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

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 ....

Leslie A. Kolodziejski

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

List of Figures

List of Tables

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 an 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 stuites 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 outide 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 interators, 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.

```
while (!end_ec(f)) {
  idx = 0;
  while (idx < 11 * columns) {
    buffer[idx] = 0;
    idx = idx + 1;
  }
  idx = 0;
  j = 0;
  x = 0;
  while (j < columns && x >= 0 && x != '\n') {
    start = idx;
    if (j > 0) {
      buffer[idx] = ',';
      idx = idx + 1;
    }
    i = 0;
    x = read_ec(f);
    while (x >= 0 && x != '\n' && x != ',' && i < 10) {
      buffer[idx] = x;
      idx = idx + 1;
      i = i + 1;
      x = read_ec(f);
    }
    if (x == '\n' || x == ',') {
      j = j + 1;
    } else { // skip unit
      while (idx > start) {
        buffer[idx] = 0;
        idx = idx - 1;
      }
    }
    while (x >= 0 && x != '\n' && x != ',') {
      x = read_ec(f);
    }
  }
  if (j == columns) {
    print(buffer);
    print('\n');
  } // skip unit
  while (x >= 0 && x != '\n') {
    x = read_ec(f);
  }
}
```

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

```
inspectt (!end(f), f, '\n') {
  j = 0;
  inspectt (j < columns, f, ',') {
    field = malloc(10);
    i = 0;
    while (!end(f)) {
      x = read(f);
      field[i] = x;
      i = i + 1;
    }
    if (j > 0) {
      print(',');
    }
    print(field);
    free(field);
    j = j + 1;
  }
  print('\n');
  assert(j == columns);
}
```

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

Chapter 3

Design

In this chapter, we first introdoce 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 collectioins 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 inputs 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.

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

$$
\begin{array}{ll}
x, y \in \mathit{IVar} & q \in \text{function names} \\
a, d_u, d_s \in \mathit{AVar} & n \in \mathit{Int} \\
f \in \mathit{FVar} & str \in \mathit{String}
\end{array}
$$

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:

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

A state $\sigma \in State$ 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 $fd \in FDesc$ 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 $ud \in UDesc$ describes the delimiters in use for text files and the cutoff positions for binary files.

For text files, a delimiter definition $dlm \in Delim$ describes three sets of delimiters that identify the broundaries between input units: $dlm = \langle eou, eos, osd \rangle$ where $eou \in EOU$ is the set of end-of-unit delimiters, $eos \in EOS$ is the set of end-of-sequence delimiters, and $osd \in osd$ 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, $osd = \varnothing$. 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 $cut \in \mathit{Cutoff}$ describes where the current input unit ends. This value also updates at runtime according to nesting `inspectb` layers.

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

The relation $\langle s, \sigma \rangle \Downarrow_{\mathrm{s}} \xi$ denotes that executing the statement s in the state σ yields the output configuration $\xi \in \mathit{State} \times \{\mathrm{ok}, \mathrm{bad}\}$. An output configuration $\xi = \langle \sigma', \mathrm{ok} \rangle$ indicates that the program execution is successful and that the resulting state is σ'. An output configuration $\xi = \langle \sigma', \mathrm{bad} \rangle$ 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 (avalid-t, avalid-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

$$\frac{}{\langle n, \sigma \rangle \Downarrow_{\mathrm{e}} n} \quad \text{(int)}$$

$$\frac{}{\langle x, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} \sigma_S(x)} \quad \text{(ivar)}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow_{\mathrm{e}} u_1 \quad \langle e_2, \sigma \rangle \Downarrow_{\mathrm{e}} u_2 \quad u_1 \text{ op } u_2 = v}{\langle e_1 \text{ op } e_2, \sigma \rangle \Downarrow_{\mathrm{e}} v} \quad \text{(iop-ok)}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow_{\mathrm{e}} u_1 \quad \langle e_2, \sigma \rangle \Downarrow_{\mathrm{e}} u_2 \quad u_1 \text{ op } u_2 = \bot}{\langle e_1 \text{ op } e_2, \sigma \rangle \Downarrow_{\mathrm{e}} \text{err}} \quad \text{(iop-bad)}$$

$$\frac{\sigma_S(a) = \text{null}}{\langle a[e], \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} \text{err}} \quad \text{(ard-null)}$$

$$\frac{\sigma_S(a) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} u \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n \rangle \quad u < 0 \vee u \geq n}{\langle a[e], \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} \text{err}} \quad \text{(ard-out)}$$

$$\frac{\sigma_S(a) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} u \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n \rangle \quad 0 \leq u < n}{\langle a[e], \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} \gamma(u)} \quad \text{(ard-ok)}$$

$$\frac{\sigma_S(a) \neq \text{null}}{\langle \texttt{valid}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} \text{true}} \quad \text{(avalid-t)}$$

$$\frac{\sigma_S(a) = \text{null}}{\langle \texttt{valid}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} \text{false}} \quad \text{(avalid-f)}$$

Figure 3-2: Semantics for simple expressions

$$\frac{\langle e, \sigma \rangle \Downarrow_{\mathrm{e}} \text{err}}{\langle x = e, \sigma \rangle \Downarrow_{\mathrm{s}} \langle \sigma, \text{bad} \rangle} \quad \text{(vwr-bad)}$$

$$\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} u}{\langle x = e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{s}} \langle \langle \sigma_S[x \mapsto u], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad \text{(vwr-ok)}$$

Figure 3-3: Semantics for simple assignments

$$\frac{\langle e, \sigma \rangle \Downarrow_{\text{e}} \text{err}}{\langle a = \texttt{malloc}(e), \sigma \rangle \Downarrow_{\text{s}} \langle \sigma, \text{bad} \rangle} \quad \text{(malloc-bad)}$$

$$\frac{\langle e, \sigma \rangle \Downarrow_{\text{e}} u \quad u \leq 0}{\langle a = \texttt{malloc}(e), \sigma \rangle \Downarrow_{\text{s}} \langle \sigma, \text{bad} \rangle} \quad \text{(malloc-neg)}$$

$$\frac{\langle e, \sigma \rangle \Downarrow_{\text{e}} u \quad u > 0 \quad \text{heap allocate}(u) = \bot}{\langle a = \texttt{malloc}(e), \sigma \rangle \Downarrow_{\text{s}} \langle \sigma, \text{bad} \rangle} \quad \text{(malloc-ovf)}$$

$$\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} u \quad u > 0 \quad \text{heap allocate}(u) = \text{addr}}{\begin{array}{c}\langle a = \texttt{malloc}(e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \\ \langle \langle \sigma_S[a \mapsto \text{addr}], \sigma_H[\text{addr} \mapsto \langle [0 \mapsto 0, 1 \mapsto 0, \ldots, u-1 \mapsto 0], u \rangle], \sigma_F, \sigma_D \rangle, \text{ok} \rangle\end{array}} \quad \text{(malloc-ok)}$$

$$\frac{\sigma_S(a) = \text{null}}{\langle \texttt{free}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad \text{(free-null)}$$

$$\frac{\sigma_S(a) \neq \text{null}}{\langle \texttt{free}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma_S[a \mapsto \text{null}], \sigma_H[\sigma_S(a) \mapsto \bot], \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad \text{(free-ok)}$$

$$\frac{\sigma_S(a) = \text{null}}{\langle a[e_1] = e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad \text{(awr-null)}$$

$$\frac{\sigma_S(a) \neq \text{null} \quad \langle e_1, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} \text{err} \vee \langle e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} \text{err}}{\langle a[e_1] = e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \sigma, \text{bad} \rangle} \quad \text{(awr-bad)}$$

$$\frac{\sigma_S(a) \neq \text{null} \quad \langle e_1, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} u \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n \rangle \quad u < 0 \vee u \geq n}{\langle a[e_1] = e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad \text{(awr-out)}$$

$$\frac{\begin{array}{c}\sigma_S(a) \neq \text{null} \quad \langle e_1, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} u_1 \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n \rangle \\ 0 \leq u_1 < n \quad \langle e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} u_2\end{array}}{\begin{array}{c}\langle a[e_1] = e_2, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \\ \langle \langle \sigma_S, \sigma_H[\sigma_S(a) \mapsto \langle \gamma[u_1 \mapsto u_2], n \rangle], \sigma_F, \sigma_D \rangle, \text{ok} \rangle\end{array}} \quad \text{(awr-ok)}$$

Figure 3-4: Semantics for arrays

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err}}{\langle x = q(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma,\mathrm{bad}\rangle} \quad \text{(fn-arg)}$$

$$\frac{\text{stack allocate}(\mathit{fr}(q)) = \bot}{\langle x = q(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma,\mathrm{bad}\rangle} \quad \text{(fn-ovf)}$$

$$\frac{\begin{array}{c}\text{stack allocate}(\mathit{fr}(q)) \neq \bot \quad \langle e,\sigma\rangle \Downarrow_{\mathrm{e}} u\\ \langle \mathit{body}(q), \langle[\mathit{arg}(q)\mapsto u],\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle\sigma'_S,\sigma'_H,\sigma'_F,\sigma'_D\rangle,\mathrm{bad}\rangle\end{array}}{\langle x = q(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma_S,\sigma_H,\sigma'_F,\sigma_D\rangle,\mathrm{bad}\rangle} \quad \text{(fn-body)}$$

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

Figure 3-5: Semantics for function calls

$$\frac{\langle s_1,\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma',\mathrm{bad}\rangle}{\langle s_1;s_2,\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma',\mathrm{bad}\rangle} \quad \text{(seq-bad)}$$

$$\frac{\langle s_1,\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma',\mathrm{ok}\rangle \quad \langle s_2,\sigma'\rangle \Downarrow_{\mathrm{s}} \xi}{\langle s_1;s_2,\sigma\rangle \Downarrow_{\mathrm{s}} \xi} \quad \text{(seq-prgr)}$$

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err}}{\langle \mathtt{if}(e)\{s_1\}\mathtt{else}\{s_2\},\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma,\mathrm{bad}\rangle} \quad \text{(if-bad)}$$

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{true} \quad \langle s_1,\sigma\rangle \Downarrow_{\mathrm{s}} \xi}{\langle \mathtt{if}(e)\{s_1\}\mathtt{else}\{s_2\},\sigma\rangle \Downarrow_{\mathrm{s}} \xi} \quad \text{(if-t)}$$

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{false} \quad \langle s_2,\sigma\rangle \Downarrow_{\mathrm{s}} \xi}{\langle \mathtt{if}(e)\{s_1\}\mathtt{else}\{s_2\},\sigma\rangle \Downarrow_{\mathrm{s}} \xi} \quad \text{(if-f)}$$

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err}}{\langle \mathtt{while}(e)\{s\},\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma,\mathrm{bad}\rangle} \quad \text{(while-bad)}$$

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{false}}{\langle \mathtt{while}(e)\{s\},\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma,\mathrm{ok}\rangle} \quad \text{(while-end)}$$

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{true} \quad \langle s,\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma',\mathrm{bad}\rangle}{\langle \mathtt{while}(e)\{s\},\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma',\mathrm{bad}\rangle} \quad \text{(while-body)}$$

$$\frac{\begin{array}{c}\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{true} \quad \langle s,\sigma\rangle \Downarrow_{\mathrm{s}} \langle\sigma',\mathrm{ok}\rangle\\ \langle \mathtt{while}(e)\{s\},\sigma'\rangle \Downarrow_{\mathrm{s}} \xi\end{array}}{\langle \mathtt{while}(e)\{s\},\sigma\rangle \Downarrow_{\mathrm{s}} \xi} \quad \text{(while-prgr)}$$

Figure 3-6: Semantics for basic control structures

$$\frac{\langle e, \sigma \rangle \Downarrow_{\mathrm{e}} \mathrm{err}}{\langle \mathtt{assert}(e), \sigma \rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad} \rangle} \quad \text{(assert-bad)}$$

$$\frac{\langle e, \sigma \rangle \Downarrow_{\mathrm{e}} \mathrm{true}}{\langle \mathtt{assert}(e), \sigma \rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{ok} \rangle} \quad \text{(assert-t)}$$

$$\frac{\langle e, \sigma \rangle \Downarrow_{\mathrm{e}} \mathrm{false}}{\langle \mathtt{assert}(e), \sigma \rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad} \rangle} \quad \text{(assert-f)}$$

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 $\mathtt{func}\ q(x)\{s;\mathtt{return}\ y\}$,

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:

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

returns the set of elements in array a.

$$\Lambda'(l') \triangleq \operatorname*{argmin}_{k \geq l'} \{k = n' \vee \gamma'(k) \in \mathit{eou}' \cup \mathit{eos}' \cup \mathit{osd}'\} \quad (l' = 0, 1, \ldots, n')$$

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:

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

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

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

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

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

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

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

$$\frac{\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) \neq \varnothing}{\langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad \text{(inspt-dupl)}$$

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

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

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

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

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

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

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

(inspb-prgr)

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

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

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

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).

$$\frac{\begin{array}{c}\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} \mathrm{err} \vee \langle o, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} \mathrm{err} \\ \vee \langle w, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} \mathrm{err} \vee \langle c, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\mathrm{e}} \mathrm{err}\end{array}}{\langle \mathtt{inspectb}(e, f, o, w, c)\{s\}, \sigma \rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad} \rangle} \quad (\text{inspb-bad})$$

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

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

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

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

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

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

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

Figure 3-12: Semantics for input file expressions

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

$$\frac{\text{file } str \text{ exists}}{\begin{array}{c}\langle f = \texttt{opent}(str), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_{\text{s}} \\ \langle\langle \sigma_S[f \mapsto \text{hndl}], \sigma_H, \sigma_F[\text{hndl} \mapsto \langle str, 0, 0, \langle \varnothing, \varnothing, \varnothing\rangle\rangle], \sigma_D\rangle, \text{ok}\rangle\end{array}} \quad \text{(opent-ok)}$$

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

$$\frac{\text{file } str \text{ exists} \quad \sigma_D(str) = \langle \gamma, n\rangle}{\begin{array}{c}\langle f = \texttt{openb}(str), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_{\text{s}} \\ \langle\langle \sigma_S[f \mapsto \text{hndl}], \sigma_H, \sigma_F[\text{hndl} \mapsto \langle str, 0, 0, n\rangle], \sigma_D\rangle, \text{ok}\rangle\end{array}} \quad \text{(openb-ok)}$$

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:

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

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

returns the offset of the upcoming delimiter from offset l for text inputs, and returns the upcoming cutoff position for binary inputs.

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err}}{\langle \mathtt{seek}(f,e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad}\rangle} \quad \text{(sk-bad)}$$

$$\frac{\langle e,\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} u \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud\rangle \quad u < sou}{\langle \mathtt{seek}(f,e),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \mathrm{bad}\rangle} \quad \text{(sk-l)}$$

$$\frac{\langle e,\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} u \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud\rangle \quad \sigma_D(str) = \langle \gamma, n\rangle \quad sou \le u \le \Lambda(l)}{\langle \mathtt{seek}(f,e),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F[\sigma_S(f) \mapsto \langle str, u, sou, ud\rangle],\sigma_D\rangle, \mathrm{ok}\rangle} \quad \text{(sk-ok)}$$

$$\frac{\langle e,\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} u \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud\rangle \quad \sigma_D(str) = \langle \gamma, n\rangle \quad u > \Lambda(l)}{\langle \mathtt{seek}(f,e),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \mathrm{bad}\rangle} \quad \text{(sk-r)}$$

$$\frac{\sigma_F(\sigma_S(f)) = \langle str, l, sou, ud\rangle \quad \sigma_D(str) = \langle \gamma, n\rangle \quad l = \Lambda(l)}{\langle x = \mathtt{read}(f),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \mathrm{bad}\rangle} \quad \text{(frd-out)}$$

$$\frac{\sigma_F(\sigma_S(f)) = \langle str, l, sou, ud\rangle \quad \sigma_D(str) = \langle \gamma, n\rangle \quad l < \Lambda(l)}{\langle x = \mathtt{read}(f),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S[x \mapsto \gamma(l)],\sigma_H,\sigma_F[\sigma_S(f) \mapsto \langle str, l+1, ud\rangle],\sigma_D\rangle, \mathrm{ok}\rangle} \quad \text{(frd-ok)}$$

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 sucessfully 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 though 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

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

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

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

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

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

$$\frac{\sigma_S(f) = \text{null} \vee \sigma_S(d_u) = \text{null} \vee \sigma_S(d_s) = \text{null}}{\langle \texttt{lookatt}(e,f,d_u,d_s)\{s\}, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\text{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \text{bad}\rangle} \quad \text{(lookt-null)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\text{e}} \text{err}}{\langle \texttt{lookatt}(e,f,d_u,d_s)\{s\}, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\text{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \text{bad}\rangle} \quad \text{(lookt-bad)}$$

$$\frac{\begin{array}{c}\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \\ \langle e, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) \neq \varnothing\end{array}}{\langle \texttt{lookatt}(e,f,d_u,d_s)\{s\}, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\text{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \text{bad}\rangle} \quad \text{(lookt-dupl)}$$

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

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

$$\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\text{e}} \text{false}}{\langle \texttt{lookatb}(e,f,o,w,c)\{s\}, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\text{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \text{ok}\rangle} \quad \text{(lookb-end)}$$

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

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

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

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

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

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

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

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

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

$$\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e \text{err}}{\langle a = \texttt{malloc\_ec}(e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle\langle \sigma_S[a \mapsto \text{null}], \sigma_H, \sigma_F, \sigma_D\rangle, \text{ok}\rangle} \quad \text{(malloc-ec-bad)}$$

$$\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e u \quad u \le 0}{\langle a = \texttt{malloc\_ec}(e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle\langle \sigma_S[a \mapsto \text{null}], \sigma_H, \sigma_F, \sigma_D\rangle, \text{ok}\rangle} \quad \text{(malloc-ec-neg)}$$

$$\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e u \quad u > 0 \quad \text{heap allocate}(u) = \bot}{\langle a = \texttt{malloc\_ec}(e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle\langle \sigma_S[a \mapsto \text{null}], \sigma_H, \sigma_F, \sigma_D\rangle, \text{ok}\rangle} \quad \text{(malloc-ec-ovf)}$$

$$\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e u \quad u > 0 \quad \text{heap allocate}(u) = \text{addr}}{\begin{array}{c}\langle a = \texttt{malloc\_ec}(e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \\ \langle\langle \sigma_S[a \mapsto \text{addr}], \sigma_H[\text{addr} \mapsto \langle[0 \mapsto 0, 1 \mapsto 0, \ldots, u-1 \mapsto 0], u\rangle], \sigma_F, \sigma_D\rangle, \text{ok}\rangle\end{array}} \quad \text{(malloc-ec-ok)}$$

$$\frac{\sigma_S(a) = \text{null}}{\langle x = \texttt{free\_ec}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle\langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D\rangle, \text{ok}\rangle} \quad \text{(free-ec-null)}$$

$$\frac{\sigma_S(a) \ne \text{null}}{\begin{array}{c}\langle x = \texttt{free\_ec}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \\ \langle\langle \sigma_S[x \mapsto 0, a \mapsto \text{null}], \sigma_H[\sigma_S(a) \mapsto \bot], \sigma_F, \sigma_D\rangle, \text{ok}\rangle\end{array}} \quad \text{(free-ec-ok)}$$

