A Verified LL(1) Parser Generator

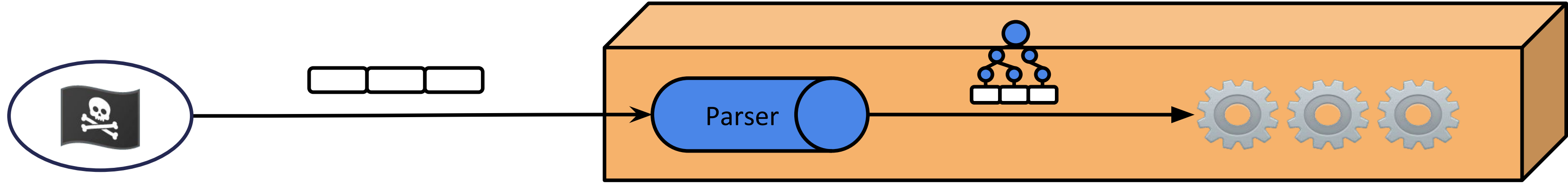
Sam Lasser, Kathleen Fisher
Tufts University

Chris Casinghino, Cody Roux
Draper

Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Why Verify Parsers?

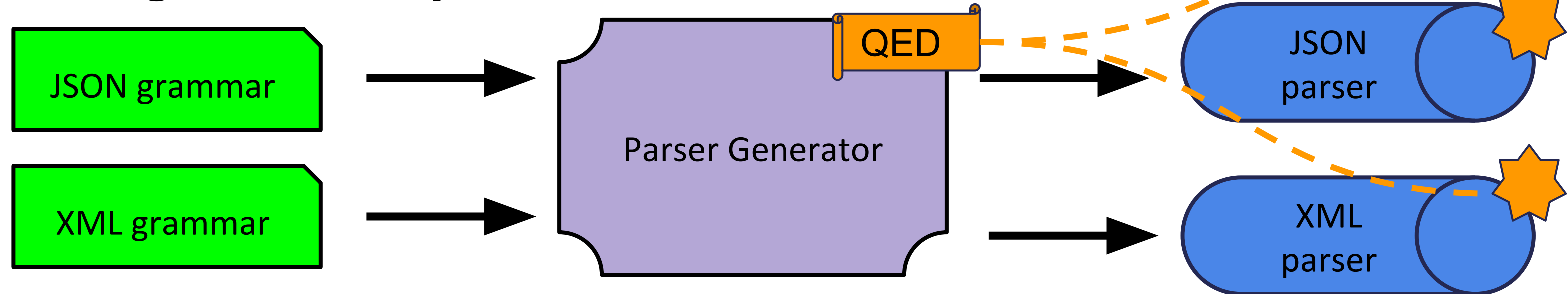
- Parsers connect application internals to the outside world



- Real-world parsers contain vulnerabilities
  - Faulty parser in web application framework: *Equifax breach* [CVE-2017-5638]
  - HTML parser vulnerability: *Data leak from online services* [Cloudflare Report 2017]
  - XML parser flaw: *Remote code execution on network device* [CVE-2016-0101]

Why Verify Parser Generators?

- Full verification of a parser generator ensures correctness of **each generated parser**



- Alternative approaches to producing correct parsers
  - Manual verification of an individual parser
  - *A posteriori* validation of generated parsers [Jourdan et al. 2012]

Main Contributions

- **Vermillion: LL(1) parser generator implemented and verified in Coq**
- **End-to-End Correctness Proofs**

  The generator produces a correct LL(1) parser whenever such a parser exists
- **Total Implementations**

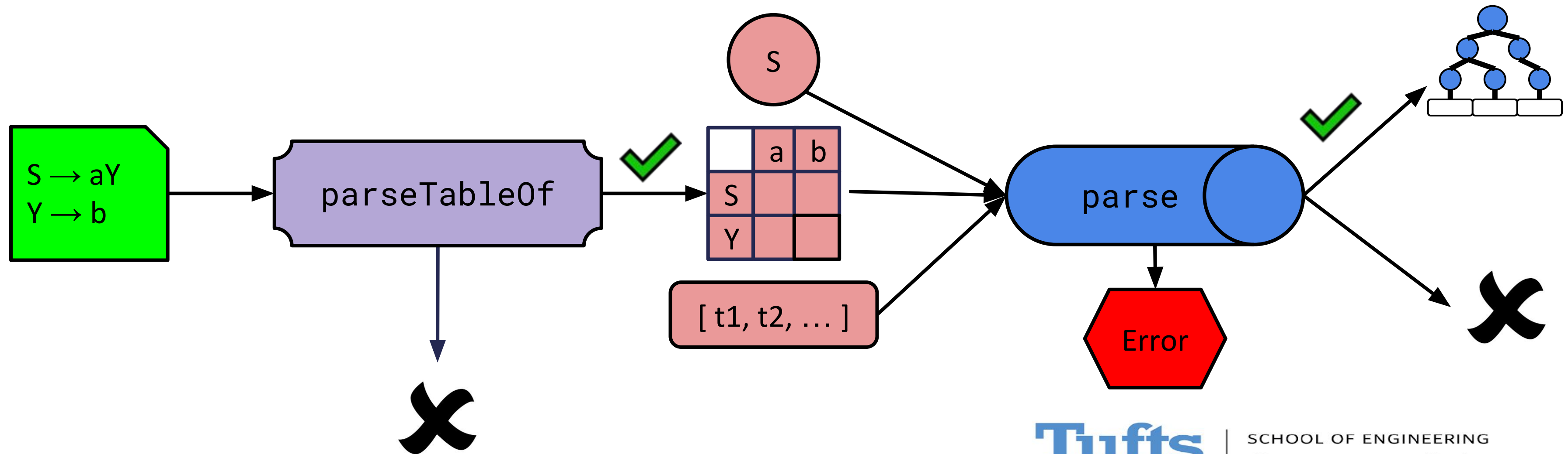
  The generator and parsers terminate without error on all inputs
- **Efficient Extractable Code**

