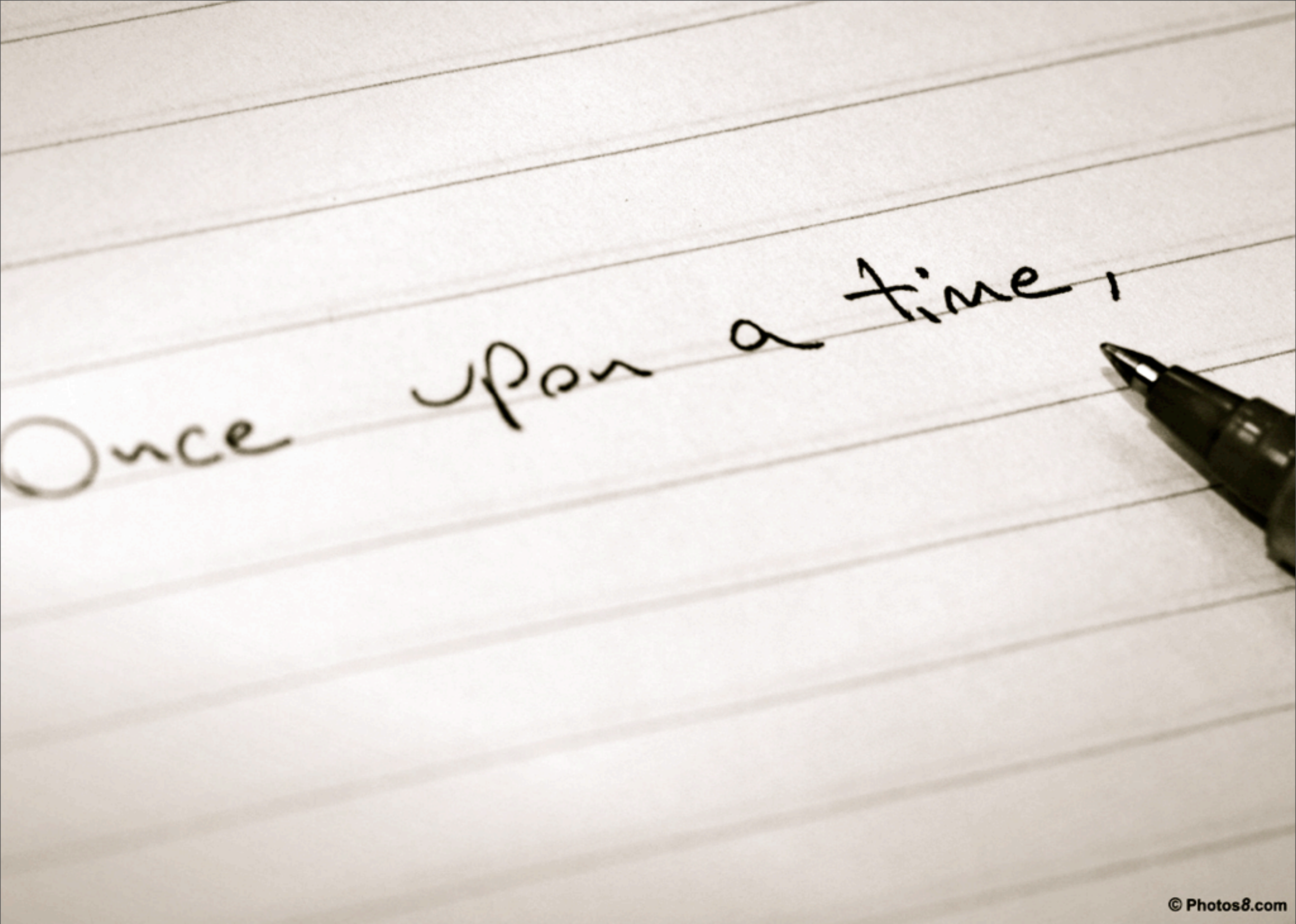
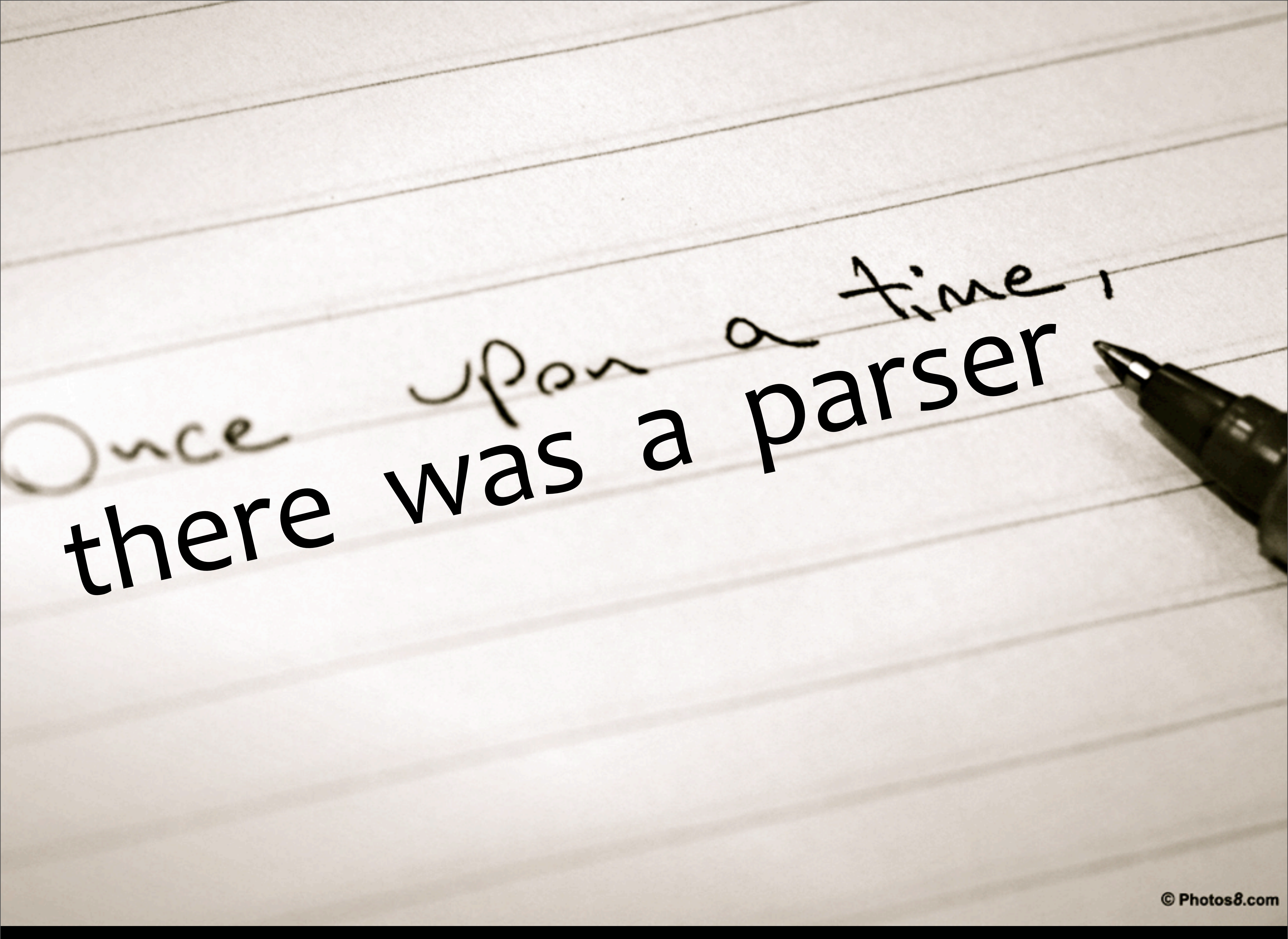
Modular rollback through control logging

