ZipLex

Verified Linear-Time Invertible Lexer

Samuel Chassot, Viktor Kunčak

27.07.2026 - CAV26

Sorting Functions

Refactor Tool Example

```
def abs(x: Int): Int =
  if (x < 0) then
    -x
  else
    x
end abs
def mul(x: Int, y: Int): Int =
  x * y
end mul
```

Sorting Functions

Refactor Tool Example

```
def abs(x: Int): Int =
  if (x < 0) then
    -x
  else
    x
end abs
def mul(x: Int, x: Int): Int =
  x * y
end mul
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, x: Int): Int =⏎
  x * y⏎
end mul
def abs(x: Int): Int =⏎
  if (x < 0) then⏎
    -x⏎
  else⏎
    x⏎
end abs⏎
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, x: Int): Int =⏎
  x * y⏎
end mul
def abs(x: Int): Int =⏎
  if (x < 0) then⏎
    -x⏎
  else⏎
    x⏎
end abs⏎
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, y: Int): Int =
  x * y
end muldef abs(x: Int): Int =
  if (x < 0) then
    -x
  else
    x
end abs
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, y: Int): Int =
  x * y
end muldef abs(x: Int): Int =
  if (x < 0) then
    -x
  else
    x
end abs
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, x: Int): Int =⏎
  x * y⏎
end muldef abs(x: Int): Int =⏎
  if (x < 0) then⏎
    -x⏎
  else⏎
    x⏎
end abs⏎
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, x: Int): Int =
  x * y
end muldef abs(x: Int): Int =
  if (x < 0) then
    -x
  else
    x
end abs
```

Lexical Analysis

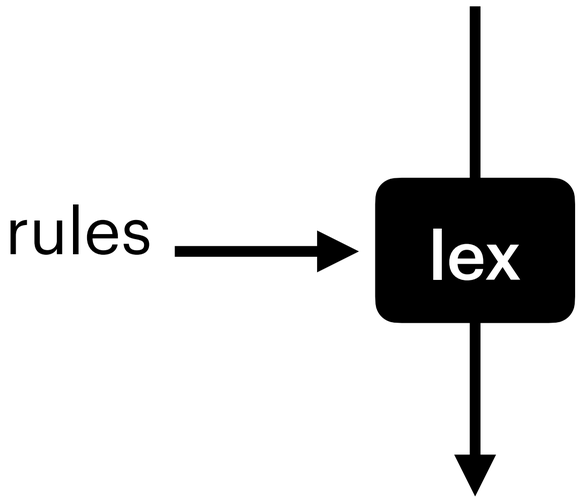
Ziplex Interface

```
def abs(x: Int): Int
```

Lexical Analysis

Ziplex Interface

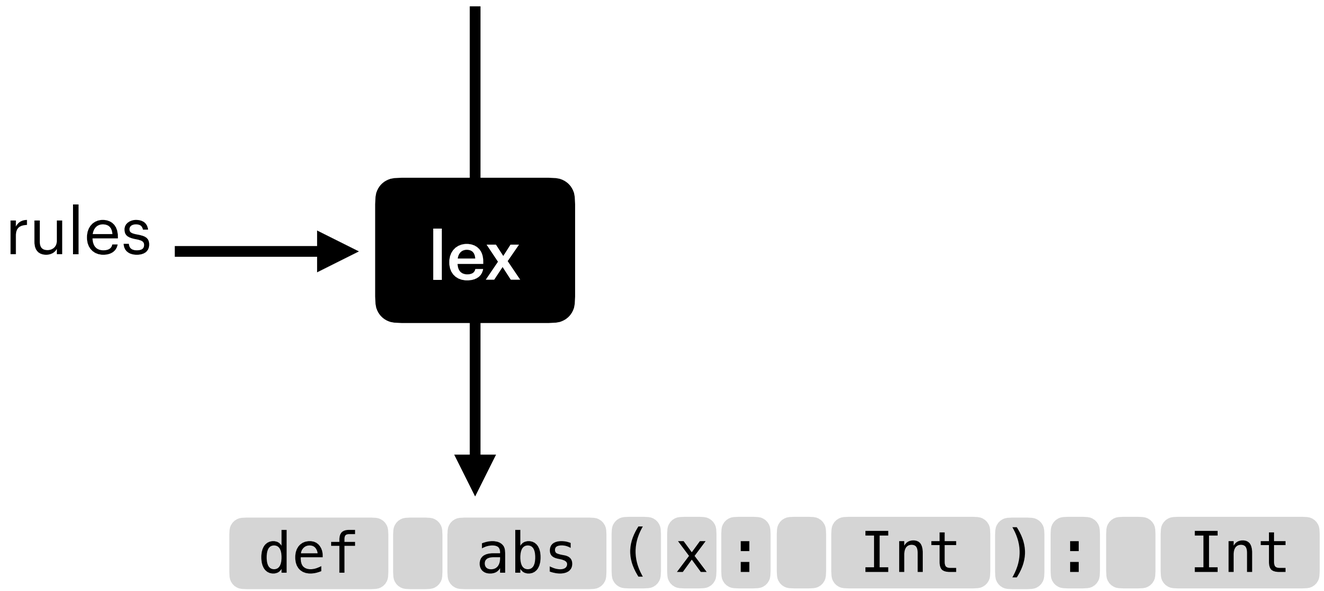
```
def abs(x: Int): Int
```



Lexical Analysis

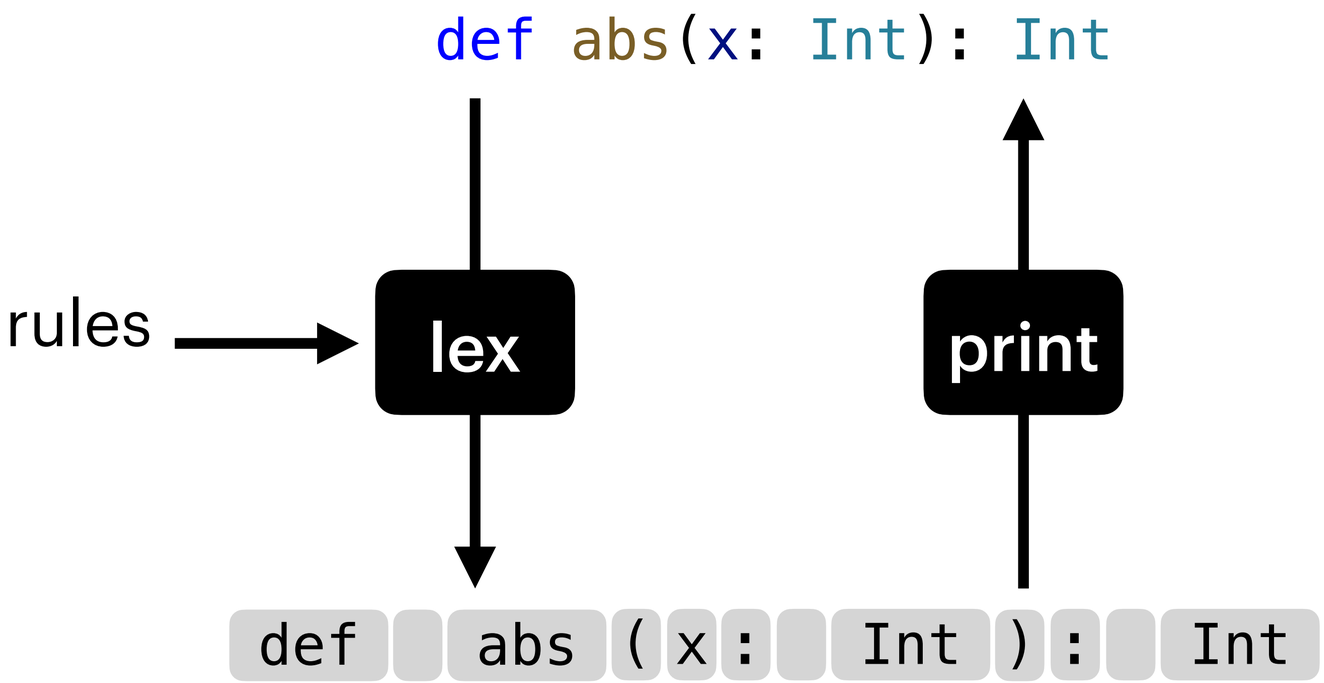
Ziplex Interface

```
def abs(x: Int): Int
```