  Generated parser is ~2-4x slower than an unverified parser for the same grammar

Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Parser Generator API

- Two-stage LL(1) parser generator workflow
  1. Generation: convert a grammar to an LL(1) parse table
  2. Parsing: use the table to build a semantic value for a token sequence



Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Implementing the Parse Table Generator

- classic dataflow analysis (e.g., Appel 1998)
- verification challenges
  - analyses performed in parallel
  - "iterate until convergence"
- solutions
  - perform analyses sequentially
  - identify well-founded decreasing measure for each analysis

Algorithm to compute FIRST, FOLLOW, *and* nullable.

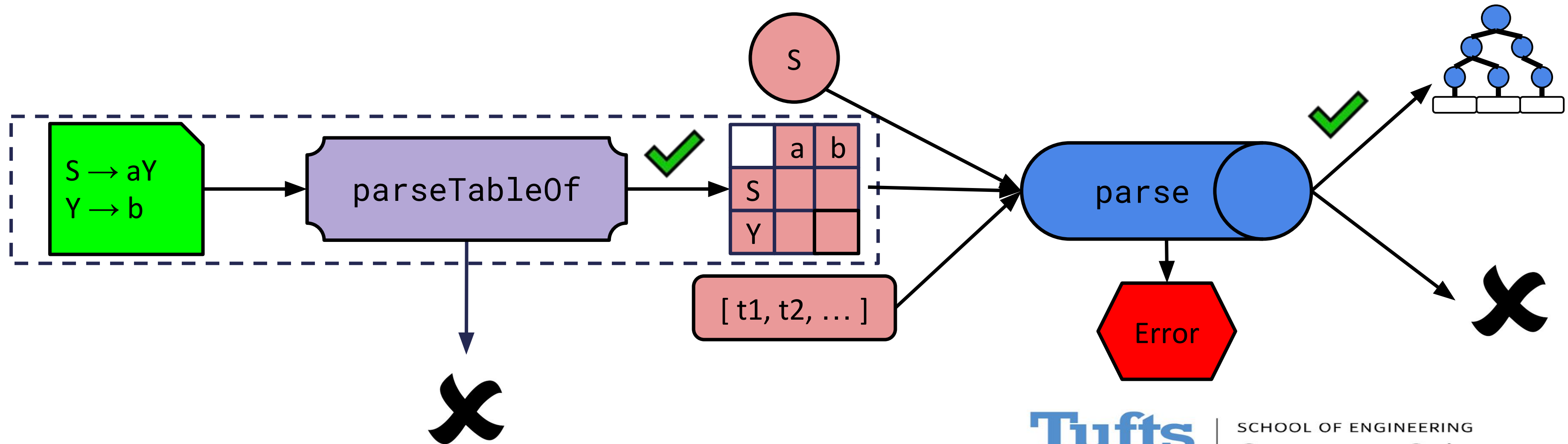
Initialize FIRST and FOLLOW to all empty sets, and nullable to all false.

for each terminal symbol Z
  FIRST[Z] $\leftarrow \{Z\}$
repeat
  for each production $X \rightarrow Y_1 Y_2 \cdots Y_k$
    for each i from 1 to k, each j from $i+1$ to k,
      if all the Y_i are nullable
        then nullable[X] $\leftarrow$ true
      if $Y_1 \cdots Y_{i-1}$ are all nullable
        then FIRST[X] $\leftarrow$ FIRST[X] $\cup$ FIRST[Y_i]
      if $Y_{i+1} \cdots Y_k$ are all nullable
        then FOLLOW[Y_i] $\leftarrow$ FOLLOW[Y_i] $\cup$ FOLLOW[X]
      if $Y_{i+1} \cdots Y_{j-1}$ are all nullable
        then FOLLOW[Y_i] $\leftarrow$ FOLLOW[Y_i] $\cup$ FIRST[Y_j]
until FIRST, FOLLOW, and nullable did not change in this iteration.

ALGORITHM 3.13. Iterative computation of *FIRST*, *FOLLOW*, and *nullable*.

Verifying the Parse Table Generator

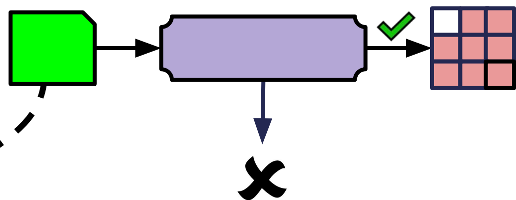
Theorem: `parseTableOf` applied to grammar G returns a correct LL(1) parse table T iff such a table exists for G



Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Verifying the Parse Table Generator

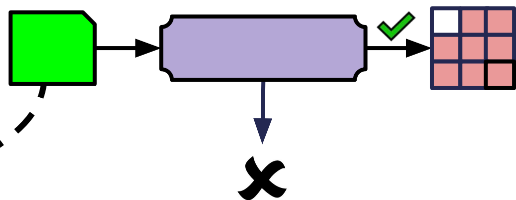
Theorem: `parseTableOf` applied to grammar G returns a correct LL(1) parse table T iff such a table exists for G