Figure 3-19: Semantics for arrays with error codes

$$\frac{\sigma_S(f) = \text{null}}{\langle \texttt{end\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e -1} \quad \text{(end-ec-null)}$$

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

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

$$\frac{\sigma_S(f) = \text{null}}{\langle \texttt{pos\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e -1} \quad \text{(pos-ec-null)}$$

$$\frac{\sigma_S(f) \ne \text{null} \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud\rangle}{\langle \texttt{pos\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e l} \quad \text{(pos-ec-ok)}$$

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

$$\frac{\text{file } str \text{ does not exist}}{\langle f = \texttt{opent\_ec}(str), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle\langle \sigma_S[f \mapsto \text{null}], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok}\rangle} \quad \text{(opent-ec-bad)}$$

$$\frac{\text{file } str \text{ exists}}{\begin{array}{c}\langle f = \texttt{opent\_ec}(str), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \\ \langle\langle \sigma_S[f \mapsto \text{hndl}], \sigma_H, \sigma_F[\text{hndl} \mapsto \langle str, 0, 0, \langle \varnothing, \varnothing, \varnothing \rangle\rangle], \sigma_D \rangle, \text{ok}\rangle\end{array}} \quad \text{(opent-ec-ok)}$$

$$\frac{\text{file } str \text{ does not exist}}{\langle f = \texttt{openb\_ec}(str), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle\langle \sigma_S[f \mapsto \text{null}], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok}\rangle} \quad \text{(openb-ec-bad)}$$

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

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
$a = \texttt{malloc}(e)$	$a = \texttt{malloc\_ec}(e)$
$\texttt{free}(a)$	$x = \texttt{free\_ec}(a)$
$\texttt{end}(f)$	$\texttt{end\_ec}(f)$
$\texttt{pos}(f)$	$\texttt{pos\_ec}(f)$
$f = \texttt{opent}(str)$	$f = \texttt{opent\_ec}(str)$
$f = \texttt{openb}(str)$	$f = \texttt{openb\_ec}(str)$
$\texttt{seek}(f, e)$	$x = \texttt{seek\_ec}(f, e)$
$x = \texttt{read}(f)$	$x = \texttt{read\_ec}(f)$

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

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

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

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

$$\frac{\begin{array}{c}\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{e}} u \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \\ \sigma_D(str) = \langle \gamma, n \rangle \quad sou \leq u \leq \Lambda(l)\end{array}}{\begin{array}{c}\langle x = \texttt{seek\_ec}(f,e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \\ \langle\langle \sigma_S[x \mapsto u], \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, u, sou, ud \rangle], \sigma_D \rangle, \text{ok}\rangle\end{array}} \quad \text{(sk-ec-ok)}$$

$$\frac{\begin{array}{c}\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{e}} u \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \\ \sigma_D(str) = \langle \gamma, n \rangle \quad u > \Lambda(l)\end{array}}{\langle x = \texttt{seek\_ec}(f,e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle\langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok}\rangle} \quad \text{(sk-ec-r)}$$

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

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

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \quad \sigma_D(str) = \langle \gamma, n \rangle \quad l < \Lambda(l)}{\begin{array}{c}\langle x = \texttt{read\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \\ \langle\langle \sigma_S[x \mapsto \gamma(l)], \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l+1, ud \rangle], \sigma_D \rangle, \text{ok}\rangle\end{array}} \quad \text{(frd-ec-ok)}$$

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

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

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

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 diagnoze 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 stragies 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 preceeding 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 impelement 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	inspectt, inspectb, lookatt, lookatb, while	assert, malloc, free, end, pos, opent, openb, seek, read
Protective-check	inspectt, inspectb, lookatt, lookatb, while	assert, malloc_ec, free_ec, end_ec, pos_ec, opent_ec, openb_ec, seek_ec, read_ec
Explicit-recovery	lookatt, lookatb, while	malloc_ec, free_ec, end_ec, pos_ec, opent_ec, openb_ec, seek_ec, read_ec
Plain-loop	while	malloc_ec, free_ec, end_ec, pos_ec, opent_ec, openb_ec, seek_ec, read_ec

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

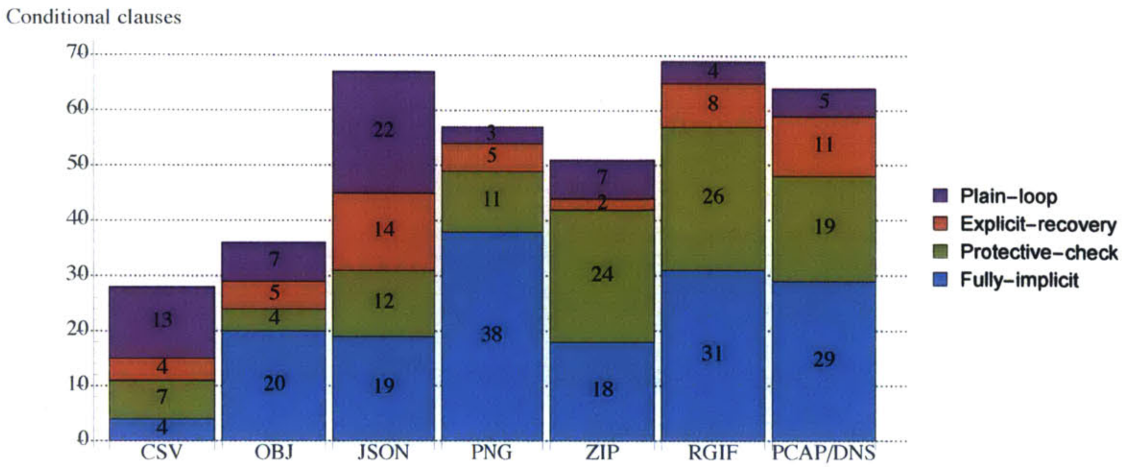

Figure 4-1: Control-flow complexity

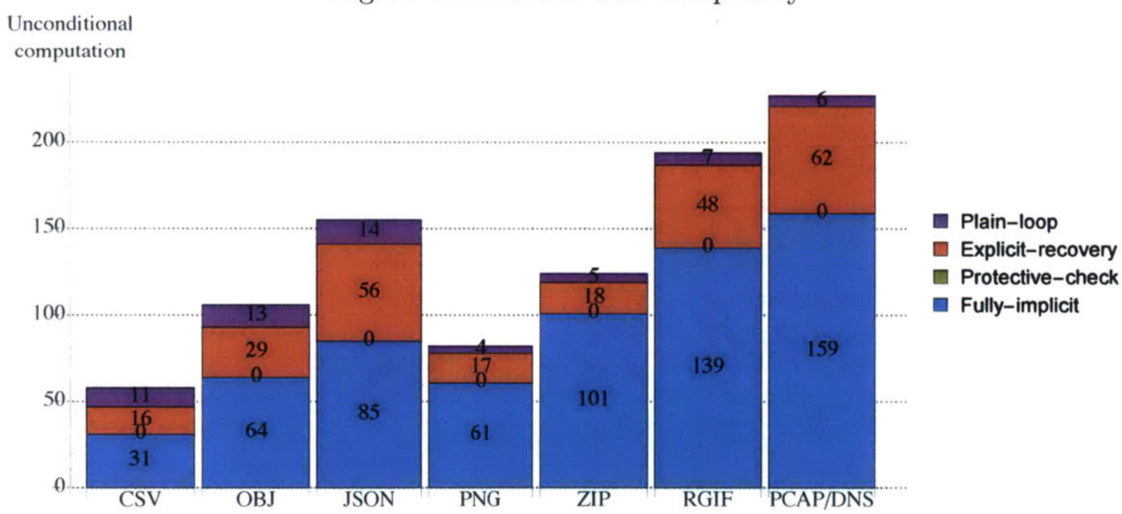

Figure 4-2: Data-manipulation complexity

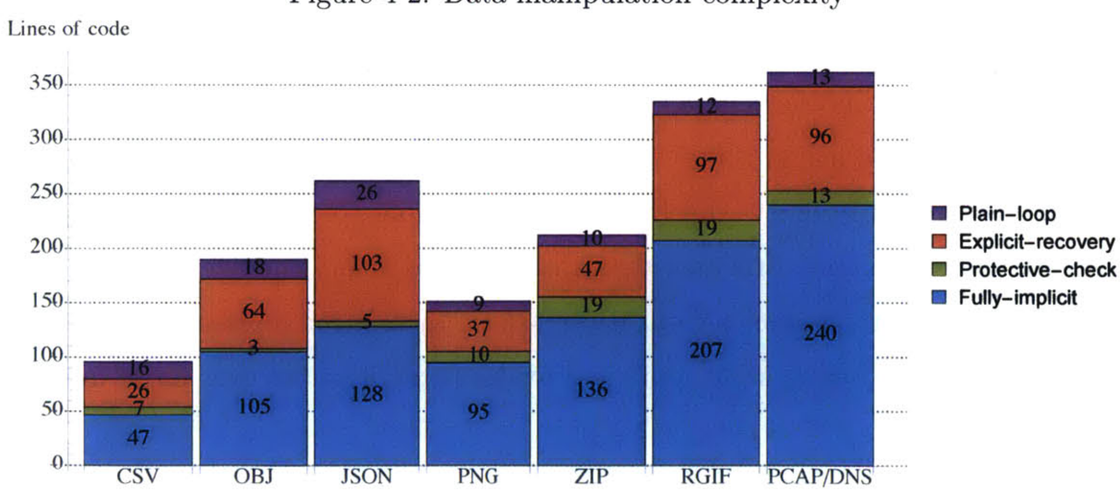

Figure 4-3: Lines of code

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

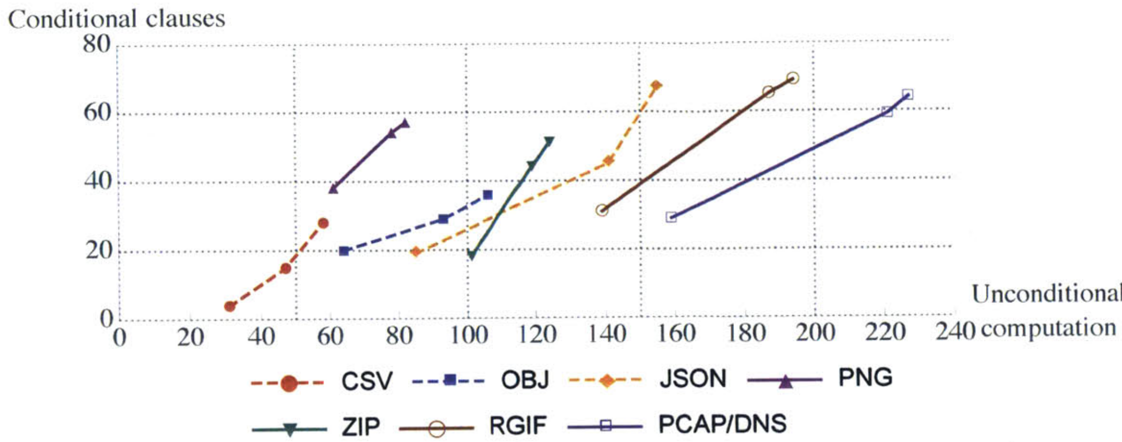

Figure 4-4: Effects of filtering and iterating

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 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 maintainance 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 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

```
main {
  f = opent("../inputs/csv/newlines.csv");

  // titles
  columns = 0;
  inspectt (1, f, ',', '\n') {
    title = malloc(10);
    i = 0;
    while (!end(f)) {
      x = read(f);
      title[i] = x;
      i = i + 1;
    }
    if (columns > 0) {
      print(',');
    }
    print(title);
    free(title);
    columns = columns + 1;
  }
  print('\n');
  print('\n');

  // contents
  assert(columns > 0);
  inspectt (!end(f), f, '\n') {
    j = 0;
    inspectt (j < columns, f, ',') {
      field = malloc(10);
      i = 0;
```

```
      while (!end(f)) {
        x = read(f);
        field[i] = x;
        i = i + 1;
      }
      if (j > 0) {
        print(',');
      }
      print(field);
      free(field);
      j = j + 1;
    }
    print('\n');
    assert(j == columns);
  }
  return 0;
}
```

A.1.2 Protective-check version

```
main {
  f = opent_ec("../inputs/csv/newlines.csv");
  assert(valid(f));

  // titles
  columns = 0;
  inspectt (1, f, ',', '\n') {
    title = malloc_ec(10);
    assert(valid(title));
    i = 0;
    while (!end_ec(f)) {
      assert(i < 10);
      x = read_ec(f);
      assert(x >= 0);
      title[i] = x;
      i = i + 1;
    }
    if (columns > 0) {
      print(',');
    }
    print(title);
    x = free_ec(title);
    columns = columns + 1;
  }
  print('\n');
  print('\n');

  // contents
  assert(columns > 0);
  inspectt (!end_ec(f), f, '\n') {
    j = 0;
    inspectt (j < columns, f, ',') {
      field = malloc_ec(10);
      assert(valid(field));
      i = 0;
      while (!end_ec(f)) {
        assert(i < 10);
        x = read_ec(f);
        assert(x >= 0);
        field[i] = x;
        i = i + 1;
      }
      if (j > 0) {
        print(',');
      }
      print(field);
      x = free_ec(field);
      j = j + 1;
    }
    print('\n');
    assert(j == columns);
  }
  return 0;
```

```
}
```

A.1.3 Explicit-recovery version

```
main {
  f = opent_ec("../inputs/csv/newlines.csv");
  if (!valid(f)) {
    exit(1);
  }

  // titles
  columns = 0;
  lookatt (1, f, ',', '\n') {
    title = malloc_ec(10);
    if (valid(title)) {
      bad = 0;
      i = 0;
      while (!end_ec(f) && i < 10) {
        x = read_ec(f);
        if (x < 0) {
          bad = 1;
        }
        title[i] = x;
        i = i + 1;
      }
      if (end_ec(f) && !bad) {
        if (columns > 0) {
          print(',');
        }
        print(title);
        columns = columns + 1;
      } // skip unit
      x = free_ec(title);
    } // skip unit
  }
  print('\n');
  print('\n');

  // contents
  if (columns <= 0) {
    exit(1);
  }
  buffer = malloc_ec(11 * columns);
  if (!valid(buffer)) {
    exit(1);
  }
  lookatt (!end_ec(f), f, '\n') {
    idx = 0;
    while (idx < 11 * columns) {
      buffer[idx] = 0;
      idx = idx + 1;
    }
    idx = 0;
    j = 0;
    lookatt (j < columns, f, ',') {
      start = idx;
      if (j > 0) {
        buffer[idx] = ',';
        idx = idx + 1;
      }
      i = 0;
      while (!end_ec(f) && x >= 0 && i < 10) {
        x = read_ec(f);
        buffer[idx] = x;
        idx = idx + 1;
        i = i + 1;
      }
      if (end_ec(f) && x >= 0) {
        j = j + 1;
      } else { // skip unit
        while (idx > start) {
          buffer[idx] = 0;
          idx = idx - 1;
```

```
        }
      }
    }
    if (j == columns) {
      print(buffer);
      print('\n');
    } // skip unit
  }
  x = free_ec(buffer);
  return 0;
}
```

A.1.4 Plain-loop version

```
main {
  f = opent_ec("../inputs/csv/newlines.csv");
  if (!valid(f)) {
    exit(1);
  }

  // titles
  columns = 0;
  finish = 0;
  while (!finish) {
    title = malloc_ec(10);
    if (valid(title)) {
      bad = 0;
      i = 1;
      x = read_ec(f);
      while (!bad && x != ',' && x != '\n' && i < 10) {
        if (x < 0) {
          bad = 1;
        }
        title[i] = x;
        i = i + 1;
        x = read_ec(f);
      }
      if (!bad && (x == ',' || x == '\n')) {
        if (columns > 0) {
          print(',');
        }
        print(title);
        columns = columns + 1;
      } // skip unit
      while (x >= 0 && x != ',' && x != '\n') {
        x = read_ec(f);
      }
      dummy = free_ec(title);
    } // skip unit
    if (x < 0 || x == '\n') {
      finish = 1;
    }
  }
  print('\n');
  print('\n');

  // contents
  if (columns <= 0) {
    exit(1);
  }
  buffer = malloc_ec(11 * columns);
  if (!valid(buffer)) {
    exit(1);
  }
  while (!end_ec(f)) {
    idx = 0;
    while (idx < 11 * columns) {
      buffer[idx] = 0;
      idx = idx + 1;
    }
    idx = 0;
    j = 0;
    x = 0;
```

```
    while (j < columns && x >= 0 && x != '\n') {
      start = idx;
      if (j > 0) {
        buffer[idx] = ',';
        idx = idx + 1;
      }
      i = 0;
      x = read_ec(f);
      while (x >= 0 && x != '\n' && x != ',' && i < 10) {
        buffer[idx] = x;
        idx = idx + 1;
        i = i + 1;
        x = read_ec(f);
      }
      if (x == '\n' || x == ',') {
        j = j + 1;
      } else { // skip unit
        while (idx > start) {
          buffer[idx] = 0;
          idx = idx - 1;
        }
      }
      while (x >= 0 && x != '\n' && x != ',') {
        x = read_ec(f);
      }
    }
    if (j == columns) {
      print(buffer);
      print('\n');
    } // skip unit
    while (x >= 0 && x != '\n') {
      x = read_ec(f);
    }
  }
  x = free_ec(buffer);
  return 0;
}
```

A.2 OBJ

A.2.1 Fully-implicit version

```
f = opent("../inputs/obj/icosahedron-2.obj");
num = malloc(10);
dim = 0;

func cleannum(dummy) {
  i = 0;
  while (i < 10) {
    num[i] = 0;
    i = i + 1;
  }
  return 0;
}

func readfloat(dummy) {
  dummy = cleannum(0);

  dot = 0;
  i = 0;
  x = 0;
  sign = 1;
  while (!end(f)) {
    c = read(f);
    if (c == '.') {
      assert(dot == 0);
      dot = 1;
    } else {
      if (c == '-') {
```

```
        assert(i == 0);
        sign = -1;
      } else {
        assert (c >= '0' && c <= '9');
        x = x * 10;
        x = x + (c - '0');
        assert(x >= 0);
      }
    }
    num[i] = c;
    i = i + 1;
  }
  return x * sign;
}

func readidx(n) {
  dummy = cleannum(0);

  idx = 0;
  i = 0;
  while (!end(f)) {
    c = read(f);
    assert(c >= '0' && c <= '9');
    idx = idx * 10;
    idx = idx + (c - '0');
    assert(idx >= 0);
    num[i] = c;
    i = i + 1;
  }

  assert(idx >= 1 && idx <= n);
  return idx;
}

main {
  n = 0;
  m = 0;
  inspectt (!end(f), f, '\n') {
    c = read(f);
    assert(c == 'v' || c == 'f');
    space = read(f);
    assert(space == ' ');
    if (c == 'v') {
      print('v');
      dim = 0;
      inspectt (1, f, ' ') {
        assert(!end(f));
        x = readfloat(0);
        print(' ');
        print(num);
        dim = dim + 1;
      }
      assert(dim == 3);
      print('\n');
      n = n + 1;
    } else {
      print('f');
      dim = 0;
      inspectt (1, f, ' ') {
        assert(!end(f));
        idx = readidx(n);
        print(' ');
        print(num);
        dim = dim + 1;
      }
      assert(dim >= 3);
      print('\n');
      m = m + 1;
    }
  }

  print(n);
  print('\n');
  print(m);
  print('\n');