Lexical Analysis

Ziplex Interface

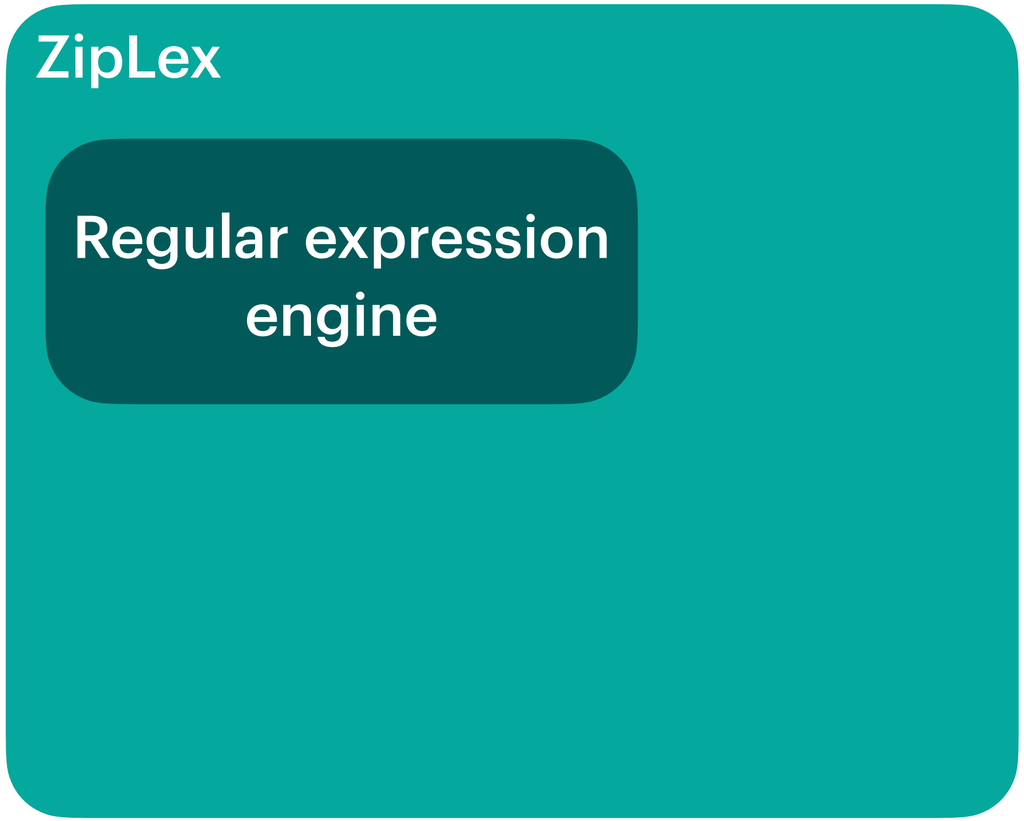


ZipLex Framework

Scala project, verified with Stainless

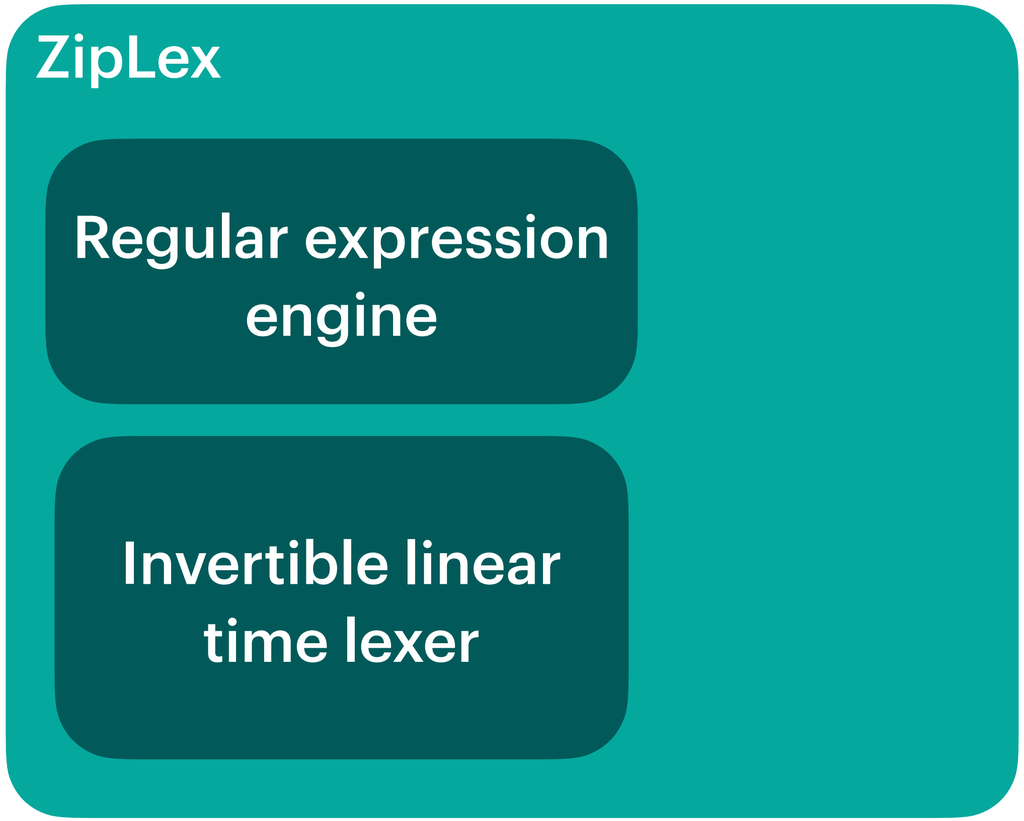
ZipLex Framework

Scala project, verified with Stainless



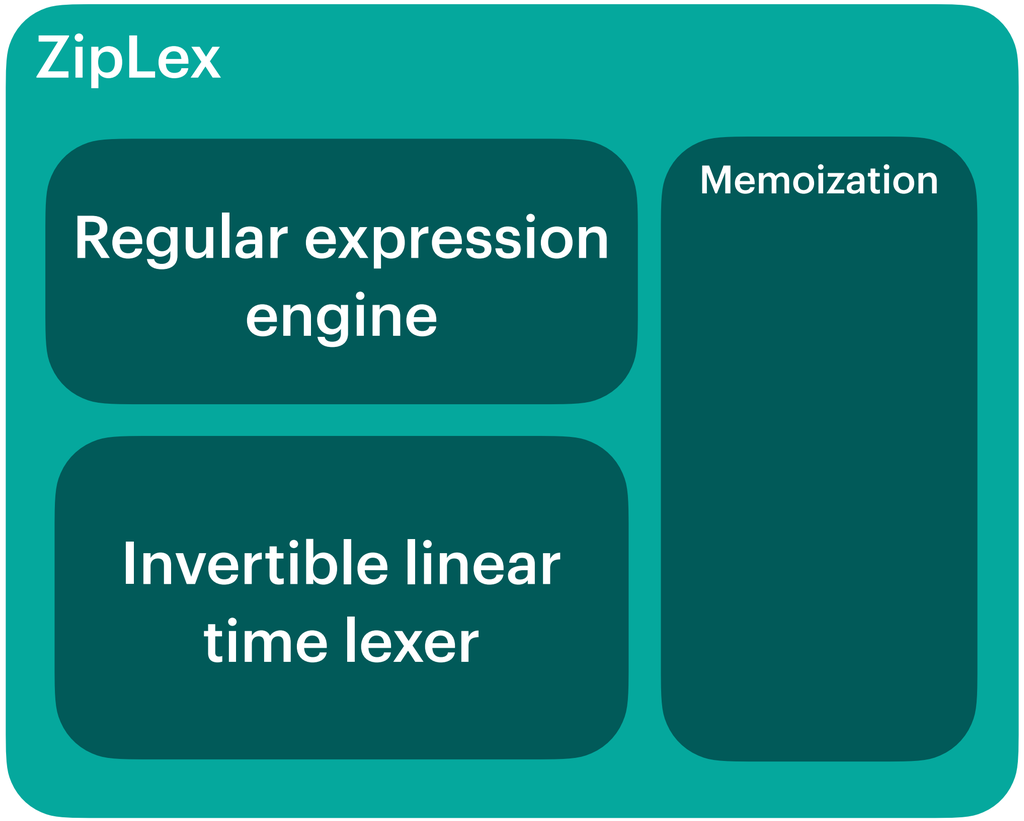
ZipLex Framework

Scala project, verified with Stainless



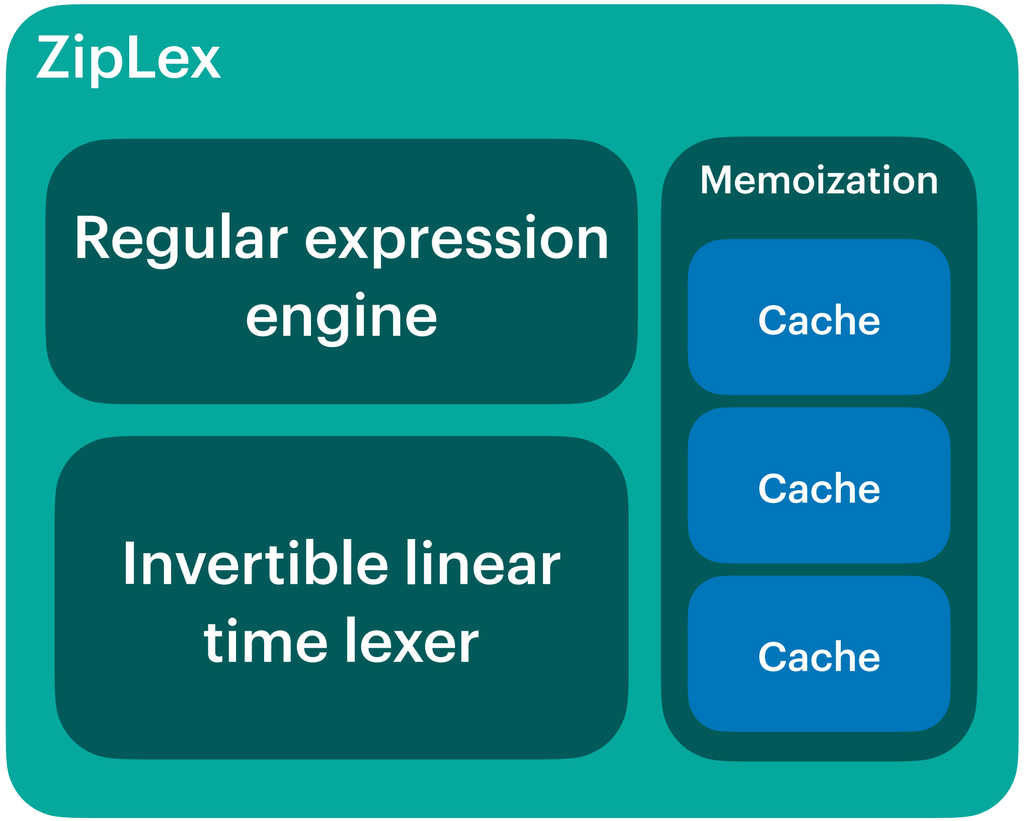
ZipLex Framework

Scala project, verified with Stainless



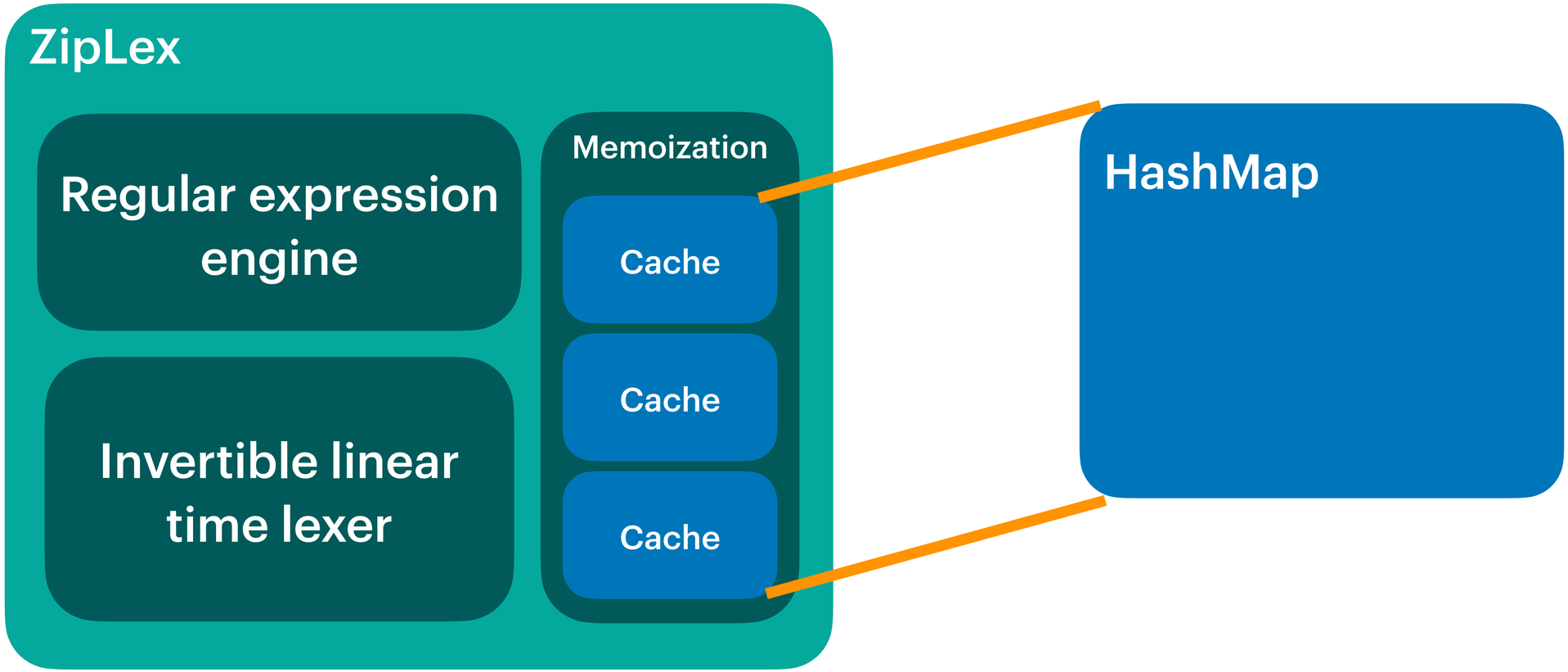
ZipLex Framework

Scala project, verified with Stainless



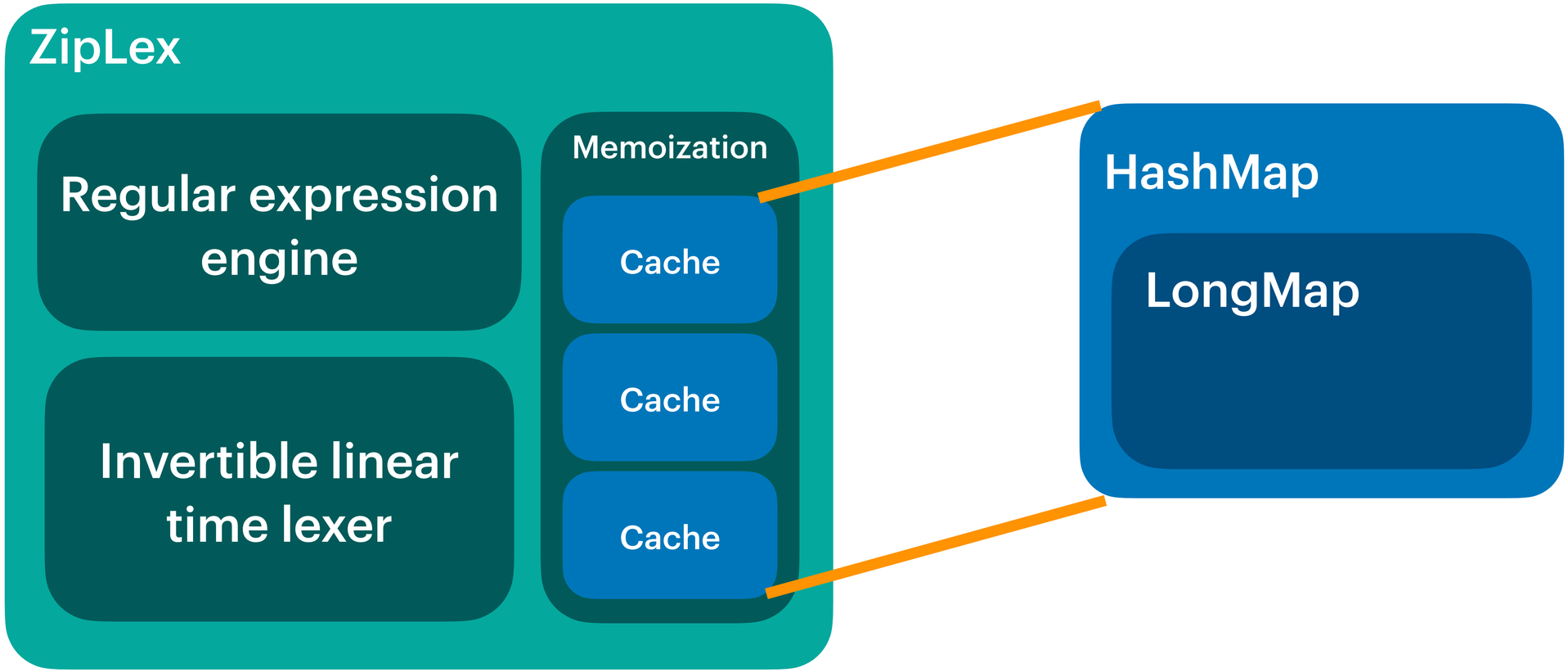
ZipLex Framework

Scala project, verified with Stainless



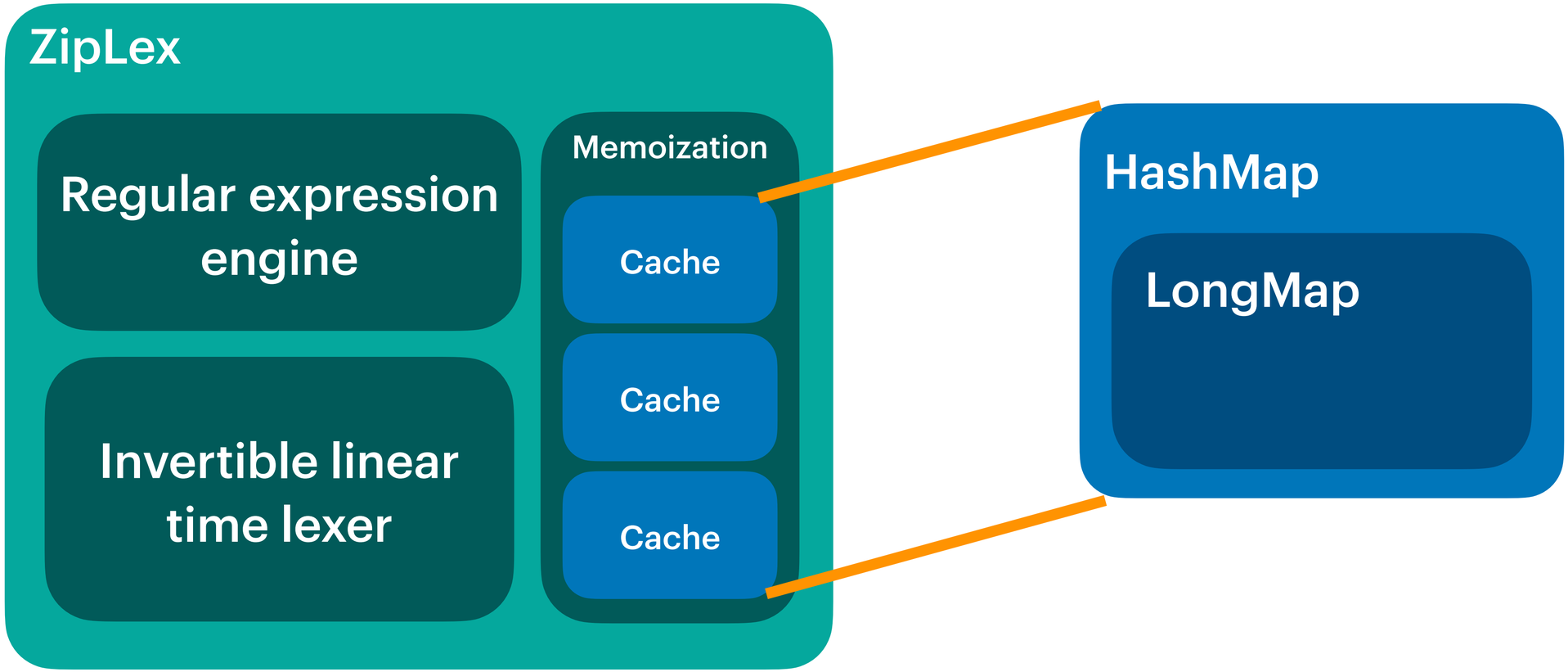
ZipLex Framework

Scala project, verified with Stainless



ZipLex Framework

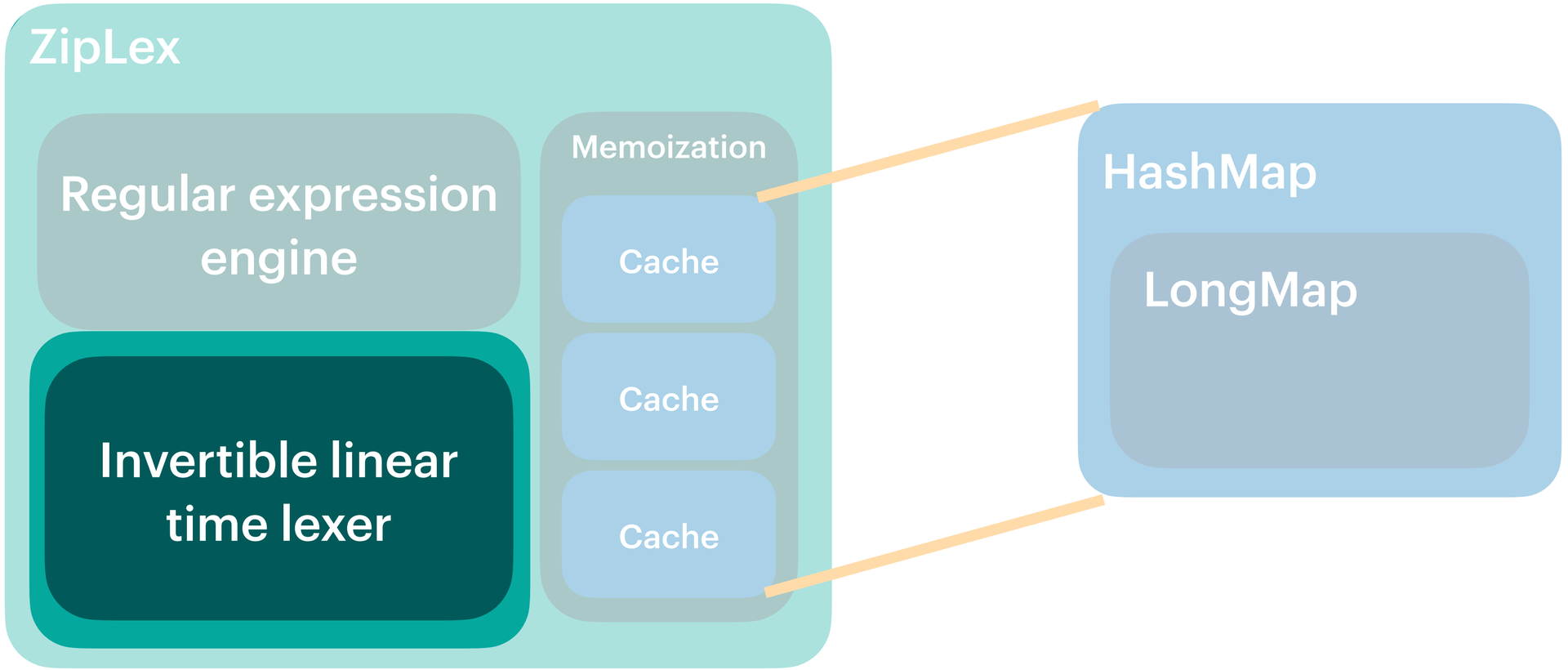
Scala project, verified with Stainless



Composition pattern for efficient programs verification

ZipLex Framework

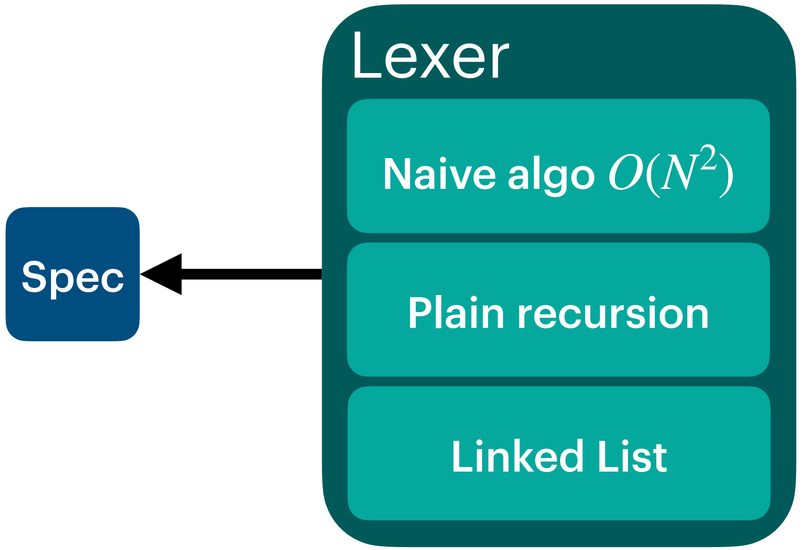
Scala project, verified with Stainless



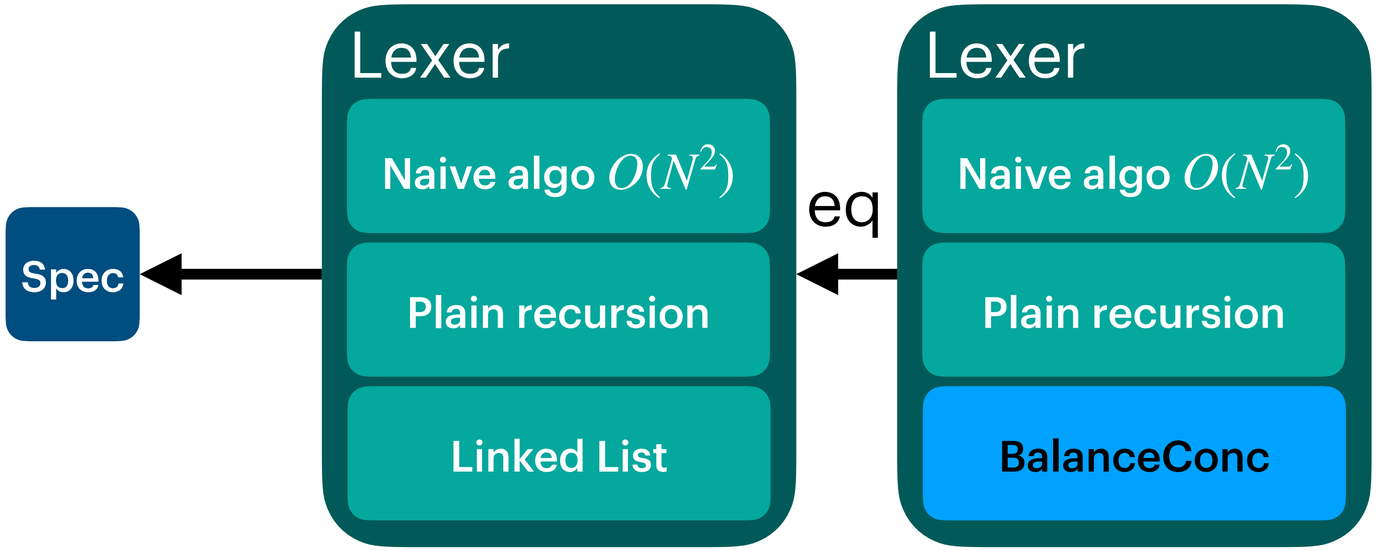
Composition pattern for efficient programs verification