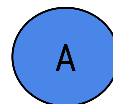
```
(1)   A → 42
(2)   A → if B then A else A
(3)   B → true
(4)   B → false
```

Verifying the Parse Table Generator

Theorem: `parseTableOf` applied to grammar G returns a correct LL(1) parse table T iff such a table exists for G



```
(1)   A → 42
(2)   A → if B then A else A
(3)   B → true
(4)   B → false
```

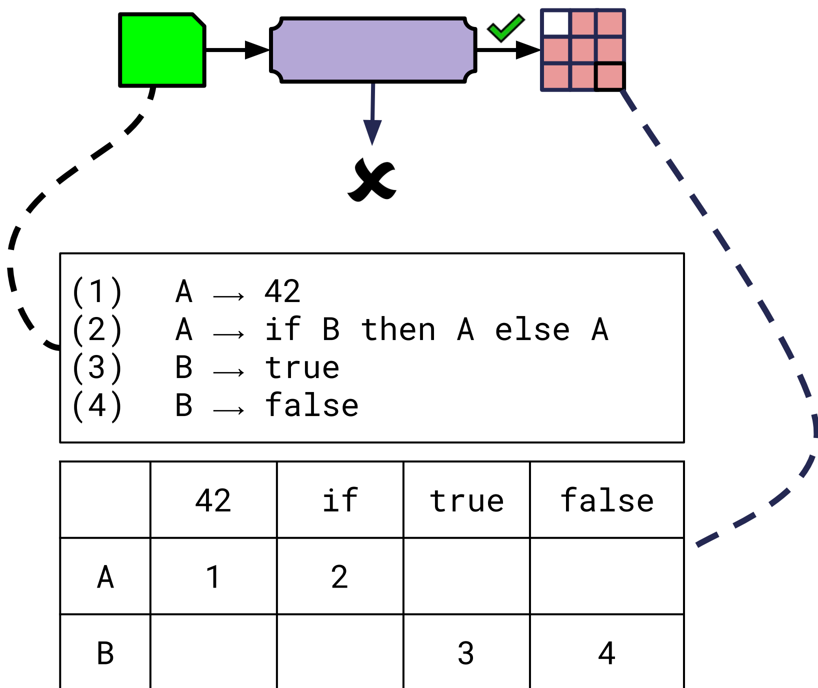


???

Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Verifying the Parse Table Generator

Theorem: `parseTableOf` applied to grammar G returns a correct LL(1) parse table T iff such a table exists for G



	42	if	true	false
A	1	2		
B			3	4

A

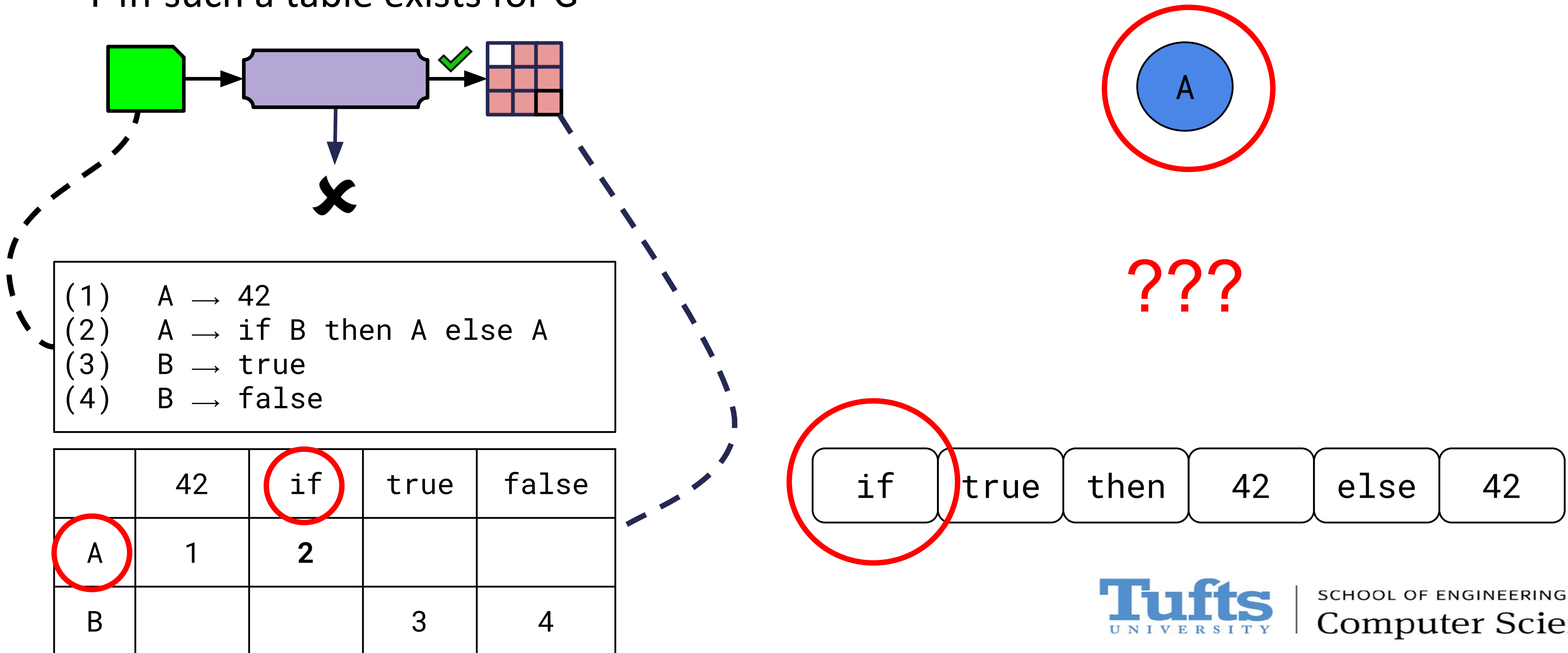
???