```

```
  return 0;
}
```

A.2.2 Protective-check version

```
f = opent_ec("../inputs/obj/icosahedron-2.obj");
num = malloc_ec(10);
dim = 0;

func cleannum(dummy) {
  i = 0;
  while (i < 10) {
    num[i] = 0;
    i = i + 1;
  }
  return 0;
}

func readfloat(dummy) {
  dummy = cleannum(0);

  dot = 0;
  i = 0;
  x = 0;
  sign = 1;
  while (!end_ec(f)) {
    assert(i < 10);
    c = read_ec(f);
    if (c == '.') {
      assert(dot == 0);
      dot = 1;
    } else {
      if (c == '-') {
        assert(i == 0);
        sign = -1;
      } else {
        assert (c >= '0' && c <= '9');
        x = x * 10;
        x = x + (c - '0');
        assert(x >= 0);
      }
    }
    num[i] = c;
    i = i + 1;
  }
  return x * sign;
}

func readidx(n) {
  dummy = cleannum(0);

  idx = 0;
  i = 0;
  while (!end_ec(f)) {
    assert(i < 10);
    c = read_ec(f);
    assert(c >= '0' && c <= '9');
    idx = idx * 10;
    idx = idx + (c - '0');
    assert(idx >= 0);
    num[i] = c;
    i = i + 1;
  }

  assert(idx >= 1 && idx <= n);
  return idx;
}

main {
  assert(valid(f) && valid(num));
  n = 0;
  m = 0;
  inspectt (!end(f), f, '\n') {
```

```
      c = read_ec(f);
      assert(c == 'v' || c == 'f');
      space = read_ec(f);
      assert(space == ' ');
      if (c == 'v') {
        print('v');
        dim = 0;
        inspectt (1, f, ' ') {
          assert(!end(f));
          x = readfloat(0);
          print(' ');
          print(num);
          dim = dim + 1;
        }
        assert(dim == 3);
        print('\n');
        n = n + 1;
      } else {
        print('f');
        dim = 0;
        inspectt (1, f, ' ') {
          assert(!end(f));
          idx = readidx(n);
          print(' ');
          print(num);
          dim = dim + 1;
        }
        assert(dim >= 3);
        print('\n');
        m = m + 1;
      }
    }

    print(n);
    print('\n');
    print(m);
    print('\n');

    return 0;
}
```

A.2.3 Explicit-recovery version

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

bad = 0;

func cleannum(dummy) {
  i = 0;
  while (i < 100) {
    num[i] = 0;
    i = i + 1;
  }
  numidx = 0;
  return 0;
}

func putnum(c) {
  if (numidx >= 100) {
    bad = 1;
  } else {
    num[numidx] = c;
    numidx = numidx + 1;
  }
  return 0;
}

func revertnum(start) {
  while (numidx > start) {
    numidx = numidx - 1;
```

```
    num[numidx] = 0;
  }
  return 0;
}

func readfloat(dummy) {
  dummy = putnum(' ');

  dot = 0;
  i = 0;
  x = 0;
  sign = 1;
  while (!end_ec(f)) {
    c = read_ec(f);
    if (i >= 10 || numidx >= 100) {
      bad = 1;
    } else {
      if (c == '.') {
        if (dot != 0) {
          bad = 1;
        }
        dot = 1;
      } else {
        if (c == '-') {
          if (i != 0) {
            bad = 1;
          }
          sign = -1;
        } else {
          if (c >= '0' && c <= '9') {
            x = x * 10;
            x = x + (c - '0');
            if (x < 0) {
              bad = 1;
            }
          } else {
            bad = 1;
          }
        }
      }
      dummy = putnum(c);
      i = i + 1;
    }
  }
  return x * sign;
}

func readidx(n) {
  dummy = putnum(' ');

  idx = 0;
  i = 0;
  while (!end_ec(f)) {
    c = read_ec(f);
    if (i >= 10 || numidx >= 100) {
      bad = 1;
    } else {
      if (c >= '0' && c <= '9') {
        idx = idx * 10;
        idx = idx + (c - '0');
        if (idx < 0) {
          bad = 1;
        }
      } else {
        bad = 1;
      }
    }
    dummy = putnum(c);
    i = i + 1;
  }

  if (!(idx >= 1 && idx <= n)) {
    bad = 1;
  }
  return idx;
}
```

```

main {
  if (!(valid(f) && valid(num))) {
    exit(1);
  }
  n = 0;
  m = 0;
  lookatt (!end_ec(f), f, '\n') {
    c = read_ec(f);
    if (c == 'v' || c == 'f') {
      space = read_ec(f);
      if (space == ' ') {
        if (c == 'v') {
          dim = 0;
          dummy = cleannum(0);
          lookatt (1, f, ' ') {
            bad = 0;
            if (!end_ec(f)) {
              start = numidx;
              x = readfloat(0);
              if (!bad) {
                dim = dim + 1;
              } else { // discard bad updates
                dummy = revertnum(start);
              }
            }
          }
          if (dim == 3) {
            print('v');
            print(num);
            print('\n');
            n = n + 1;
          } // skip unit
        } else {
          dim = 0;
          dummy = cleannum(0);
          lookatt (1, f, ' ') {
            bad = 0;
            if (!end_ec(f)) {
              start = numidx;
              idx = readidx(n);
              if (!bad) {
                dim = dim + 1;
              } else { // discard bad updates
                dummy = revertnum(start);
              }
            }
          }
          if (dim >= 3) {
            print('f');
            print(num);
            print('\n');
            m = m + 1;
          } // skip unit
        }
      } // skip unit
    } // skip unit
  }

  print(n);
  print('\n');
  print(m);
  print('\n');

  return 0;
}
```

A.2.4 Plain-loop version

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

```
bad = 0;
endline = 0;

func cleannum(dummy) {
  i = 0;
  while (i < 100) {
    num[i] = 0;
    i = i + 1;
  }
  numidx = 0;
  return 0;
}

func putnum(c) {
  if (numidx >= 100) {
    bad = 1;
  } else {
    num[numidx] = c;
    numidx = numidx + 1;
  }
  return 0;
}

func revertnum(start) {
  while (numidx > start) {
    numidx = numidx - 1;
    num[numidx] = 0;
  }
  return 0;
}

func readfloat(dummy) {
  dummy = putnum(' ');

  dot = 0;
  i = 0;
  x = 0;
  sign = 1;
  c = read_ec(f);
  while (c >= 0 && c != '\n' && c != ' ') {
    if (i >= 10 || numidx >= 100) {
      bad = 1;
    } else {
      if (c == '.') {
        if (dot != 0) {
          bad = 1;
        }
        dot = 1;
      } else {
        if (c == '-') {
          if (i != 0) {
            bad = 1;
          }
          sign = -1;
        } else {
          if (c >= '0' && c <= '9') {
            x = x * 10;
            x = x + (c - '0');
            if (x < 0) {
              bad = 1;
            }
          } else {
            bad = 1;
          }
        }
      }
      dummy = putnum(c);
      i = i + 1;
    }
    c = read_ec(f);
  }

  if (c != ' ') {
    endline = 1;
  }
```

```
  if (i <= 0) {
    bad = 1;
  }
  return x * sign;
}

func readidx(n) {
  dummy = putnum(' ');

  idx = 0;
  i = 0;
  c = read_ec(f);
  while (c >= 0 && c != '\n' && c != ' ') {
    if (i >= 10 || numidx >= 100) {
      bad = 1;
    } else {
      if (c >= '0' && c <= '9') {
        idx = idx * 10;
        idx = idx + (c - '0');
        if (idx < 0) {
          bad = 1;
        }
      } else {
        bad = 1;
      }
    }
    dummy = putnum(c);
    i = i + 1;
    c = read_ec(f);
  }

  if (c != ' ') {
    endline = 1;
  }
  if (!(i > 0 && idx >= 1 && idx <= n)) {
    bad = 1;
  }
  return idx;
}

main {
  if (!(valid(f) && valid(num))) {
    exit(1);
  }
  n = 0;
  m = 0;
  while (!end_ec(f)) {
    c = read_ec(f);
    if (c == 'v' || c == 'f') {
      space = read_ec(f);
      if (space == ' ') {
        if (c == 'v') {
          dim = 0;
          dummy = cleannum(0);
          endline = 0;
          while (!endline) {
            bad = 0;
            start = numidx;
            x = readfloat(0);
            if (!bad) {
              dim = dim + 1;
            } else { // discard bad updates
              dummy = revertnum(start);
            }
          }
          if (dim == 3) {
            print('v');
            print(num);
            print('\n');
            n = n + 1;
          } // skip unit
        } else {
          dim = 0;
          dummy = cleannum(0);
          endline = 0;
          while (!endline) {
```

```
            bad = 0;
            start = numidx;
            idx = readidx(n);
            if (!bad) {
              dim = dim + 1;
            } else { // discard bad updates
              dummy = revertnum(start);
            }
          }
          if (dim >= 3) {
            print('f');
            print(num);
            print('\n');
            m = m + 1;
          } // skip unit
        }
        dummy = seek_ec(f, pos_ec(f) - 1);
        c = read_ec(f);
      } else { // skip unit
        c = space;
      }
    } // skip unit
    while (c >= 0 && c != '\n') {
      c = read_ec(f);
    }
  }

  print(n);
  print('\n');
  print(m);
  print('\n');

  return 0;
}
```

A.3 JSON

A.3.1 Fully-implicit version

```
tokenstart = malloc(10);
tokenlen = malloc(10);
tokentype = malloc(10);
tokencount = 0;

func DFS(f, level) {
  first = read(f);
  // remove leading whitespace
  while (first == ' ' || first == '\n') {
    first = read(f);
  }
  assert (first != ',' && first != ':' && first != '}' && first != ']');
  if (first == '{') { // object
    objlen = 0;
    inspectt (1, f, ',', '}') {
      c = read(f);
      key = malloc(20);
      while (c == ' ' || c == '\n') {
        c = read(f);
      }
      // parse key token
      tokenstart[tokencount] = pos(f);
      i = 0;
      while (c != ':') {
        assert(c != '[' && c != '{');
        if (c == '\n') {
          key[i] = ' ';
        } else {
          key[i] = c;
        }
```

```
          i = i + 1;
          c = read(f);
        }
        // remove trailing whitespace
        while (i > 0 && (key[i-1] == ' ' || key[i-1] == '\n')) {
          key[i-1] = 0;
          i = i - 1;
        }
        // maintain global arrays
        tokenlen[tokencount] = i;
        tokentype[tokencount] = 'k';
        tokencount = tokencount + 1;
        // print key token
        j = 0;
        print('\n');
        while (j < level) {
          print(' ');
          print(' ');
          j = j + 1;
        }
        print(key);
        free(key);
        print(':');
        print(' ');
        // parse value
        vallen = DFS(f, level + 1);
        assert (vallen > 0);
        objlen = objlen + vallen + i;
      }
      return objlen;
    } else {
      if (first == '[') { // array
        arrlen = 0;
        print('\n');
        // parse array elements
        inspectt (1, f, ',', ']') {
          i = 0;
          while (i < level) {
            print(' ');
            print(' ');
            i = i + 1;
          }
          print('-');
          print(' ');
          elemlen = DFS(f, level + 1);
          assert (elemlen > 0);
          print('\n');
          arrlen = arrlen + elemlen;
        }
        return arrlen;
      } else { // single token
        word = malloc(20);
        // parse value token
        tokenstart[tokencount] = pos(f);
        word[0] = first;
        i = 1;
        while (!end(f)) {
          c = read(f);
          assert(c != ':');
          if (c == '\n') {
            word[i] = ' ';
          } else {
            word[i] = c;
          }
          i = i + 1;
        }
        // remove trailing whitespace
        while (i > 0 && (word[i-1] == ' ' || word[i-1] == '\n')) {
          word[i-1] = 0;
          i = i - 1;
        }
        // maintain global arrays
        tokenlen[tokencount] = i;
        tokentype[tokencount] = 'v';
        tokencount = tokencount + 1;
        // print value token
```

```
      print(word);
      free(word);
      return i;
    }
  }
}

main {
  f = opent("../inputs/json/widget.json");
  total = DFS(f, 0);
  print('\n');

  print(tokencount);
  print('\n');
  print(tokenstart);
  print('\n');
  print(tokenlen);
  print('\n');
  print(tokentype);
  print('\n');
  return total;
}
```

A.3.2 Protective-check version

```
tokenstart = malloc_ec(10);
tokenlen = malloc_ec(10);
tokentype = malloc_ec(10);
tokencount = 0;

func DFS(f, level) {
  first = read_ec(f);
  // remove leading whitespace
  while (first == ' ' || first == '\n') {
    first = read_ec(f);
  }
  assert (first >= 0 && first != ',' && first != ':' && first != '}' && first
      != ']');
  if (first == '{') { // object
    objlen = 0;
    inspectt (tokencount < 9, f, ',', '}') {
      c = read_ec(f);
      key = malloc_ec(20);
      assert(valid(key));
      while (c == ' ' || c == '\n') {
        c = read_ec(f);
      }
      // parse key token
      tokenstart[tokencount] = pos_ec(f);
      assert(tokenstart[tokencount] >= 0);
      i = 0;
      while (c != ':') {
        assert(i < 20 && c >= 0 && c != '[' && c != '{');
        if (c == '\n') {
          key[i] = ' ';
        } else {
          key[i] = c;
        }
        i = i + 1;
        c = read_ec(f);
      }
      // remove trailing whitespace
      while (i > 0 && (key[i-1] == ' ' || key[i-1] == '\n')) {
        key[i-1] = 0;
        i = i - 1;
      }
      // maintain global arrays
      tokenlen[tokencount] = i;
      tokentype[tokencount] = 'k';
      tokencount = tokencount + 1;
      // print key token
      j = 0;
      print('\n');
```

```
      while (j < level) {
        print(' ');
        print(' ');
        j = j + 1;
      }
      print(key);
      dummy = free_ec(key);
      print(':');
      print(' ');
      // parse value
      vallen = DFS(f, level + 1);
      assert (vallen > 0);
      objlen = objlen + vallen + i;
    }
    return objlen;
  } else {
    if (first == '[') { // array
      arrlen = 0;
      print('\n');
      // parse array elements
      inspectt (1, f, ',', ']') {
        i = 0;
        while (i < level) {
          print(' ');
          print(' ');
          i = i + 1;
        }
        print('-');
        print(' ');
        elemlen = DFS(f, level + 1);
        assert (elemlen > 0);
        print('\n');
        arrlen = arrlen + elemlen;
      }
      return arrlen;
    } else { // single token
      word = malloc_ec(20);
      assert(valid(word));
      // parse value token
      tokenstart[tokencount] = pos_ec(f);
      word[0] = first;
      i = 1;
      while (!end_ec(f)) {
        assert(i < 20);
        c = read_ec(f);
        assert(c >= 0 && c != ':');
        if (c == '\n') {
          word[i] = ' ';
        } else {
          word[i] = c;
        }
        i = i + 1;
      }
      // remove trailing whitespace
      while (i > 0 && (word[i-1] == ' ' || word[i-1] == '\n')) {
        word[i-1] = 0;
        i = i - 1;
      }
      // maintain global arrays
      tokenlen[tokencount] = i;
      tokentype[tokencount] = 'v';
      tokencount = tokencount + 1;
      // print value token
      print(word);
      dummy = free_ec(word);
      return i;
    }
  }
}

main {
  f = opent_ec("../inputs/json/widget.json");
  assert(valid(f) && valid(tokenstart) && valid(tokenlen) && valid(tokentype))
    ;
  total = DFS(f, 0);
  print('\n');
```

```
  print(tokencount);
  print('\n');
  print(tokenstart);
  print('\n');
  print(tokenlen);
  print('\n');
  print(tokentype);
  print('\n');
  return total;
}
```

A.3.3 Explicit-recovery version

```
tokenstart = malloc_ec(10);
tokenlen = malloc_ec(10);
tokentype = malloc_ec(10);
tokencount = 0;

stack = malloc_ec(150);
stackidx = 0;
stackbad = 0;

func putstack(c) {
  if (stackidx >= 150) {
    stackbad = 1;
  } else {
    stack[stackidx] = c;
    stackidx = stackidx + 1;
  }
  return 0;
}

func revertstack(start) {
  while (stackidx > start) {
    if (stackidx < 150) {
      stack[stackidx] = 0;
    }
    stackidx = stackidx - 1;
  }
  if (stackidx < 150) {
    stackbad = 0;
  }
  return 0;
}

func reverttoken(start) {
  while (tokencount > start) {
    if (tokencount < 10) {
      tokenstart[tokencount] = 0;
      tokenlen[tokencount] = 0;
      tokentype[tokencount] = 0;
    }
    tokencount = tokencount - 1;
  }
  return 0;
}

func DFS(f, level) {
  first = read_ec(f);
  // remove leading whitespace
  while (first == ' ' || first == '\n') {
    first = read_ec(f);
  }
  if (!(first >= 0 && first != ',' && first != ':' && first != '}' && first !=
      ']')) {
    return -1;
  }
  if (first == '{') { // object
    objlen = 0;
    lookatt (tokencount < 9, f, ',', '}') {
      bad = 0;
      c = read_ec(f);
```

```
        key = malloc_ec(20);
        if (valid(key)) {
          while (c == ' ' || c == '\n') {
            c = read_ec(f);
          }
          // parse key token
          position = pos_ec(f);
          if (position < 0) {
            bad = 1;
          }
          i = 0;
          while (!bad && c != ':') {
            if (!(i < 20 && c >= 0 && c != '[' && c != '{')) {
              bad = 1;
            }
            if (c == '\n') {
              key[i] = ' ';
            } else {
              key[i] = c;
            }
            i = i + 1;
            c = read_ec(f);
          }
          if (!bad) {
            stackok = stackidx;
            tokenok = tokencount;
            // remove trailing whitespace
            while (i > 0 && (key[i-1] == ' ' || key[i-1] == '\n')) {
              key[i-1] = 0;
              i = i - 1;
            }
            // maintain global arrays
            tokenstart[tokencount] = position;
            tokenlen[tokencount] = i;
            tokentype[tokencount] = 'k';
            tokencount = tokencount + 1;
            // print key token
            j = 0;
            dummy = putstack('\n');
            while (j < level) {
              dummy = putstack(' ');
              dummy = putstack(' ');
              j = j + 1;
            }
            j = 0;
            while (j < i) {
              x = key[j];
              dummy = putstack(x);
              j = j + 1;
            }
            dummy = putstack(':');
            dummy = putstack(' ');
            // parse value
            vallen = DFS(f, level + 1);
            if (!stackbad && vallen > 0) {
              objlen = objlen + vallen + i;
            } else { // discard bad updates
              dummy = revertstack(stackok);
              dummy = reverttoken(tokenok);
            }
          } // skip unit
          dummy = free_ec(key);
        } // skip unit
      }
      return objlen;
    } else {
      if (first == '[') { // array
        arrlen = 0;
        dummy = putstack('\n');
        // parse array elements
        lookatt (1, f, ',', ']') {
          stackok = stackidx;
          tokenok = tokencount;
          i = 0;
          while (i < level) {
            dummy = putstack(' ');
```