Implementation and verification methodology

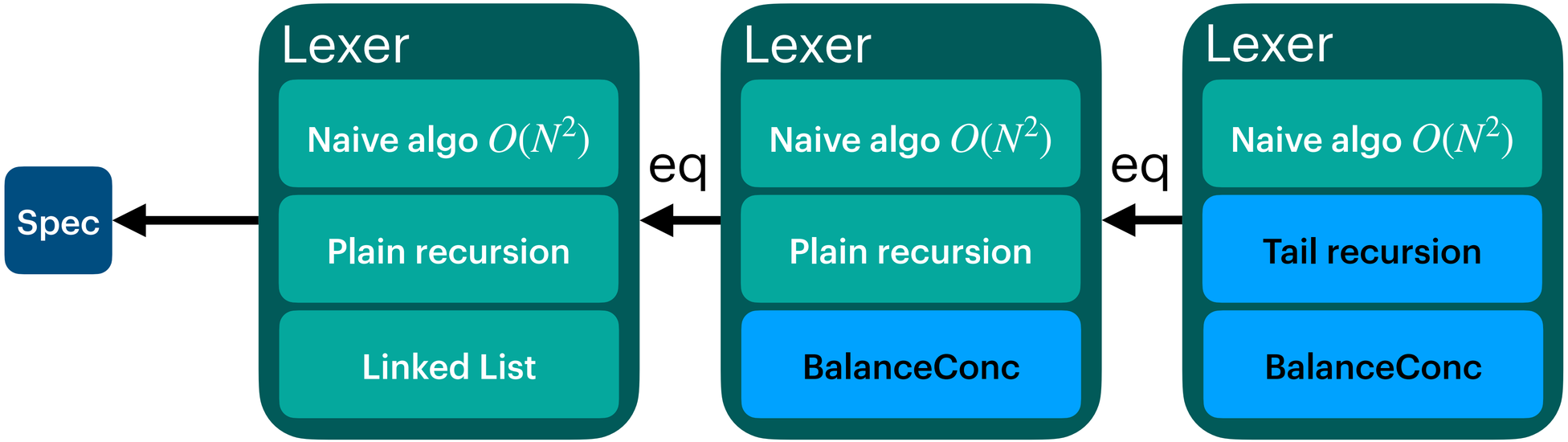
Implementation and verification methodology



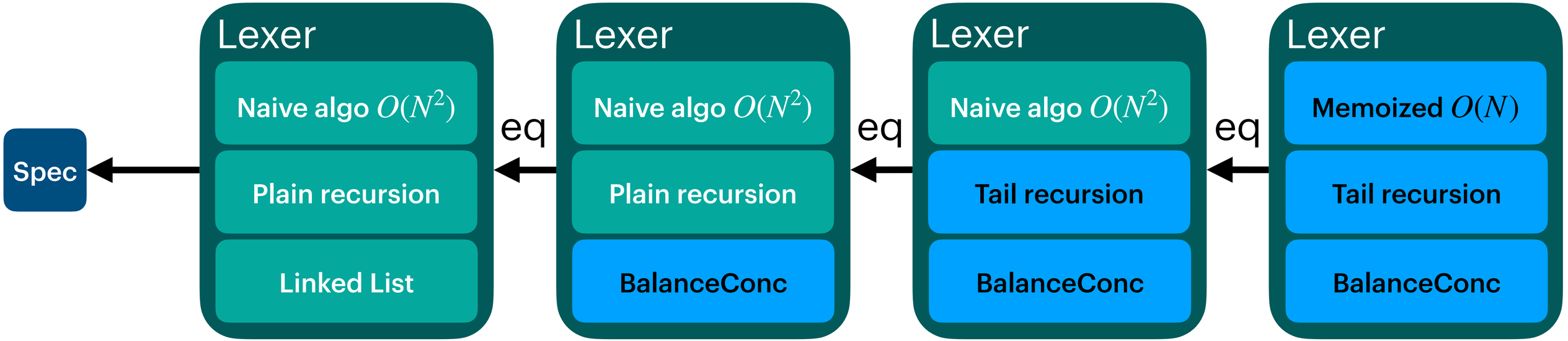
Implementation and verification methodology



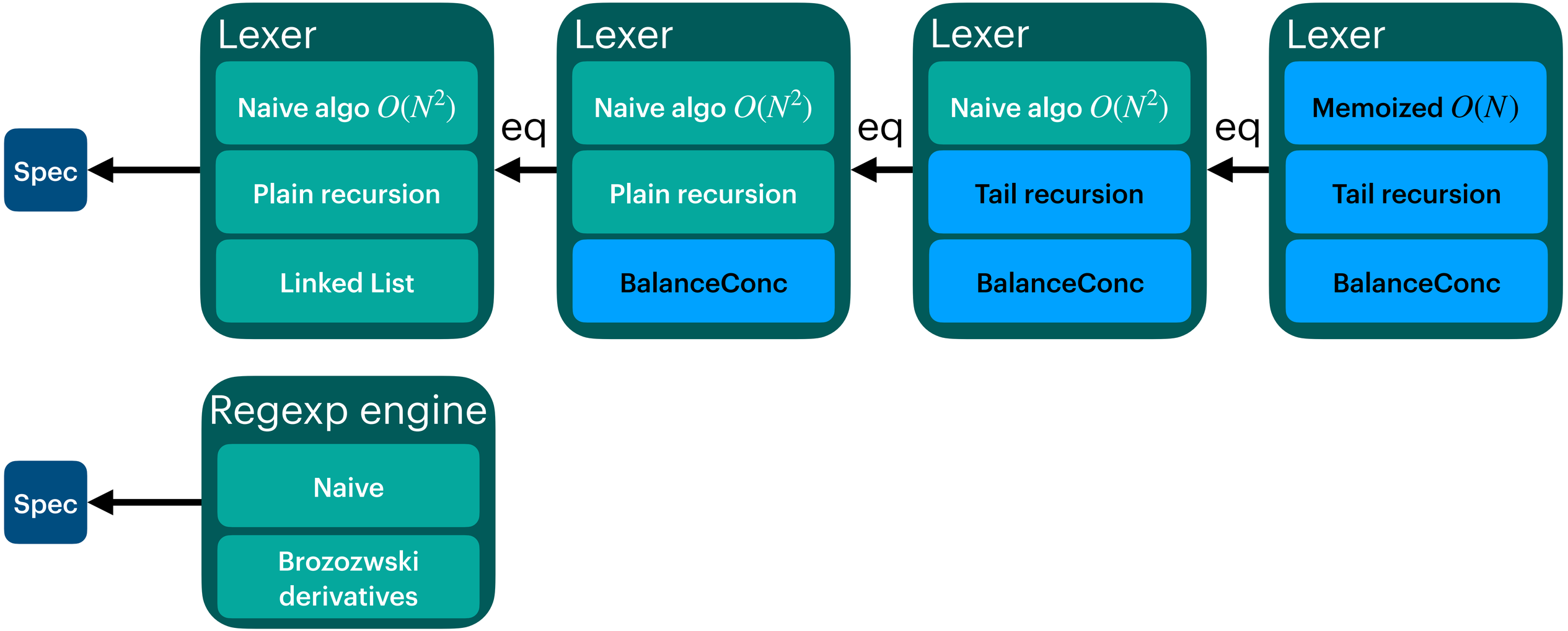
Implementation and verification methodology



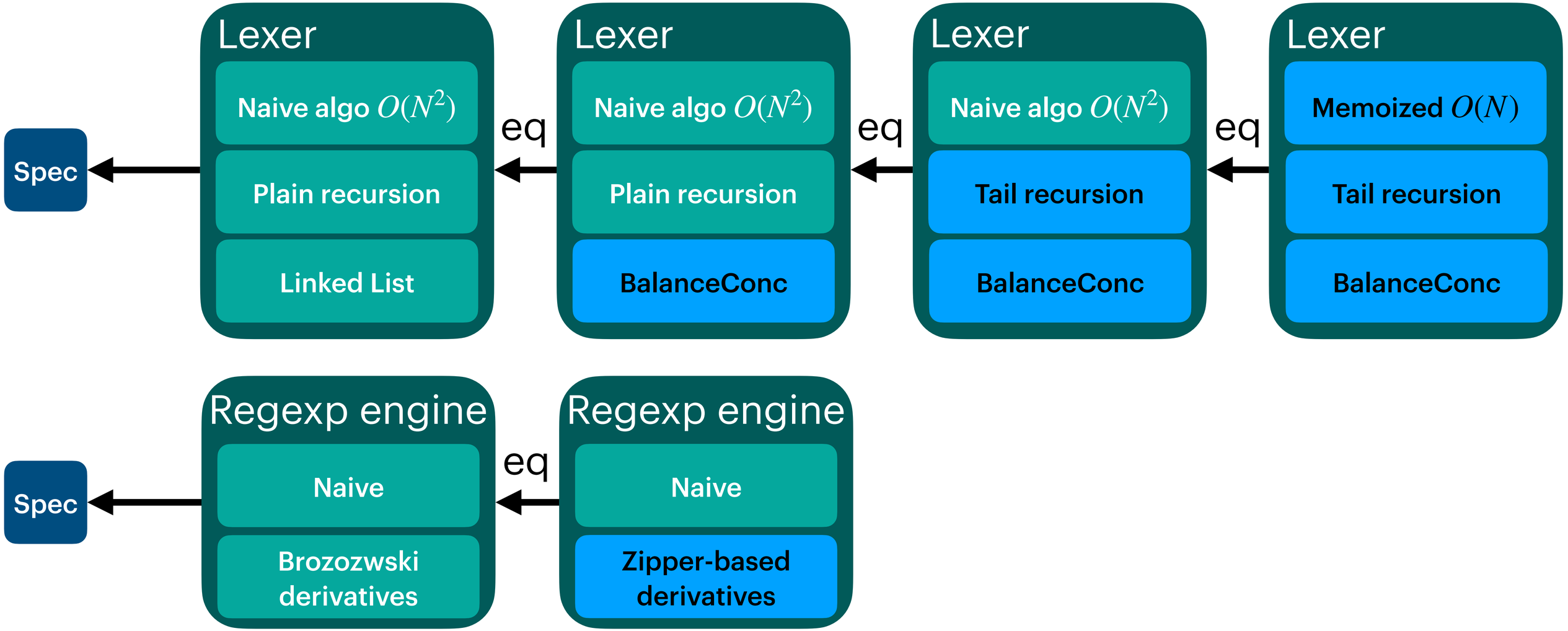
Implementation and verification methodology



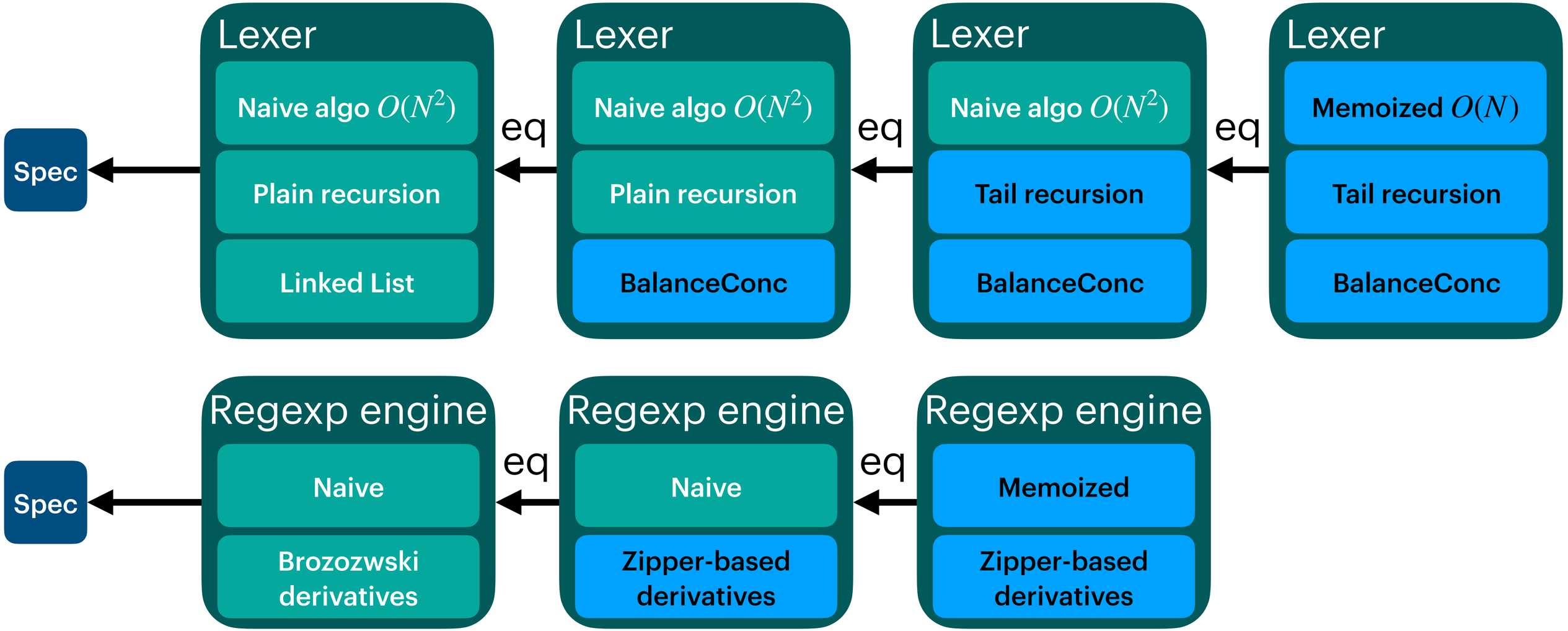
Implementation and verification methodology



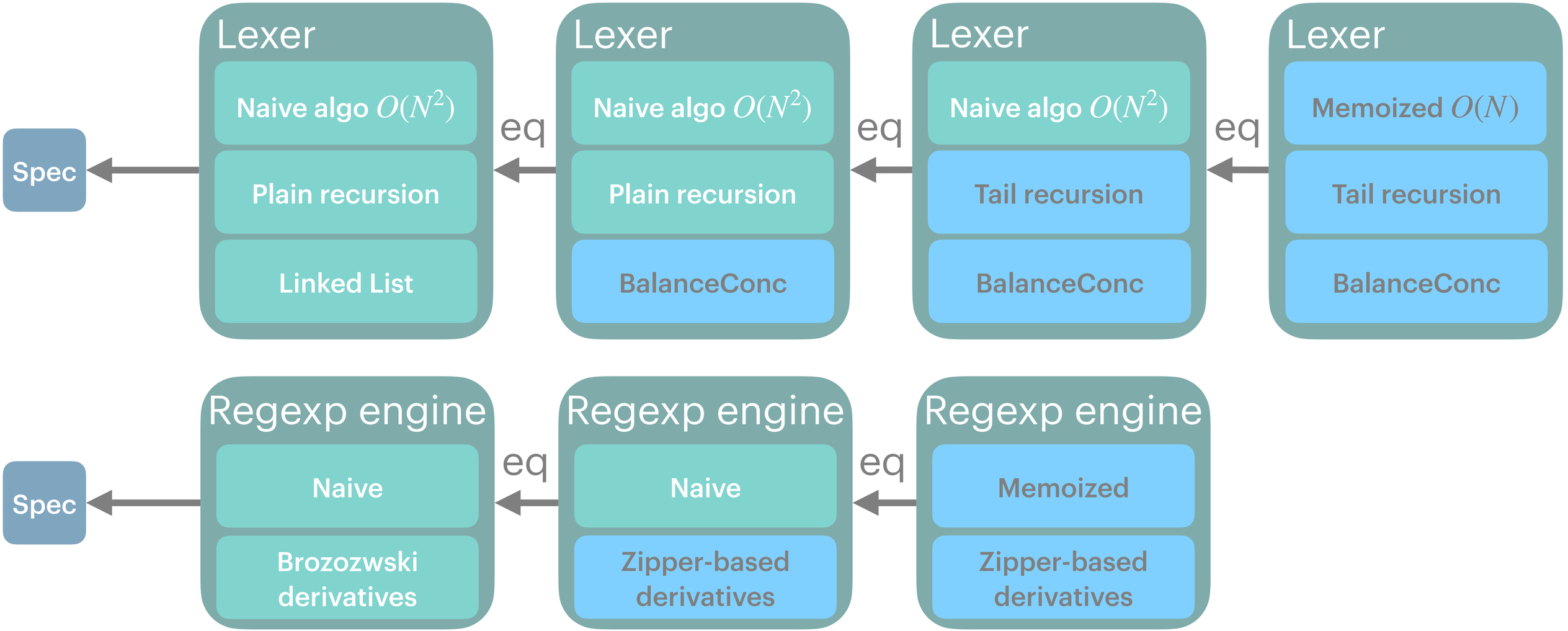
Implementation and verification methodology