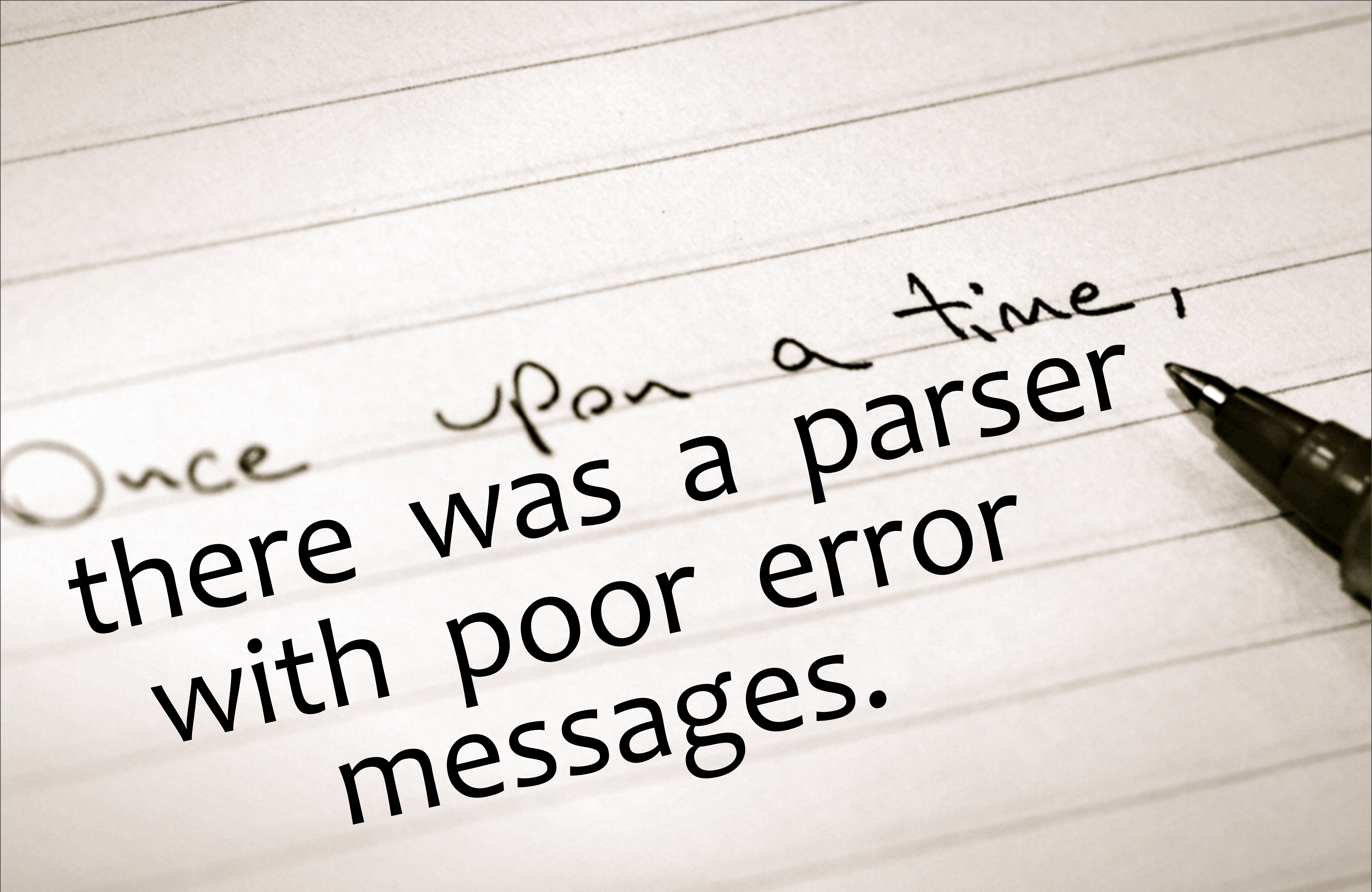
- A pair of twin functional pearls -

Olin Shivers & **Aaron Turon**
Northeastern University



Tuesday, September 27, 2011


Once upon a time,
there was a parser
© Photos8.com


Once upon a time,
there was a parser
with poor error
messages.
© Photos8.com

```
> val f(x) = x + 1;
```

```
> val f(x) = x + 1;
```

SYNTAX ERROR at 1:6

```
val f(x) = x + 1;
```

```
val f(x) = x + 1;
```

decl ::= **val** *id* **=** *exp* **;**
  | **fun** *id* **(** *ids* **)** **=** *exp* **;**
  | ...

```
val f(x) = x + 1;
```

Point of discovery

```
decl ::= val id = exp ;
       | fun id ( ids ) = exp ;
       | ...
```

The real error: `val` should be `fun`

```
val f(x) = x + 1;
```

Point of discovery

decl ::= **val** *id* = *exp* ;
　　| **fun** *id* (*ids*) = *exp* ;
　　| ...

A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery

MICHAEL G. BURKE and GERALD A. FISHER
Thomas J. Watson Research Center

1987

This paper presents a powerful, practical, and essentially language-independent syntactic error diagnosis and recovery method that is applicable within the frameworks of LR and LL parsing. The method