if	true	then	42	else	42

Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Verifying the Parse Table Generator

Theorem: `parseTableOf` applied to grammar G returns a correct LL(1) parse table T iff such a table exists for G

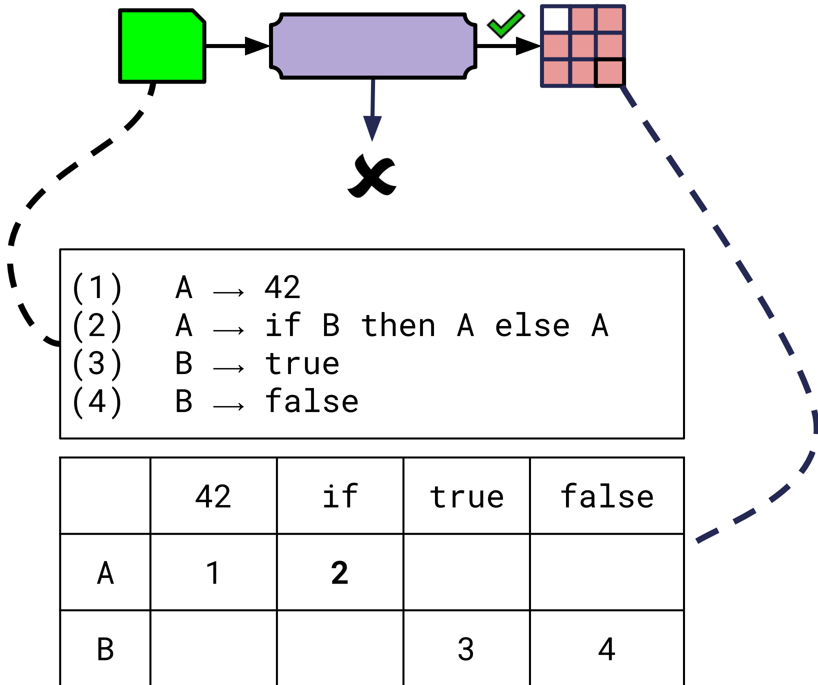


	42	if	true	false
A	1	2		
B			3	4

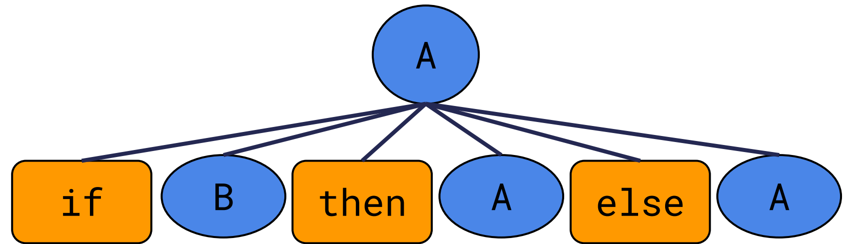
Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

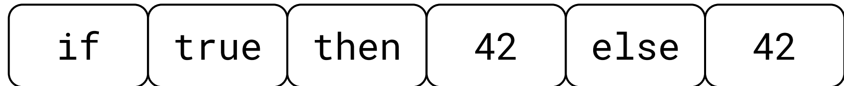
Verifying the Parse Table Generator

Theorem: `parseTableOf` applied to grammar G returns a correct LL(1) parse table T iff such a table exists for G