Implementation and verification methodology



Implementation and verification methodology



Iterative approach to efficiently verify efficient programs

Specification

Introduction

Specification

Introduction

Longest match

Specification

Introduction

Longest match

Longest possible prefix for each token

Specification

Introduction

Longest match

Longest possible prefix for each token

Invertibility

Specification

Introduction

Longest match

Longest possible prefix for each token

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Specification

Introduction

Longest match

Longest possible prefix for each token

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Invertible lexer if both

Specification

Introduction

Longest match

Longest possible prefix for each token

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Invertible lexer if both

$$print(lex(s)) = s$$

Specification

Introduction

Longest match

Longest possible prefix for each token

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Invertible lexer if both

$$print(lex(s)) = s \qquad \text{and}$$

Specification

Introduction

Longest match

Longest possible prefix for each token

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Invertible lexer if both

$$print(lex(s)) = s \qquad \text{and} \qquad lex(print(ts)) = ts$$

Specification

Introduction

Longest match

Longest possible prefix for each token

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Invertible lexer if both

$$print(lex(s)) = s \quad \text{and} \quad lex(print(ts)) = ts$$

By induction

Specification

Introduction

Longest match

Longest possible prefix for each token

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Invertible lexer if both

$$print(lex(s)) = s \qquad \text{and} \qquad lex(print(ts)) = ts$$

Specification

Introduction

Longest match

Longest possible prefix for each token

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Invertible lexer if both

$$print(lex(s)) = s \qquad \text{and} \qquad lex(print(ts)) = ts$$

Lexing Invertibility

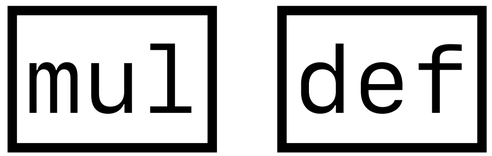
2. *lex*(*print*(ts)) = ts

Tokens can ***collapse***

Lexing Invertibility

2. *lex*(*print*(ts)) = ts

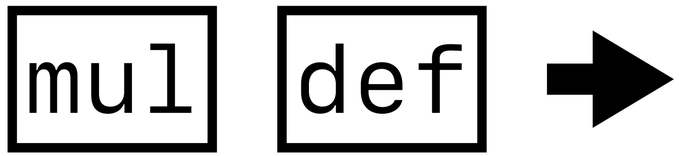
Tokens can ***collapse***



Lexing Invertibility

2. *lex*(*print*(ts)) = ts

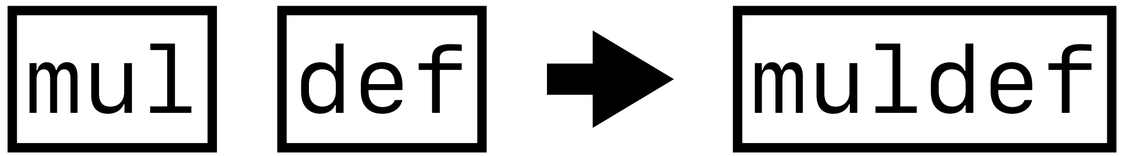
Tokens can ***collapse***



Lexing Invertibility

2. *lex*(*print*(ts)) = ts

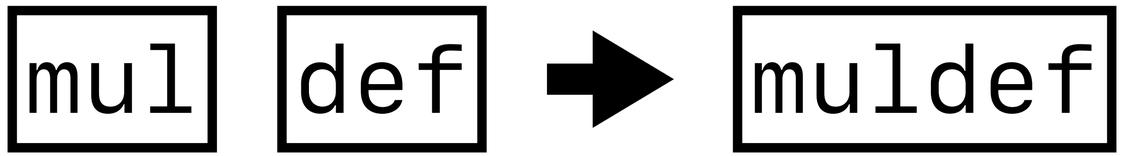
Tokens can ***collapse***



Lexing Invertibility

2. *lex*(*print*(ts)) = ts

Tokens can ***collapse***

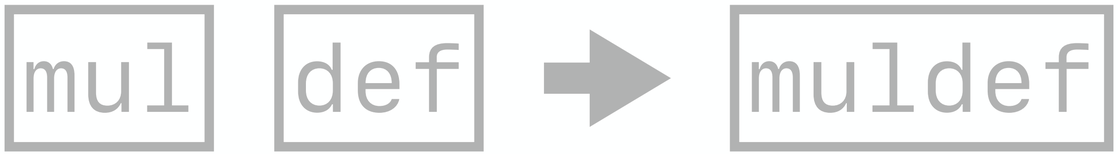


$lex(print(ts)) = ts$ NOT true in general

Lexing Invertibility

2. *lex*(*print*(ts)) = ts

Tokens can ***collapse***



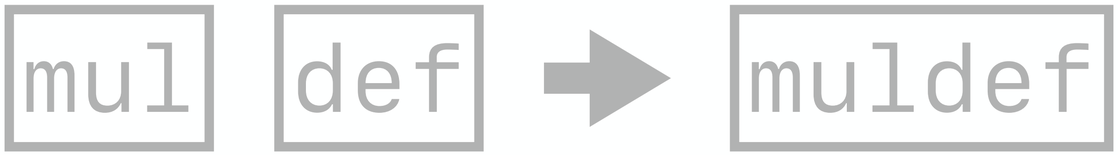
$lex(print(ts)) = ts$ NOT true in general

We define the notion of ***separability***

Lexing Invertibility

2. *lex*(*print*(ts)) = ts

Tokens can ***collapse***



$$lex(print(ts)) = ts$$ NOT true in general

We define the notion of ***separability***

$$sep(ts) \implies lex(print(ts)) = ts$$

Token Separability

Predicate definition

Token Separability

Predicate definition

In general, $sep(ts)$ depends on

Token Separability

Predicate definition

In general, $sep(ts)$ depends on

1. ALL the rules

Token Separability

Predicate definition

In general, $sep(ts)$ depends on

1. ALL the rules
2. ALL tokens in the sequence

Token Separability

Predicate definition

In general, $sep(ts)$ depends on

1. ALL the rules
2. ALL tokens in the sequence

We need a predicate that is efficient to compute

Token Separability

Predicate definition

In general, $sep(ts)$ depends on

1. ALL the rules
2. ~~ALL tokens in the sequence~~

   Look only at adjacent tokens

We need a predicate that is efficient to compute

R-Paths

Introduction

Given a binary relation R, a sequence $s = s_1, s_2, \ldots, s_n$ is an *R-Path* if the relation holds for each consecutive pair

R-Paths

Introduction

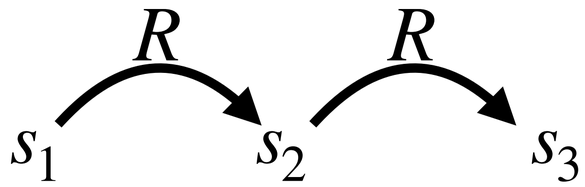
Given a binary relation R, a sequence $s = s_1, s_2, \ldots, s_n$ is an *R-Path* if the relation holds for each consecutive pair

s_1 s_2 s_3

R-Paths

Introduction

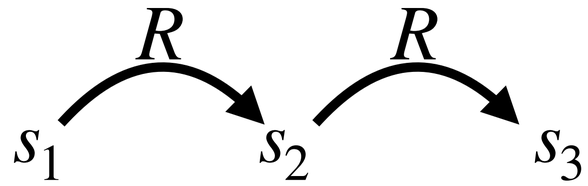
Given a binary relation R, a sequence $s = s_1, s_2, \ldots, s_n$ is an *R-Path* if the relation holds for each consecutive pair



R-Paths

Introduction

Given a binary relation R, a sequence $s = s_1, s_2, \ldots, s_n$ is an *R-Path* if the relation holds for each consecutive pair



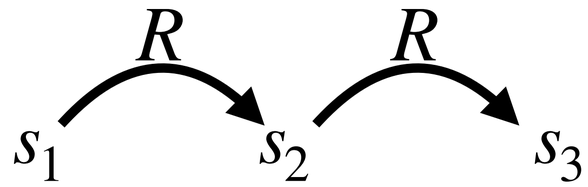
Formally, *R-Path predicate*

$$p_R(s) = \forall i,\ 1 \leq i < |s| - 1 \implies R(s_i, s_{i+1})$$

R-Paths

Introduction

Given a binary relation R, a sequence $s = s_1, s_2, \ldots, s_n$ is an *R-Path* if the relation holds for each consecutive pair



Formally, *R-Path predicate*

$$p_R(s) = \forall i,\ 1 \leq i < |s| - 1 \implies R(s_i, s_{i+1})$$

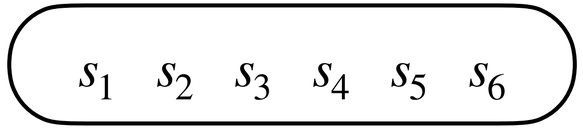
Predicates over sequences requiring only local computations

R-Path

Efficient operations with statically checked invariant

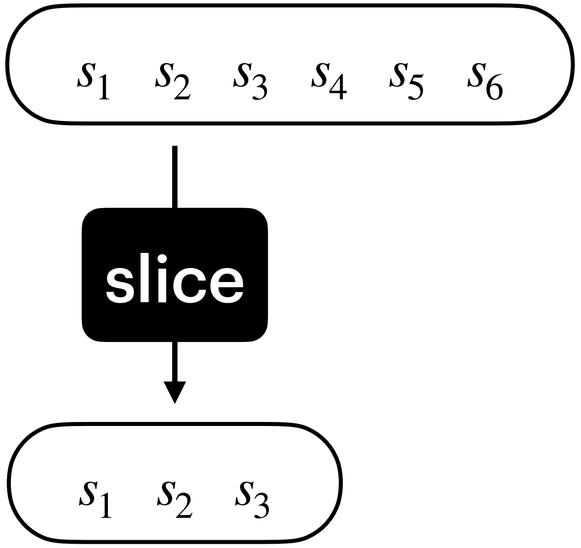
R-Path

Efficient operations with statically checked invariant



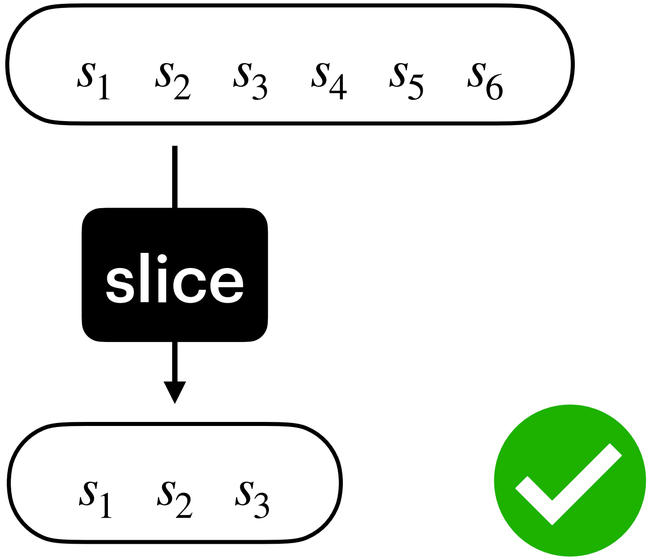
R-Path

Efficient operations with statically checked invariant



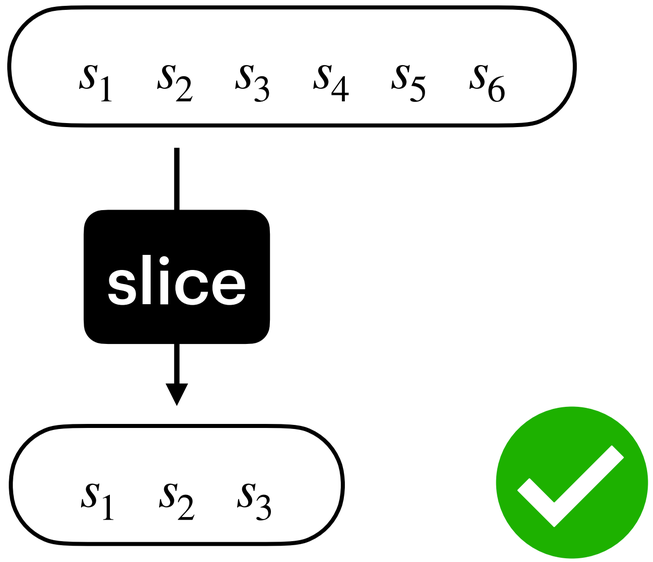
R-Path

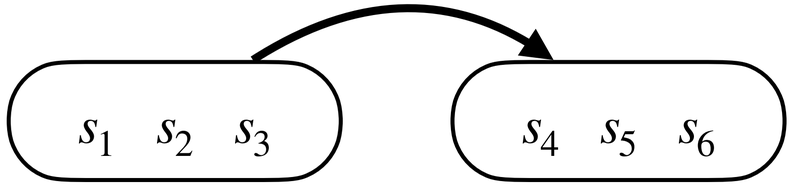
Efficient operations with statically checked invariant



R-Path

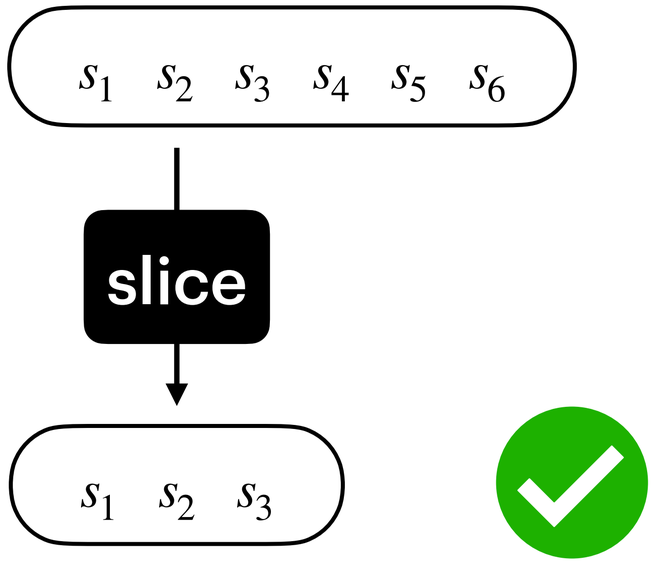
Efficient operations with statically checked invariant

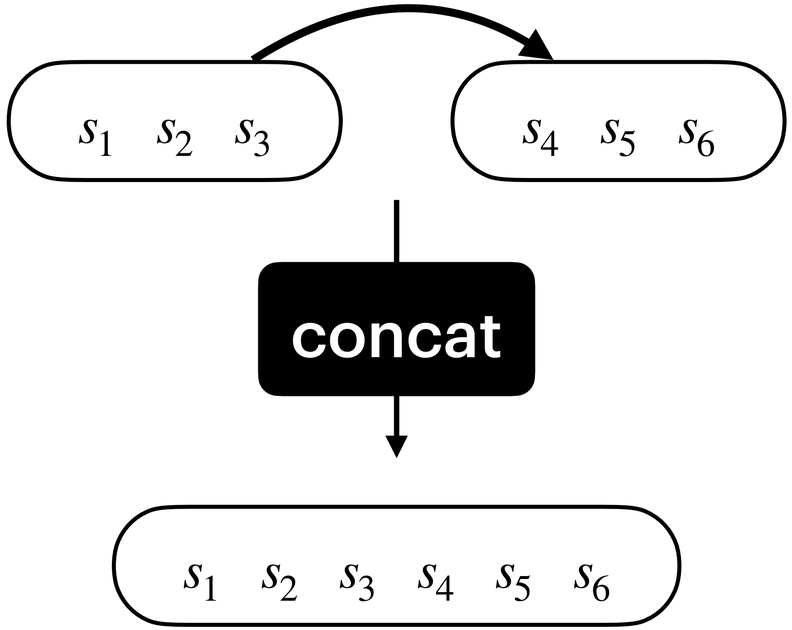




R-Path

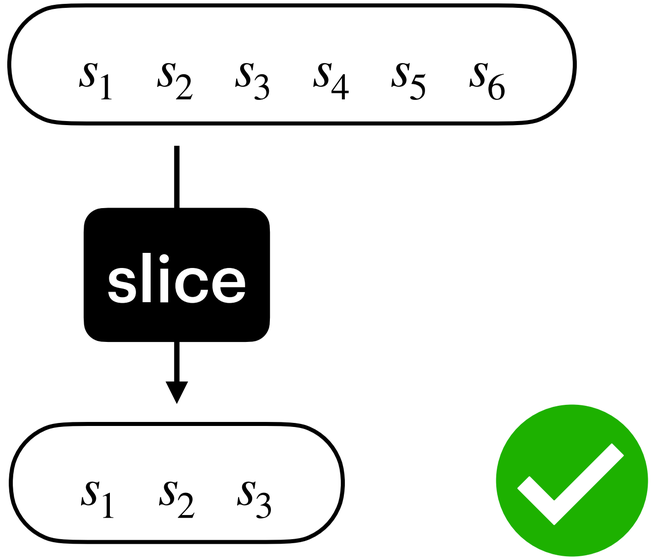
Efficient operations with statically checked invariant