```
          dummy = putstack(' ');
          i = i + 1;
        }
        dummy = putstack('-');
        dummy = putstack(' ');
        elemlen = DFS(f, level + 1);
        dummy = putstack('\n');
        if (!stackbad && elemlen > 0) {
          arrlen = arrlen + elemlen;
        } else { // discard bad updates
          dummy = revertstack(stackok);
          dummy = reverttoken(tokenok);
        }
      }
      return arrlen;
    } else { // single token
      word = malloc_ec(20);
      bad = 0;
      if (valid(word)) {
        // parse value token
        position = pos_ec(f);
        if (position < 0) {
          bad = 1;
        }
        word[0] = first;
        i = 1;
        while (!bad && !end_ec(f)) {
          if (i >= 20) {
            bad = 1;
          } else {
            c = read_ec(f);
            if (!(c >= 0 && c != ':')) {
              bad = 1;
            } else {
              if (c == '\n') {
                word[i] = ' ';
              } else {
                word[i] = c;
              }
            }
            i = i + 1;
          }
        }
        if (!bad) {
          stackok = stackidx;
          // remove trailing whitespace
          while (i > 0 && (word[i-1] == ' ' || word[i-1] == '\n')) {
            word[i-1] = 0;
            i = i - 1;
          }
          // maintain global arrays
          tokenstart[tokencount] = position;
          tokenlen[tokencount] = i;
          tokentype[tokencount] = 'v';
          tokencount = tokencount + 1;
          // print value token
          j = 0;
          while (j < i) {
            x = word[j];
            dummy = putstack(x);
            j = j + 1;
          }
          if (stackbad) { // discard bad updates
            dummy = revertstack(stackok);
            dummy = reverttoken(tokencount - 1);
            i = 0;
          }
        } else { // skip unit
          i = 0;
        }
        dummy = free_ec(word);
        return i;
      } else { // skip unit
        return -1;
      }
    }
```

```
  }
}

main {
  f = opent_ec("../inputs/json/widget.json");
  if (!(valid(f) && valid(tokenstart) && valid(tokenlen) && valid(tokentype)
      && valid(stack))) {
    exit(1);
  }
  total = DFS(f, 0);
  if (total < 0) {
    exit(1);
  } else {
    print(stack);
    print('\n');

    print(tokencount);
    print('\n');
    print(tokenstart);
    print('\n');
    print(tokenlen);
    print('\n');
    print(tokentype);
    print('\n');
  }
  return total;
}
```

A.3.4 Plain-loop version

```
tokenstart = malloc_ec(10);
tokenlen = malloc_ec(10);
tokentype = malloc_ec(10);
tokencount = 0;

stack = malloc_ec(150);
stackidx = 0;
stackbad = 0;

func putstack(c) {
  if (stackidx >= 150) {
    stackbad = 1;
  } else {
    stack[stackidx] = c;
    stackidx = stackidx + 1;
  }
  return 0;
}

func revertstack(start) {
  while (stackidx > start) {
    if (stackidx < 150) {
      stack[stackidx] = 0;
    }
    stackidx = stackidx - 1;
  }
  if (stackidx < 150) {
    stackbad = 0;
  }
  return 0;
}

func reverttoken(start) {
  while (tokencount > start) {
    if (tokencount < 10) {
      tokenstart[tokencount] = 0;
      tokenlen[tokencount] = 0;
      tokentype[tokencount] = 0;
    }
    tokencount = tokencount - 1;
  }
  return 0;
}
```

```

func DFS(f, level) {
  first = read_ec(f);
  // remove leading whitespace
  while (first == ' ' || first == '\n') {
    first = read_ec(f);
  }
  if (!(first >= 0 && first != ',' && first != ':' && first != '}' && first !=
      ']')) {
    return -1;
  }
  if (first == '{') { // object
    objlen = 0;
    finish = 0;
    while (tokencount < 9 && !finish) {
      bad = 0;
      c = read_ec(f);
      key = malloc_ec(20);
      if (valid(key)) {
        while (c == ' ' || c == '\n') {
          c = read_ec(f);
        }
        // parse key token
        position = pos_ec(f);
        if (position < 0) {
          bad = 1;
        }
        i = 0;
        while (!bad && c != ':' && c != ',' && c != '}' && c != ']') {
          if (!(i < 20 && c >= 0 && c != '[' && c != '{')) {
            bad = 1;
          }
          if (c == '\n') {
            key[i] = ' ';
          } else {
            key[i] = c;
          }
          i = i + 1;
          c = read_ec(f);
        }
        if (!bad) {
          stackok = stackidx;
          tokenok = tokencount;
          // remove trailing whitespace
          while (i > 0 && (key[i-1] == ' ' || key[i-1] == '\n')) {
            key[i-1] = 0;
            i = i - 1;
          }
          // maintain global arrays
          tokenstart[tokencount] = position;
          tokenlen[tokencount] = i;
          tokentype[tokencount] = 'k';
          tokencount = tokencount + 1;
          // print key token
          j = 0;
          dummy = putstack('\n');
          while (j < level) {
            dummy = putstack(' ');
            dummy = putstack(' ');
            j = j + 1;
          }
          j = 0;
          while (j < i) {
            x = key[j];
            dummy = putstack(x);
            j = j + 1;
          }
          dummy = putstack(':');
          dummy = putstack(' ');
          // parse value
          vallen = DFS(f, level + 1);
          if (!stackbad && vallen > 0) {
            objlen = objlen + vallen + i;
          } else { // discard bad updates
            dummy = revertstack(stackok);
            dummy = reverttoken(tokenok);
```

```
          }
        } // skip unit
        dummy = free_ec(key);
      } // skip unit
      while (c >= 0 && c != ',' && c != '}' && c != ']') { // end of unit
        c = read_ec(f);
      }
      if (c < 0 || c == '}') { // end of sequence
        finish = 1;
      }
      if (c == ']') { // outside delimiters
        finish = 1;
        dummy = seek_ec(f, pos_ec(f) - 1);
      }
    }
    return objlen;
  } else {
    if (first == '[') { // array
      arrlen = 0;
      dummy = putstack('\n');
      // parse array elements
      while (!finish) {
        stackok = stackidx;
        tokenok = tokencount;
        i = 0;
        while (i < level) {
          dummy = putstack(' ');
          dummy = putstack(' ');
          i = i + 1;
        }
        dummy = putstack('-');
        dummy = putstack(' ');
        elemlen = DFS(f, level + 1);
        dummy = putstack('\n');
        if (!stackbad && elemlen > 0) {
          arrlen = arrlen + elemlen;
        } else { // discard bad updates
          dummy = revertstack(stackok);
          dummy = reverttoken(tokenok);
        }
        c = read_ec(f);
        while (c >= 0 && c != ',' && c != ']' && c != '}') { // end of unit
          c = read_ec(f);
        }
        if (c < 0 || c == ']') { // end of sequence
          finish = 1;
        }
        if (c == '}') { // outside delimiters
          finish = 1;
          dummy = seek_ec(f, pos_ec(f) - 1);
        }
      }
      return arrlen;
    } else { // single token
      word = malloc_ec(20);
      bad = 0;
      if (valid(word)) {
        // parse value token
        position = pos_ec(f);
        if (position < 0) {
          bad = 1;
        }
        word[0] = first;
        i = 1;
        c = read_ec(f);
        while (!bad && c >= 0 && c != ',' && c != ']' && c != '}') {
          if (i >= 20) {
            bad = 1;
          } else {
            if (!(c >= 0 && c != ':')) {
              bad = 1;
            } else {
              if (c == '\n') {
                word[i] = ' ';
              } else {
                word[i] = c;
```

```
              }
            }
            i = i + 1;
            c = read_ec(f);
          }
        }
        if (c == ',' || c == ']' || c == '}') {
          dummy = seek_ec(f, pos_ec(f) - 1);
        }
        if (!bad) {
          stackok = stackidx;
          // remove trailing whitespace
          while (i > 0 && (word[i-1] == ' ' || word[i-1] == '\n')) {
            word[i-1] = 0;
            i = i - 1;
          }
          // maintain global arrays
          tokenstart[tokencount] = position;
          tokenlen[tokencount] = i;
          tokentype[tokencount] = 'v';
          tokencount = tokencount + 1;
          // print value token
          j = 0;
          while (j < i) {
            x = word[j];
            dummy = putstack(x);
            j = j + 1;
          }
          if (stackbad) { // discard bad updates
            dummy = revertstack(stackok);
            dummy = reverttoken(tokencount - 1);
            i = 0;
          }
        } else { // skip unit
          i = 0;
        }
        dummy = free_ec(word);
        return i;
      } else { // skip unit
        return -1;
      }
    }
  }
}

main {
  f = opent_ec("../inputs/json/widget.json");
  if (!(valid(f) && valid(tokenstart) && valid(tokenlen) && valid(tokentype)
      && valid(stack))) {
    exit(1);
  }
  total = DFS(f, 0);
  if (total < 0) {
    exit(1);
  } else {
    print(stack);
    print('\n');

    print(tokencount);
    print('\n');
    print(tokenstart);
    print('\n');
    print(tokenlen);
    print('\n');
    print(tokentype);
    print('\n');
  }
  return total;
}
```

A.4 PNG

A.4.1 Fully-implicit version

```
func readintbytes (f, n) {
  x = 0;
  i = 0;
  while (i < n) {
    byte = read(f);
    x = x << 8;
    x = x | byte;
    i = i + 1;
  }
  return x;
}

main {
  f = openb("../inputs/png0/oi4n0g16-2.png", 0);

  magic = malloc(8);
  i = 0;
  while (i < 8) {
    x = read(f);
    magic[i] = x;
    i = i + 1;
  }
  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);
  free(magic);

  n = readintbytes(f, 4);
  assert(n == 13);

  ihdr = malloc(4);
  i = 0;
  while (i < 4) {
    x = read(f);
    ihdr[i] = x;
    i = i + 1;
  }
  assert(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr[2] == 'D' && ihdr[3] == 'R')
       ;
  free(ihdr);

  w = readintbytes(f, 4);
  h = readintbytes(f, 4);
  assert(w > 0 && h > 0);

  depth = read(f);
  color = read(f);
  compression = read(f);
  filter = read(f);
  interlace = read(f);
  assert((depth == 1 || depth == 2 || depth == 4 || depth == 8 || depth == 16)
        && color == 0 && compression == 0 && filter == 0 && (interlace == 0 ||
       interlace == 1));

  crc = readintbytes(f, 4);
  // check CRC

  ndata = 0;
  finish = 0;
  type = malloc(4);
  inspectb (!finish, f, 0, 4, 8) {
    length = readintbytes(f, 4);

    i = 0;
    while (i < 4) {
      x = read(f);
      type[i] = x;
      i = i + 1;
```

```
    }
    // check type legal

    if (type[0] == 'I' && type[1] == 'E' && type[2] == 'N' && type[3] == 'D')
        { //
        IEND
      a = read(f);
      b = read(f);
      c = read(f);
      d = read(f);
      assert(length == 0 && a == 174 && b == 66 && c == 96 && d == 130);
      finish = 1;
    } else {
      bytes = malloc(length);
      i = 0;
      while (i < length) {
        x = read(f);
        bytes[i] = x;
        i = i + 1;
      }
      crc = readintbytes(f, 4);
      // check CRC
      if (type[0] == 'I' && type[1] == 'D' && type[2] == 'A' && type[3] == 'T
          ') { //
          IDAT
        ndata = ndata + 1;
        // decompress
        // decode
        print(bytes);
      }
      free(bytes);
    }
  }
  free(type);
  return 0;
}
```

A.4.2 Protective-check version

```
func readintbytes (f, n) {
  x = 0;
  i = 0;
  while (i < n) {
    byte = read_ec(f);
    assert(byte >= 0);
    x = x << 8;
    x = x | byte;
    i = i + 1;
  }
  return x;
}

main {
  f = openb_ec("../inputs/png0/oi4n0g16-2.png", 0);

  magic = malloc_ec(8);
  assert(valid(f) && valid(magic));
  i = 0;
  while (i < 8) {
    x = read_ec(f);
    assert(x >= 0);
    magic[i] = x;
    i = i + 1;
  }
  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);
  x = free_ec(magic);

  n = readintbytes(f, 4);
  assert(n == 13);

  ihdr = malloc_ec(4);
```

```
assert(valid(ihdr));
i = 0;
while (i < 4) {
  x = read_ec(f);
  assert(x >= 0);
  ihdr[i] = x;
  i = i + 1;
}
assert(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr[2] == 'D' && ihdr[3] == 'R');
x = free_ec(ihdr);

w = readintbytes(f, 4);
h = readintbytes(f, 4);
assert(w > 0 && h > 0);

depth = read_ec(f);
color = read_ec(f);
compression = read_ec(f);
filter = read_ec(f);
interlace = read_ec(f);
assert((depth == 1 || depth == 2 || depth == 4 || depth == 8 || depth == 16) && color == 0 && compression == 0 && filter == 0 && (interlace == 0 || interlace == 1));

crc = readintbytes(f, 4);
// check CRC

ndata = 0;
finish = 0;
type = malloc_ec(4);
assert(valid(type));
inspectb (!finish, f, 0, 4, 8) {
  length = readintbytes(f, 4);

  i = 0;
  while (i < 4) {
    x = read_ec(f);
    assert(x >= 0);
    type[i] = x;
    i = i + 1;
  }
  // check type legal

  if (type[0] == 'I' && type[1] == 'E' && type[2] == 'N' && type[3] == 'D') { // IEND
    a = read_ec(f);
    b = read_ec(f);
    c = read_ec(f);
    d = read_ec(f);
    assert(length == 0 && a == 174 && b == 66 && c == 96 && d == 130);
    finish = 1;
  } else {
    assert(length > 0);
    bytes = malloc_ec(length);
    assert(valid(bytes));
    i = 0;
    while (i < length) {
      x = read_ec(f);
      assert(x >= 0);
      bytes[i] = x;
      i = i + 1;
    }
    crc = readintbytes(f, 4);
    // check CRC
    if (type[0] == 'I' && type[1] == 'D' && type[2] == 'A' && type[3] == 'T') { // IDAT
      ndata = ndata + 1;
      // decompress
      // decode
      print(bytes);
    }
    x = free_ec(bytes);
  }
```

```
  }
  x = free_ec(type);
  return 0;
}
```

A.4.3 Explicit-recovery version

```
bad = 0;

func readintbytes (f, n) {
  x = 0;
  i = 0;
  while (i < n && !bad) {
    byte = read_ec(f);
    if (byte >= 0) {
      x = x << 8;
      x = x | byte;
      i = i + 1;
    } else {
      bad = 1;
      return -1;
    }
  }
  return x;
}

main {
  f = openb_ec("../inputs/png0/oi4n0g16-2.png", 0);

  magic = malloc_ec(8);
  if (!valid(f) || !valid(magic)) {
    exit(1);
  }
  i = 0;
  while (i < 8) {
    x = read_ec(f);
    if (x < 0) {
      exit(1);
    }
    magic[i] = x;
    i = i + 1;
  }
  if (!(magic[0] == 137 && magic[1] == 'P' && magic[2] == 'N' && magic[3] == '
      G' && magic[4] == 13 && magic[5] == 10 && magic[6] == 26 && magic[7] ==
      10)) {
    exit(1);
  }
  x = free_ec(magic);

  n = readintbytes(f, 4);
  if (bad || n != 13) {
    exit(1);
  }

  ihdr = malloc_ec(4);
  if (!valid(ihdr)) {
    exit(1);
  }
  i = 0;
  while (i < 4) {
    x = read_ec(f);
    if (x < 0) {
      exit(1);
    }
    ihdr[i] = x;
    i = i + 1;
  }
  if (!(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr[2] == 'D' && ihdr[3] == 'R'))
      {
    exit(1);
  }
  x = free_ec(ihdr);

```

```
  w = readintbytes(f, 4);
  h = readintbytes(f, 4);
  if (bad || w <= 0 || h <= 0) {
    exit(1);
  }

  depth = read_ec(f);
  color = read_ec(f);
  compression = read_ec(f);
  filter = read_ec(f);
  interlace = read_ec(f);
  if (!(depth == 1 || depth == 2 || depth == 4 || depth == 8 || depth == 16)
      || color != 0 || compression != 0 || filter != 0 || !(interlace == 0 ||
      interlace == 1)) {
    exit(1);
  }

  crc = readintbytes(f, 4);
  if (bad) {
    exit(1);
  }
  // check CRC

  ndata = 0;
  finish = 0;
  type = malloc_ec(4);
  if (!valid(type)) {
    exit(1);
  }
  lookatb (!finish, f, 0, 4, 8) {
    bad = 0;
    length = readintbytes(f, 4);

    i = 0;
    while (i < 4) {
      x = read_ec(f);
      if (x < 0) {
        bad = 1;
      }
      type[i] = x;
      i = i + 1;
    }
    // check type legal

    if (type[0] == 'I' && type[1] == 'E' && type[2] == 'N' && type[3] == 'D')
        { //
        IEND
      a = read_ec(f);
      b = read_ec(f);
      c = read_ec(f);
      d = read_ec(f);
      if (length == 0 && a == 174 && b == 66 && c == 96 && d == 130) { // good
        finish = 1;
      } // skip unit
    } else {
      if (length > 0) { // good
        bytes = malloc_ec(length);
        if (valid(bytes)) {
          i = 0;
          while (i < length) {
            x = read_ec(f);
            if (x < 0) {
              bad = 1;
            }
            bytes[i] = x;
            i = i + 1;
          }
          crc = readintbytes(f, 4);
          // check CRC
          if (!bad && type[0] == 'I' && type[1] == 'D' && type[2] == 'A' &&
              type[3] == 'T') { //
              IDAT
            ndata = ndata + 1;
            // decompress
            // decode
            print(bytes);
```

```
          } // skip unit
          x = free_ec(bytes);
        } // skip unit
      } // skip unit
    }
  }
  x = free_ec(type);
  return 0;
}
```

A.4.4 Plain-loop version

```
bad = 0;

func readintbytes (f, n) {
  x = 0;
  i = 0;
  while (i < n && !bad) {
    byte = read_ec(f);
    if (byte >= 0) {
      x = x << 8;
      x = x | byte;
      i = i + 1;
    } else {
      bad = 1;
      return -1;
    }
  }
  return x;
}

main {
  f = openb_ec("../inputs/png0/oi4n0g16-2.png", 0);

  magic = malloc_ec(8);
  if (!valid(f) || !valid(magic)) {
    exit(1);
  }
  i = 0;
  while (i < 8) {
    x = read_ec(f);
    if (x < 0) {
      exit(1);
    }
    magic[i] = x;
    i = i + 1;
  }
  if (!(magic[0] == 137 && magic[1] == 'P' && magic[2] == 'N' && magic[3] == '
      G' && magic[4] == 13 && magic[5] == 10 && magic[6] == 26 && magic[7] ==
      10)) {
    exit(1);
  }
  x = free_ec(magic);

  n = readintbytes(f, 4);
  if (bad || n != 13) {
    exit(1);
  }

  ihdr = malloc_ec(4);
  if (!valid(ihdr)) {
    exit(1);
  }
  i = 0;
  while (i < 4) {
    x = read_ec(f);
    if (x < 0) {
      exit(1);
    }
    ihdr[i] = x;
    i = i + 1;
  }
```