generally issues accurate diagnoses even where multiple errors occur within close proximity, yet seldom issues spurious error messages. It employs a new technique, parse action deferral, that allows the most appropriate recovery in cases where this would ordinarily be precluded by late detection of the error. The method is practical in that it does not impose substantial space or time overhead on the parsing of correct programs, and in that its time efficiency in processing an error allows for its incorporation in a production compiler. The method is language independent, but it does allow for tuning with respect to particular languages and implementations through the setting of language-specific parameters.

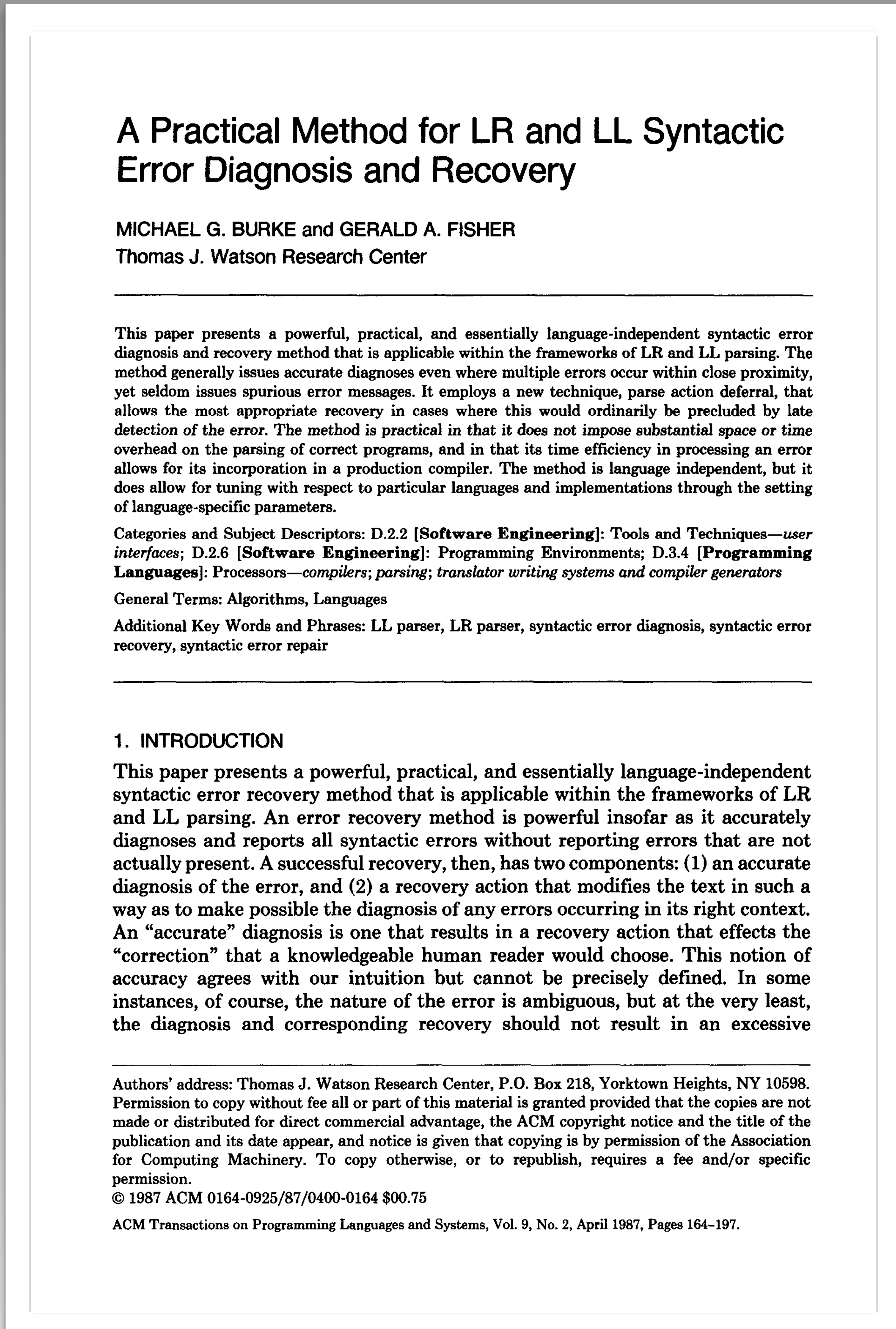
Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; D.2.6 [**Software Engineering**]: Programming Environments; D.3.4 [**Programming Languages**]: Processors—*compilers*; *parsing*; *translator writing systems and compiler generators*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: LL parser, LR parser, syntactic error diagnosis, syntactic error recovery, syntactic error repair