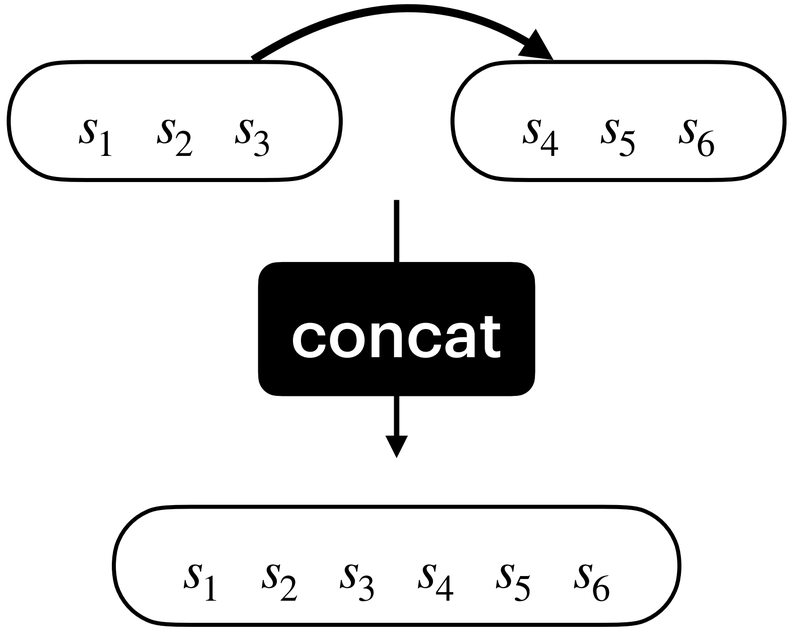




R-Path

Efficient operations with statically checked invariant

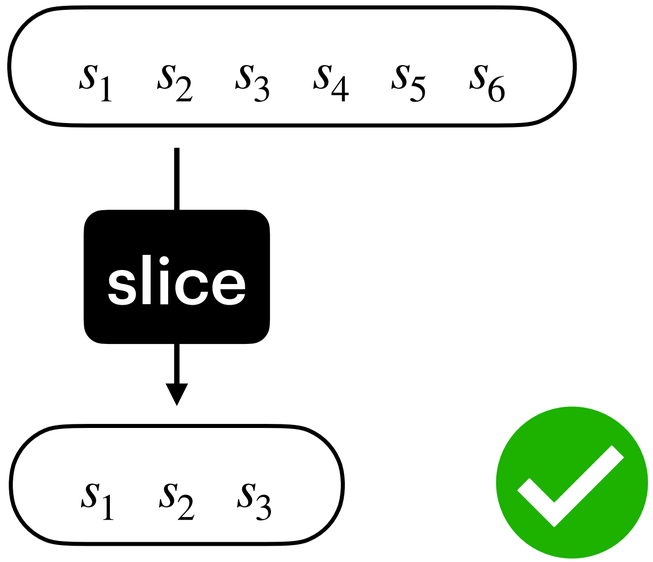


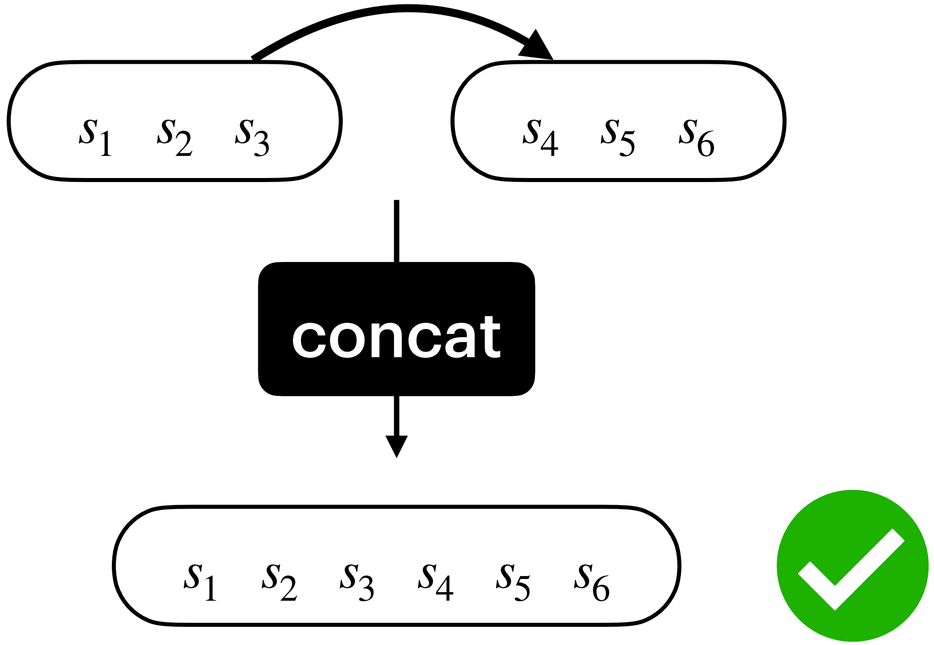


if $R(s_3, s_4)$ holds

R-Path

Efficient operations with statically checked invariant

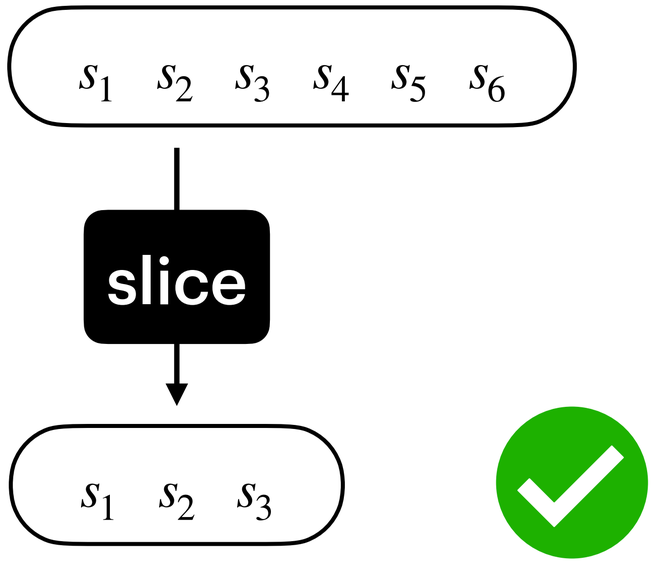


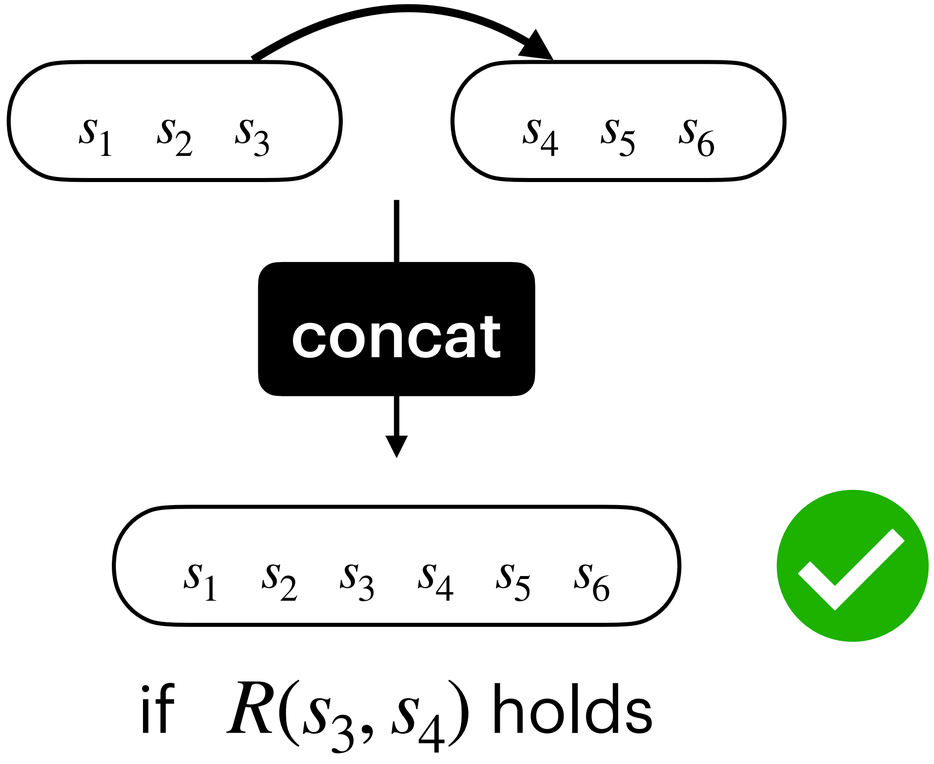


if $R(s_3, s_4)$ holds

R-Path

Efficient operations with statically checked invariant





if $R(s_3, s_4)$ holds

Efficient to manipulate with a mix of static and runtime checks

Separability

As R-Paths

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right).\mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

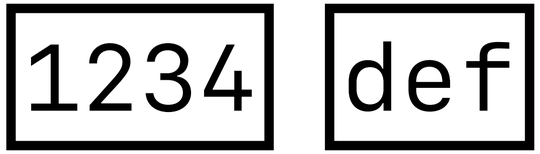
1234

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

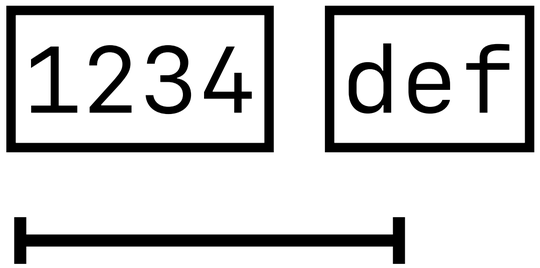


Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

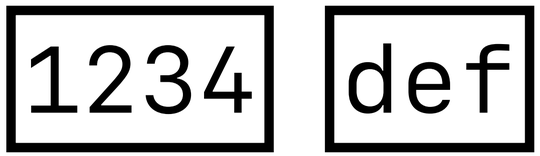


Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*



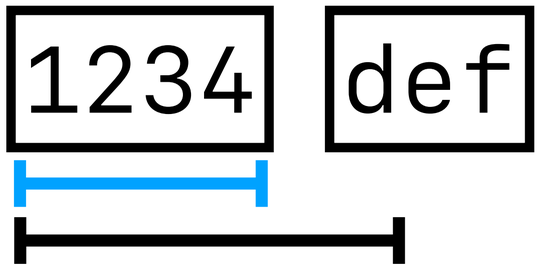
No rules match any string
starting with 1234d

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*



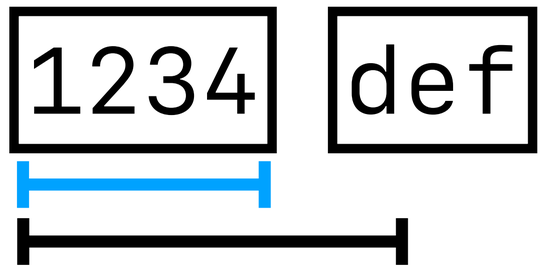
No rules match any string
starting with 1234d

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*



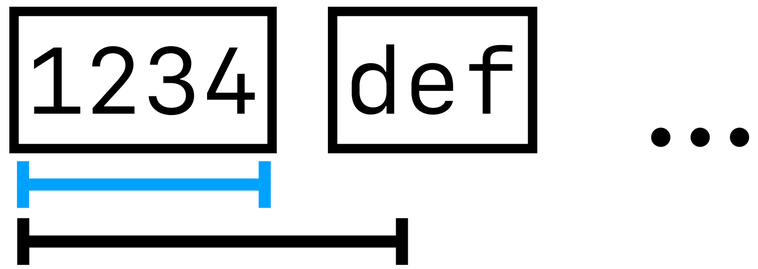
No rules match any string
starting with 1234d

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*



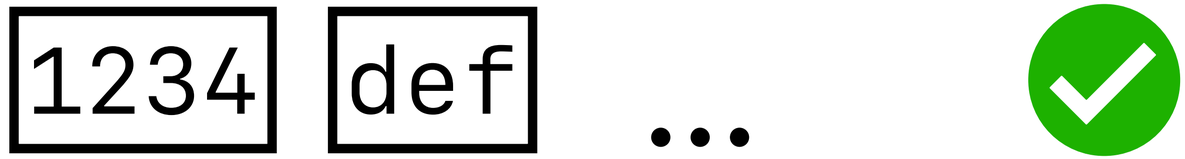
No rules match any string
starting with 1234d

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

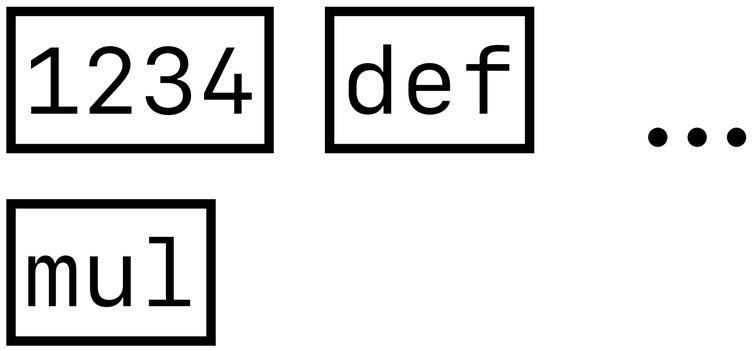


Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

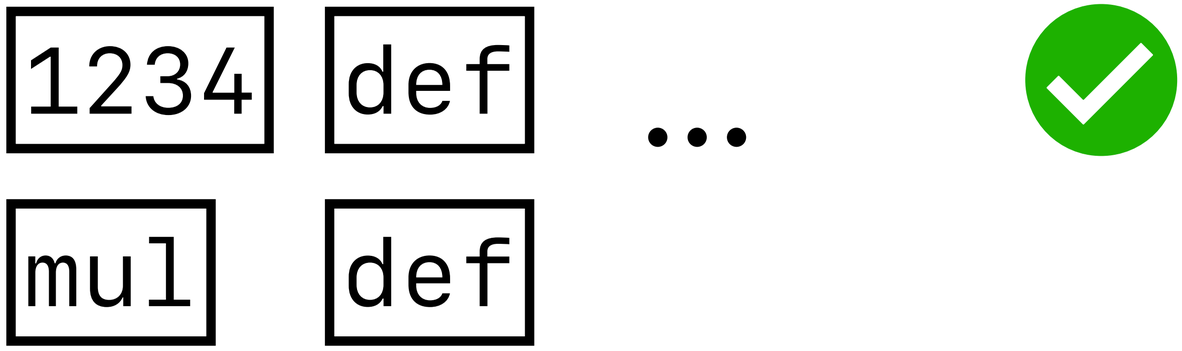


Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

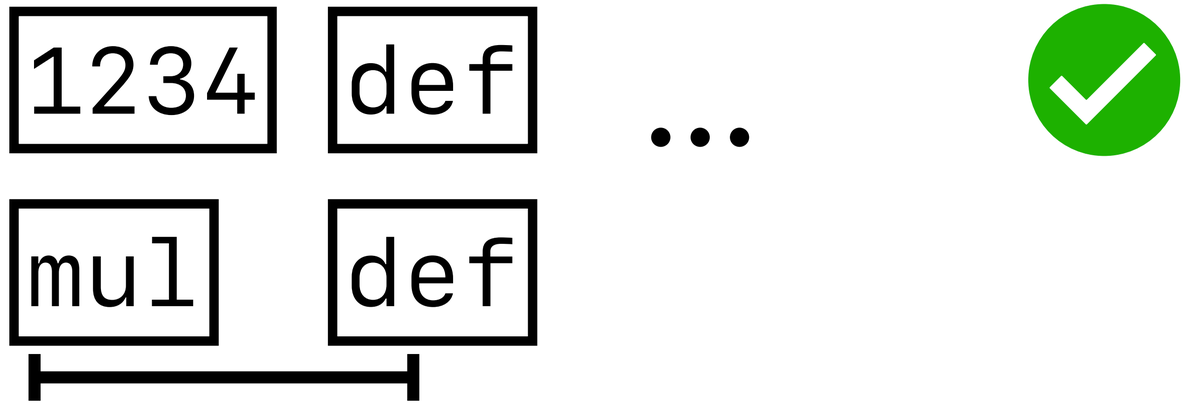


Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*



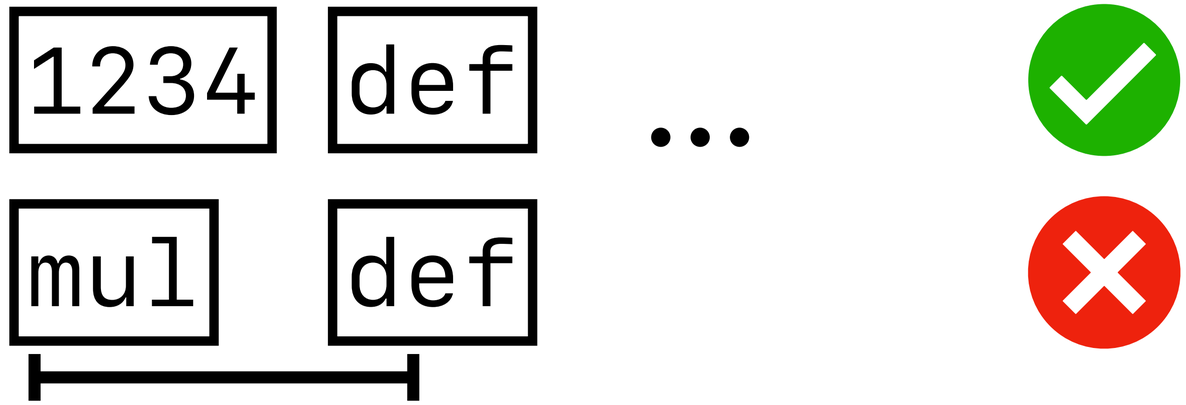
Identifier rule matches strings starting with `muld`

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*



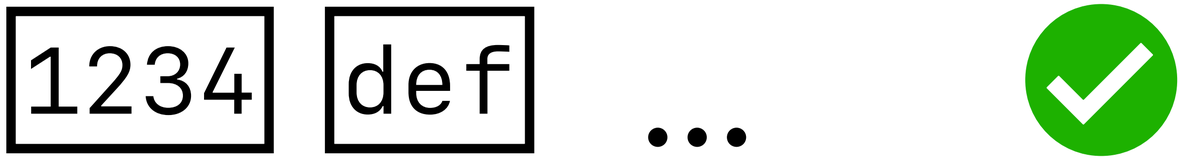
Identifier rule matches strings starting with `muld`

Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

r.prefixMatch(s): true if *r* matches a string starting with *s*

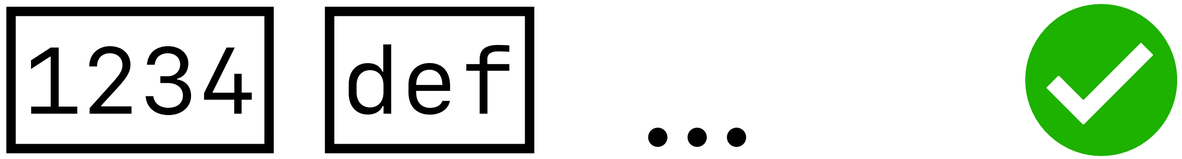


Separability

As R-Paths

$$R_{sep}(t_1, t_2) := \neg\left(\bigcup_{i \in m} r_i\right) . \mathit{prefixMatch}(t_1 ::: t_2(0))$$

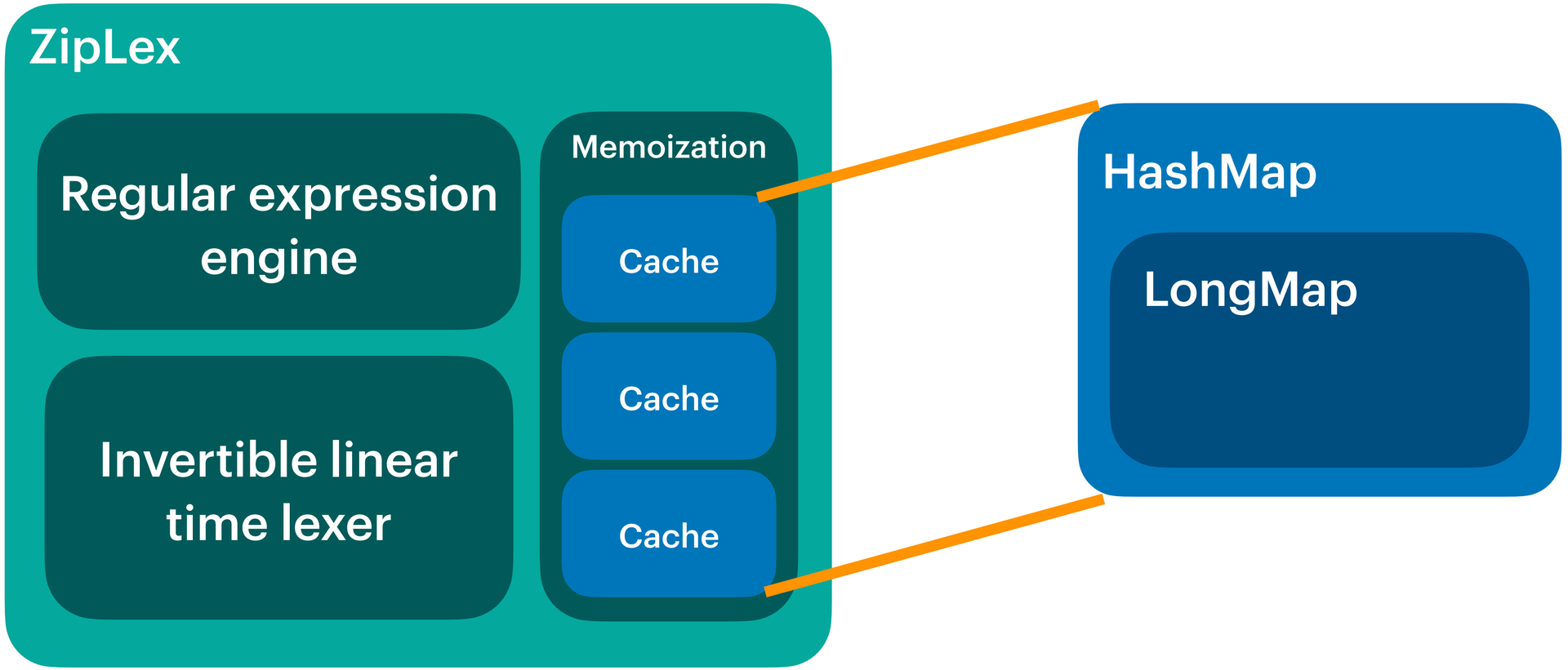
r.prefixMatch(s): true if *r* matches a string starting with *s*



When 2 tokens are separable, the longest prefix by any rule is always the first token

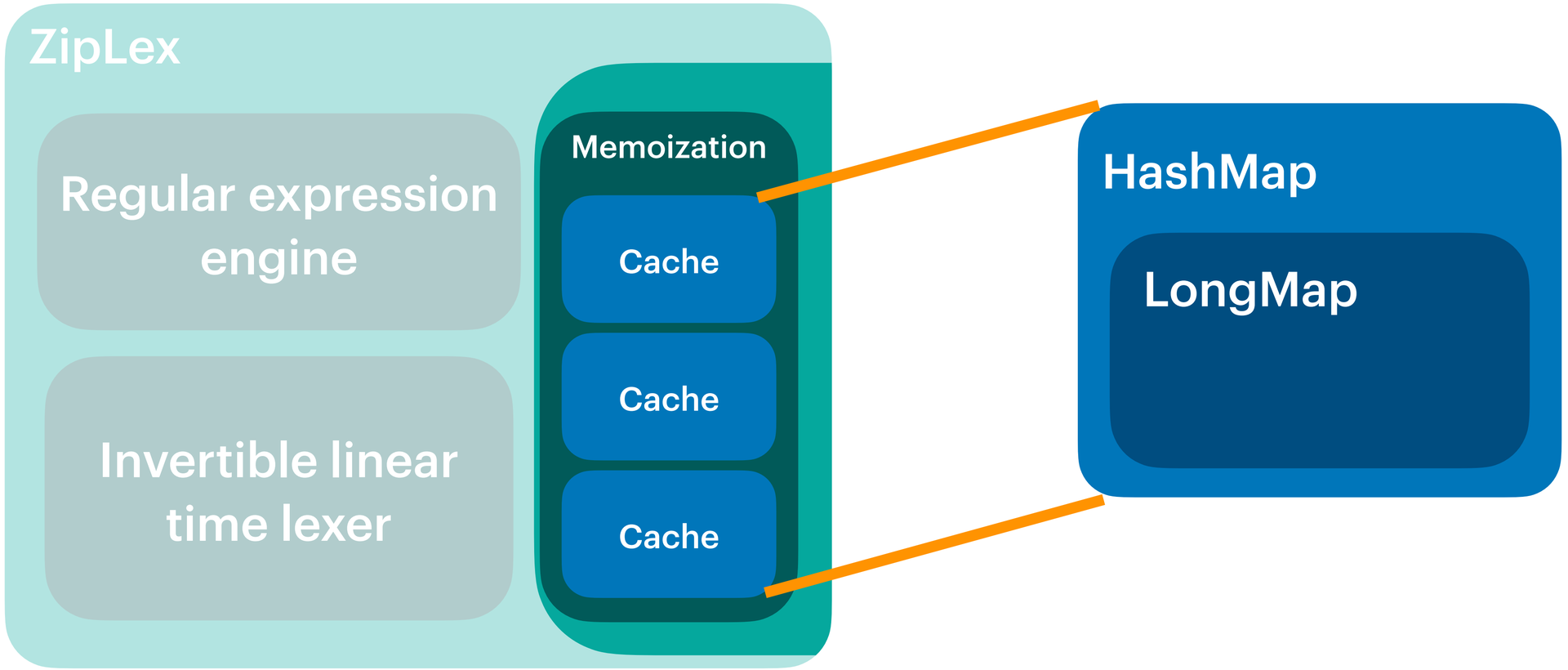
ZipLex Framework

Overview



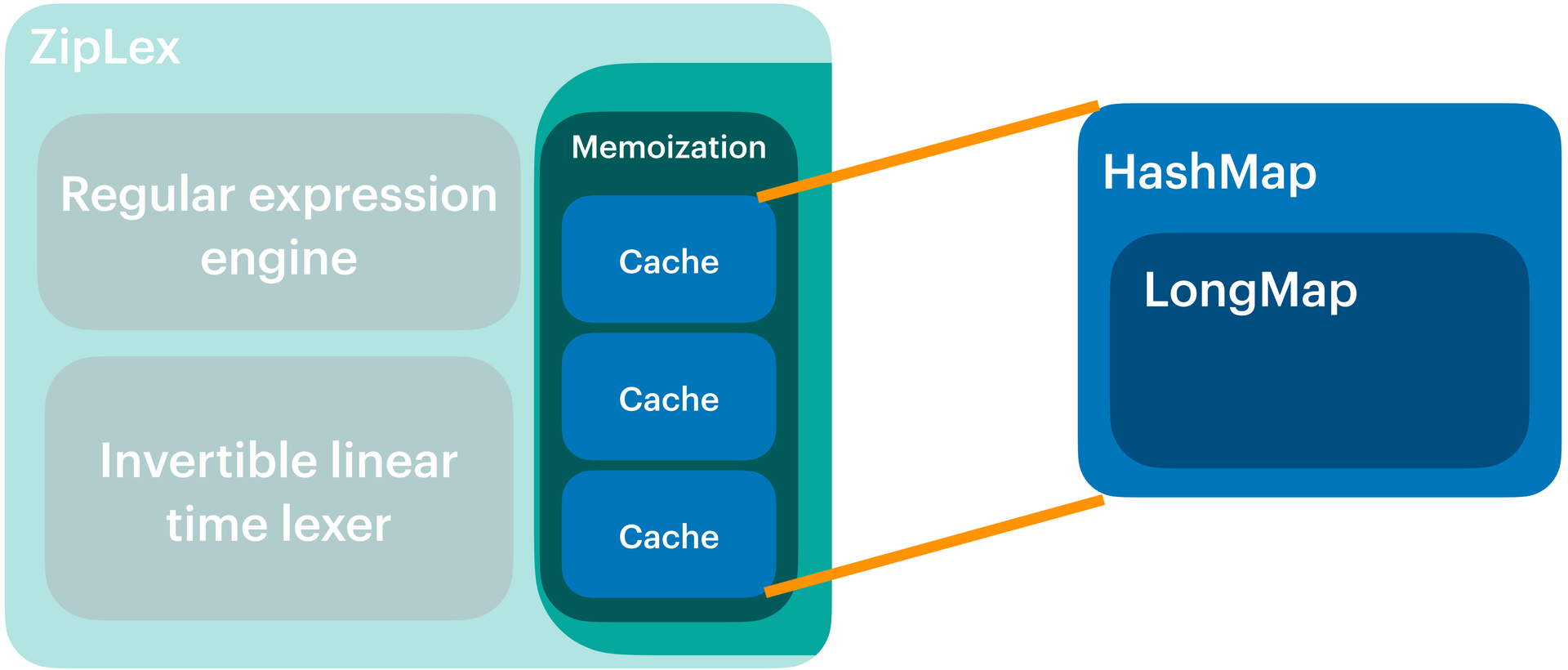
ZipLex Framework

Overview



ZipLex Framework

Overview



Reusable Verified Memoization Framework

Implementation

Reuse LongMap[1]

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

Implementation

Reuse LongMap[1]

LongMap[List[(K, V)]]

- **Verified mutable** Hash Table

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

Implementation

Reuse LongMap[1]

LongMap[List[(K, V)]]

- **Verified mutable** Hash Table
- **Open addressing**, non-linear probing

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

Implementation

Reuse LongMap[1]

LongMap[List[(K, V)]]

- **Verified mutable** Hash Table
- **Open addressing**, non-linear probing
- From Scala standard library

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

Implementation

Reuse LongMap[1]

LongMap[List[(K, V)]]

- **Verified mutable** Hash Table
- **Open addressing**, non-linear probing
- From Scala standard library
- **64-bit integer** keys

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

Implementation

Reuse LongMap[1]

HashMap[K, V]

LongMap[List[(K, V)]]

- **Verified mutable** Hash Table
- **Open addressing**, non-linear probing
- From Scala standard library
- **64-bit integer** keys

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

Implementation

Reuse LongMap[1]

HashMap[K, V]

LongMap[List[(K, V)]]

- **Verified mutable** Hash Table
- **Open addressing**, non-linear probing
- From Scala standard library
- **64-bit integer** keys

- **Generically typed** keys

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

Implementation

Reuse LongMap[1]

HashMap[K, V]

LongMap[List[(K, V)]]

- **Verified mutable** Hash Table
- **Open addressing**, non-linear probing
- From Scala standard library
- **64-bit integer** keys

- **Generically typed** keys
- **Fully verified**

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

Implementation

Reuse LongMap[1]

HashMap[K, V]

LongMap[List[(K, V)]]

- **Verified mutable** Hash Table
- **Open addressing**, non-linear probing
- From Scala standard library
- **64-bit integer** keys

- **Generically typed** keys
- **Fully verified**
- Lemmas to verify memoization of **any pure function**

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

Implementation

Reuse LongMap[1]

HashMap[K, V]

LongMap[List[(K, V)]]

- **Verified mutable** Hash Table
- **Open addressing**, non-linear probing
- From Scala standard library
- **64-bit integer** keys

- **Generically typed** keys
- **Fully verified**
- Lemmas to verify memoization of **any pure function**
- Around **2800 LOC** vs more than 8000 for LongMap

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

Implementation

Reuse LongMap[1]

HashMap[K, V]

LongMap[List[(K, V)]]

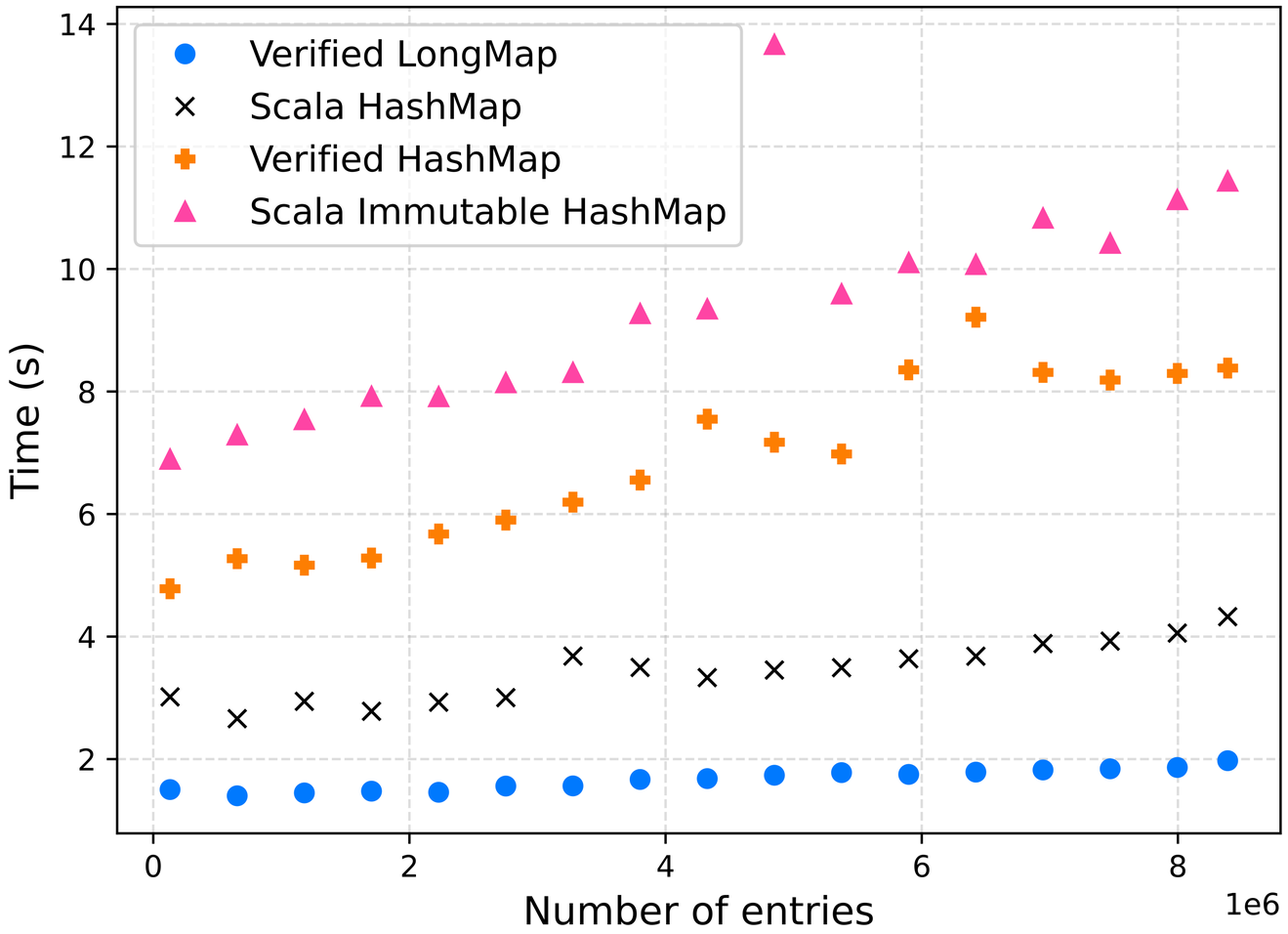
- **Verified mutable** Hash Table
- **Open addressing**, non-linear probing
- From Scala standard library
- **64-bit integer** keys

- **Generically typed** keys
- **Fully verified**
- Lemmas to verify memoization of **any pure function**
- Around **2800 LOC** vs more than 8000 for LongMap

Composition allowed for reuse and saved resources

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

Hash Tables Performance



⇒ Our **verified mutable** hash table outperforms the Scala library **unverified immutable** map

Insert N pairs, remove N/2, insert N

Memoization

Regular expression matching and lexing

Memoization

Regular expression matching and lexing

Regular expression engine

Memoization

Regular expression matching and lexing

Regular expression engine

Memoized zipper-**derivatives computation** yielding **DFAs**

Memoization

Regular expression matching and lexing

Regular expression engine

Memoized zipper-**derivatives computation** yielding **DFAs**

Lexing

Memoization

Regular expression matching and lexing

Regular expression engine

Memoized zipper-**derivatives computation** yielding **DFAs**

Lexing

Memoization enables **linear complexity** lexing (Reps 1998)[1]

[1]**Reps**, Thomas. 1998. **"'Maximal-Munch' Tokenization in Linear Time."** *ACM Trans. Program. Lang. Syst.* 20 (2): 259–73. https://doi.org/10.1145/276393.276394.