```
if (!(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr[2] == 'D' && ihdr[3] == 'R')) {
  exit(1);
}
x = free_ec(ihdr);

w = readintbytes(f, 4);
h = readintbytes(f, 4);
if (bad || w <= 0 || h <= 0) {
  exit(1);
}

depth = read_ec(f);
color = read_ec(f);
compression = read_ec(f);
filter = read_ec(f);
interlace = read_ec(f);
if (!(depth == 1 || depth == 2 || depth == 4 || depth == 8 || depth == 16) || color != 0 || compression != 0 || filter != 0 || !(interlace == 0 || interlace == 1)) {
  exit(1);
}

crc = readintbytes(f, 4);
if (bad) {
  exit(1);
}
// check CRC

ndata = 0;
finish = 0;
type = malloc_ec(4);
if (!valid(type)) {
  exit(1);
}
while (!finish) {
  bad = 0;
  length = readintbytes(f, 4);
  eou = pos_ec(f) + length + 8;
  if (!bad && length >= 0) {
    i = 0;
    while (i < 4) {
      x = read_ec(f);
      if (x < 0) {
        bad = 1;
      }
      type[i] = x;
      i = i + 1;
    }
    // check type legal

    if (type[0] == 'I' && type[1] == 'E' && type[2] == 'N' && type[3] == 'D') { // IEND
      a = read_ec(f);
      b = read_ec(f);
      c = read_ec(f);
      d = read_ec(f);
      if (length == 0 && a == 174 && b == 66 && c == 96 && d == 130) { // good
        finish = 1;
      } // skip unit
    } else { // other types
      if (length > 0) { // good
        bytes = malloc_ec(length);
        if (valid(bytes)) {
          i = 0;
          while (i < length) {
            x = read_ec(f);
            if (x < 0) {
              bad = 1;
            }
            bytes[i] = x;
            i = i + 1;
          }
          crc = readintbytes(f, 4);
```

```
            // check CRC
            if (!bad && type[0] == 'I' && type[1] == 'D' && type[2] == 'A' && type[3] == 'T') { // IDAT
              ndata = ndata + 1;
              // decompress
              // decode
              print(bytes);
            } // skip unit
            x = free_ec(bytes);
          } // skip unit
        } // skip unit
      }
      // go to next unit according to length
      x = seek_ec(f, eou);
      if (x < 0) { // give up
        finish = 1;
      }
    } else { // give up
      finish = 1;
    }
  }
  x = free_ec(type);
  return 0;
}
```

A.5 ZIP

A.5.1 Fully-implicit version

```
func readintbytesl (f, n) { // little endian
  x = read(f);
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

main {
  fdir = openb("../inputs/zip/stuff-1.zip", 1);
  fent = openb("../inputs/zip/stuff-1.zip", 1);

  eocd = size(fdir) - 4;
  found = 0;
  while (!found) {
    seek(fdir, eocd);
    dir_sig = readintbytesl(fdir, 4);
    if (dir_sig == 101010256) { // 0x06054b50
      found = 1;
    } else {
      eocd = eocd - 1;
    }
  }
  assert(found);
  diskid = readintbytesl(fdir, 2);
  ndisk = readintbytesl(fdir, 2);
  dirid = readintbytesl(fdir, 2);
  ndir = readintbytesl(fdir, 2);

  dir_size = readintbytesl(fdir, 4);
  dir_start = readintbytesl(fdir, 4);
  eocd_comm_size = readintbytesl(fdir, 2);
  if (eocd_comm_size > 0) {
    eocd_comm = malloc(eocd_comm_size);
    i = 0;
    while (i < eocd_comm_size) {
      b = read(fdir);
```

```
      eocd_comm[i] = b;
      i = i + 1;
    }
  }

  seek(fdir, dir_start);
  inspectb (pos(fdir) < eocd, fdir, 28, 2, 40) {
    dir_sig = readintbytesl(fdir, 4);
    dir_ver_made = readintbytesl(fdir, 2);
    dir_ver_extr = readintbytesl(fdir, 2);
    dir_flag = readintbytesl(fdir, 2);
    dir_comp = readintbytesl(fdir, 2);
    dir_modif = readintbytesl(fdir, 4);
    dir_crc = readintbytesl(fdir, 4);
    dir_ent_size = readintbytesl(fdir, 4);
    dir_ent_size_uncomp = readintbytesl(fdir, 4);
    dir_name_size = readintbytesl(fdir, 2);
    dir_extr_size = readintbytesl(fdir, 2);
    dir_comm_size = readintbytesl(fdir, 2);
    dir_diskid = readintbytesl(fdir, 2);
    dir_attr = malloc(6);
    i = 0;
    while (i < 6) {
      b = read(fdir);
      dir_attr[i] = b;
      i = i + 1;
    }
    dir_ent_start = readintbytesl(fdir, 4);
    assert(dir_sig == 33639248 && dir_extr_size == 24 && dir_comm_size == 0);
    dir_name = malloc(dir_name_size);
    i = 0;
    while (i < dir_name_size) {
      b = read(fdir);
      dir_name[i] = b;
      i = i + 1;
    }
    print(dir_name);
    print('\n');
    dir_extr = malloc(dir_extr_size);
    i = 0;
    while (i < dir_extr_size) {
      b = read(fdir);
      dir_extr[i] = b;
      i = i + 1;
    }

    seek(fent, dir_ent_start);

    ent_sig = readintbytesl(fent, 4);
    ent_ver_extr = readintbytesl(fent, 2);
    ent_flag = readintbytesl(fent, 2);
    ent_comp = readintbytesl(fent, 2);
    ent_modif = readintbytesl(fent, 4);
    ent_crc = readintbytesl(fent, 4);
    ent_ent_size = readintbytesl(fent, 4);
    ent_ent_size_uncomp = readintbytesl(fent, 4);
    ent_name_size = readintbytesl(fent, 2);
    ent_extr_size = readintbytesl(fent, 2);

    assert(ent_sig == 67324752 && ent_ver_extr == dir_ver_extr && ent_flag ==
        dir_flag && ent_comp == dir_comp && ent_modif == dir_modif &&
        ent_name_size == dir_name_size);
    ent_name = malloc(ent_name_size);
    i = 0;
    while (i < ent_name_size) {
      b = read(fent);
      ent_name[i] = b;
      i = i + 1;
    }
    print(ent_name);
    print('\n');
    if (ent_extr_size > 0) {
      ent_extr = malloc(ent_extr_size);
      i = 0;
      while (i < ent_extr_size) {
        b = read(fent);
```

```
          ent_extr[i] = b;
          i = i + 1;
        }
      }
      assert(ent_flag & 8 == 0 && ent_crc == dir_crc && ent_ent_size ==
          dir_ent_size && ent_ent_size_uncomp == dir_ent_size_uncomp);

      raw_data = malloc(ent_ent_size);
      i = 0;
      while (i < ent_ent_size) {
        b = read(fent);
        raw_data[i] = b;
        i = i + 1;
      }
      // decompress
      // check crc
      print(raw_data);
      print('\n');
      free(ent_name);
      free(raw_data);
    }
    seek(fdir, eocd);
    return 0;
  }
```

A.5.2 Protective-check version

```
func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  assert(x >= 0);
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

main {
  fdir = openb_ec("../inputs/zip/stuff-1.zip", 1);
  fent = openb_ec("../inputs/zip/stuff-1.zip", 1);
  assert(valid(fdir) && valid(fent));

  eocd = size_ec(fdir) - 4;
  found = 0;
  while (!found) {
    x = seek_ec(fdir, eocd);
    assert(x >= 0);
    dir_sig = readintbytesl(fdir, 4);
    if (dir_sig == 101010256) { // 0x06054b50
      found = 1;
    } else {
      eocd = eocd - 1;
    }
  }
  assert(found);
  diskid = readintbytesl(fdir, 2);
  ndisk = readintbytesl(fdir, 2);
  dirid = readintbytesl(fdir, 2);
  ndir = readintbytesl(fdir, 2);

  dir_size = readintbytesl(fdir, 4);
  dir_start = readintbytesl(fdir, 4);
  eocd_comm_size = readintbytesl(fdir, 2);
  if (eocd_comm_size > 0) {
    eocd_comm = malloc_ec(eocd_comm_size);
    assert(valid(eocd_comm));
    i = 0;
    while (i < eocd_comm_size) {
      b = read_ec(fdir);
      assert(b >= 0);
      eocd_comm[i] = b;
      i = i + 1;
```

```
      }
    }

    x = seek_ec(fdir, dir_start);
    assert(x >= 0);
    inspectb (pos_ec(fdir) < eocd && pos_ec(fdir) >= 0, fdir, 28, 2, 40) {
      dir_sig = readintbytesl(fdir, 4);
      dir_ver_made = readintbytesl(fdir, 2);
      dir_ver_extr = readintbytesl(fdir, 2);
      dir_flag = readintbytesl(fdir, 2);
      dir_comp = readintbytesl(fdir, 2);
      dir_modif = readintbytesl(fdir, 4);
      dir_crc = readintbytesl(fdir, 4);
      dir_ent_size = readintbytesl(fdir, 4);
      dir_ent_size_uncomp = readintbytesl(fdir, 4);
      dir_name_size = readintbytesl(fdir, 2);
      dir_extr_size = readintbytesl(fdir, 2);
      dir_comm_size = readintbytesl(fdir, 2);
      dir_diskid = readintbytesl(fdir, 2);
      dir_attr = malloc_ec(6);
      assert(valid(dir_attr));
      i = 0;
      while (i < 6) {
        b = read_ec(fdir);
        assert(b >= 0);
        dir_attr[i] = b;
        i = i + 1;
      }
      dir_ent_start = readintbytesl(fdir, 4);
      assert(dir_sig == 33639248 && dir_extr_size == 24 && dir_comm_size == 0 &&
           dir_name_size > 0);
      dir_name = malloc_ec(dir_name_size);
      assert(valid(dir_name));
      i = 0;
      while (i < dir_name_size) {
        b = read_ec(fdir);
        assert(b >= 0);
        dir_name[i] = b;
        i = i + 1;
      }
      print(dir_name);
      print('\n');
      dir_extr = malloc_ec(dir_extr_size);
      assert(valid(dir_extr));
      i = 0;
      while (i < dir_extr_size) {
        b = read_ec(fdir);
        assert(b >= 0);
        dir_extr[i] = b;
        i = i + 1;
      }

      x = seek_ec(fent, dir_ent_start);
      assert(x >= 0);

      ent_sig = readintbytesl(fent, 4);
      ent_ver_extr = readintbytesl(fent, 2);
      ent_flag = readintbytesl(fent, 2);
      ent_comp = readintbytesl(fent, 2);
      ent_modif = readintbytesl(fent, 4);
      ent_crc = readintbytesl(fent, 4);
      ent_ent_size = readintbytesl(fent, 4);
      ent_ent_size_uncomp = readintbytesl(fent, 4);
      ent_name_size = readintbytesl(fent, 2);
      ent_extr_size = readintbytesl(fent, 2);

      assert(ent_sig == 67324752 && ent_ver_extr == dir_ver_extr && ent_flag ==
          dir_flag && ent_comp == dir_comp && ent_modif == dir_modif &&
          ent_name_size == dir_name_size && ent_name_size > 0);
      ent_name = malloc_ec(ent_name_size);
      assert(valid(ent_name));
      i = 0;
      while (i < ent_name_size) {
        b = read_ec(fent);
        assert(b >= 0);
        ent_name[i] = b;
```

```
      i = i + 1;
    }
    print(ent_name);
    print('\n');
    if (ent_extr_size > 0) {
      ent_extr = malloc_ec(ent_extr_size);
      assert(valid(ent_extr));
      i = 0;
      while (i < ent_extr_size) {
        b = read_ec(fent);
        assert(b >= 0);
        ent_extr[i] = b;
        i = i + 1;
      }
    }
    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);

    raw_data = malloc_ec(ent_ent_size);
    assert(valid(raw_data));
    i = 0;
    while (i < ent_ent_size) {
      b = read_ec(fent);
      assert(b >= 0);
      raw_data[i] = b;
      i = i + 1;
    }
    // decompress
    // check crc
    print(raw_data);
    print('\n');
    x = free_ec(ent_name);
    x = free_ec(raw_data);
  }
  x = seek_ec(fdir, eocd);
  return 0;
}
```

A.5.3 Explicit-recovery version

```
bad = 0;

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  if (x >= 0) {
    if (n == 1) {
      return x;
    } else {
      y = readintbytesl(f, n-1);
      if (bad) {
        return -1;
      }
      return (y << 8) + x;
    }
  } else {
    bad = 1;
    return -1;
  }
}

main {
  fdir = openb_ec("../inputs/zip/stuff-1.zip", 1);
  fent = openb_ec("../inputs/zip/stuff-1.zip", 1);
  if (!valid(fdir) || !valid(fent)) {
    exit(1);
  }

  eocd = size_ec(fdir) - 4;
  found = 0;
  while (!found) {
    x = seek_ec(fdir, eocd);
    if (x < 0) {
```

```
      exit(1);
    }
    dir_sig = readintbytesl(fdir, 4);
    if (dir_sig == 101010256) { // 0x06054b50
      found = 1;
    } else {
      eocd = eocd - 1;
    }
  }
  if (!found) {
    exit(1);
  }
  diskid = readintbytesl(fdir, 2);
  ndisk = readintbytesl(fdir, 2);
  dirid = readintbytesl(fdir, 2);
  ndir = readintbytesl(fdir, 2);

  dir_size = readintbytesl(fdir, 4);
  dir_start = readintbytesl(fdir, 4);
  eocd_comm_size = readintbytesl(fdir, 2);
  if (eocd_comm_size > 0) {
    eocd_comm = malloc_ec(eocd_comm_size);
    if (!valid(eocd_comm)) {
      exit(1);
    }
    i = 0;
    while (i < eocd_comm_size) {
      b = read_ec(fdir);
      if (b < 0) {
        exit(1);
      }
      eocd_comm[i] = b;
      i = i + 1;
    }
  }

  x = seek_ec(fdir, dir_start);
  if (x < 0) {
    exit(1);
  }
  lookatb (pos_ec(fdir) < eocd && pos_ec(fdir) >= 0, fdir, 28, 2, 40) {
    bad = 0;
    dir_sig = readintbytesl(fdir, 4);
    dir_ver_made = readintbytesl(fdir, 2);
    dir_ver_extr = readintbytesl(fdir, 2);
    dir_flag = readintbytesl(fdir, 2);
    dir_comp = readintbytesl(fdir, 2);
    dir_modif = readintbytesl(fdir, 4);
    dir_crc = readintbytesl(fdir, 4);
    dir_ent_size = readintbytesl(fdir, 4);
    dir_ent_size_uncomp = readintbytesl(fdir, 4);
    dir_name_size = readintbytesl(fdir, 2);
    dir_extr_size = readintbytesl(fdir, 2);
    dir_comm_size = readintbytesl(fdir, 2);
    dir_diskid = readintbytesl(fdir, 2);
    dir_attr = malloc_ec(6);
    if (valid(dir_attr)) {
      i = 0;
      while (i < 6) {
        b = read_ec(fdir);
        if (b < 0) {
          bad = 1;
        }
        dir_attr[i] = b;
        i = i + 1;
      }
      dir_ent_start = readintbytesl(fdir, 4);
      if (dir_sig == 33639248 && dir_extr_size == 24 && dir_comm_size == 0 &&
          dir_name_size > 0) {
        dir_name = malloc_ec(dir_name_size);
        if (valid(dir_name)) {
          i = 0;
          while (i < dir_name_size) {
            b = read_ec(fdir);
            if (b < 0) {
              bad = 1;
```

```
  }
  dir_name[i] = b;
  i = i + 1;
}
dir_extr = malloc_ec(dir_extr_size);
if (valid(dir_extr)) {
  i = 0;
  while (i < dir_extr_size) {
    b = read_ec(fdir);
    if (b < 0) {
      bad = 1;
    }
    dir_extr[i] = b;
    i = i + 1;
  }

  x = seek_ec(fent, dir_ent_start);
  if (x >= 0) {
    ent_sig = readintbytesl(fent, 4);
    ent_ver_extr = readintbytesl(fent, 2);
    ent_flag = readintbytesl(fent, 2);
    ent_comp = readintbytesl(fent, 2);
    ent_modif = readintbytesl(fent, 4);
    ent_crc = readintbytesl(fent, 4);
    ent_ent_size = readintbytesl(fent, 4);
    ent_ent_size_uncomp = readintbytesl(fent, 4);
    ent_name_size = readintbytesl(fent, 2);
    ent_extr_size = readintbytesl(fent, 2);

    print(dir_name);
    print('\n');
    if (ent_sig == 67324752 && ent_ver_extr == dir_ver_extr &&
        ent_flag == dir_flag && ent_comp == dir_comp && ent_modif ==
         dir_modif && ent_name_size == dir_name_size &&
        ent_name_size > 0) {
      ent_name = malloc_ec(ent_name_size);
      if (valid(ent_name)) {
        i = 0;
        while (i < ent_name_size) {
          b = read_ec(fent);
          if (b < 0) {
            bad = 1;
          }
          ent_name[i] = b;
          i = i + 1;
        }
        if (ent_extr_size > 0) {
          ent_extr = malloc_ec(ent_extr_size);
          if (valid(ent_extr)) {
            i = 0;
            while (i < ent_extr_size) {
              b = read_ec(fent);
              if (b < 0) {
                bad = 1;
              }
              ent_extr[i] = b;
              i = i + 1;
            }
          } else {
            bad = 1;
          }
        }

        if (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) {
          raw_data = malloc_ec(ent_ent_size);
          if (valid(raw_data)) {
            i = 0;
            while (i < ent_ent_size) {
              b = read_ec(fent);
              if (b < 0) {
                bad = 1;
              }
              raw_data[i] = b;
              i = i + 1;
```

```
                      }
                      // decompress
                      // check crc
                      if (!bad) {
                        print(ent_name);
                        print('\n');
                        print(raw_data);
                        print('\n');
                      } // skip unit
                      x = free_ec(ent_name);
                      x = free_ec(raw_data);
                    } // skip unit
                  } // skip unit
                } // skip unit
              } // skip unit
            } // skip unit
          } // skip unit
        } // skip unit
      } // skip unit
    } // skip unit
  }
  x = seek_ec(fdir, eocd);
  return 0;
}
```

A.5.4 Plain-loop version

```
bad = 0;

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  if (x >= 0) {
    if (n == 1) {
      return x;
    } else {
      y = readintbytesl(f, n-1);
      if (bad) {
        return -1;
      }
      return (y << 8) + x;
    }
  } else {
    bad = 1;
    return -1;
  }
}

main {
  fdir = openb_ec("../inputs/zip/stuff-1.zip", 1);
  fent = openb_ec("../inputs/zip/stuff-1.zip", 1);
  if (!valid(fdir) || !valid(fent)) {
    exit(1);
  }

  eocd = size_ec(fdir) - 4;
  found = 0;
  while (!found) {
    x = seek_ec(fdir, eocd);
    if (x < 0) {
      exit(1);
    }
    dir_sig = readintbytesl(fdir, 4);
    if (dir_sig == 101010256) { // 0x06054b50
      found = 1;
    } else {
      eocd = eocd - 1;
    }
  }
  if (!found) {
    exit(1);
  }
  diskid = readintbytesl(fdir, 2);
  ndisk = readintbytesl(fdir, 2);
```

