);
212
213     x = read(f);
214     assert(x == 212);
215     x = read(f);
216     assert(x == 195);
217     x = read(f);
218     assert(x == 178);
219     x = read(f);
220     assert(x == 161);
221     maj_ver = readintbytesl(f, 2);
222     min_ver = readintbytesl(f, 2);
223     this_zone = readintbytesl(f, 4);
224     sigfigs = readintbytesl(f, 4);
225     snap_len = readintbytesl(f, 4); // number of packets
226     link_type = readintbytesl(f, 4);
227     assert(link_type == 1); // ethernet
228
229     inspectb (1, f, 12, 4, 0) {
230         ts_epoch = readintbytesl(f, 4);
231         ts_nanosec = readintbytesl(f, 4);
232         caplen = readintbytesl(f, 4);
233         length = readintbytesl(f, 4);
234         assert(caplen == length);
235
236         dummy = parse_ethernet(f, 0);
237     }
238
239     return 0;
240 }

```

A.7.2 Protective-check version

```

1  dst_mac = malloc_ec(6);
2  src_mac = malloc_ec(6);
3  data = malloc_ec(1);
4
5  func readintbytesl (f, n) { // little endian
6      x = read_ec(f);
7      assert(x >= 0);
8      if (n == 1) {
9          return x;
10     } else {
11         y = readintbytesl(f, n-1);
12         return (y << 8) + x;
13     }
14 }
15
16 func readintbytesb (f, n) { // big endian
17     x = 0;
18     i = 0;
19     while (i < n) {
20         byte = read_ec(f);
21         assert(byte >= 0);
22         x = x << 8;
23         x = x | byte;
24         i = i + 1;
25     }
26     return x;
27 }
28
29 func parse_ethernet (f, dummy) {
30     i = 0;
31     while (i < 6) {
32         x = read_ec(f);

```

```

33     assert(x >= 0);
34     dst_mac[i] = x;
35     i = i + 1;
36 }
37 i = 0;
38 while (i < 6) {
39     x = read_ec(f);
40     assert(x >= 0);
41     src_mac[i] = x;
42     i = i + 1;
43 }
44
45 ether_type = readintbytesl(f, 2);
46 assert(ether_type == 8); // IPv4
47
48 dummy = parse_ipv4(f, 0);
49 return 0;
50 }
51
52 func check_sum (f, len) {
53     start = pos_ec(f);
54     assert(len > 0 && len % 2 == 0 && start >= 0);
55     sum = 0;
56     i = 0;
57     while (i < len / 2) {
58         x = readintbytesb(f, 2);
59         sum = sum + x;
60         i = i + 1;
61     }
62     hi = (sum >> 16) & 65535;
63     lo = sum & 65535;
64     x = seek_ec(f, start);
65     assert(hi + lo == 65535 && x >= 0);
66     return 0;
67 }
68
69 func print_ip (addr) {
70     b = (addr >> 24) & 255;
71     print(b);
72     print('.');
73     b = (addr >> 16) & 255;
74     print(b);
75     print('.');
76     b = (addr >> 8) & 255;
77     print(b);
78     print('.');
79     b = addr & 255;
80     print(b);
81     return 0;
82 }
83
84 func parse_ipv4 (f, dummy) {
85     dummy = check_sum(f, 20); // no options
86
87     x = read_ec(f);
88     assert(x >= 0);
89     version = (x & 240) >> 4;
90     hdr_size = x & 15;
91     x = read_ec(f);
92     assert(x >= 0);
93     total = readintbytesb(f, 2);
94
95     id = readintbytesb(f, 2);
96     fragment = readintbytesb(f, 2);
97     dont_frag = (fragment & 16384) >> 14;
98     more_frag = (fragment & 8192) >> 13;
99     offs_frag = fragment & 8191;
100
101     ttl = read_ec(f);
102     ip_type = read_ec(f);
103     checksum = readintbytesb(f, 2);
104
105     src_ip = readintbytesb(f, 4);
106     dst_ip = readintbytesb(f, 4);
107

```