	42	if	true	false
A	1	**2**		
B			3	4

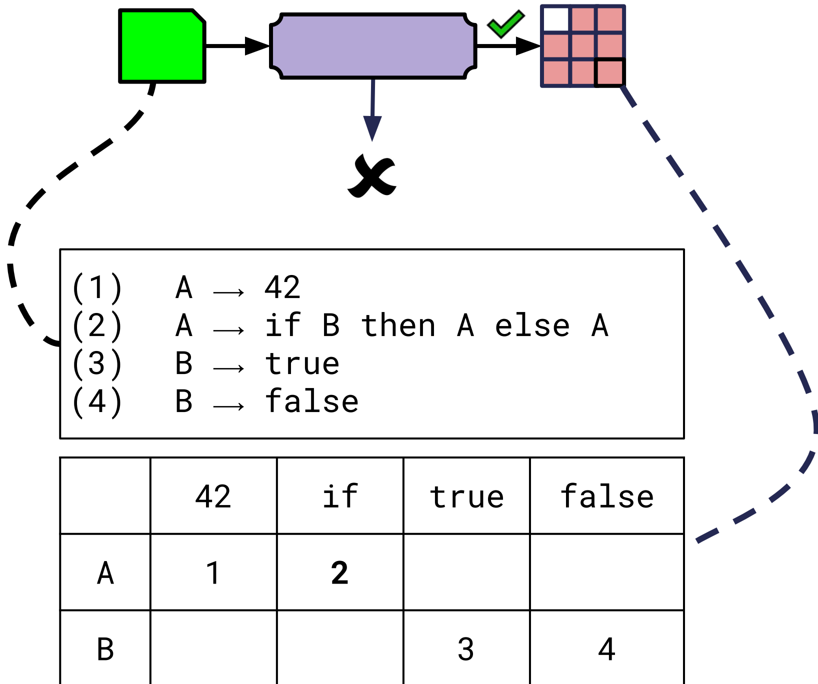




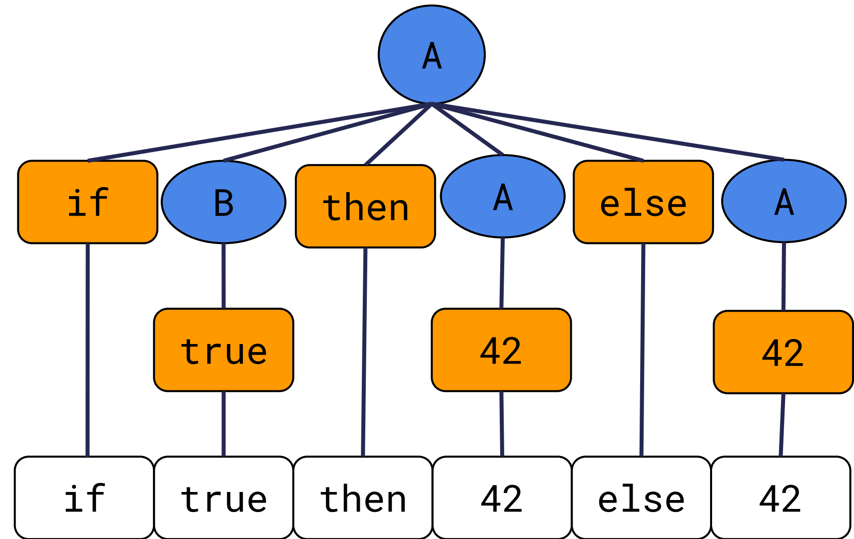
Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Verifying the Parse Table Generator

Theorem: `parseTableOf` applied to grammar G returns a correct LL(1) parse table T iff such a table exists for G



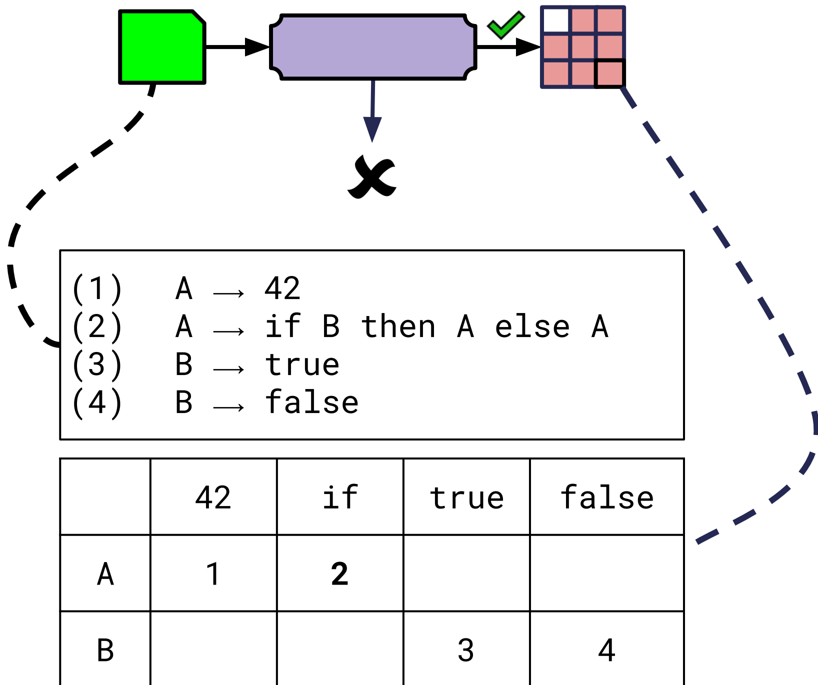
	42	if	true	false
A	1	**2**		
B			3	4



Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Verifying the Parse Table Generator

Theorem: `parseTableOf` applied to grammar G returns a **correct** LL(1) parse table T iff such a table exists for G