Memoization

Regular expression matching and lexing

Regular expression engine

Memoized zipper-**derivatives computation** yielding **DFAs**

Lexing

Memoization enables **linear complexity** lexing (Reps 1998)[1]

Longest prefix computation

[1]**Reps**, Thomas. 1998. **"'Maximal-Munch' Tokenization in Linear Time."** *ACM Trans. Program. Lang. Syst.* 20 (2): 259–73. https://doi.org/10.1145/276393.276394.

Project Scale

Scala project formally verified using Stainless

Module	**Program**	**Spec + Proof**	**Total**
Longest Match Lexing	580	2030	2610
Lexer Printing + Invertibility	130	1430	1560
Regex Derivative Based Matching	470	2440	2910
Zipper Based Matching	390	2670	3060
BalanceConc	255	320	575
List & Set lemmas	-	1610	1610
Mutable HashMap (excluding LongMap)	90	2700	2790
Total	**1'915**	**13'200**	**15'115**

Project Scale

Scala project formally verified using Stainless

Module	**Program**	**Spec + Proof**	**Total**
Longest Match Lexing	580	2030	2610
Lexer Printing + Invertibility	130	1430	1560
Regex Derivative Based Matching	470	2440	2910
Zipper Based Matching	390	2670	3060
BalanceConc	255	320	575
List & Set lemmas	-	1610	1610
Mutable HashMap (excluding LongMap)	90	2700	2790
Total	**1'915**	**13'200**	**15'115**
Mutable LongMap	410	8200	8610

Performance analysis

Performance

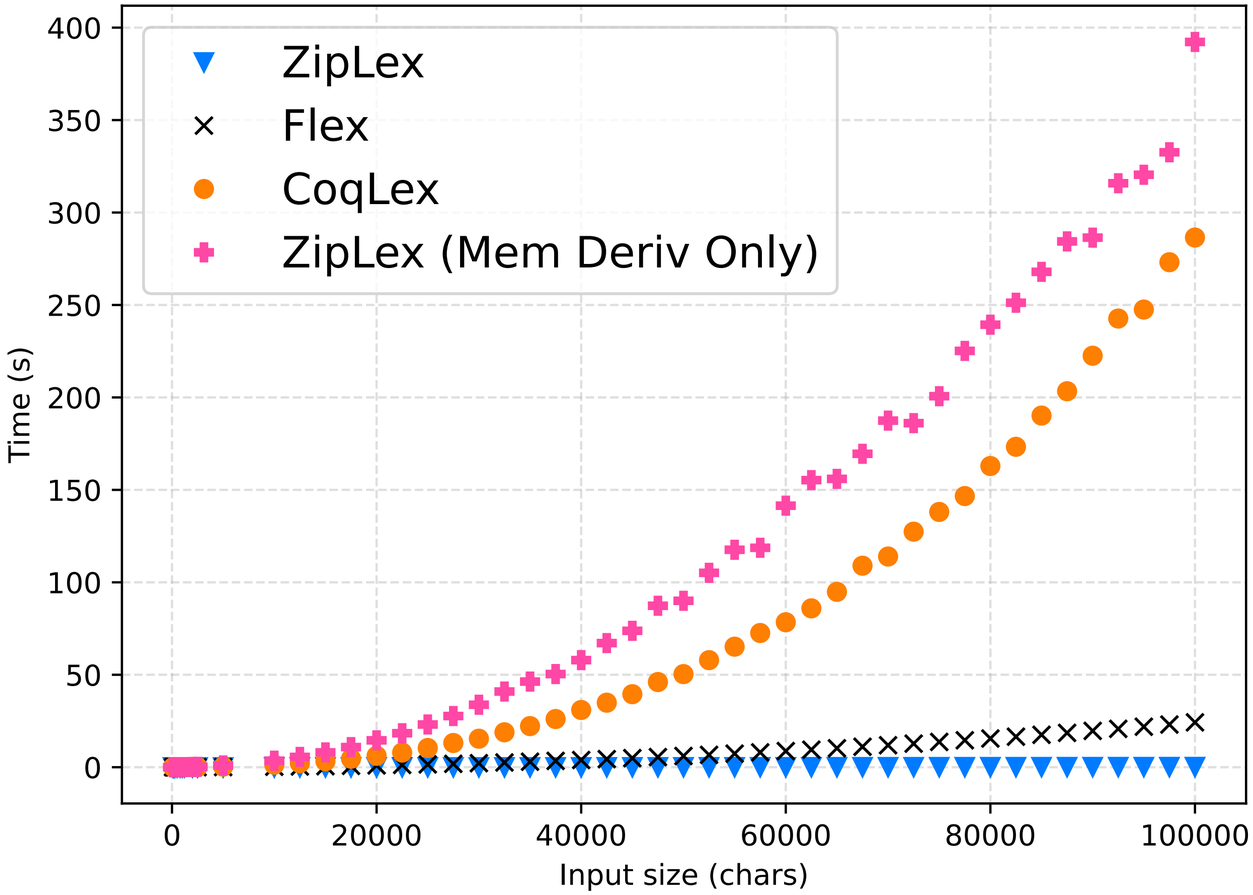
a and a^*b Benchmark

Input: “$aaa\ldots a$”

Performance

a and a^*b Benchmark

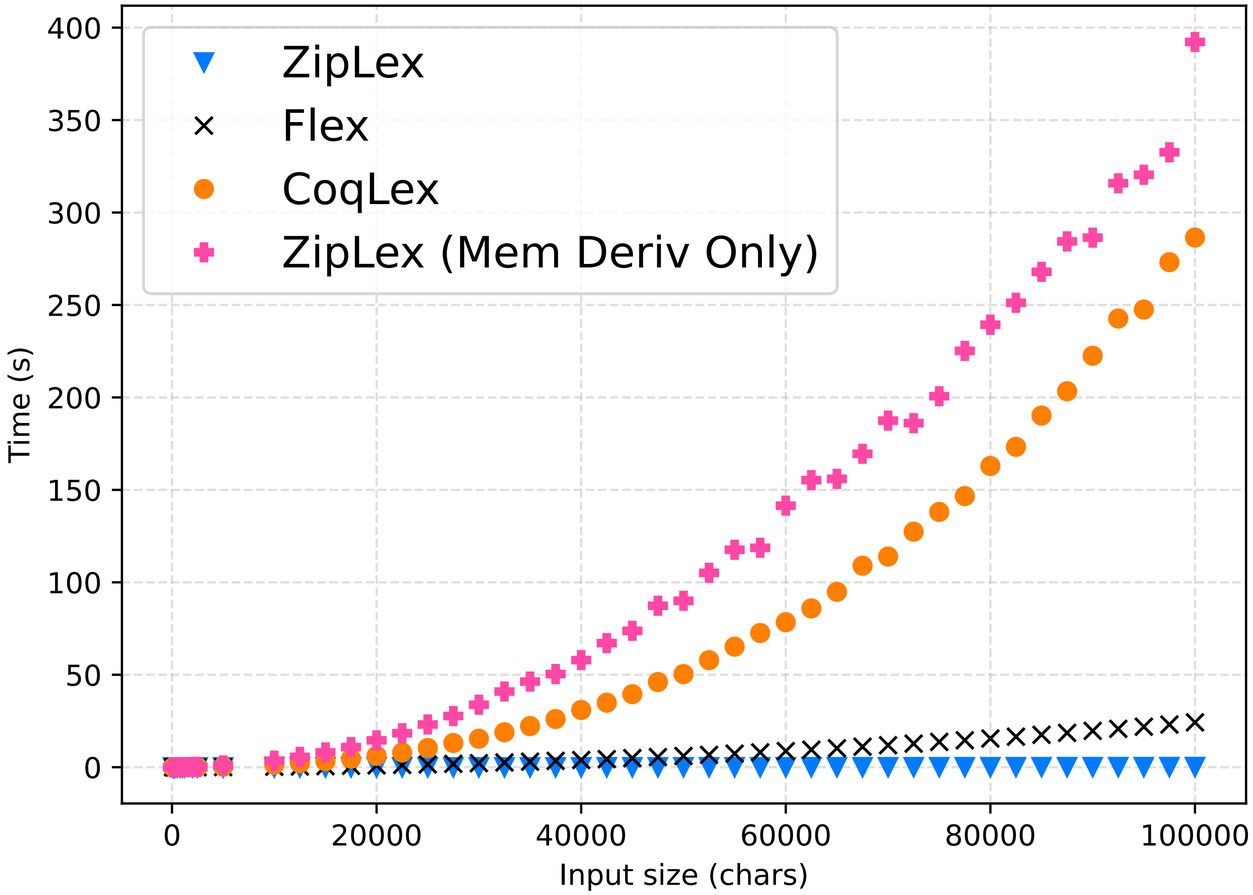
Input: "$aaa\ldots a$"

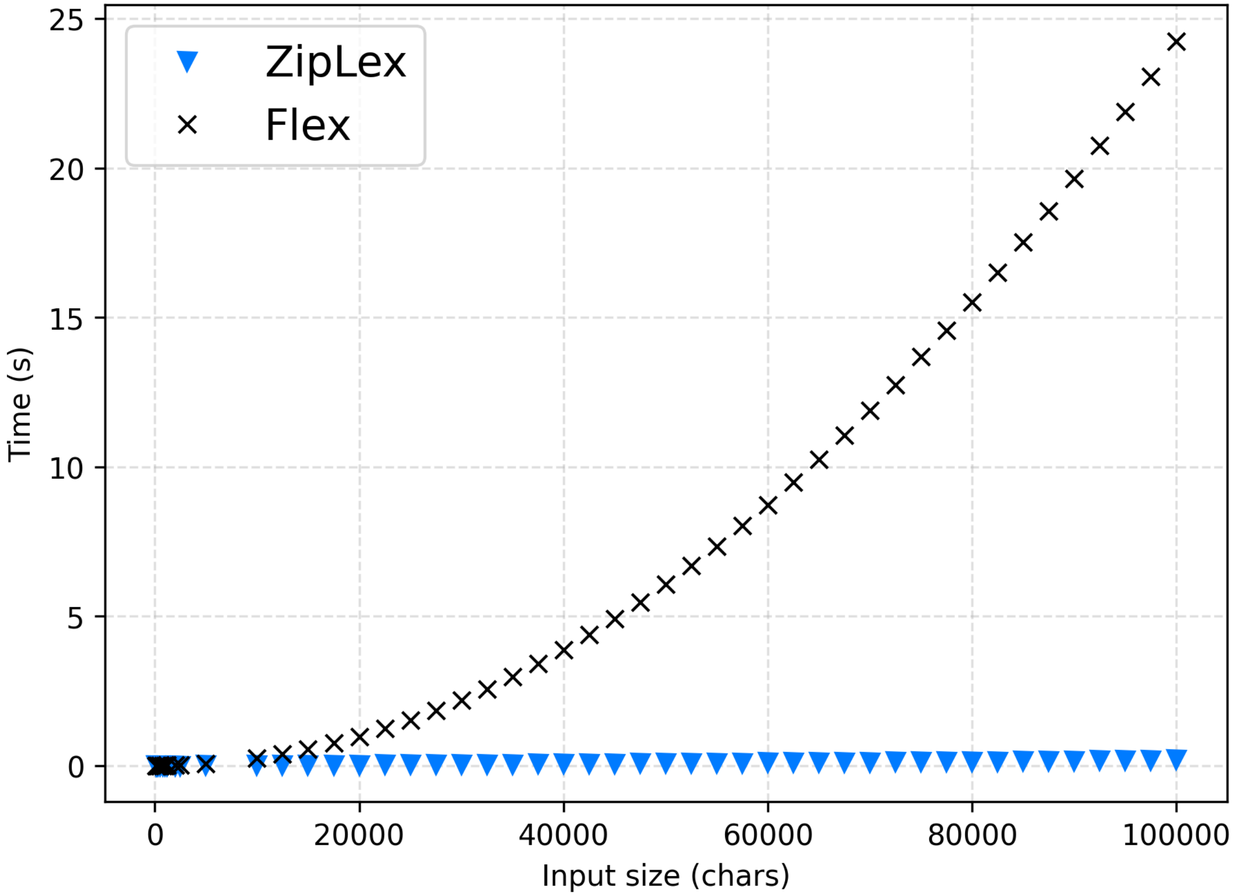


Performance

a and a^*b Benchmark

Input: "$aaa\ldots a$"