```

108     assert(version == 4 && hdr_size == 5 && total >= hdr_size * 4 && fragment &
        32768 == 0 && more_frag == 0 && offs_frag == 0 && ttl >= 0 && ip_type ==
        17); // IPv4, no options,
        UDP
109
110     print('\n');
111     dummy = print_ip(src_ip);
112     print(' ');
113     dummy = print_ip(dst_ip);
114     print(' ');
115     dummy = parse_udp(f, total - hdr_size * 4);
116
117     return 0;
118 }
119
120 func parse_udp (f, pkt_size) {
121     src_port = readintbytesb(f, 2);
122     dst_port = readintbytesb(f, 2);
123     udp_size = readintbytesb(f, 2);
124     checksum = readintbytesb(f, 2);
125
126     assert(pkt_size <= udp_size && pkt_size > 8);
127     if (src_port == 53 || dst_port == 53) { // DNS
128         dummy = parse_dns(f, pkt_size - 8);
129     } else { // others
130         if (valid(data)) {
131             dummy = free_ec(data);
132         }
133         data_size = pkt_size - 8;
134         data = malloc_ec(data_size);
135         assert(valid(data));
136         i = 0;
137         while (i < data_size) {
138             x = read_ec(f);
139             assert(x >= 0);
140             data[i] = x;
141             i = i + 1;
142         }
143         print('\n');
144     }
145     return 0;
146 }
147
148 func parse_dns (f, data_size) {
149     id = readintbytesb(f, 2);
150     print(id);
151     print('\n');
152     flags = readintbytesb(f, 2);
153     n_q = readintbytesb(f, 2);
154     n_ans = readintbytesb(f, 2);
155     n_auth = readintbytesb(f, 2);
156     n_add = readintbytesb(f, 2);
157
158     // questions
159     i = 0;
160     while (i < n_q) {
161         stop = 0;
162         first = 1;
163         while (!stop) {
164             len = read_ec(f);
165             assert(len >= 0);
166             if (len == 0) {
167                 stop = 1;
168             } else {
169                 if (first) {
170                     first = 0;
171                 } else {
172                     print(' ');
173                 }
174                 j = 0;
175                 while (j < len) {
176                     x = read_ec(f);
177                     assert(x >= 0);
178                     print(x);
179                     j = j + 1;
180                 }

```

```

181     }
182   }
183   print('\n');
184   type_q = readintbytesb(f, 2);
185   class_q = readintbytesb(f, 2);
186   i = i + 1;
187 }
188
189 // answers
190 i = 0;
191 while (i < n_ans) {
192   ptr = readintbytesb(f, 2);
193   assert(ptr == 49164);
194   type_ans = readintbytesb(f, 2);
195   class_ans = readintbytesb(f, 2);
196   ttl = readintbytesb(f, 4);
197   len = readintbytesb(f, 2);
198   if (type_ans == 1) {
199     assert(len == 4);
200     addr = readintbytesb(f, 4);
201     dummy = print_ip(addr);
202   } else {
203     assert(type_ans == 5 && len > 0);
204     j = 0;
205     while (j < len) {
206       x = read_ec(f);
207       assert(x >= 0);
208       print(x);
209       j = j + 1;
210     }
211   }
212   print('\n');
213   i = i + 1;
214 }
215
216 // authority
217 // additional
218 return 0;
219 }
220
221 main {
222   assert(valid(dst_mac) && valid(src_mac) && valid(data));
223   f = openb_ec("../inputs/pcap/bad.pcap", 1);
224   assert(valid(f));
225
226   x = read_ec(f);
227   assert(x == 212);
228   x = read_ec(f);
229   assert(x == 195);
230   x = read_ec(f);
231   assert(x == 178);
232   x = read_ec(f);
233   assert(x == 161);
234   maj_ver = readintbytesl(f, 2);
235   min_ver = readintbytesl(f, 2);
236   this_zone = readintbytesl(f, 4);
237   sigfigs = readintbytesl(f, 4);
238   snap_len = readintbytesl(f, 4); // number of packets
239   link_type = readintbytesl(f, 4);
240   assert(link_type == 1); // ethernet
241
242   inspectb (1, f, 12, 4, 0) {
243     ts_epoch = readintbytesl(f, 4);
244     ts_nanosec = readintbytesl(f, 4);
245     caplen = readintbytesl(f, 4);
246     length = readintbytesl(f, 4);
247     assert(caplen == length);
248
249     dummy = parse_ethernet(f, 0);
250   }
251
252   return 0;
253 }

```

A.7.3 Explicit-recovery version