1. INTRODUCTION

This paper presents a powerful, practical, and essentially language-independent



A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery

MICHAEL G. BURKE and GERALD A. FISHER
Thomas J. Watson Research Center

This paper presents a powerful, practical, and essentially language-independent syntactic error diagnosis and recovery method that is applicable within the frameworks of LR and LL parsing. The method generally issues accurate diagnoses even where multiple errors occur within close proximity, yet seldom issues spurious error messages. It employs a new technique, parse action deferral, that allows the most appropriate recovery in cases where this would ordinarily be precluded by late detection of the error. The method is practical in that it does not impose substantial space or time overhead on the parsing of correct programs, and in that its time efficiency in processing an error allows for its incorporation in a production compiler. The method is language independent, but it does allow for tuning with respect to particular languages and implementations through the setting of language-specific parameters.

Categories and Subject Descriptors: D.2.2 **[Software Engineering]**: Tools and Techniques—*user interfaces*; D.2.6 **[Software Engineering]**: Programming Environments; D.3.4 **[Programming Languages]**: Processors—*compilers*; *parsing*; *translator writing systems and compiler generators*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: LL parser, LR parser, syntactic error diagnosis, syntactic error recovery, syntactic error repair

1. INTRODUCTION

This paper presents a powerful, practical, and essentially language-independent syntactic error recovery method that is applicable within the frameworks of LR and LL parsing. An error recovery method is powerful insofar as it accurately diagnoses and reports all syntactic errors without reporting errors that are not actually present. A successful recovery, then, has two components: (1) an accurate diagnosis of the error, and (2) a recovery action that modifies the text in such a way as to make possible the diagnosis of any errors occurring in its right context. An "accurate" diagnosis is one that results in a recovery action that effects the "correction" that a knowledgeable human reader would choose. This notion of accuracy agrees with our intuition but cannot be precisely defined. In some instances, of course, the nature of the error is ambiguous, but at the very least, the diagnosis and corresponding recovery should not result in an excessive

Authors' address: Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

ACM Transactions on Programming Languages and Systems, Vol. 9, No. 2, April 1987, Pages 164–197.

The Burke-Fisher Principle:

Explain syntax errors by finding small, nearby edits that enable the parser to make substantial progress

2. THE METHOD

2.1 Overview

2.1.1 *The Parsing Framework.* The method assumes a framework in which an LR or LL parser maintains an input token buffer TOKENS, a state or prediction stack, and a parse stack. The *parse configuration* thus has three components: the configuration of TOKENS, that of the state or prediction stack, and that of the parse stack. TOKENS is a queue containing part or all of the sequence of remaining input tokens. The *current token*, denoted CURTOK, is the front element of TOKENS. The token immediately preceding CURTOK in the source program shall be denoted as PREVTOK.

2. THE METHOD

2.1 Overview

2.1.1 *The Parsing Framework.* The method assumes a framework in which an LR or LL parser maintains an input token buffer TOKENS, a state or prediction stack, and a parse stack. The *parse configuration* thus has three components: the configuration of TOKENS, that of the state or prediction stack, and that of the parse stack. TOKENS is a queue containing part or all of the sequence of remaining input tokens. The *current token*, denoted CURTOK, is the front element of TOKENS. The token immediately preceding CURTOK in the source program shall be denoted as PREVTOK.

By backing down the parse stack and considering the possible simple repairs at each of its elements, one can effect a simple repair at a point in the prefix.

2. THE METHOD

2.1 Overview

2.1.1 *The Parsing Framework.* The method assumes a framework in which an LR or LL parser maintains an input token buffer TOKENS, a state or prediction stack, and a parse stack. The *parse configuration* thus has three components: the configuration of TOKENS, that of the state or prediction stack, and that of the parse stack. TOKENS is a queue containing part or all of the sequence of remaining input tokens. The *current token*, denoted CURTOK, is the front element of TOKENS. The token immediately preceding CURTOK in the source program shall be denoted as PREVTOK.

By backing down the parse stack and considering the possible simple repairs at each of its elements, one can effect a simple repair at a point in the prefix.

Token deferral may also be viewed as double parsing. One parser simply checks for syntactic correctness and performs no real reduce actions. The second parser is always $k-1$ tokens behind, always has correct input, and performs reduce actions on the parse stack. In our implementation the deferred tokens and sequences of reductions are maintained in a deferred tokens queue and a deferred rules queue, respectively.