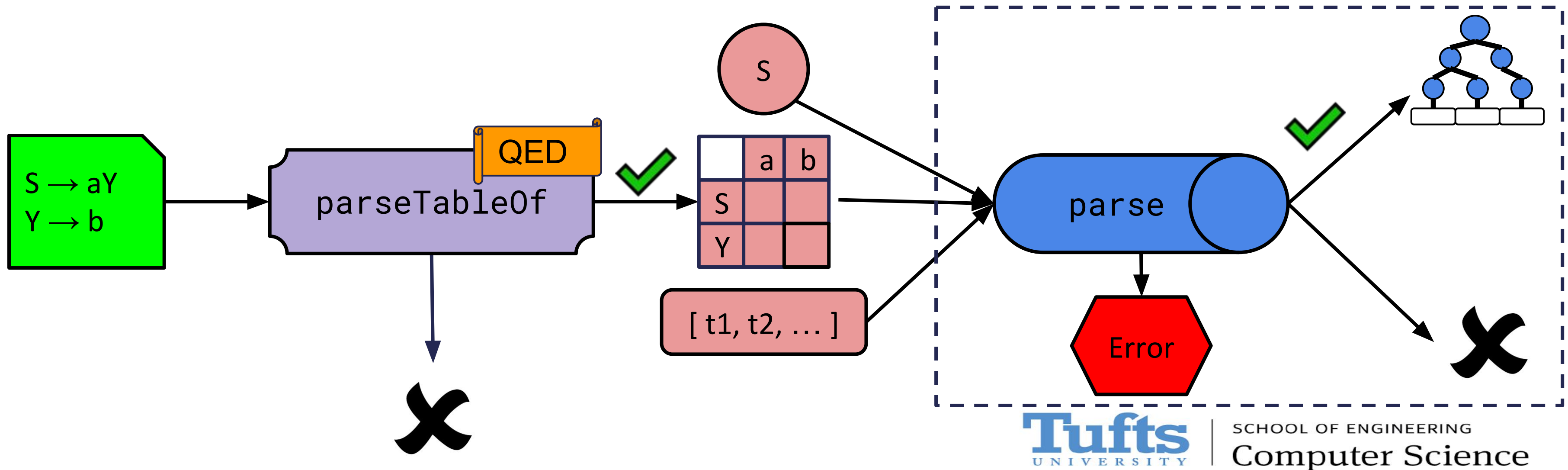
	42	if	true	false
A	1	**2**		
B			3	4

T predicts production P iff P may result in a successful derivation

Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Parser Correctness

Goal: if T is a correct LL(1) parse table for G, then the parser applied to T is **correct with respect to G**



Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Parser Correctness Spec: Derivation Relation

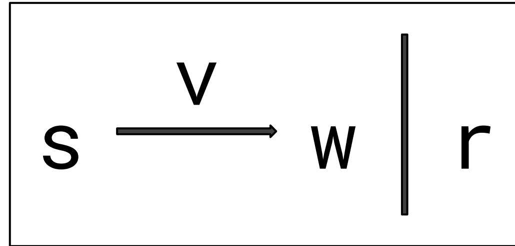
s : grammar symbol

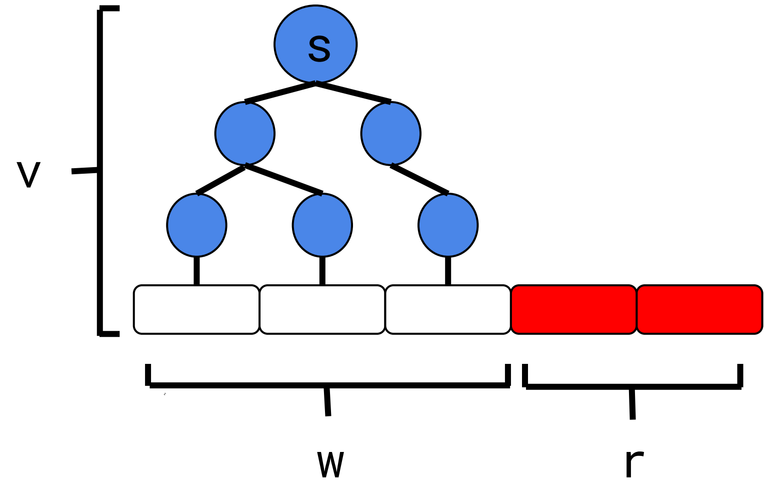
w : word (token sequence)

v : semantic value

r : remainder (unparsed suffix)

Judgment form:



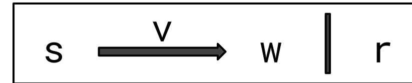


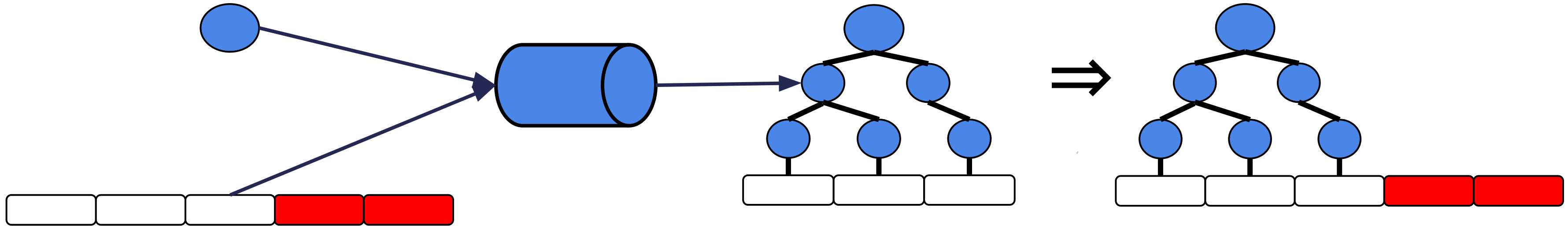
Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Parser Correctness Properties