```
1  dst_mac = malloc_ec(6);
2  src_mac = malloc_ec(6);
3  dst_ip = 0;
4  src_ip = 0;
5  data = malloc_ec(1);
6
7  bad = 0;
8  idx = 0;
9  out = malloc_ec(1000);
10
11 func readintbytesl (f, n) { // little endian
12     x = read_ec(f);
13     if (x < 0) {
14         bad = 1;
15         return -1;
16     }
17     if (n == 1) {
18         return x;
19     } else {
20         y = readintbytesl(f, n-1);
21         return (y << 8) + x;
22     }
23 }
24
25 func readintbytesb (f, n) { // big endian
26     x = 0;
27     i = 0;
28     while (i < n) {
29         byte = read_ec(f);
30         if (byte < 0) {
31             bad = 1;
32             return -1;
33         }
34         x = x << 8;
35         x = x | byte;
36         i = i + 1;
37     }
38     return x;
39 }
40
41 func parse_ethernet (f, dummy) {
42     i = 0;
43     while (i < 6) {
44         x = read_ec(f);
45         if (x < 0) {
46             bad = 1;
47             return -1;
48         }
49         dst_mac[i] = x;
50         i = i + 1;
51     }
52     i = 0;
53     while (i < 6) {
54         x = read_ec(f);
55         if (x < 0) {
56             bad = 1;
57             return -1;
58         }
59         src_mac[i] = x;
60         i = i + 1;
61     }
62
63     ether_type = readintbytesl(f, 2);
64     if (bad || ether_type != 8) {
65         bad = 1;
66         return -1;
67     } // IPv4
68     dummy = parse_ipv4(f, 0);
69     return 0;
70 }
71
72 func check_sum (f, len) {
```

```

73 start = pos_ec(f);
74 if (len <= 0 || len % 2 != 0 || start < 0) {
75     bad = 1;
76     return -1;
77 }
78 sum = 0;
79 i = 0;
80 while (i < len / 2) {
81     x = readintbytesb(f, 2);
82     sum = sum + x;
83     i = i + 1;
84 }
85 hi = (sum >> 16) & 65535;
86 lo = sum & 65535;
87 x = seek_ec(f, start);
88 if (hi + lo != 65535 || x < 0) {
89     bad = 1;
90     return -1;
91 }
92 return 0;
93 }
94
95 func prtbuf (x) {
96     if (idx >= 1000) {
97         bad = 1;
98         return -1;
99     }
100    out[idx] = x;
101    idx = idx + 1;
102    return 0;
103 }
104
105 func prtbuf_ip (addr) {
106     dummy = prtbuf((addr >> 24) & 255);
107     dummy = prtbuf('.')';
108     dummy = prtbuf((addr >> 16) & 255);
109     dummy = prtbuf('.')';
110     dummy = prtbuf((addr >> 8) & 255);
111     dummy = prtbuf('.')';
112     dummy = prtbuf(addr & 255);
113     return 0;
114 }
115
116 func parse_ipv4 (f, dummy) {
117     dummy = check_sum(f, 20); // no options
118     if (bad) {
119         return -1;
120     }
121
122     x = read_ec(f);
123     if (x < 0) {
124         bad = 1;
125         return -1;
126     }
127     version = (x & 240) >> 4;
128     hdr_size = x & 15;
129     x = read_ec(f);
130     if (x < 0) {
131         bad = 1;
132         return -1;
133     }
134     total = readintbytesb(f, 2);
135
136     id = readintbytesb(f, 2);
137     fragment = readintbytesb(f, 2);
138     dont_frag = (fragment & 16384) >> 14;
139     more_frag = (fragment & 8192) >> 13;
140     offs_frag = fragment & 8191;
141
142     ttl = read_ec(f);
143     ip_type = read_ec(f);
144     checksum = readintbytesb(f, 2);
145
146     src_ip = readintbytesb(f, 4);
147     dst_ip = readintbytesb(f, 4);
148

```