Modular rollback through control logging

Our mission:
Infiltrate the parser by impersonating its lexer

Our plan:
`BurkeFisher: PARSER→PARSER`
a functor that wraps a parser, and spies on its control flow

Our agent:

Control.
Delimited Control.

Parser **Repairer** **Lexer**

```
parse lex s
```

Parser

Repairer

Lexer

```
parse lex s
```

```
reset (fn () =>
  parse wrapLex s)
```

Parser **Repairer** **Lexer**

```
parse lex s
```

```
reset (fn () =>
  parse wrapLex s)
```

```
wrapLex s
```

Parser **Repairer** **Lexer**

```
parse lex s
```

```
reset (fn () =>
  parse wrapLex s)
```

```
wrapLex s
```

checkpoint

Parser **Repairer** **Lexer**

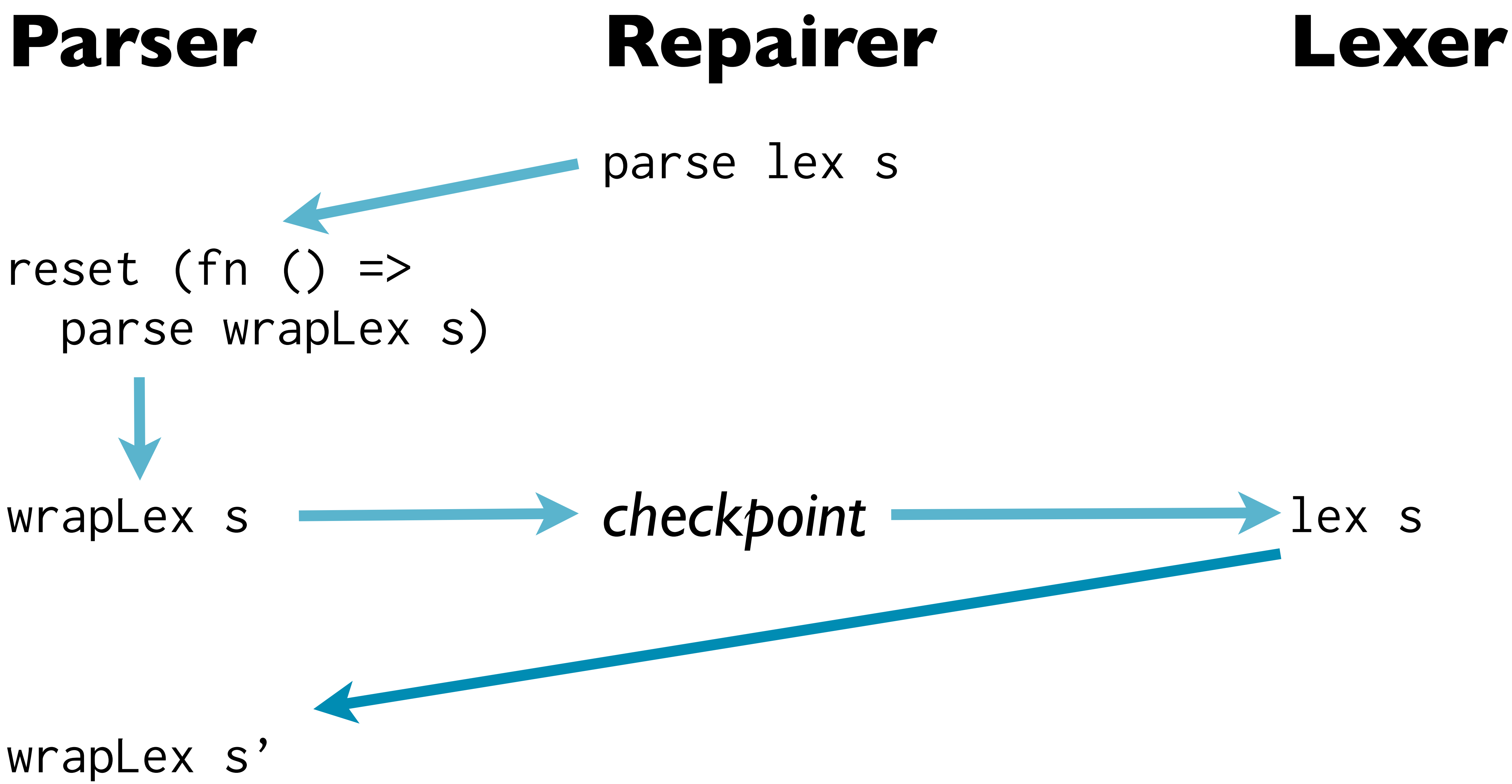
```
parse lex s
```

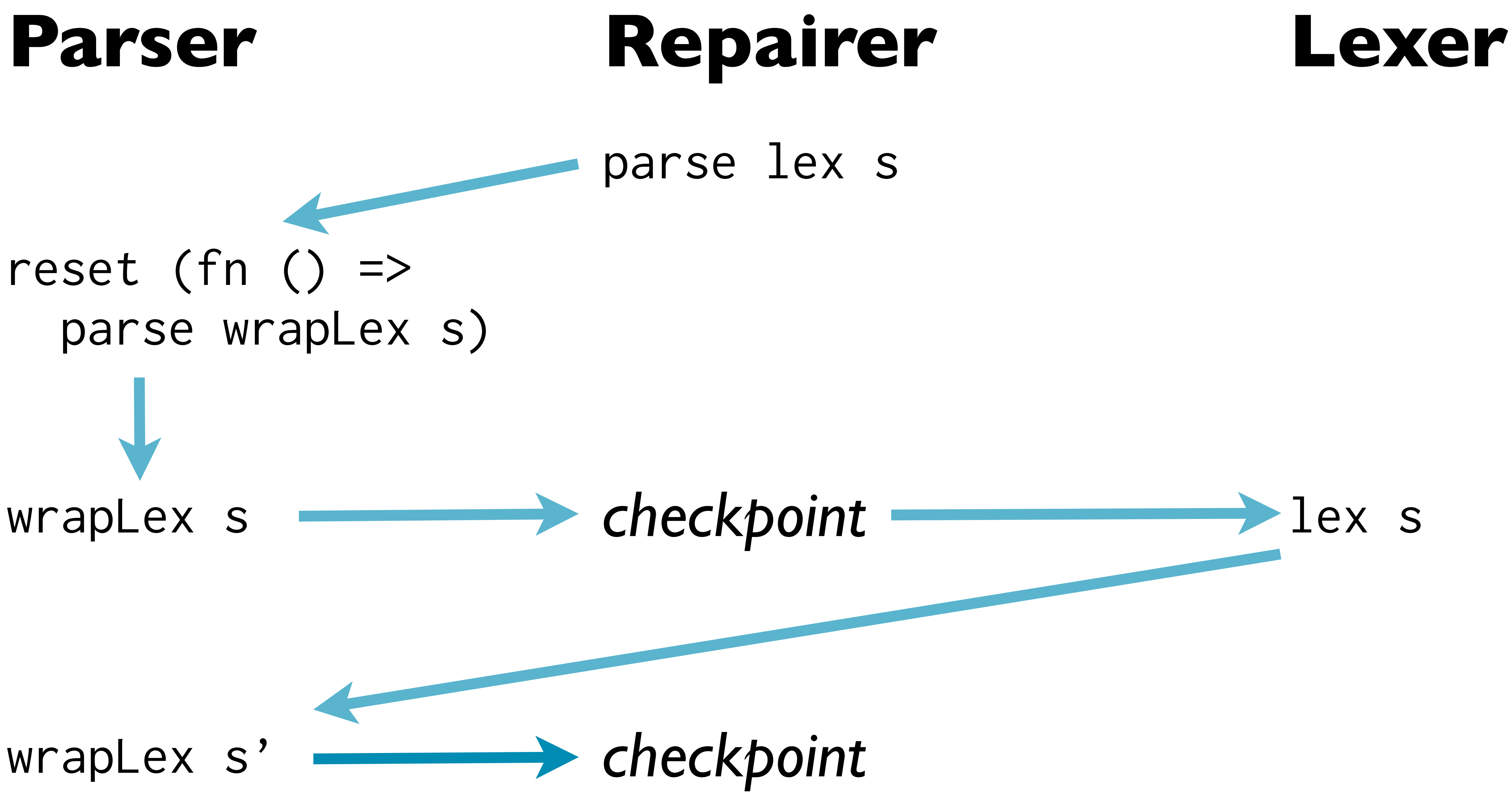
```
reset (fn () =>
  parse wrapLex s)
```