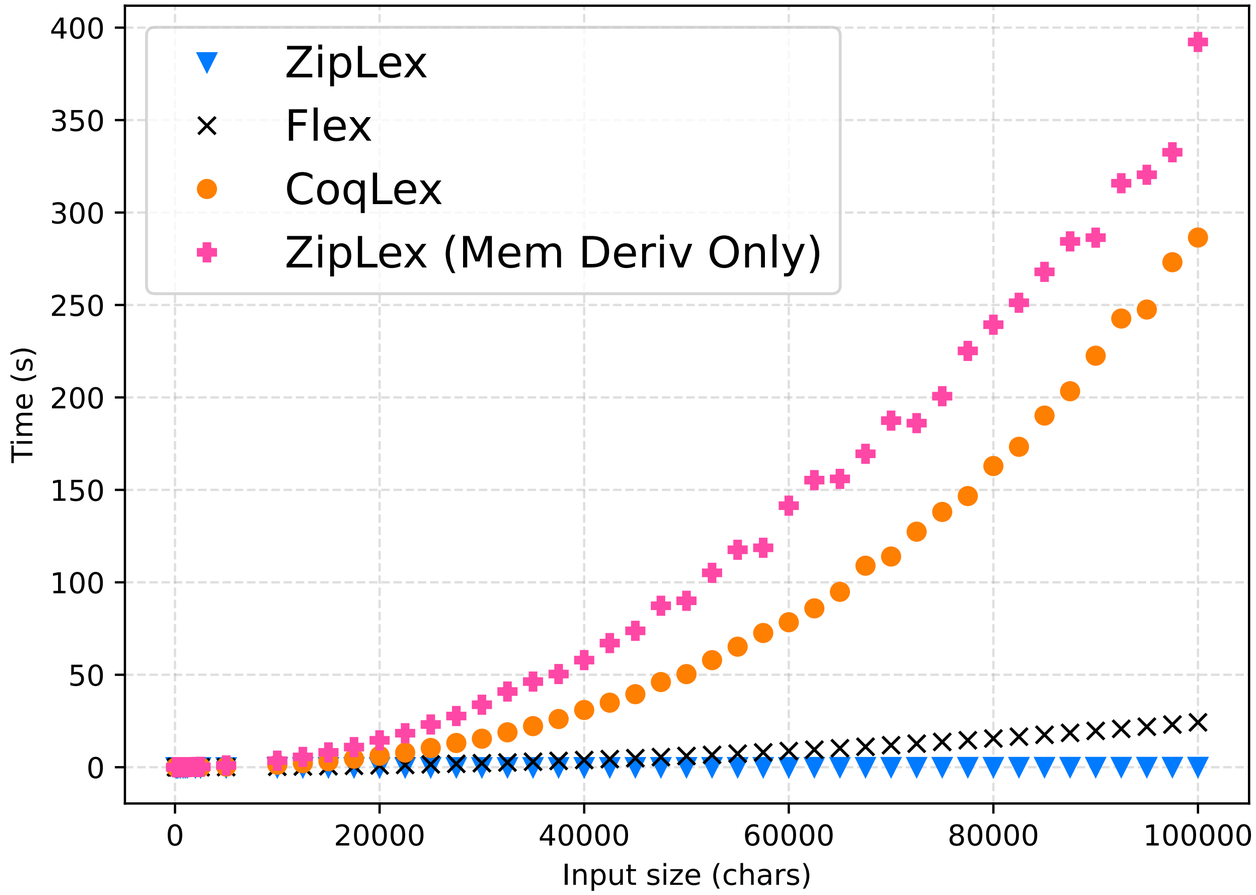


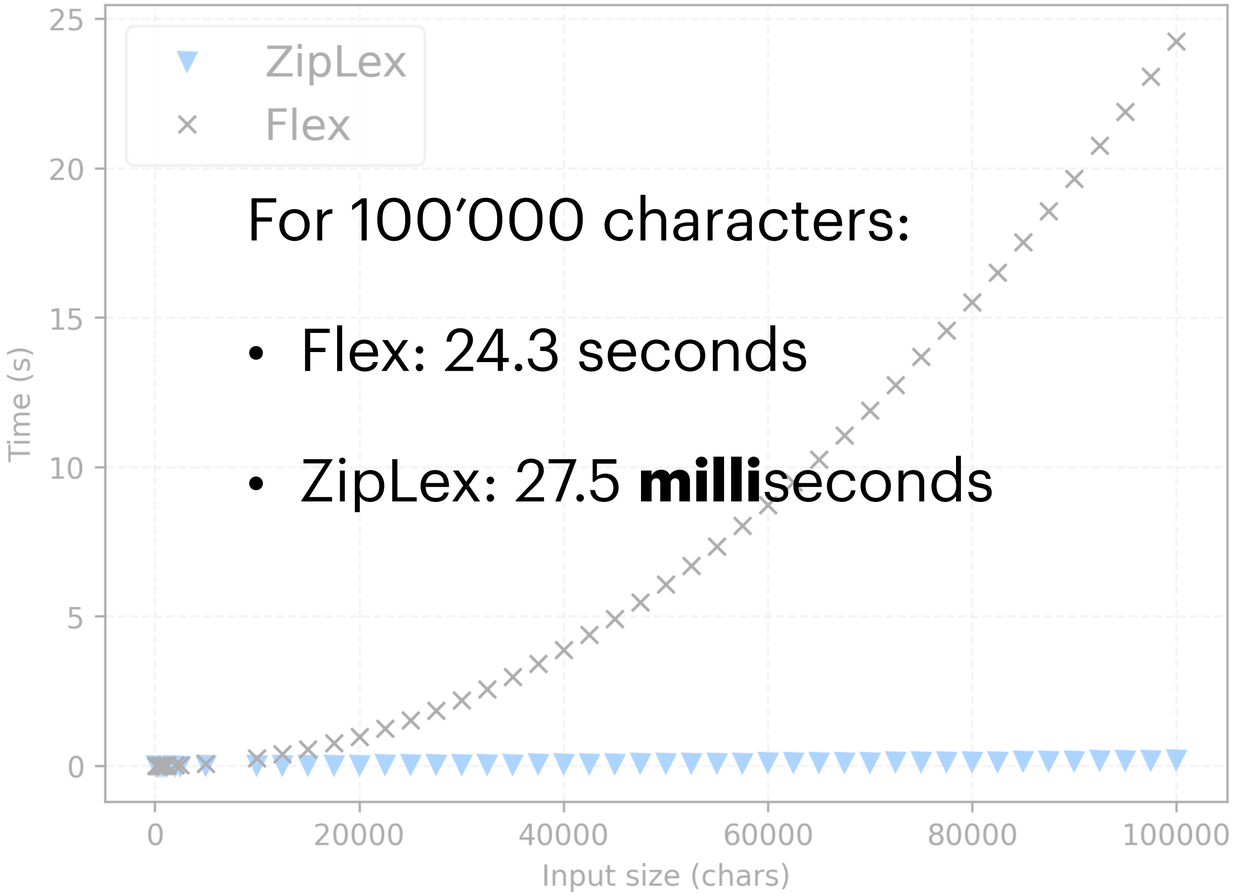


Performance

a and a^*b Benchmark

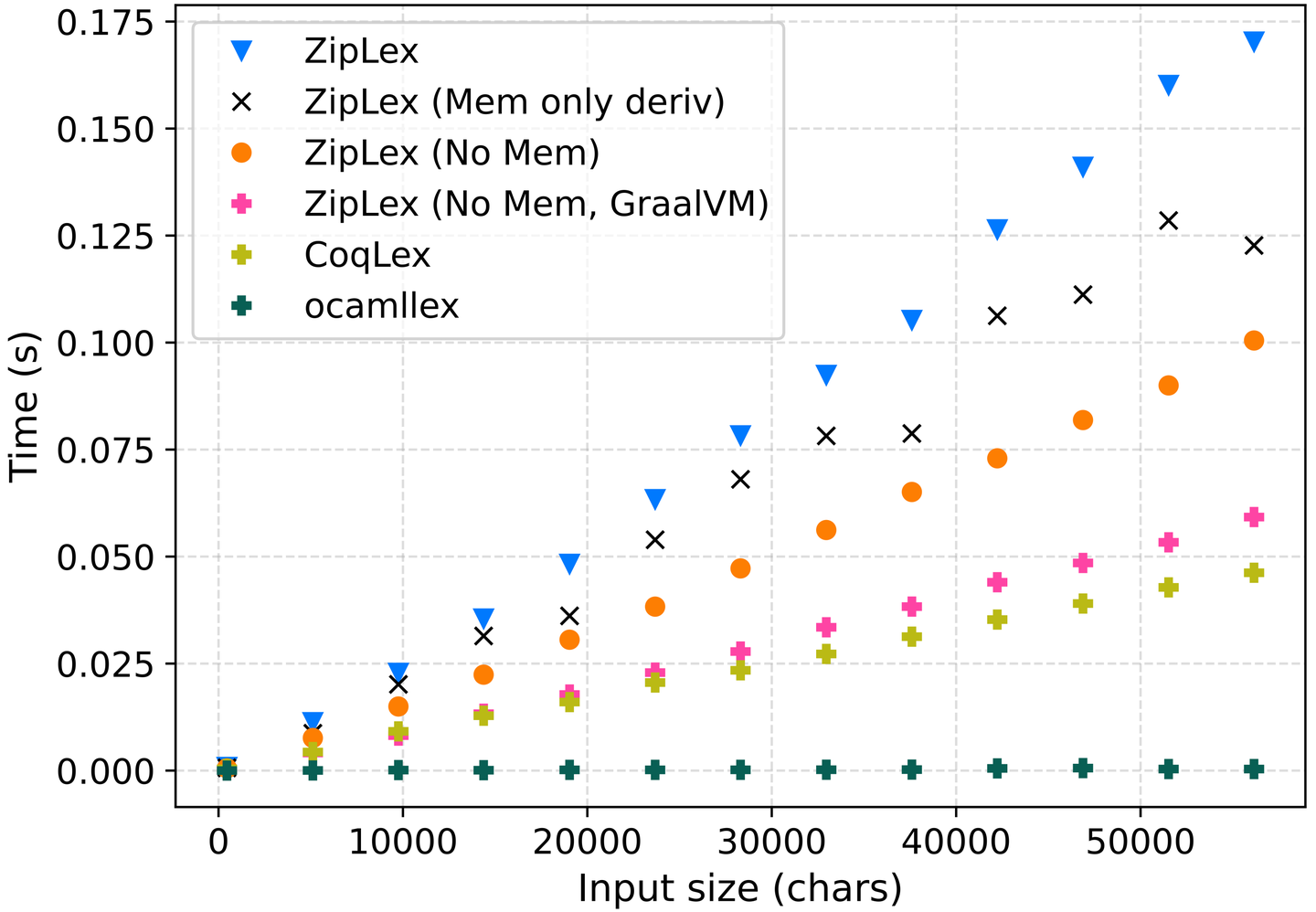
Input: “$aaa\ldots a$”





Performance

JSON Benchmark



Input: JSON objects and arrays[1]

Coqlex works on ASCII alphabet and does not verify invertibility

[1]Ouedraogo, Wendlasida, Gabriel Scherer, and Lutz Strassburger. 2023. **"Coqlex: Generating Formally Verified Lexers."** *The Art, Science, and Engineering of Programming* 8 (1): 3.

Conclusion

ZipLex

Conclusion

ZipLex

Fully verified and **linear-time** lexer

Conclusion

ZipLex

Fully verified and **linear-time** lexer

Longest match and **invertible** along with **R-Path** predicates

Conclusion

ZipLex

Fully verified and **linear-time** lexer

Longest match and **invertible** along with **R-Path** predicates

Regular expression engine, using derivatives with **zippers** and **memoization**

Conclusion

ZipLex

Fully verified and **linear-time** lexer

Longest match and **invertible** along with **R-Path** predicates

Regular expression engine, using derivatives with **zippers** and **memoization**

Reusable memoization with **verified mutable hash table**

Conclusion

ZipLex

Fully verified and **linear-time** lexer

Longest match and **invertible** along with **R-Path** predicates

Regular expression engine, using derivatives with **zippers** and **memoization**

Reusable memoization with **verified mutable hash table**

Leading performance on **adversarial grammars**

Conclusion

ZipLex

Fully verified and **linear-time** lexer

Longest match and **invertible** along with **R-Path** predicates

Regular expression engine, using derivatives with **zippers** and **memoization**

Reusable memoization with **verified mutable hash table**

Leading performance on **adversarial grammars**

On par performance with **Coqlex** on **JSON**

Conclusion

ZipLex

Fully verified and **linear-time** lexer

Longest match and **invertible** along with **R-Path** predicates

Regular expression engine, using derivatives with **zippers** and **memoization**

Reusable memoization with **verified mutable hash table**

Leading performance on **adversarial grammars**

On par performance with **Coqlex** on **JSON**

Personal page

samuel.chassot@epfl.ch

Stainless

ZipLex

Backup

Token Separability

Examples

Token Separability

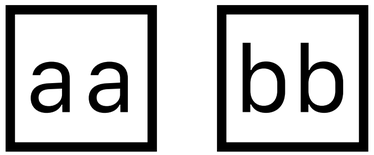
Examples

$$r_1 = a^+ \quad r_2 = b^+$$

Token Separability

Examples

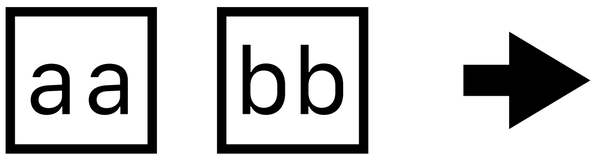
$$r_1 = a^+ \quad r_2 = b^+$$



Token Separability

Examples

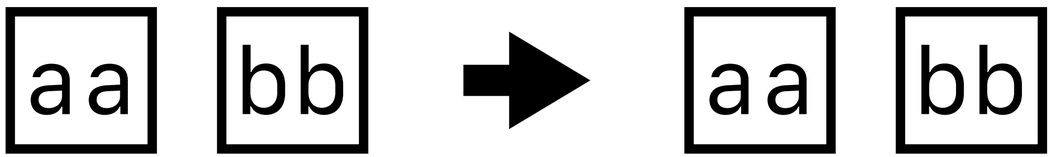
$$r_1 = a^+ \quad r_2 = b^+$$



Token Separability

Examples

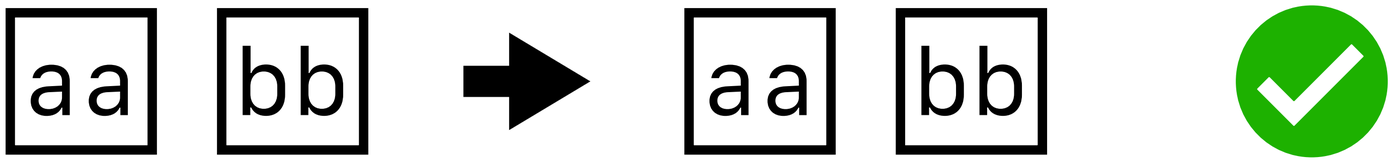
$$r_1 = a^+ \quad r_2 = b^+$$



Token Separability

Examples

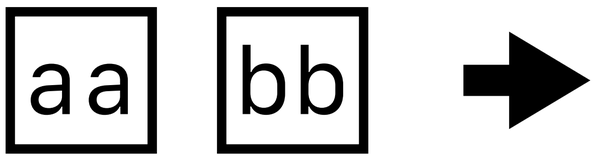
$r_1 = a^+ \quad r_2 = b^+$



Token Separability

Examples

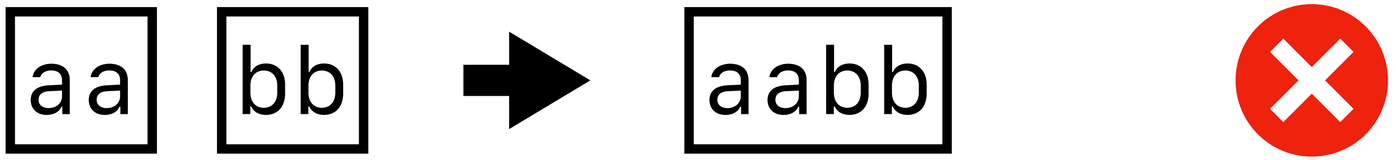
$$r_1 = a^+ \quad r_2 = b^+ \quad r_3 = (a \mid b)^+$$



Token Separability

Examples

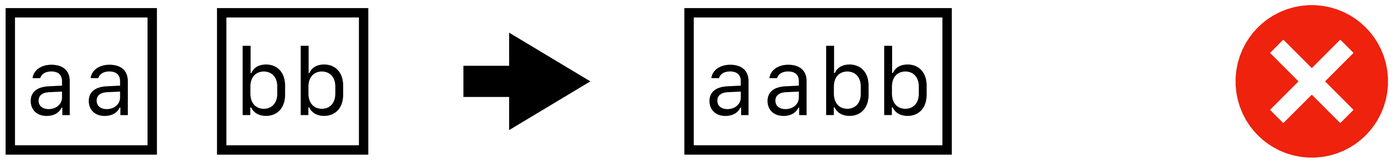
$$r_1 = a^+ \quad r_2 = b^+ \quad r_3 = (a \mid b)^+$$



Token Separability

Examples

$$r_1 = a^+ \quad r_2 = b^+ \quad r_3 = (a \mid b)^+$$

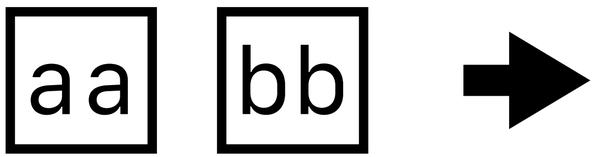


Separability is affected by ALL the rules

Token Separability

Examples

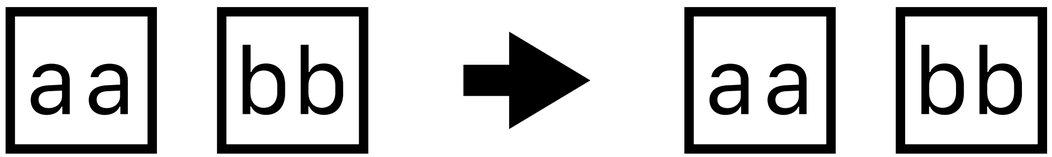
$$r_1 = a^+ \quad r_2 = b^+ \quad r_3 = (a \mid b)^+ \cdot c$$



Token Separability

Examples

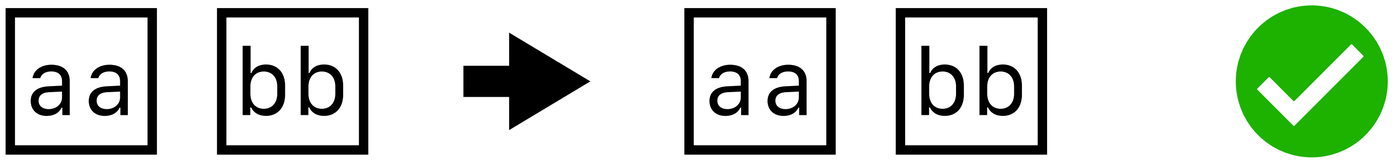
$$r_1 = a^+ \quad r_2 = b^+ \quad r_3 = (a \mid b)^+ \cdot c$$



Token Separability

Examples

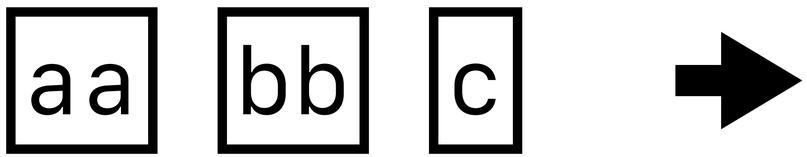
$$r_1 = a^+ \quad r_2 = b^+ \quad r_3 = (a \mid b)^+ \cdot c$$



Token Separability

Examples

$$r_1 = a^+ \quad r_2 = b^+ \quad r_3 = (a \mid b)^+ \cdot c$$



Token Separability

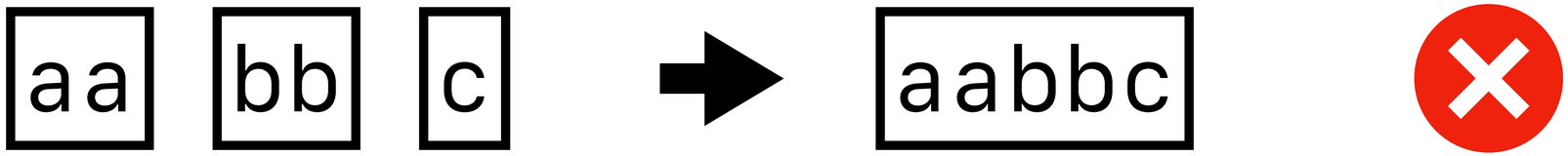
Examples

$$r_1 = a^+ \quad r_2 = b^+ \quad r_3 = (a \mid b)^+ \cdot c$$

Token Separability

Examples

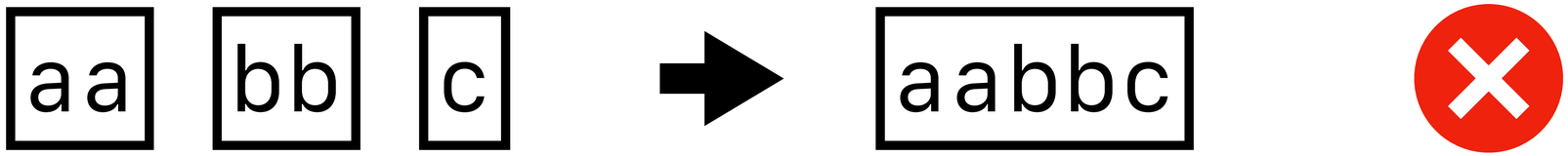
$$r_1 = a^+ \quad r_2 = b^+ \quad r_3 = (a \mid b)^+ \cdot c$$



Token Separability

Examples

$$r_1 = a^+ \quad r_2 = b^+ \quad r_3 = (a \mid b)^+ \cdot c$$



Separability is affected by following tokens

Performance

a and a^*b Benchmark

Input: “$aaa\ldots a$”

Performance

a and a^*b Benchmark

Input: "$aaa\ldots a$"