```

149     if (bad || version != 4 || hdr_size != 5 || total < hdr_size * 4 || fragment
        & 32768 != 0 || more_frag != 0 || offs_frag != 0 || ttl < 0 || ip_type
        != 17) { // IPv4, no options,
150         UDP
        bad = 1;
151         return -1;
152     }
153     dummy = prtbuf('\n');
154     dummy = prtbuf_ip(src_ip);
155     dummy = prtbuf(' ');
156     dummy = prtbuf_ip(dst_ip);
157     dummy = prtbuf(' ');
158     dummy = parse_udp(f, total - hdr_size * 4);
159     return 0;
160 }
161
162 func parse_udp (f, pkt_size) {
163     src_port = readintbytesb(f, 2);
164     dst_port = readintbytesb(f, 2);
165     udp_size = readintbytesb(f, 2);
166     checksum = readintbytesb(f, 2);
167
168     if (bad || pkt_size > udp_size || pkt_size <= 8) {
169         bad = 1;
170         return -1;
171     }
172
173     if (src_port == 53 || dst_port == 53) { // DNS
174         dummy = parse_dns(f, pkt_size - 8);
175     } else { // others
176         if (valid(data)) {
177             dummy = free_ec(data);
178         }
179         data_size = pkt_size - 8;
180         data = malloc_ec(data_size);
181         if (!valid(data)) {
182             bad = 1;
183             return -1;
184         }
185         i = 0;
186         while (i < data_size) {
187             x = read_ec(f);
188             if (x < 0) {
189                 bad = 1;
190                 return -1;
191             }
192             data[i] = x;
193             i = i + 1;
194         }
195         dummy = prtbuf('\n');
196     }
197     return 0;
198 }
199
200 func parse_dns (f, data_size) {
201     id = readintbytesb(f, 2);
202     dummy = prtbuf(id);
203     dummy = prtbuf('\n');
204     flags = readintbytesb(f, 2);
205     n_q = readintbytesb(f, 2);
206     n_ans = readintbytesb(f, 2);
207     n_auth = readintbytesb(f, 2);
208     n_add = readintbytesb(f, 2);
209
210     // questions
211     i = 0;
212     while (i < n_q) {
213         stop = 0;
214         first = 1;
215         while (!stop) {
216             len = read_ec(f);
217             if (len < 0) {
218                 bad = 1;
219                 return -1;
220             }
221             if (len == 0) {

```

```

222     stop = 1;
223 } else {
224     if (first) {
225         first = 0;
226     } else {
227         dummy = prtbuf(' ');
228     }
229     j = 0;
230     while (j < len) {
231         x = read_ec(f);
232         if (x < 0 || idx >= 1000) {
233             bad = 1;
234             return -1;
235         }
236         out[idx] = x;
237         j = j + 1;
238         idx = idx + 1;
239     }
240 }
241 }
242 dummy = prtbuf('\n');
243 type_q = readintbytesb(f, 2);
244 class_q = readintbytesb(f, 2);
245 i = i + 1;
246 }
247
248 // answers
249 i = 0;
250 while (i < n_ans) {
251     ptr = readintbytesb(f, 2);
252     if (ptr != 49164) {
253         bad = 1;
254         return -1;
255     }
256     type_ans = readintbytesb(f, 2);
257     class_ans = readintbytesb(f, 2);
258     ttl = readintbytesb(f, 4);
259     len = readintbytesb(f, 2);
260     if (type_ans == 1) {
261         if (len != 4) {
262             bad = 1;
263             return -1;
264         }
265         addr = readintbytesb(f, 4);
266         dummy = prtbuf_ip(addr);
267     } else {
268         if (type_ans != 5 || len <= 0) {
269             bad = 1;
270             return -1;
271         }
272         j = 0;
273         while (j < len) {
274             x = read_ec(f);
275             if (x < 0 || idx >= 1000) {
276                 bad = 1;
277                 return -1;
278             }
279             out[idx] = x;
280             j = j + 1;
281             idx = idx + 1;
282         }
283     }
284     dummy = prtbuf('\n');
285     i = i + 1;
286 }
287
288 // authority
289 // additional
290 return 0;
291 }
292
293 main {
294     if (!valid(dst_mac) || !valid(src_mac) || !valid(data) || !valid(out)) {
295         exit(1);
296     }
297     f = openb_ec("../inputs/pcap/bad.pcap", 1);

```