```
  dirid = readintbytesl(fdir, 2);
  ndir = readintbytesl(fdir, 2);

  dir_size = readintbytesl(fdir, 4);
  dir_start = readintbytesl(fdir, 4);
  eocd_comm_size = readintbytesl(fdir, 2);
  if (eocd_comm_size > 0) {
    eocd_comm = malloc_ec(eocd_comm_size);
    if (!valid(eocd_comm)) {
      exit(1);
    }
    i = 0;
    while (i < eocd_comm_size) {
      b = read_ec(fdir);
      if (b < 0) {
        exit(1);
      }
      eocd_comm[i] = b;
      i = i + 1;
    }
  }

  x = seek_ec(fdir, dir_start);
  if (x < 0) {
    exit(1);
  }
  finish = 0;
  while (!finish && pos_ec(fdir) < eocd && pos_ec(fdir) >= 0) {
    bad = 0;
    dir_sig = readintbytesl(fdir, 4);
    dir_ver_made = readintbytesl(fdir, 2);
    dir_ver_extr = readintbytesl(fdir, 2);
    dir_flag = readintbytesl(fdir, 2);
    dir_comp = readintbytesl(fdir, 2);
    dir_modif = readintbytesl(fdir, 4);
    dir_crc = readintbytesl(fdir, 4);
    dir_ent_size = readintbytesl(fdir, 4);
    dir_ent_size_uncomp = readintbytesl(fdir, 4);
    dir_name_size = readintbytesl(fdir, 2);
    eou = pos_ec(fdir) + dir_name_size + 40;
    if (!bad && dir_name_size >= 0) {
      dir_extr_size = readintbytesl(fdir, 2);
      dir_comm_size = readintbytesl(fdir, 2);
      dir_diskid = readintbytesl(fdir, 2);
      dir_attr = malloc_ec(6);
      if (!bad && valid(dir_attr)) {
        i = 0;
        while (i < 6) {
          b = read_ec(fdir);
          if (b < 0) {
            bad = 1;
          }
          dir_attr[i] = b;
          i = i + 1;
        }
        dir_ent_start = readintbytesl(fdir, 4);
        if (!bad && dir_sig == 33639248 && dir_extr_size == 24 &&
            dir_comm_size == 0 && dir_name_size > 0) {
          dir_name = malloc_ec(dir_name_size);
          if (valid(dir_name)) {
            i = 0;
            while (i < dir_name_size) {
              b = read_ec(fdir);
              if (b < 0) {
                bad = 1;
              }
              dir_name[i] = b;
              i = i + 1;
            }
            dir_extr = malloc_ec(dir_extr_size);
            if (!bad && valid(dir_extr)) {
              i = 0;
              while (i < dir_extr_size) {
                b = read_ec(fdir);
                if (b < 0) {
                  bad = 1;
```

```
  }
  dir_extr[i] = b;
  i = i + 1;
}

x = seek_ec(fent, dir_ent_start);
if (x >= 0) {
  ent_sig = readintbytesl(fent, 4);
  ent_ver_extr = readintbytesl(fent, 2);
  ent_flag = readintbytesl(fent, 2);
  ent_comp = readintbytesl(fent, 2);
  ent_modif = readintbytesl(fent, 4);
  ent_crc = readintbytesl(fent, 4);
  ent_ent_size = readintbytesl(fent, 4);
  ent_ent_size_uncomp = readintbytesl(fent, 4);
  ent_name_size = readintbytesl(fent, 2);
  ent_extr_size = readintbytesl(fent, 2);

  print(dir_name);
  print('\n');
  if (ent_sig == 67324752 && ent_ver_extr == dir_ver_extr &&
      ent_flag == dir_flag && ent_comp == dir_comp && ent_modif
      == dir_modif && ent_name_size == dir_name_size &&
      ent_name_size > 0) {
    ent_name = malloc_ec(ent_name_size);
    if (valid(ent_name)) {
      i = 0;
      while (i < ent_name_size) {
        b = read_ec(fent);
        if (b < 0) {
          bad = 1;
        }
        ent_name[i] = b;
        i = i + 1;
      }
      if (ent_extr_size > 0) {
        ent_extr = malloc_ec(ent_extr_size);
        if (valid(ent_extr)) {
          i = 0;
          while (i < ent_extr_size) {
            b = read_ec(fent);
            if (b < 0) {
              bad = 1;
            }
            ent_extr[i] = b;
            i = i + 1;
          }
        } else {
          bad = 1;
        }
      }

      if (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) {
        raw_data = malloc_ec(ent_ent_size);
        if (valid(raw_data)) {
          i = 0;
          while (i < ent_ent_size) {
            b = read_ec(fent);
            if (b < 0) {
              bad = 1;
            }
            raw_data[i] = b;
            i = i + 1;
          }
          // decompress
          // check crc
          if (!bad) {
            print(ent_name);
            print('\n');
            print(raw_data);
            print('\n');
          } // skip unit
          x = free_ec(ent_name);
          x = free_ec(raw_data);
```

```
                        } // skip unit
                      } // skip unit
                    } // skip unit
                  } // skip unit
                } // skip unit
              } // skip unit
            } // skip unit
          } // skip unit
        } // skip unit
        x = seek_ec(fdir, eou);
        if (x < 0) { // give up
          finish = 1;
        }
      } else { // give up
        finish = 1;
      }
    }
    x = seek_ec(fdir, eocd);
    return 0;
}
```

A.6 RGIF

A.6.1 Fully-implicit version

```
buffer = malloc(7500);
number = 0;

func readintbytesl (f, n) { // little endian
  x = read(f);
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

func readblocks (f, echo) {
  i = 0;
  stop = 0;
  lookatb (!stop, f, 0, 1, 0) {
    len = read(f);
    if (echo) {
      print(len);
    }
    if (len == 0) {
      stop = 1;
    } else {
      while (!end(f)) {
        x = read(f);
        if (echo) {
          print(x);
        }
        buffer[i] = x;
        i = i + 1;
      }
    }
  }
  assert(stop);
  return i;
}

func printintbytesl (n) {
  b = number & 255;
  print(b);
  if (n == 1) {
    return 0;
  } else {
```

```
      number = number >> 8;
      dummy = printintbytesl(n - 1);
      return 0;
    }
}

main {
  f = openb("../inputs/rgif/welcome2-block.rgif", 1);

  // header block
  header = malloc(6);
  i = 0;
  while (i < 6) {
    x = read(f);
    print(x);
    header[i] = x;
    i = i + 1;
  }
  assert(header[0] == 'G' && header[1] == 'I' && header[2] == 'F' && header[3]
       == '8' && (header[4] == '9' || header[4] == '7') && header[5] == 'a');
  free(header);

  // logical screen descriptor
  canvasw = readintbytesl(f, 2);
  number = canvasw;
  dummy = printintbytesl(2);
  canvash = readintbytesl(f, 2);
  number = canvash;
  dummy = printintbytesl(2);
  x = read(f);
  print(x);
  gflag = (x & 128) >> 7;
  bpp = (x & 112) >> 4;
  gsort = (x & 8) >> 3;
  background = read(f);
  print(background);
  aspect = read(f);
  print(aspect);
  assert(gflag && x & 7 == bpp);

  // global color table
  nglobal = 2 << bpp;
  gcolors = malloc(nglobal);
  i = 0;
  while (i < nglobal) {
    x = readintbytesl(f, 3);
    number = x;
    dummy = printintbytesl(3);
    gcolors[i] = x;
    i = i + 1;
  }

  trailer = 0;
  inspectb (!trailer, f, 0, 2, 1) {
    length = readintbytesl(f, 2);
    x = read(f);
    if (x == 59) { // trailer block
      print(x);
      trailer = 1;
    } else {
      label = read(f);
      if (x == 33 && (label == 255 || label == 254)) {
        if (label == 255) { // appliation extension
          idlen = read(f);
          id = malloc(idlen);
          i = 0;
          while (i < idlen) {
            x = read(f);
            id[i] = x;
            i = i + 1;
          }
          free(id);
          applen = readblocks(f, 0);
          app = malloc(applen);
          i = 0;
          while (i < applen) {
```

```
      app[i] = buffer[i];
      i = i + 1;
    }
  } else { // comment extension
    commentlen = readblocks(f, 0);
    comments = malloc(commentlen);
    i = 0;
    while (i < commentlen) {
      comments[i] = buffer[i];
      i = i + 1;
    }
    free(comments);
  }
} else {
  if (x == 33 && label == 249) { // graphics control extension
    print(x);
    print(label);
    blocksize = read(f);
    print(blocksize);
    ctrlext = read(f);
    print(ctrlext);
    delay = readintbytesl(f, 2);
    number = delay;
    dummy = printintbytesl(2);
    transparent = read(f);
    print(transparent);
    terminator = read(f);
    print(terminator);
    assert(terminator == 0);
  } else {
    assert(x != 33);
    seek(f, pos(f)-2);
  }

  // image descriptor
  x = read(f);
  assert(x == 44);
  print(x);
  left = readintbytesl(f, 2);
  number = left;
  dummy = printintbytesl(2);
  top = readintbytesl(f, 2);
  number = top;
  dummy = printintbytesl(2);
  width = readintbytesl(f, 2);
  number = width;
  dummy = printintbytesl(2);
  height = readintbytesl(f, 2);
  number = height;
  dummy = printintbytesl(2);
  assert(left == 0 && top == 0 && width == canvasw && height == canvash);
  x = read(f);
  print(x);
  lflag = (x & 128) >> 7;
  linterlace = (x & 64) >> 6;
  lsort = (x & 32) >> 5;
  lctsize = x & 7;

  // local color table
  if (lflag) {
    nlocal = 2 << bpp;
    lcolors = malloc(nlocal);
    i = 0;
    while (i < nlocal) {
      x = readintbytesl(f, 3);
      number = x;
      dummy = printintbytesl(3);
      lcolors[i] = x;
      i = i + 1;
    }
  }

  // data sub-blocks
  lzw = read(f);
  print(lzw);
```

```
        datalen = readblocks(f, 1);
        // decompress
        // decode
      }
    }
  }
  free(buffer);
  if (!trailer) {
    x = 59;
    print(x);
  }
  return 0;
}
```

A.6.2 Protective-check version

```
buffer = malloc_ec(7500);
number = 0;

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  assert(x >= 0);
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

func readblocks (f, echo) {
  i = 0;
  stop = 0;
  lookatb (!stop, f, 0, 1, 0) {
    len = read_ec(f);
    assert(len >= 0);
    if (echo) {
      print(len);
    }
    if (len == 0) {
      stop = 1;
    } else {
      while (!end_ec(f)) {
        x = read_ec(f);
        assert(x >= 0 && i < 7500);
        if (echo) {
          print(x);
        }
        buffer[i] = x;
        i = i + 1;
      }
    }
  }
  assert(stop);
  return i;
}

func printintbytesl (n) {
  b = number & 255;
  print(b);
  if (n == 1) {
    return 0;
  } else {
    number = number >> 8;
    dummy = printintbytesl(n - 1);
    return 0;
  }
}

main {
  f = openb_ec("../inputs/rgif/welcome2-block.rgif", 1);
  assert(valid(buffer) && valid(f));

```

```
// header block
header = malloc_ec(6);
assert(valid(header));
i = 0;
while (i < 6) {
  x = read_ec(f);
  assert(x >= 0);
  print(x);
  header[i] = x;
  i = i + 1;
}
assert(header[0] == 'G' && header[1] == 'I' && header[2] == 'F' && header[3]
     == '8' && (header[4] == '9' || header[4] == '7') && header[5] == 'a');
x = free_ec(header);

// logical screen descriptor
canvasw = readintbytesl(f, 2);
number = canvasw;
dummy = printintbytesl(2);
canvash = readintbytesl(f, 2);
number = canvash;
dummy = printintbytesl(2);
x = read_ec(f);
assert(x >= 0);
print(x);
gflag = (x & 128) >> 7;
bpp = (x & 112) >> 4;
gsort = (x & 8) >> 3;
background = read_ec(f);
print(background);
aspect = read_ec(f);
print(aspect);
assert(gflag && x & 7 == bpp && background >= 0 && aspect >= 0);

// global color table
nglobal = 2 << bpp;
gcolors = malloc_ec(nglobal);
assert(valid(gcolors));
i = 0;
while (i < nglobal) {
  x = readintbytesl(f, 3);
  number = x;
  dummy = printintbytesl(3);
  gcolors[i] = x;
  i = i + 1;
}

trailer = 0;
inspectb (!trailer, f, 0, 2, 1) {
  length = readintbytesl(f, 2);
  x = read_ec(f);
  assert(x >= 0);
  if (x == 59) { // trailer block
    print(x);
    trailer = 1;
  } else {
    label = read_ec(f);
    assert(label >= 0);
    if (x == 33 && (label == 255 || label == 254)) {
      if (label == 255) { // appliation extension
        idlen = read_ec(f);
        assert(idlen >= 0);
        id = malloc_ec(idlen);
        assert(valid(id));
        i = 0;
        while (i < idlen) {
          x = read_ec(f);
          assert(x >= 0);
          id[i] = x;
          i = i + 1;
        }
        x = free_ec(id);
        applen = readblocks(f, 0);
        app = malloc_ec(applen);
        assert(valid(app));
        i = 0;
```

```
          while (i < applen) {
            app[i] = buffer[i];
            i = i + 1;
          }
        } else { // comment extension
          commentlen = readblocks(f, 0);
          comments = malloc_ec(commentlen);
          assert(valid(comments));
          i = 0;
          while (i < commentlen) {
            comments[i] = buffer[i];
            i = i + 1;
          }
          x = free_ec(comments);
        }
      } else {
        if (x == 33 && label == 249) { // graphics control extension
          print(x);
          print(label);
          blocksize = read_ec(f);
          print(blocksize);
          ctrlext = read_ec(f);
          print(ctrlext);
          delay = readintbytesl(f, 2);
          number = delay;
          dummy = printintbytesl(2);
          transparent = read_ec(f);
          print(transparent);
          terminator = read_ec(f);
          print(terminator);
          assert(blocksize >= 0 && ctrlext >= 0 && transparent >= 0 &&
              terminator == 0);
        } else {
          assert(x != 33);
          x = seek_ec(f, pos_ec(f)-2);
          assert(x >= 0);
        }

        // image descriptor
        x = read_ec(f);
        assert(x == 44);
        print(x);
        left = readintbytesl(f, 2);
        number = left;
        dummy = printintbytesl(2);
        top = readintbytesl(f, 2);
        number = top;
        dummy = printintbytesl(2);
        width = readintbytesl(f, 2);
        number = width;
        dummy = printintbytesl(2);
        height = readintbytesl(f, 2);
        number = height;
        dummy = printintbytesl(2);
        assert(left == 0 && top == 0 && width == canvasw && height == canvash)
            ;
        x = read_ec(f);
        assert(x >= 0);
        print(x);
        lflag = (x & 128) >> 7;
        linterlace = (x & 64) >> 6;
        lsort = (x & 32) >> 5;
        lctsize = x & 7;

        // local color table
        if (lflag) {
          nlocal = 2 << bpp;
          lcolors = malloc_ec(nlocal);
          assert(valid(lcolors));
          i = 0;
          while (i < nlocal) {
            x = readintbytesl(f, 3);
            number = x;
            dummy = printintbytesl(3);
            lcolors[i] = x;
            i = i + 1;
```

```
          }
        }

        // data sub-blocks
        lzw = read_ec(f);
        assert(lzw >= 0);
        print(lzw);
        datalen = readblocks(f, 1);
        // decompress
        // decode
      }
    }
  }
  x = free_ec(buffer);
  if (!trailer) {
    x = 59;
    print(x);
  }
  return 0;
}
```

A.6.3 Explicit-recovery version

```
buffer = malloc_ec(7500);
number = 0;
prt = 1;

bad = 0;
out = malloc_ec(7500);
outidx = 0;

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    if (bad) {
      return -1;
    }
    return (y << 8) + x;
  }
}

func readblocks (f, echo) {
  i = 0;
  stop = 0;
  lookatb (!stop, f, 0, 1, 0) {
    len = read_ec(f);
    if (len >= 0) {
      if (echo) {
        dummy = write(len);
      }
      if (len == 0) {
        stop = 1;
      } else {
        while (!end_ec(f) && !bad) {
          x = read_ec(f);
          if (x >= 0 && i < 7500) {
            if (echo) {
              dummy = write(x);
            }
            buffer[i] = x;
            i = i + 1;
          } else {
            bad = 1;
          }
        }
      }
```

```
    } else {
      bad = 1;
    }
  }
  if (!stop) {
    bad = 1;
  }
  return i;
}

func write (x) {
  if (outidx < 7500) {
    out[outidx] = x;
    outidx = outidx + 1;
    return 0;
  } else {
    bad = 1;
    return -1;
  }
}

func writeintbytesl (n) {
  b = number & 255;
  if (prt) {
    print(b);
  } else {
    dummy = write(b);
  }
  if (n == 1) {
    return 0;
  } else {
    number = number >> 8;
    dummy = writeintbytesl(n - 1);
    return 0;
  }
}

main {
  f = openb_ec("../inputs/rgif/welcome2-block.rgif", 1);
  if (!valid(buffer) || !valid(out) || !valid(f)) {
    exit(1);
  }

  // header block
  header = malloc_ec(6);
  if (!valid(header)) {
    exit(1);
  }
  i = 0;
  while (i < 6) {
    x = read_ec(f);
    if (x < 0) {
      exit(1);
    }
    print(x);
    header[i] = x;
    i = i + 1;
  }
  if (!(header[0] == 'G' && header[1] == 'I' && header[2] == 'F' && header[3] == '8' && (header[4] == '9' || header[4] == '7') && header[5] == 'a')) {
    exit(1);
  }
  x = free_ec(header);

  // logical screen descriptor
  canvasw = readintbytesl(f, 2);
  number = canvasw;
  dummy = writeintbytesl(2);
  canvash = readintbytesl(f, 2);
  number = canvash;
  dummy = writeintbytesl(2);
  x = read_ec(f);
  if (bad || x < 0) {
    exit(1);
  }
  print(x);
```

```
  gflag = (x & 128) >> 7;
  bpp = (x & 112) >> 4;
  gsort = (x & 8) >> 3;
  background = read_ec(f);
  print(background);
  aspect = read_ec(f);
  print(aspect);
  if (!gflag || x & 7 != bpp || background < 0 || aspect < 0) {
    exit(1);
  }

  // global color table
  nglobal = 2 << bpp;
  gcolors = malloc_ec(nglobal);
  if (!valid(gcolors)) {
    exit(1);
  }
  i = 0;
  while (i < nglobal) {
    x = readintbytesl(f, 3);
    if (bad) {
      exit(1);
    }
    number = x;
    dummy = writeintbytesl(3);
    gcolors[i] = x;
    i = i + 1;
  }

  prt = 0;
  trailer = 0;
  lookatb (!trailer, f, 0, 2, 1) {
    bad = 0;
    outidx = 0;
    length = readintbytesl(f, 2);
    x = read_ec(f);
    if (x >= 0) {
      if (x == 59) { // trailer block
        dummy = write(x);
        trailer = 1;
      } else {
        label = read_ec(f);
        if (label < 0) {
          bad = 1;
        }
        if (x == 33 && (label == 255 || label == 254)) {
          if (label == 255) { // appliation extension
            idlen = read_ec(f);
            if (idlen < 0) {
              bad = 1;
            }
            id = malloc_ec(idlen);
            if (valid(id)) {
              i = 0;
              while (i < idlen) {
                x = read_ec(f);
                if (x < 0) {
                  bad = 1;
                }
                id[i] = x;
                i = i + 1;
              }
              x = free_ec(id);
              applen = readblocks(f, 0);
              app = malloc_ec(applen);
              if (valid(app)) {
                i = 0;
                while (i < applen) {
                  app[i] = buffer[i];
                  i = i + 1;
                }
              } else {
                bad = 1;
              }
            } else {
              bad = 1;
```