T is a correct LL(1) parse table for G ⇒

- **Soundness:** `parse T s (w ++ r) = v` ⇒

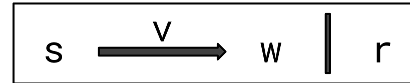




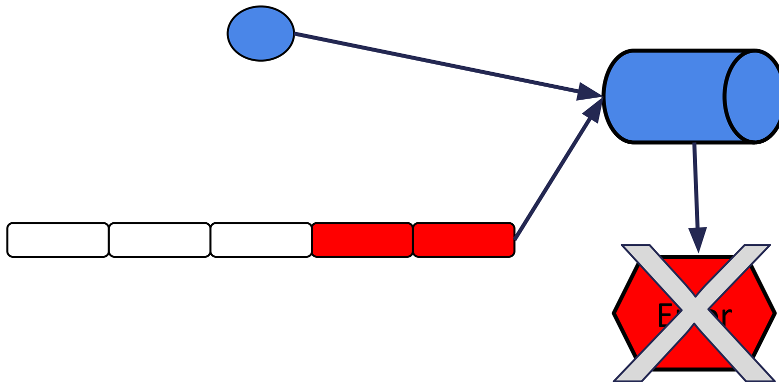
Parser Correctness Properties

T is a correct LL(1) parse table for G ⇒

- **Soundness:** `parse T s (w ++ r) = v` ⇒



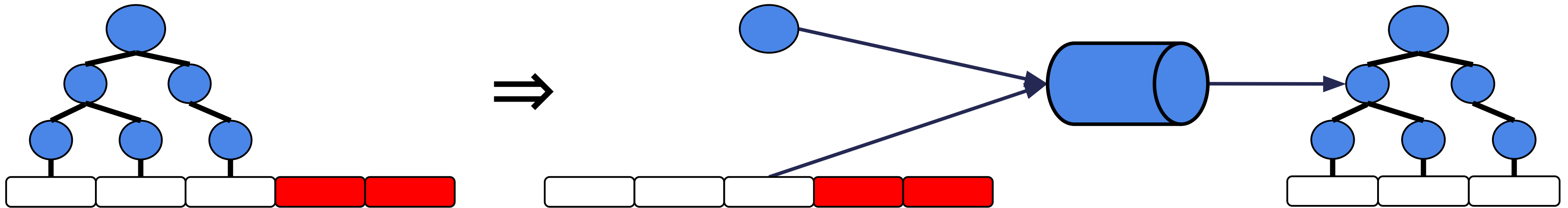
- **Error-Free Termination:** `parse T s toks ≠ Error`



Parser Correctness Properties

T is a correct LL(1) parse table for G ⇒

- **Soundness:** `parse T s (w ++ r) = v` ⇒ [box: s —v→ w | r]
- **Error-Free Termination:** `parse T s toks ≠ Error`
- **Completeness:** [box: s —v→ w | r] ⇒ `parse T s (w ++ r) = v`

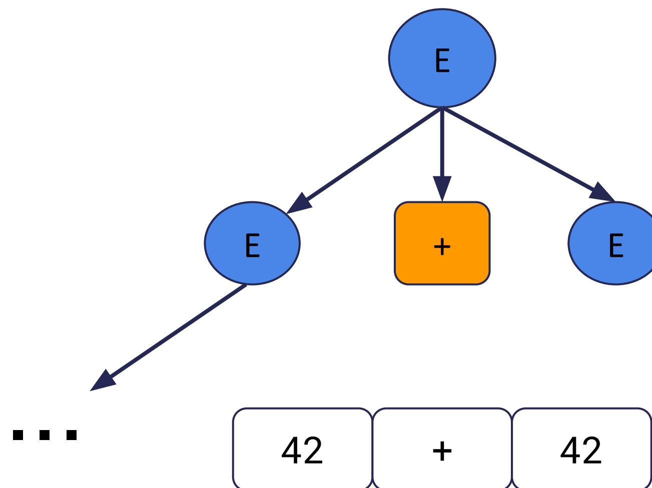


Defining a Provably Terminating Parser

- Left-recursive grammars cause top-down parsers to diverge

```
(1)    E → 42
(2)    E → E + E
```

- How can we define a seemingly non-terminating algorithm in Coq?

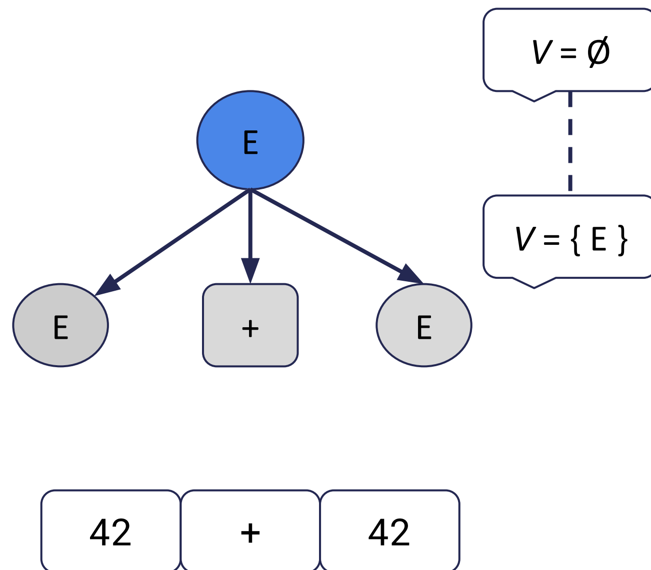


Defining a Provably Terminating Parser

- Left-recursive grammars cause top-down parsers to diverge

(1)	E	$\longrightarrow$	42
(2)	E	$\longrightarrow$	E + E

- How can we define a seemingly non-terminating algorithm in Coq?
  - maintain a set of visited nonterminals V
  - return Error if the parser visits an NT twice without consuming a token
  - rule out the error case *a posteriori*