```

298     if (!valid(f)) {
299         exit(1);
300     }
301
302     x = read_ec(f);
303     if (x != 212) {
304         exit(1);
305     }
306     x = read_ec(f);
307     if (x != 195) {
308         exit(1);
309     }
310     x = read_ec(f);
311     if (x != 178) {
312         exit(1);
313     }
314     x = read_ec(f);
315     if (x != 161) {
316         exit(1);
317     }
318     maj_ver = readintbytes1(f, 2);
319     min_ver = readintbytes1(f, 2);
320     this_zone = readintbytes1(f, 4);
321     sigfigs = readintbytes1(f, 4);
322     snap_len = readintbytes1(f, 4); // number of packets
323     link_type = readintbytes1(f, 4);
324     if (bad || link_type != 1) {
325         exit(1);
326     } // ethernet
327
328     lookatb (1, f, 12, 4, 0) {
329         bad = 0;
330         idx = 0;
331         ts_epoch = readintbytes1(f, 4);
332         ts_nanosec = readintbytes1(f, 4);
333         caplen = readintbytes1(f, 4);
334         length = readintbytes1(f, 4);
335         if (!bad && caplen == length) {
336             dummy = parse_ethernet(f, 0);
337             if (!bad) {
338                 i = 0;
339                 while (i < idx) {
340                     x = out[i];
341                     print(x);
342                     i = i + 1;
343                 }
344             }
345         } // skip unit
346     }
347
348     return 0;
349 }

```

A.7.4 Plain-loop version

```

1  dst_mac = malloc_ec(6);
2  src_mac = malloc_ec(6);
3  dst_ip = 0;
4  src_ip = 0;
5  data = malloc_ec(1);
6
7  bad = 0;
8  idx = 0;
9  out = malloc_ec(1000);
10
11 func readintbytes1 (f, n) { // little endian
12     x = read_ec(f);
13     if (x < 0) {
14         bad = 1;
15         return -1;
16     }
17     if (n == 1) {
18         return x;

```

```

19 } else {
20     y = readintbytesl(f, n-1);
21     return (y << 8) + x;
22 }
23 }
24
25 func readintbytesb (f, n) { // big endian
26     x = 0;
27     i = 0;
28     while (i < n) {
29         byte = read_ec(f);
30         if (byte < 0) {
31             bad = 1;
32             return -1;
33         }
34         x = x << 8;
35         x = x | byte;
36         i = i + 1;
37     }
38     return x;
39 }
40
41 func parse_ethernet (f, dummy) {
42     i = 0;
43     while (i < 6) {
44         x = read_ec(f);
45         if (x < 0) {
46             bad = 1;
47             return -1;
48         }
49         dst_mac[i] = x;
50         i = i + 1;
51     }
52     i = 0;
53     while (i < 6) {
54         x = read_ec(f);
55         if (x < 0) {
56             bad = 1;
57             return -1;
58         }
59         src_mac[i] = x;
60         i = i + 1;
61     }
62
63     ether_type = readintbytesl(f, 2);
64     if (bad || ether_type != 8) {
65         bad = 1;
66         return -1;
67     } // IPv4
68     dummy = parse_ipv4(f, 0);
69     return 0;
70 }
71
72 func check_sum (f, len) {
73     start = pos_ec(f);
74     if (len <= 0 || len % 2 != 0 || start < 0) {
75         bad = 1;
76         return -1;
77     }
78     sum = 0;
79     i = 0;
80     while (i < len / 2) {
81         x = readintbytesb(f, 2);
82         sum = sum + x;
83         i = i + 1;
84     }
85     hi = (sum >> 16) & 65535;
86     lo = sum & 65535;
87     x = seek_ec(f, start);
88     if (hi + lo != 65535 || x < 0) {
89         bad = 1;
90         return -1;
91     }
92     return 0;
93 }
94

```