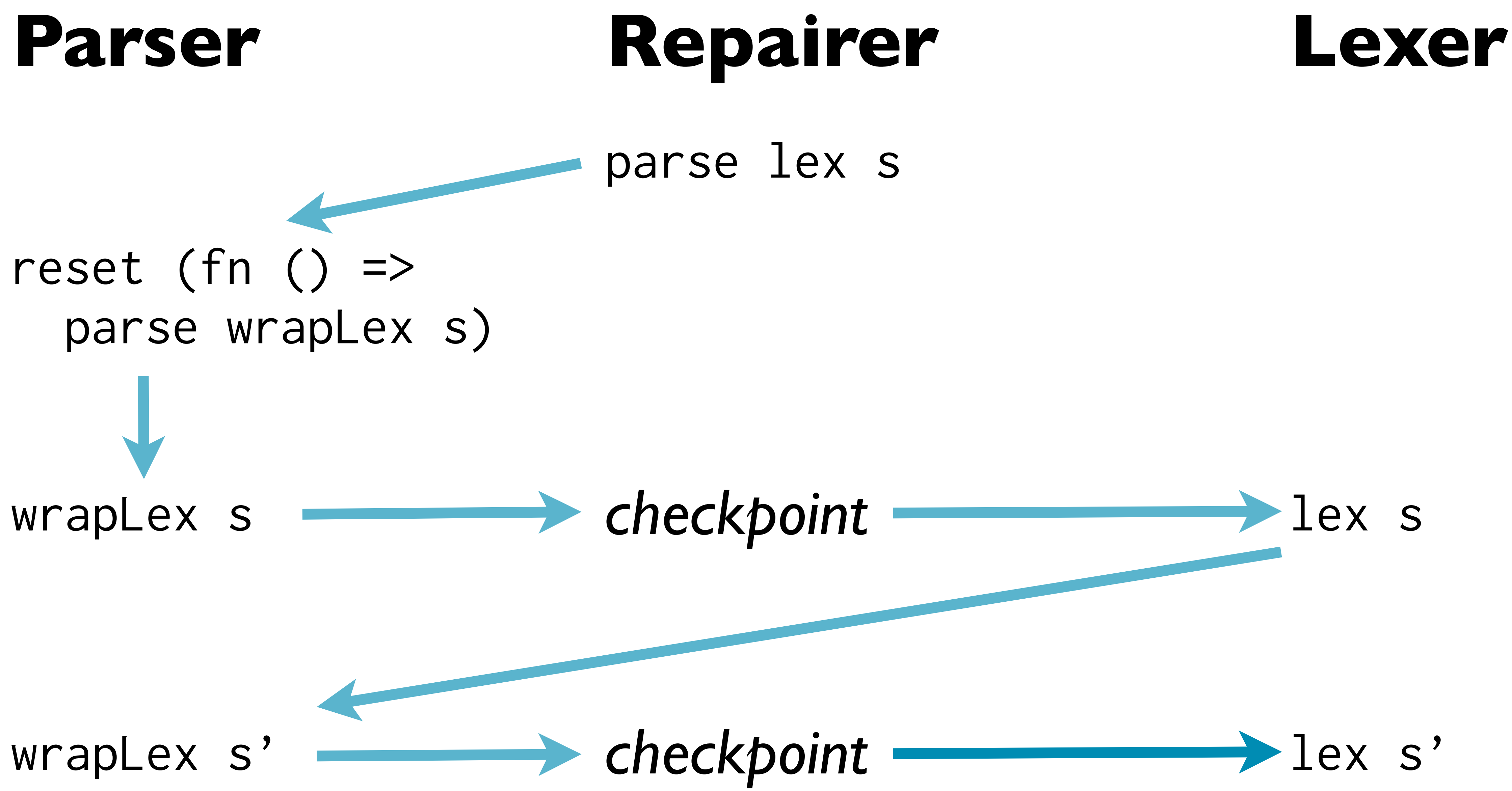
```
wrapLex s
```

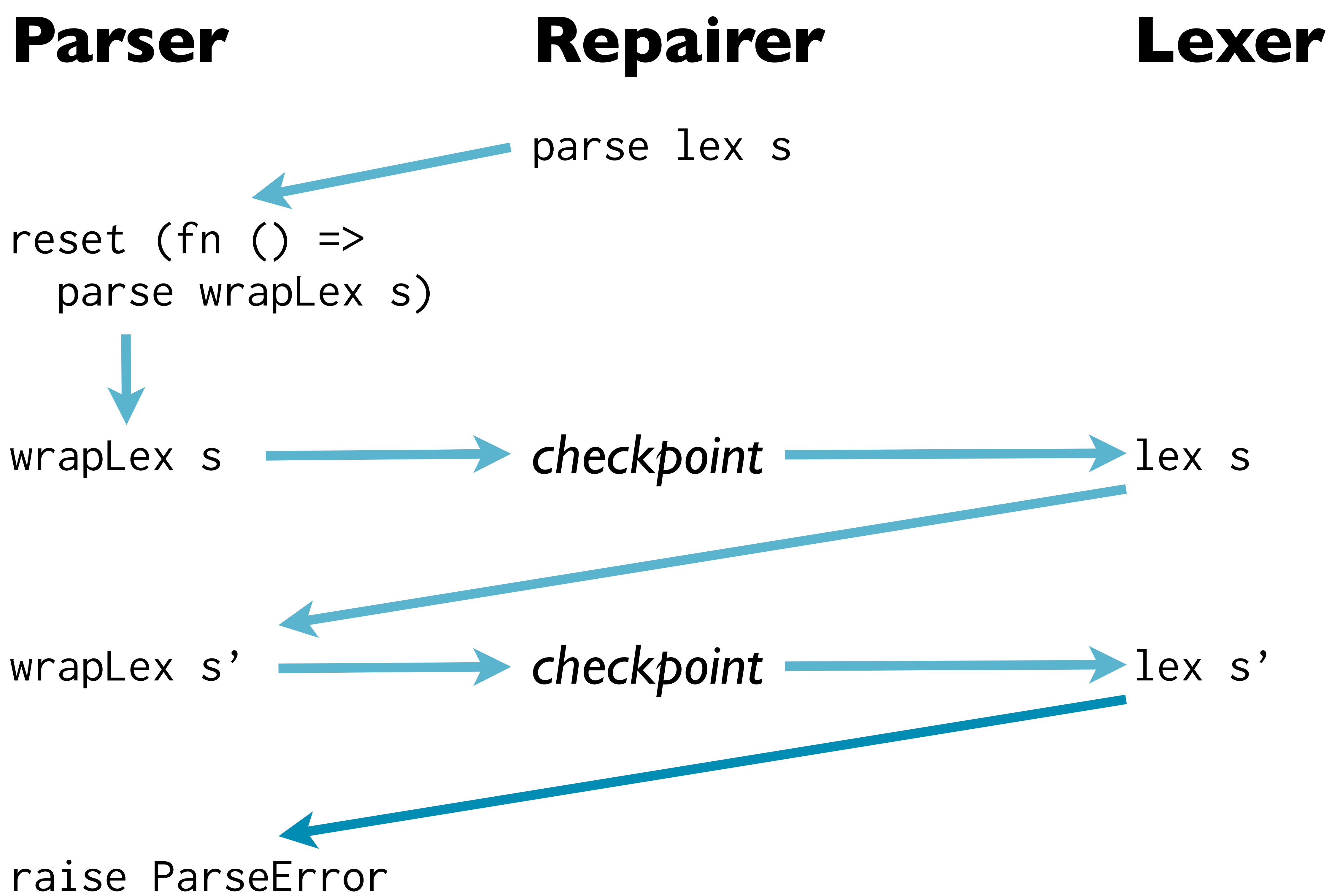
checkpoint

```
lex s
```


Parser
Repairer
Lexer
parse lex s
reset (fn () =>
  parse wrapLex s)
wrapLex s
checkpoint
lex s
wrapLex s’