```
          }
        } else { // comment extension
          commentlen = readblocks(f, 0);
          comments = malloc_ec(commentlen);
          if (valid(comments)) {
            i = 0;
            while (i < commentlen) {
              comments[i] = buffer[i];
              i = i + 1;
            }
            x = free_ec(comments);
          } else {
            bad = 1;
          }
        }
      } else {
        if (x == 33 && label == 249) { // graphics control extension
          dummy = write(x);
          dummy = write(label);
          blocksize = read_ec(f);
          dummy = write(blocksize);
          ctrlext = read_ec(f);
          dummy = write(ctrlext);
          delay = readintbytesl(f, 2);
          number = delay;
          dummy = writeintbytesl(2);
          transparent = read_ec(f);
          dummy = write(transparent);
          terminator = read_ec(f);
          dummy = write(terminator);
          if (blocksize < 0 || ctrlext < 0 || transparent < 0 || terminator != 0) {
            bad = 1;
          }
        } else {
          if (x == 33) {
            bad = 1;
          }
          x = seek_ec(f, pos_ec(f)-2);
          if (x < 0) {
            bad = 1;
          }
        }

        // image descriptor
        x = read_ec(f);
        if (x != 44) {
          bad = 1;
        }
        dummy = write(x);
        left = readintbytesl(f, 2);
        number = left;
        dummy = writeintbytesl(2);
        top = readintbytesl(f, 2);
        number = top;
        dummy = writeintbytesl(2);
        width = readintbytesl(f, 2);
        number = width;
        dummy = writeintbytesl(2);
        height = readintbytesl(f, 2);
        number = height;
        dummy = writeintbytesl(2);
        if (left != 0 || top != 0 || width != canvasw || height != canvash) {
          bad = 1;
        }
        x = read_ec(f);
        if (x < 0) {
          bad = 1;
        }
        dummy = write(x);
        lflag = (x & 128) >> 7;
        linterlace = (x & 64) >> 6;
        lsort = (x & 32) >> 5;
        lctsize = x & 7;

```

```
          // local color table
          if (lflag) {
            nlocal = 2 << bpp;
            lcolors = malloc_ec(nlocal);
            if (valid(lcolors)) {
              i = 0;
              while (i < nlocal) {
                x = readintbytesl(f, 3);
                number = x;
                dummy = writeintbytesl(3);
                lcolors[i] = x;
                i = i + 1;
              }
            } else {
              bad = 1;
            }
          }

          // data sub-blocks
          lzw = read_ec(f);
          if (lzw < 0) {
            bad = 1;
          }
          dummy = write(lzw);
          datalen = readblocks(f, 1);
          // decompress
          // decode
        }
      }
    } else {
      bad = 1;
    }
    if (!bad) {
      i = 0;
      while (i < outidx) {
        x = out[i];
        print(x);
        i = i + 1;
      }
    } // skip unit
  }
  x = free_ec(buffer);
  if (!trailer) {
    x = 59;
    print(x);
  }
  return 0;
}
```

A.6.4 Plain-loop version

```
buffer = malloc_ec(7500);
number = 0;
prt = 1;

bad = 0;
out = malloc_ec(7500);
outidx = 0;

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    if (bad) {
      return -1;
    }
    return (y << 8) + x;
```

```
  }
}

func readblocks (f, echo) {
  i = 0;
  stop = 0;
  while (!stop) {
    len = read_ec(f);
    if (len >= 0) {
      if (echo) {
        dummy = write(len);
      }
      if (len == 0) {
        stop = 1;
      } else {
        j = 0;
        while (j < len) {
          x = read_ec(f);
          if (x >= 0 && i < 7500) {
            if (echo) {
              dummy = write(x);
            }
            buffer[i] = x;
            i = i + 1;
          } else {
            bad = 1;
          }
          j = j + 1;
        }
      }
    } else {
      bad = 1;
    }
  }
  return i;
}

func write (x) {
  if (outidx < 7500) {
    out[outidx] = x;
    outidx = outidx + 1;
    return 0;
  } else {
    bad = 1;
    return -1;
  }
}

func writeintbytesl (n) {
  b = number & 255;
  if (prt) {
    print(b);
  } else {
    dummy = write(b);
  }
  if (n == 1) {
    return 0;
  } else {
    number = number >> 8;
    dummy = writeintbytesl(n - 1);
    return 0;
  }
}

main {
  f = openb_ec("../inputs/rgif/welcome2-block.rgif", 1);
  if (!valid(buffer) || !valid(out) || !valid(f)) {
    exit(1);
  }

  // header block
  header = malloc_ec(6);
  if (!valid(header)) {
    exit(1);
  }
  i = 0;
```

```
  while (i < 6) {
    x = read_ec(f);
    if (x < 0) {
      exit(1);
    }
    print(x);
    header[i] = x;
    i = i + 1;
  }
  if (!(header[0] == 'G' && header[1] == 'I' && header[2] == 'F' && header[3]
      == '8' && (header[4] == '9' || header[4] == '7') && header[5] == 'a')) {
    exit(1);
  }
  x = free_ec(header);

  // logical screen descriptor
  canvasw = readintbytesl(f, 2);
  number = canvasw;
  dummy = writeintbytesl(2);
  canvash = readintbytesl(f, 2);
  number = canvash;
  dummy = writeintbytesl(2);
  x = read_ec(f);
  if (bad || x < 0) {
    exit(1);
  }
  print(x);
  gflag = (x & 128) >> 7;
  bpp = (x & 112) >> 4;
  gsort = (x & 8) >> 3;
  background = read_ec(f);
  print(background);
  aspect = read_ec(f);
  print(aspect);
  if (!gflag || x & 7 != bpp || background < 0 || aspect < 0) {
    exit(1);
  }

  // global color table
  nglobal = 2 << bpp;
  gcolors = malloc_ec(nglobal);
  if (!valid(gcolors)) {
    exit(1);
  }
  i = 0;
  while (i < nglobal) {
    x = readintbytesl(f, 3);
    if (bad) {
      exit(1);
    }
    number = x;
    dummy = writeintbytesl(3);
    gcolors[i] = x;
    i = i + 1;
  }

  prt = 0;
  trailer = 0;
  finish = 0;
  while (!trailer && !finish) {
    bad = 0;
    outidx = 0;
    length = readintbytesl(f, 2);
    if (length >= 0) {
      eou = pos_ec(f) + length + 1;
      x = read_ec(f);
      if (x >= 0) {
        if (x == 59) { // trailer block
          dummy = write(x);
          trailer = 1;
        } else {
          label = read_ec(f);
          if (label < 0) {
            bad = 1;
          }
          if (x == 33 && (label == 255 || label == 254)) {
```

```
  if (label == 255) { // appliation extension
    idlen = read_ec(f);
    if (idlen < 0) {
      bad = 1;
    }
    id = malloc_ec(idlen);
    if (valid(id)) {
      i = 0;
      while (i < idlen) {
        x = read_ec(f);
        if (x < 0) {
          bad = 1;
        }
        id[i] = x;
        i = i + 1;
      }
      x = free_ec(id);
      applen = readblocks(f, 0);
      app = malloc_ec(applen);
      if (valid(app)) {
        i = 0;
        while (i < applen) {
          app[i] = buffer[i];
          i = i + 1;
        }
      } else {
        bad = 1;
      }
    } else {
      bad = 1;
    }
  } else { // comment extension
    commentlen = readblocks(f, 0);
    comments = malloc_ec(commentlen);
    if (valid(comments)) {
      i = 0;
      while (i < commentlen) {
        comments[i] = buffer[i];
        i = i + 1;
      }
      x = free_ec(comments);
    } else {
      bad = 1;
    }
  }
} else {
  if (x == 33 && label == 249) { // graphics control extension
    dummy = write(x);
    dummy = write(label);
    blocksize = read_ec(f);
    dummy = write(blocksize);
    ctrlext = read_ec(f);
    dummy = write(ctrlext);
    delay = readintbytesl(f, 2);
    number = delay;
    dummy = writeintbytesl(2);
    transparent = read_ec(f);
    dummy = write(transparent);
    terminator = read_ec(f);
    dummy = write(terminator);
    if (blocksize < 0 || ctrlext < 0 || transparent < 0 ||
        terminator != 0) {
      bad = 1;
    }
  } else {
    if (x == 33) {
      bad = 1;
    }
    x = seek_ec(f, pos_ec(f)-2);
    if (x < 0) {
      bad = 1;
    }
  }

  // image descriptor
  x = read_ec(f);
```

```
      if (x != 44) {
        bad = 1;
      }
      dummy = write(x);
      left = readintbytesl(f, 2);
      number = left;
      dummy = writeintbytesl(2);
      top = readintbytesl(f, 2);
      number = top;
      dummy = writeintbytesl(2);
      width = readintbytesl(f, 2);
      number = width;
      dummy = writeintbytesl(2);
      height = readintbytesl(f, 2);
      number = height;
      dummy = writeintbytesl(2);
      if (left != 0 || top != 0 || width != canvasw || height != canvash ) {
        bad = 1;
      }
      x = read_ec(f);
      if (x < 0) {
        bad = 1;
      }
      dummy = write(x);
      lflag = (x & 128) >> 7;
      linterlace = (x & 64) >> 6;
      lsort = (x & 32) >> 5;
      lctsize = x & 7;

      // local color table
      if (lflag) {
        nlocal = 2 << bpp;
        lcolors = malloc_ec(nlocal);
        if (valid(lcolors)) {
          i = 0;
          while (i < nlocal) {
            x = readintbytesl(f, 3);
            number = x;
            dummy = writeintbytesl(3);
            lcolors[i] = x;
            i = i + 1;
          }
        } else {
          bad = 1;
        }
      }

      // data sub-blocks
      lzw = read_ec(f);
      if (lzw < 0) {
        bad = 1;
      }
      dummy = write(lzw);
      datalen = readblocks(f, 1);
      // decompress
      // decode
    }
  }
} else {
  bad = 1;
}
if (pos_ec(f) > eou || pos_ec(f) < 0) {
  bad = 1;
}
x = seek_ec(f, eou);
if (x < 0) { // give up
  finish = 1;
}
if (!bad && !finish) {
  i = 0;
  while (i < outidx) {
    x = out[i];
    print(x);
    i = i + 1;
  }
```

```
      } // skip unit
    } else { // give up
      finish = 1;
    }
  }
  x = free_ec(buffer);
  if (!trailer) {
    x = 59;
    print(x);
  }
  return 0;
}
```

A.7 PCAP/DNS

A.7.1 Fully-implicit version

```
dst_mac = malloc(6);
src_mac = malloc(6);
data = malloc(1);

func readintbytesl (f, n) { // little endian
  x = read(f);
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

func readintbytesb (f, n) { // big endian
  x = 0;
  i = 0;
  while (i < n) {
    byte = read(f);
    x = x << 8;
    x = x | byte;
    i = i + 1;
  }
  return x;
}

func parse_ethernet (f, dummy) {
  i = 0;
  while (i < 6) {
    x = read(f);
    dst_mac[i] = x;
    i = i + 1;
  }
  i = 0;
  while (i < 6) {
    x = read(f);
    src_mac[i] = x;
    i = i + 1;
  }

  ether_type = readintbytesl(f, 2);
  assert(ether_type == 8); // IPv4

  dummy = parse_ipv4(f, 0);
  return 0;
}

func check_sum (f, len) {
  start = pos(f);
  assert(len > 0 && len % 2 == 0);
  sum = 0;
  i = 0;
```

```
  while (i < len / 2) {
    x = readintbytesb(f, 2);
    sum = sum + x;
    i = i + 1;
  }
  hi = (sum >> 16) & 65535;
  lo = sum & 65535;
  assert(hi + lo == 65535);
  seek(f, start);
  return 0;
}

func print_ip (addr) {
  b = (addr >> 24) & 255;
  print(b);
  print('.');
  b = (addr >> 16) & 255;
  print(b);
  print('.');
  b = (addr >> 8) & 255;
  print(b);
  print('.');
  b = addr & 255;
  print(b);
  return 0;
}

func parse_ipv4 (f, dummy) {
  dummy = check_sum(f, 20); // no options

  x = read(f);
  version = (x & 240) >> 4;
  hdr_size = x & 15;
  x = read(f);
  total = readintbytesb(f, 2);

  id = readintbytesb(f, 2);
  fragment = readintbytesb(f, 2);
  dont_frag = (fragment & 16384) >> 14;
  more_frag = (fragment & 8192) >> 13;
  offs_frag = fragment & 8191;

  ttl = read(f);
  ip_type = read(f);
  checksum = readintbytesb(f, 2);

  src_ip = readintbytesb(f, 4);
  dst_ip = readintbytesb(f, 4);

  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

  print('\n');
  dummy = print_ip(src_ip);
  print(' ');
  dummy = print_ip(dst_ip);
  print(' ');
  dummy = parse_udp(f, total - hdr_size * 4);

  return 0;
}

func parse_udp (f, pkt_size) {
  src_port = readintbytesb(f, 2);
  dst_port = readintbytesb(f, 2);
  udp_size = readintbytesb(f, 2);
  checksum = readintbytesb(f, 2);

  assert(pkt_size <= udp_size && pkt_size > 8);
  if (src_port == 53 || dst_port == 53) { // DNS
    dummy = parse_dns(f, pkt_size - 8);
  } else { // others
    if (valid(data)) {
      free(data);
    }
```

```
      data_size = pkt_size - 8;
      data = malloc(data_size);
      i = 0;
      while (i < data_size) {
        x = read(f);
        data[i] = x;
        i = i + 1;
      }
      print('\n');
    }
    return 0;
}

func parse_dns (f, data_size) {
    id = readintbytesb(f, 2);
    print(id);
    print('\n');
    flags = readintbytesb(f, 2);
    n_q = readintbytesb(f, 2);
    n_ans = readintbytesb(f, 2);
    n_auth = readintbytesb(f, 2);
    n_add = readintbytesb(f, 2);

    // questions
    i = 0;
    while (i < n_q) {
      stop = 0;
      first = 1;
      while (!stop) {
        len = read(f);
        if (len == 0) {
          stop = 1;
        } else {
          if (first) {
            first = 0;
          } else {
            print('.');
          }
          j = 0;
          while (j < len) {
            x = read(f);
            print(x);
            j = j + 1;
          }
        }
      }
      print('\n');
      type_q = readintbytesb(f, 2);
      class_q = readintbytesb(f, 2);
      i = i + 1;
    }

    // answers
    i = 0;
    while (i < n_ans) {
      ptr = readintbytesb(f, 2);
      assert(ptr == 49164);
      type_ans = readintbytesb(f, 2);
      class_ans = readintbytesb(f, 2);
      ttl = readintbytesb(f, 4);
      len = readintbytesb(f, 2);
      if (type_ans == 1) {
        assert(len == 4);
        addr = readintbytesb(f, 4);
        dummy = print_ip(addr);
      } else {
        assert(type_ans == 5);
        j = 0;
        while (j < len) {
          x = read(f);
          print(x);
          j = j + 1;
        }
      }
      print('\n');
      i = i + 1;
```

```
  }

  // authority
  // additional
  return 0;
}

main {
  f = openb("../inputs/pcap/bad.pcap", 1);

  x = read(f);
  assert(x == 212);
  x = read(f);
  assert(x == 195);
  x = read(f);
  assert(x == 178);
  x = read(f);
  assert(x == 161);
  maj_ver = readintbytesl(f, 2);
  min_ver = readintbytesl(f, 2);
  this_zone = readintbytesl(f, 4);
  sigfigs = readintbytesl(f, 4);
  snap_len = readintbytesl(f, 4); // number of packets
  link_type = readintbytesl(f, 4);
  assert(link_type == 1); // ethernet

  inspectb (1, f, 12, 4, 0) {
    ts_epoch = readintbytesl(f, 4);
    ts_nanosec = readintbytesl(f, 4);
    caplen = readintbytesl(f, 4);
    length = readintbytesl(f, 4);
    assert(caplen == length);

    dummy = parse_ethernet(f, 0);
  }

  return 0;
}
```

A.7.2 Protective-check version

```
dst_mac = malloc_ec(6);
src_mac = malloc_ec(6);
data = malloc_ec(1);

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  assert(x >= 0);
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

func readintbytesb (f, n) { // big endian
  x = 0;
  i = 0;
  while (i < n) {
    byte = read_ec(f);
    assert(byte >= 0);
    x = x << 8;
    x = x | byte;
    i = i + 1;
  }
  return x;
}

func parse_ethernet (f, dummy) {
  i = 0;
  while (i < 6) {
    x = read_ec(f);
```

```
    assert(x >= 0);
    dst_mac[i] = x;
    i = i + 1;
  }
  i = 0;
  while (i < 6) {
    x = read_ec(f);
    assert(x >= 0);
    src_mac[i] = x;
    i = i + 1;
  }

  ether_type = readintbytesl(f, 2);
  assert(ether_type == 8); // IPv4

  dummy = parse_ipv4(f, 0);
  return 0;
}

func check_sum (f, len) {
  start = pos_ec(f);
  assert(len > 0 && len % 2 == 0 && start >= 0);
  sum = 0;
  i = 0;
  while (i < len / 2) {
    x = readintbytesb(f, 2);
    sum = sum + x;
    i = i + 1;
  }
  hi = (sum >> 16) & 65535;
  lo = sum & 65535;
  x = seek_ec(f, start);
  assert(hi + lo == 65535 && x >= 0);
  return 0;
}

func print_ip (addr) {
  b = (addr >> 24) & 255;
  print(b);
  print('.');
  b = (addr >> 16) & 255;
  print(b);
  print('.');
  b = (addr >> 8) & 255;
  print(b);
  print('.');
  b = addr & 255;
  print(b);
  return 0;
}

func parse_ipv4 (f, dummy) {
  dummy = check_sum(f, 20); // no options

  x = read_ec(f);
  assert(x >= 0);
  version = (x & 240) >> 4;
  hdr_size = x & 15;
  x = read_ec(f);
  assert(x >= 0);
  total = readintbytesb(f, 2);

  id = readintbytesb(f, 2);
  fragment = readintbytesb(f, 2);
  dont_frag = (fragment & 16384) >> 14;
  more_frag = (fragment & 8192) >> 13;
  offs_frag = fragment & 8191;

  ttl = read_ec(f);
  ip_type = read_ec(f);
  checksum = readintbytesb(f, 2);

  src_ip = readintbytesb(f, 4);
  dst_ip = readintbytesb(f, 4);

```

```
  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

  print('\n');
  dummy = print_ip(src_ip);
  print(' ');
  dummy = print_ip(dst_ip);
  print(' ');
  dummy = parse_udp(f, total - hdr_size * 4);

  return 0;
}

func parse_udp (f, pkt_size) {
  src_port = readintbytesb(f, 2);
  dst_port = readintbytesb(f, 2);
  udp_size = readintbytesb(f, 2);
  checksum = readintbytesb(f, 2);

  assert(pkt_size <= udp_size && pkt_size > 8);
  if (src_port == 53 || dst_port == 53) { // DNS
    dummy = parse_dns(f, pkt_size - 8);
  } else { // others
    if (valid(data)) {
      dummy = free_ec(data);
    }
    data_size = pkt_size - 8;
    data = malloc_ec(data_size);
    assert(valid(data));
    i = 0;
    while (i < data_size) {
      x = read_ec(f);
      assert(x >= 0);
      data[i] = x;
      i = i + 1;
    }
    print('\n');
  }
  return 0;
}

func parse_dns (f, data_size) {
  id = readintbytesb(f, 2);
  print(id);
  print('\n');
  flags = readintbytesb(f, 2);
  n_q = readintbytesb(f, 2);
  n_ans = readintbytesb(f, 2);
  n_auth = readintbytesb(f, 2);
  n_add = readintbytesb(f, 2);