```

95 func prtbuf (x) {
96     if (idx >= 1000) {
97         bad = 1;
98         return -1;
99     }
100     out[idx] = x;
101     idx = idx + 1;
102     return 0;
103 }
104
105 func prtbuf_ip (addr) {
106     dummy = prtbuf((addr >> 24) & 255);
107     dummy = prtbuf('.')';
108     dummy = prtbuf((addr >> 16) & 255);
109     dummy = prtbuf('.')';
110     dummy = prtbuf((addr >> 8) & 255);
111     dummy = prtbuf('.')';
112     dummy = prtbuf(addr & 255);
113     return 0;
114 }
115
116 func parse_ipv4 (f, dummy) {
117     dummy = check_sum(f, 20); // no options
118     if (bad) {
119         return -1;
120     }
121
122     x = read_ec(f);
123     if (x < 0) {
124         bad = 1;
125         return -1;
126     }
127     version = (x & 240) >> 4;
128     hdr_size = x & 15;
129     x = read_ec(f);
130     if (x < 0) {
131         bad = 1;
132         return -1;
133     }
134     total = readintbytesb(f, 2);
135
136     id = readintbytesb(f, 2);
137     fragment = readintbytesb(f, 2);
138     dont_frag = (fragment & 16384) >> 14;
139     more_frag = (fragment & 8192) >> 13;
140     offs_frag = fragment & 8191;
141
142     ttl = read_ec(f);
143     ip_type = read_ec(f);
144     checksum = readintbytesb(f, 2);
145
146     src_ip = readintbytesb(f, 4);
147     dst_ip = readintbytesb(f, 4);
148
149     if (bad || version != 4 || hdr_size != 5 || total < hdr_size * 4 || fragment
        & 32768 != 0 || more_frag != 0 || offs_frag != 0 || ttl < 0 || ip_type
        != 17) { // IPv4, no options,
        UDP
150         bad = 1;
151         return -1;
152     }
153     dummy = prtbuf('\n');
154     dummy = prtbuf_ip(src_ip);
155     dummy = prtbuf(' ');
156     dummy = prtbuf_ip(dst_ip);
157     dummy = prtbuf(' ');
158     dummy = parse_udp(f, total - hdr_size * 4);
159     return 0;
160 }
161
162 func parse_udp (f, pkt_size) {
163     src_port = readintbytesb(f, 2);
164     dst_port = readintbytesb(f, 2);
165     udp_size = readintbytesb(f, 2);
166     checksum = readintbytesb(f, 2);
167

```

```

168 if (bad || pkt_size > udp_size || pkt_size <= 8) {
169     bad = 1;
170     return -1;
171 }
172
173 if (src_port == 53 || dst_port == 53) { // DNS
174     dummy = parse_dns(f, pkt_size - 8);
175 } else { // others
176     if (valid(data)) {
177         dummy = free_ec(data);
178     }
179     data_size = pkt_size - 8;
180     data = malloc_ec(data_size);
181     if (!valid(data)) {
182         bad = 1;
183         return -1;
184     }
185     i = 0;
186     while (i < data_size) {
187         x = read_ec(f);
188         if (x < 0) {
189             bad = 1;
190             return -1;
191         }
192         data[i] = x;
193         i = i + 1;
194     }
195     dummy = prtbuf('\n');
196 }
197 return 0;
198 }
199
200 func parse_dns (f, data_size) {
201     id = readintbytesb(f, 2);
202     dummy = prtbuf(id);
203     dummy = prtbuf('\n');
204     flags = readintbytesb(f, 2);
205     n_q = readintbytesb(f, 2);
206     n_ans = readintbytesb(f, 2);
207     n_auth = readintbytesb(f, 2);
208     n_add = readintbytesb(f, 2);
209
210     // questions
211     i = 0;
212     while (i < n_q) {
213         stop = 0;
214         first = 1;
215         while (!stop) {
216             len = read_ec(f);
217             if (len < 0) {
218                 bad = 1;
219                 return -1;
220             }
221             if (len == 0) {
222                 stop = 1;
223             } else {
224                 if (first) {
225                     first = 0;
226                 } else {
227                     dummy = prtbuf('.')';
228                 }
229                 j = 0;
230                 while (j < len) {
231                     x = read_ec(f);
232                     if (x < 0 || idx >= 1000) {
233                         bad = 1;
234                         return -1;
235                     }
236                     out[idx] = x;
237                     j = j + 1;
238                     idx = idx + 1;
239                 }
240             }
241         }
242         dummy = prtbuf('\n');
243         type_q = readintbytesb(f, 2);

```