Parser
Repairer
Lexer
parse lex s
reset (fn () =>
parse wrapLex s)
wrapLex s
checkpoint
lex s
wrapLex s’
checkpoint


Parser
Repairer
Lexer
parse lex s
reset (fn () =>
parse wrapLex s)
wrapLex s
checkpoint
lex s
wrapLex s’
checkpoint
lex s’


Parser
Repairer
Lexer
parse lex s
reset (fn () =>
parse wrapLex s)
wrapLex s
checkpoint
lex s
wrapLex s’
checkpoint
lex s’
raise ParseError

Parser | **Repairer** | **Lexer**

`parse lex s`

```
reset (fn () =>
  parse wrapLex s)
```

`wrapLex s` → *checkpoint* → `lex s`

`wrapLex s'` → *checkpoint* → `lex s'`

`raise ParseError` → ***search for repair***

```
signature PARSER =
sig
  type token
  val exampleToks: token list

  type stream
  type lexer = stream -> token * stream
  type result
  exception ParseError

  val parse: lexer -> stream -> result
end
```

```
signature PARSER =
sig
  type token
  val exampleToks: token list

  type stream
  type lexer = stream -> token * stream
  type result
  exception ParseError

  val parse: lexer -> stream -> result
end
```

We can't change these types

```
signature PARSER =
sig
  type token
  val exampleToks: token list

  type stream
  type lexer = stream -> token * stream
  type result
  exception ParseError

  val parse: lexer -> stream -> result
end
```

But we can add effects

We can't change these types

The Burke-Fisher Functor

```
functor BurkeFisher (P: PARSER) =
struct
  open P (* we'll shadow result and parse,
          * but otherwise be just like P *)

  datatype result
    = RESULT of P.result
    | REPAIR of token   (* replace this token *)
              * token   (* with this token    *)
    | UNREPAIRABLE
```

We'll ignore position information for simplicity

The Burke-Fisher Functor

```
type checkPt = lexResult *
  (lexResult -> P.result)
```

```
fun parse lex strm = let
  val chkPts = ref []
  fun push chkPt = chkPts := (chkPt :: !chkPts)
```

The Burke-Fisher Functor

```
type checkPt = lexResult *
  (lexResult -> P.result)
```

```
fun parse lex strm = let
  val chkPts = ref []
  fun push chkPt = chkPts := (chkPt :: !chkPts)

  fun wrapLex strm = let
    val lexResult = lex strm
    in shift (fn k => push (lexResult, k);
                      k lexResult)
    end
```

The Burke-Fisher Functor

```
type checkPt = lexResult *
  (lexResult -> P.result)
```

```
fun parse lex strm = let
  val chkPts = ref []
  fun push chkPt = chkPts := (chkPt :: !chkPts)

  fun wrapLex strm = let
    val lexResult = lex strm
    in shift (fn k => push (lexResult, k);
                      k lexResult)
    end

  in RESULT (reset (fn () =>
       P.parse wrapLex strm))
  handle ParseError => repair (!chkPts)
```

The Burke-Fisher Functor

```
mapFind: (α -> β option) -> α list -> β option
```

```
repair: checkPt list -> result
```

```
fun repair [] = UNREPAIRABLE
  | repair (chkPt::chkPts) =
      case mapFind (retry chkPt) exampleToks
       of NONE                    => repair chkPts
        | SOME replacement => REPAIR replacement
```

The Burke-Fisher Functor

```
type replacement = token * token
```

```
retry: checkPt -> token -> replacement option
```

```
fun retry ((oldTok, strm), k) newTok =
  k (newTok, strm);      (* execute for effect *)
  SOME (oldTok, newTok)
handle ParseError => NONE
```

```
Syntax error:
val f(x) = 1+x;
^^^
Did you mean 'fun'?
```

Yes, but:

Yes, but:

- What about deletions, insertions?

Yes, but:

- What about deletions, insertions?
- What about metrics and heuristics?

Yes, but:

- What about deletions, insertions?
- What about metrics and heuristics?
- What about space usage?

Yes, but:

- What about deletions, insertions?
- What about metrics and heuristics?
- What about space usage?
- What about side effects?

The twin pearl: “prompt reading” in Scheme

Some “lost art” from ‘70s-era LISP systems:

- REPL handles TTY line driver
- Parsing concurrent with input
- Syntax errors are ***impossible***
- Last closing paren fires off the s-expression

The twin pearl:
"prompt reading" in Scheme

The challenge: the ***backspace*** key

Need to roll back parser control state,
and TTY state

The twin pearl: “prompt reading” in Scheme

The challenge: the ***backspace*** key

Need to roll back parser control state, *and* TTY state

The solution:

Weld ***performace*** of effects to ***logging*** of their reversal

Requires exposing effectful operations

What have we done?

Rollback can be *functorized*, using infiltration:

- Clear separation of concerns: can change input and rollback strategy independently
- Clean interface between the concerns
- Sketched dealing with side-effects

This is a general technique!

What more can we do?

- BurkeFisher(YourTypechecker)
    - cf SEMINAL
- Web development?
- Understand all of this through Filinski's lens
    - Come to the Continuation Workshop!

What more can we do?

- BurkeFisher(YourTypechecker)
  - cf SEMINAL
- Web development?
- Understand all of this through Filinski's lens
  - Come to the Continuation Workshop!

Thank you