  // questions
  i = 0;
  while (i < n_q) {
    stop = 0;
    first = 1;
    while (!stop) {
      len = read_ec(f);
      assert(len >= 0);
      if (len == 0) {
        stop = 1;
      } else {
        if (first) {
          first = 0;
        } else {
          print('.');
        }
        j = 0;
        while (j < len) {
          x = read_ec(f);
          assert(x >= 0);
          print(x);
          j = j + 1;
        }
```

```
        }
      }
      print('\n');
      type_q = readintbytesb(f, 2);
      class_q = readintbytesb(f, 2);
      i = i + 1;
    }

    // answers
    i = 0;
    while (i < n_ans) {
      ptr = readintbytesb(f, 2);
      assert(ptr == 49164);
      type_ans = readintbytesb(f, 2);
      class_ans = readintbytesb(f, 2);
      ttl = readintbytesb(f, 4);
      len = readintbytesb(f, 2);
      if (type_ans == 1) {
        assert(len == 4);
        addr = readintbytesb(f, 4);
        dummy = print_ip(addr);
      } else {
        assert(type_ans == 5 && len > 0);
        j = 0;
        while (j < len) {
          x = read_ec(f);
          assert(x >= 0);
          print(x);
          j = j + 1;
        }
      }
      print('\n');
      i = i + 1;
    }

    // authority
    // additional
    return 0;
  }

  main {
    assert(valid(dst_mac) && valid(src_mac) && valid(data));
    f = openb_ec("../inputs/pcap/bad.pcap", 1);
    assert(valid(f));

    x = read_ec(f);
    assert(x == 212);
    x = read_ec(f);
    assert(x == 195);
    x = read_ec(f);
    assert(x == 178);
    x = read_ec(f);
    assert(x == 161);
    maj_ver = readintbytesl(f, 2);
    min_ver = readintbytesl(f, 2);
    this_zone = readintbytesl(f, 4);
    sigfigs = readintbytesl(f, 4);
    snap_len = readintbytesl(f, 4); // number of packets
    link_type = readintbytesl(f, 4);
    assert(link_type == 1); // ethernet

    inspectb (1, f, 12, 4, 0) {
      ts_epoch = readintbytesl(f, 4);
      ts_nanosec = readintbytesl(f, 4);
      caplen = readintbytesl(f, 4);
      length = readintbytesl(f, 4);
      assert(caplen == length);

      dummy = parse_ethernet(f, 0);
    }

    return 0;
  }
```

A.7.3 Explicit-recovery version

```
dst_mac = malloc_ec(6);
src_mac = malloc_ec(6);
dst_ip = 0;
src_ip = 0;
data = malloc_ec(1);

bad = 0;
idx = 0;
out = malloc_ec(1000);

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

func readintbytesb (f, n) { // big endian
  x = 0;
  i = 0;
  while (i < n) {
    byte = read_ec(f);
    if (byte < 0) {
      bad = 1;
      return -1;
    }
    x = x << 8;
    x = x | byte;
    i = i + 1;
  }
  return x;
}

func parse_ethernet (f, dummy) {
  i = 0;
  while (i < 6) {
    x = read_ec(f);
    if (x < 0) {
      bad = 1;
      return -1;
    }
    dst_mac[i] = x;
    i = i + 1;
  }
  i = 0;
  while (i < 6) {
    x = read_ec(f);
    if (x < 0) {
      bad = 1;
      return -1;
    }
    src_mac[i] = x;
    i = i + 1;
  }

  ether_type = readintbytesl(f, 2);
  if (bad || ether_type != 8) {
    bad = 1;
    return -1;
  } // IPv4
  dummy = parse_ipv4(f, 0);
  return 0;
}

func check_sum (f, len) {
```

```
  start = pos_ec(f);
  if (len <= 0 || len % 2 != 0 || start < 0) {
    bad = 1;
    return -1;
  }
  sum = 0;
  i = 0;
  while (i < len / 2) {
    x = readintbytesb(f, 2);
    sum = sum + x;
    i = i + 1;
  }
  hi = (sum >> 16) & 65535;
  lo = sum & 65535;
  x = seek_ec(f, start);
  if (hi + lo != 65535 || x < 0) {
    bad = 1;
    return -1;
  }
  return 0;
}

func prtbuf (x) {
  if (idx >= 1000) {
    bad = 1;
    return -1;
  }
  out[idx] = x;
  idx = idx + 1;
  return 0;
}

func prtbuf_ip (addr) {
  dummy = prtbuf((addr >> 24) & 255);
  dummy = prtbuf('.');
  dummy = prtbuf((addr >> 16) & 255);
  dummy = prtbuf('.');
  dummy = prtbuf((addr >> 8) & 255);
  dummy = prtbuf('.');
  dummy = prtbuf(addr & 255);
  return 0;
}

func parse_ipv4 (f, dummy) {
  dummy = check_sum(f, 20); // no options
  if (bad) {
    return -1;
  }

  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  version = (x & 240) >> 4;
  hdr_size = x & 15;
  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  total = readintbytesb(f, 2);

  id = readintbytesb(f, 2);
  fragment = readintbytesb(f, 2);
  dont_frag = (fragment & 16384) >> 14;
  more_frag = (fragment & 8192) >> 13;
  offs_frag = fragment & 8191;

  ttl = read_ec(f);
  ip_type = read_ec(f);
  checksum = readintbytesb(f, 2);

  src_ip = readintbytesb(f, 4);
  dst_ip = readintbytesb(f, 4);

```

```
  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
    bad = 1;
    return -1;
  }
  dummy = prtbuf('\n');
  dummy = prtbuf_ip(src_ip);
  dummy = prtbuf(' ');
  dummy = prtbuf_ip(dst_ip);
  dummy = prtbuf(' ');
  dummy = parse_udp(f, total - hdr_size * 4);
  return 0;
}

func parse_udp (f, pkt_size) {
  src_port = readintbytesb(f, 2);
  dst_port = readintbytesb(f, 2);
  udp_size = readintbytesb(f, 2);
  checksum = readintbytesb(f, 2);

  if (bad || pkt_size > udp_size || pkt_size <= 8) {
    bad = 1;
    return -1;
  }

  if (src_port == 53 || dst_port == 53) { // DNS
    dummy = parse_dns(f, pkt_size - 8);
  } else { // others
    if (valid(data)) {
      dummy = free_ec(data);
    }
    data_size = pkt_size - 8;
    data = malloc_ec(data_size);
    if (!valid(data)) {
      bad = 1;
      return -1;
    }
    i = 0;
    while (i < data_size) {
      x = read_ec(f);
      if (x < 0) {
        bad = 1;
        return -1;
      }
      data[i] = x;
      i = i + 1;
    }
    dummy = prtbuf('\n');
  }
  return 0;
}

func parse_dns (f, data_size) {
  id = readintbytesb(f, 2);
  dummy = prtbuf(id);
  dummy = prtbuf('\n');
  flags = readintbytesb(f, 2);
  n_q = readintbytesb(f, 2);
  n_ans = readintbytesb(f, 2);
  n_auth = readintbytesb(f, 2);
  n_add = readintbytesb(f, 2);

  // questions
  i = 0;
  while (i < n_q) {
    stop = 0;
    first = 1;
    while (!stop) {
      len = read_ec(f);
      if (len < 0) {
        bad = 1;
        return -1;
      }
      if (len == 0) {
```

```
        stop = 1;
      } else {
        if (first) {
          first = 0;
        } else {
          dummy = prtbuf('.');
        }
        j = 0;
        while (j < len) {
          x = read_ec(f);
          if (x < 0 || idx >= 1000) {
            bad = 1;
            return -1;
          }
          out[idx] = x;
          j = j + 1;
          idx = idx + 1;
        }
      }
    }
    dummy = prtbuf('\n');
    type_q = readintbytesb(f, 2);
    class_q = readintbytesb(f, 2);
    i = i + 1;
  }

  // answers
  i = 0;
  while (i < n_ans) {
    ptr = readintbytesb(f, 2);
    if (ptr != 49164) {
      bad = 1;
      return -1;
    }
    type_ans = readintbytesb(f, 2);
    class_ans = readintbytesb(f, 2);
    ttl = readintbytesb(f, 4);
    len = readintbytesb(f, 2);
    if (type_ans == 1) {
      if (len != 4) {
        bad = 1;
        return -1;
      }
      addr = readintbytesb(f, 4);
      dummy = prtbuf_ip(addr);
    } else {
      if (type_ans != 5 || len <= 0) {
        bad = 1;
        return -1;
      }
      j = 0;
      while (j < len) {
        x = read_ec(f);
        if (x < 0 || idx >= 1000) {
          bad = 1;
          return -1;
        }
        out[idx] = x;
        j = j + 1;
        idx = idx + 1;
      }
    }
    dummy = prtbuf('\n');
    i = i + 1;
  }

  // authority
  // additional
  return 0;
}

main {
  if (!valid(dst_mac) || !valid(src_mac) || !valid(data) || !valid(out)) {
    exit(1);
  }
  f = openb_ec("../inputs/pcap/bad.pcap", 1);
```

```
  if (!valid(f)) {
    exit(1);
  }

  x = read_ec(f);
  if (x != 212) {
    exit(1);
  }
  x = read_ec(f);
  if (x != 195) {
    exit(1);
  }
  x = read_ec(f);
  if (x != 178) {
    exit(1);
  }
  x = read_ec(f);
  if (x != 161) {
    exit(1);
  }
  maj_ver = readintbytesl(f, 2);
  min_ver = readintbytesl(f, 2);
  this_zone = readintbytesl(f, 4);
  sigfigs = readintbytesl(f, 4);
  snap_len = readintbytesl(f, 4); // number of packets
  link_type = readintbytesl(f, 4);
  if (bad || link_type != 1) {
    exit(1);
  } // ethernet

  lookatb (1, f, 12, 4, 0) {
    bad = 0;
    idx = 0;
    ts_epoch = readintbytesl(f, 4);
    ts_nanosec = readintbytesl(f, 4);
    caplen = readintbytesl(f, 4);
    length = readintbytesl(f, 4);
    if (!bad && caplen == length) {
      dummy = parse_ethernet(f, 0);
      if (!bad) {
        i = 0;
        while (i < idx) {
          x = out[i];
          print(x);
          i = i + 1;
        }
      }
    } // skip unit
  }

  return 0;
}
```

A.7.4 Plain-loop version

```
dst_mac = malloc_ec(6);
src_mac = malloc_ec(6);
dst_ip = 0;
src_ip = 0;
data = malloc_ec(1);

bad = 0;
idx = 0;
out = malloc_ec(1000);

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  if (n == 1) {
    return x;
```

```
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

func readintbytesb (f, n) { // big endian
  x = 0;
  i = 0;
  while (i < n) {
    byte = read_ec(f);
    if (byte < 0) {
      bad = 1;
      return -1;
    }
    x = x << 8;
    x = x | byte;
    i = i + 1;
  }
  return x;
}

func parse_ethernet (f, dummy) {
  i = 0;
  while (i < 6) {
    x = read_ec(f);
    if (x < 0) {
      bad = 1;
      return -1;
    }
    dst_mac[i] = x;
    i = i + 1;
  }
  i = 0;
  while (i < 6) {
    x = read_ec(f);
    if (x < 0) {
      bad = 1;
      return -1;
    }
    src_mac[i] = x;
    i = i + 1;
  }

  ether_type = readintbytesl(f, 2);
  if (bad || ether_type != 8) {
    bad = 1;
    return -1;
  } // IPv4
  dummy = parse_ipv4(f, 0);
  return 0;
}

func check_sum (f, len) {
  start = pos_ec(f);
  if (len <= 0 || len % 2 != 0 || start < 0) {
    bad = 1;
    return -1;
  }
  sum = 0;
  i = 0;
  while (i < len / 2) {
    x = readintbytesb(f, 2);
    sum = sum + x;
    i = i + 1;
  }
  hi = (sum >> 16) & 65535;
  lo = sum & 65535;
  x = seek_ec(f, start);
  if (hi + lo != 65535 || x < 0) {
    bad = 1;
    return -1;
  }
  return 0;
}

```

```
func prtbuf (x) {
  if (idx >= 1000) {
    bad = 1;
    return -1;
  }
  out[idx] = x;
  idx = idx + 1;
  return 0;
}

func prtbuf_ip (addr) {
  dummy = prtbuf((addr >> 24) & 255);
  dummy = prtbuf('.');
  dummy = prtbuf((addr >> 16) & 255);
  dummy = prtbuf('.');
  dummy = prtbuf((addr >> 8) & 255);
  dummy = prtbuf('.');
  dummy = prtbuf(addr & 255);
  return 0;
}

func parse_ipv4 (f, dummy) {
  dummy = check_sum(f, 20); // no options
  if (bad) {
    return -1;
  }

  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  version = (x & 240) >> 4;
  hdr_size = x & 15;
  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  total = readintbytesb(f, 2);

  id = readintbytesb(f, 2);
  fragment = readintbytesb(f, 2);
  dont_frag = (fragment & 16384) >> 14;
  more_frag = (fragment & 8192) >> 13;
  offs_frag = fragment & 8191;

  ttl = read_ec(f);
  ip_type = read_ec(f);
  checksum = readintbytesb(f, 2);

  src_ip = readintbytesb(f, 4);
  dst_ip = readintbytesb(f, 4);

  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
    bad = 1;
    return -1;
  }
  dummy = prtbuf('\n');
  dummy = prtbuf_ip(src_ip);
  dummy = prtbuf(' ');
  dummy = prtbuf_ip(dst_ip);
  dummy = prtbuf(' ');
  dummy = parse_udp(f, total - hdr_size * 4);
  return 0;
}

func parse_udp (f, pkt_size) {
  src_port = readintbytesb(f, 2);
  dst_port = readintbytesb(f, 2);
  udp_size = readintbytesb(f, 2);
  checksum = readintbytesb(f, 2);

```

```
  if (bad || pkt_size > udp_size || pkt_size <= 8) {
    bad = 1;
    return -1;
  }

  if (src_port == 53 || dst_port == 53) { // DNS
    dummy = parse_dns(f, pkt_size - 8);
  } else { // others
    if (valid(data)) {
      dummy = free_ec(data);
    }
    data_size = pkt_size - 8;
    data = malloc_ec(data_size);
    if (!valid(data)) {
      bad = 1;
      return -1;
    }
    i = 0;
    while (i < data_size) {
      x = read_ec(f);
      if (x < 0) {
        bad = 1;
        return -1;
      }
      data[i] = x;
      i = i + 1;
    }
    dummy = prtbuf('\n');
  }
  return 0;
}

func parse_dns (f, data_size) {
  id = readintbytesb(f, 2);
  dummy = prtbuf(id);
  dummy = prtbuf('\n');
  flags = readintbytesb(f, 2);
  n_q = readintbytesb(f, 2);
  n_ans = readintbytesb(f, 2);
  n_auth = readintbytesb(f, 2);
  n_add = readintbytesb(f, 2);

  // questions
  i = 0;
  while (i < n_q) {
    stop = 0;
    first = 1;
    while (!stop) {
      len = read_ec(f);
      if (len < 0) {
        bad = 1;
        return -1;
      }
      if (len == 0) {
        stop = 1;
      } else {
        if (first) {
          first = 0;
        } else {
          dummy = prtbuf('.');
        }
        j = 0;
        while (j < len) {
          x = read_ec(f);
          if (x < 0 || idx >= 1000) {
            bad = 1;
            return -1;
          }
          out[idx] = x;
          j = j + 1;
          idx = idx + 1;
        }
      }
    }
    dummy = prtbuf('\n');
    type_q = readintbytesb(f, 2);
```

```
      class_q = readintbytesb(f, 2);
      i = i + 1;
    }

    // answers
    i = 0;
    while (i < n_ans) {
      ptr = readintbytesb(f, 2);
      if (ptr != 49164) {
        bad = 1;
        return -1;
      }
      type_ans = readintbytesb(f, 2);
      class_ans = readintbytesb(f, 2);
      ttl = readintbytesb(f, 4);
      len = readintbytesb(f, 2);
      if (type_ans == 1) {
        if (len != 4) {
          bad = 1;
          return -1;
        }
        addr = readintbytesb(f, 4);
        dummy = prtbuf_ip(addr);
      } else {
        if (type_ans != 5 || len <= 0) {
          bad = 1;
          return -1;
        }
        j = 0;
        while (j < len) {
          x = read_ec(f);
          if (x < 0 || idx >= 1000) {
            bad = 1;
            return -1;
          }
          out[idx] = x;
          j = j + 1;
          idx = idx + 1;
        }
      }
      dummy = prtbuf('\n');
      i = i + 1;
    }

    // authority
    // additional
    return 0;
}

main {
  if (!valid(dst_mac) || !valid(src_mac) || !valid(data) || !valid(out)) {
    exit(1);
  }
  f = openb_ec("../inputs/pcap/bad.pcap", 1);
  if (!valid(f)) {
    exit(1);
  }

  x = read_ec(f);
  if (x != 212) {
    exit(1);
  }
  x = read_ec(f);
  if (x != 195) {
    exit(1);
  }
  x = read_ec(f);
  if (x != 178) {
    exit(1);
  }
  x = read_ec(f);
  if (x != 161) {
    exit(1);
  }
  maj_ver = readintbytesl(f, 2);
  min_ver = readintbytesl(f, 2);
```

```
    this_zone = readintbytesl(f, 4);
    sigfigs = readintbytesl(f, 4);
    snap_len = readintbytesl(f, 4); // number of packets
    link_type = readintbytesl(f, 4);
    if (bad || link_type != 1) {
      exit(1);
    } // ethernet

    finish = 0;
    while (!finish) {
      bad = 0;
      idx = 0;
      ts_epoch = readintbytesl(f, 4);
      ts_nanosec = readintbytesl(f, 4);
      caplen = readintbytesl(f, 4);
      length = readintbytesl(f, 4);
      if (!bad && length >= 0) {
        eou = pos_ec(f) + length;
        if (!bad && caplen == length) {
          dummy = parse_ethernet(f, 0);
          if (pos_ec(f) < 0 || pos_ec(f) > eou) {
            bad = 1;
          }
          if (!bad) {
            i = 0;
            while (i < idx) {
              x = out[i];
              print(x);
              i = i + 1;
            }
          }
        } // skip unit
        x = seek_ec(f, eou);
        if (x < 0) {
          finish = 1;
        }
      } else { // give up
        finish = 1;
      }
    }

    return 0;
  }
```

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. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Opearting 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.