```

244     class_q = readintbytesb(f, 2);
245     i = i + 1;
246 }
247
248 // answers
249 i = 0;
250 while (i < n_ans) {
251     ptr = readintbytesb(f, 2);
252     if (ptr != 49164) {
253         bad = 1;
254         return -1;
255     }
256     type_ans = readintbytesb(f, 2);
257     class_ans = readintbytesb(f, 2);
258     ttl = readintbytesb(f, 4);
259     len = readintbytesb(f, 2);
260     if (type_ans == 1) {
261         if (len != 4) {
262             bad = 1;
263             return -1;
264         }
265         addr = readintbytesb(f, 4);
266         dummy = prtbuf_ip(addr);
267     } else {
268         if (type_ans != 5 || len <= 0) {
269             bad = 1;
270             return -1;
271         }
272         j = 0;
273         while (j < len) {
274             x = read_ec(f);
275             if (x < 0 || idx >= 1000) {
276                 bad = 1;
277                 return -1;
278             }
279             out[idx] = x;
280             j = j + 1;
281             idx = idx + 1;
282         }
283     }
284     dummy = prtbuf('\n');
285     i = i + 1;
286 }
287
288 // authority
289 // additional
290 return 0;
291 }
292
293 main {
294     if (!valid(dst_mac) || !valid(src_mac) || !valid(data) || !valid(out)) {
295         exit(1);
296     }
297     f = openb_ec("../inputs/pcap/bad.pcap", 1);
298     if (!valid(f)) {
299         exit(1);
300     }
301
302     x = read_ec(f);
303     if (x != 212) {
304         exit(1);
305     }
306     x = read_ec(f);
307     if (x != 195) {
308         exit(1);
309     }
310     x = read_ec(f);
311     if (x != 178) {
312         exit(1);
313     }
314     x = read_ec(f);
315     if (x != 161) {
316         exit(1);
317     }
318     maj_ver = readintbytesl(f, 2);
319     min_ver = readintbytesl(f, 2);

```

```

320 this_zone = readintbytesl(f, 4);
321 sigfigs = readintbytesl(f, 4);
322 snap_len = readintbytesl(f, 4); // number of packets
323 link_type = readintbytesl(f, 4);
324 if (bad || link_type != 1) {
325     exit(1);
326 } // ethernet
327
328 finish = 0;
329 while (!finish) {
330     bad = 0;
331     idx = 0;
332     ts_epoch = readintbytesl(f, 4);
333     ts_nanosec = readintbytesl(f, 4);
334     caplen = readintbytesl(f, 4);
335     length = readintbytesl(f, 4);
336     if (!bad && length >= 0) {
337         eou = pos_ec(f) + length;
338         if (!bad && caplen == length) {
339             dummy = parse_ethernet(f, 0);
340             if (pos_ec(f) < 0 || pos_ec(f) > eou) {
341                 bad = 1;
342             }
343             if (!bad) {
344                 i = 0;
345                 while (i < idx) {
346                     x = out[i];
347                     print(x);
348                     i = i + 1;
349                 }
350             }
351             // skip unit
352             x = seek_ec(f, eou);
353             if (x < 0) {
354                 finish = 1;
355             }
356         } else { // give up
357             finish = 1;
358         }
359     }
360 }
361 return 0;
362 }

```

Bibliography

- [1] T. Anderson and R. Kerr. Recovery blocks in action: A system supporting high reliability. In *Proceedings of the 2Nd International Conference on Software Engineering, ICSE '76*, pages 447–457, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
- [2] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11(12):1491–1501, December 1985.
- [3] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF '07*, pages 311–325, Washington, DC, USA, 2007. IEEE Computer Society.
- [4] K.M. Chandy and C.V. Ramamoorthy. Rollback and recovery strategies for computer programs. *Computers, IEEE Transactions on*, C-21(6):546–556, June 1972.
- [5] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07*, pages 117–130, New York, NY, USA, 2007. ACM.
- [6] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP '05*, pages 133–147, New York, NY, USA, 2005. ACM.
- [7] Flaviu Cristian. Exception handling and software fault tolerance. *IEEE Trans. Comput.*, 31(6):531–540, June 1982.
- [8] Brian Demsky and Alokika Dash. Bristlecone: A language for robust software systems. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming, ECOOP '08*, pages 490–515, Berlin, Heidelberg, 2008. Springer-Verlag.
- [9] Brian Demsky and Sivaji Sundaramurthy. Bristlecone: Language support for robust software applications. *IEEE Trans. Softw. Eng.*, 37(1):4–23, January 2011.

- [10] Brian Demsky, Jin Zhou, and William Montaz. Recovery tasks: An automated approach to failure recovery. In *Proceedings of the First International Conference on Runtime Verification, RV'10*, pages 229–244, Berlin, Heidelberg, 2010. Springer-Verlag.
- [11] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, November 1976.
- [12] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
- [13] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.
- [14] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the system r database manager. *ACM Computing Surveys (CSUR)*, 13(2):223–242, 1981.
- [15] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.
- [16] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.
- [17] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA '93*, pages 289–300, New York, NY, USA, 1993. ACM.
- [18] Ke Huang, Jie Wu, and Eduardo B. Fernández. A generalized forward recovery checkpointing scheme. In *IPPS/SPDP Workshops*, pages 623–643, 1998.
- [19] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, HOPL III*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.
- [20] Barbara Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, March 1988.
- [21] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in clu. *Commun. ACM*, 20(8):564–576, August 1977.

- [22] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 128–137, New York, NY, USA, 1977. ACM.
- [23] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, pages 80–90, Piscataway, NJ, USA, 2012. IEEE Press.
- [24] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, pages 439–452, New York, NY, USA, 2014. ACM.
- [25] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 227–238, New York, NY, USA, 2014. ACM.
- [26] E. B. Moss. *Nested transactions: An approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.
- [27] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. *SIGOPS Oper. Syst. Rev.*, 39(5):235–248, October 2005.
- [28] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebe, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI'04*, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.
- [29] Martin C. Rinard. Living in the comfort zone. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA '07*, pages 611–622, New York, NY, USA, 2007. ACM.
- [30] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, PODC '95*, pages 204–213, New York, NY, USA, 1995. ACM.
- [31] Stelios Sidiroglou and Angelos D. Keromytis. Using Execution Transactions To Recover From Buffer Overflow Attacks. Technical report, Columbia University Computer Science Department, 2004. CUCS-031-04.
- [32] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: Automatic software self-healing using rescue

- points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV*, pages 37–48, New York, NY, USA, 2009. ACM.
- [33] Guy L. Steele, Jr. *Common LISP: The Language (2Nd Ed.)*. Digital Press, Newton, MA, USA, 1990.
- [34] Janice M. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the oklahoma update. *IEEE Parallel Distrib. Technol.*, 1(4):58–71, November 1993.
- [35] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 115–128, New York, NY, USA, 2007. ACM.
- [36] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011.
- [37] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '04*, pages 193–204, New York, NY, USA, 2004. ACM.