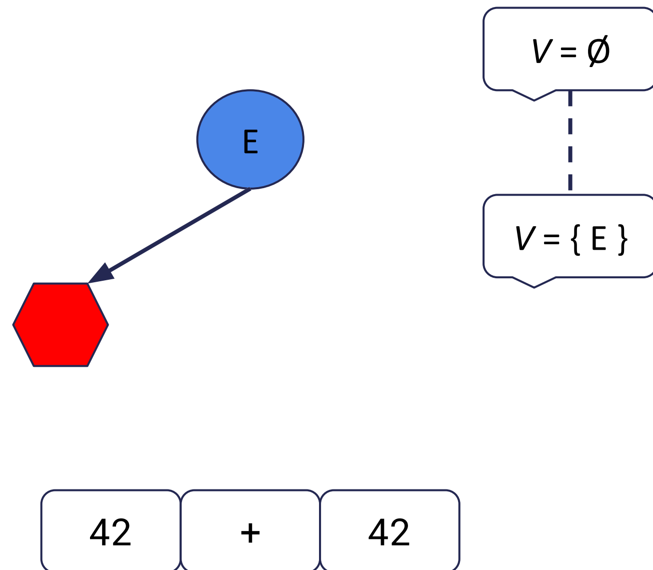


Defining a Provably Terminating Parser

- Left-recursive grammars cause top-down parsers to diverge

(1)	E	$\longrightarrow$ 42
(2)	E	$\longrightarrow$ E + E

- How can we define a seemingly non-terminating algorithm in Coq?
  - maintain a set of visited nonterminals *V*
  - return Error if the parser visits an NT twice without consuming a token
  - rule out the error case *a posteriori*



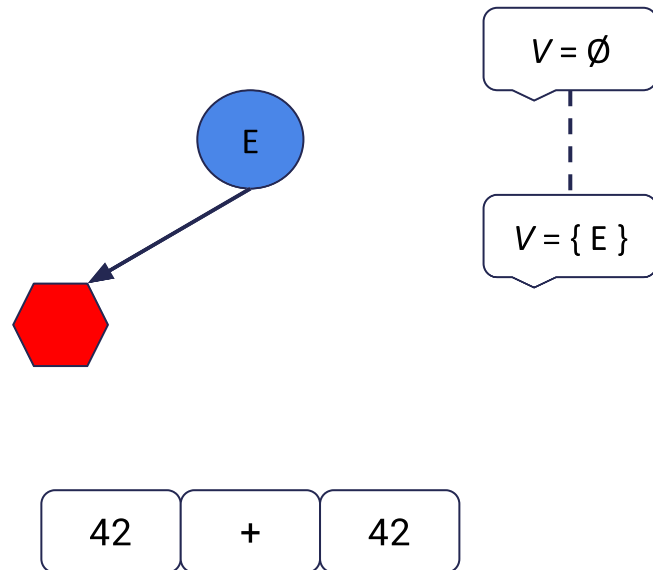
Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Defining a Provably Terminating Parser

- Left-recursive grammars cause top-down parsers to diverge

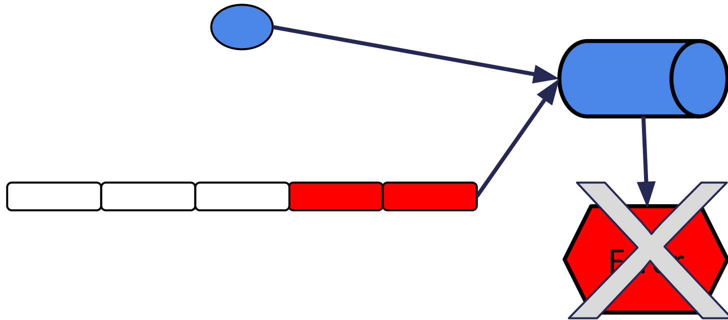
(1)	E	$\longrightarrow$	42
(2)	E	$\longrightarrow$	E + E

- How can we define a seemingly non-terminating algorithm in Coq?
  - maintain a set of visited nonterminals *V*
  - return Error if the parser visits an NT twice without consuming a token
  - rule out the error case *a posteriori*
- Lexicographic termination measure based on length of tokens and |*V*|



Parser Error-Free Termination

Theorem: when applied to a correct LL(1) parse table for grammar G, the parser never returns an Error value

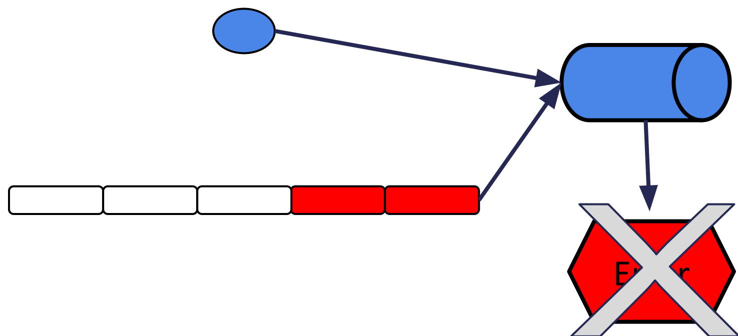


soundness + **termination** + completeness = *decidability*

Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Parser Error-Free Termination

Theorem: when applied to a correct LL(1) parse table for grammar G, the parser never returns an Error value



Proof based on two lemmas:

(1) Parser returns Error ⇒ G contains a left-recursive symbol

(2) T is a correct LL(1) parse table for G ⇒ G contains no left-recursive nonterminals

Tufts UNIVERSITY | SCHOOL OF ENGINEERING Computer Science

Performance Evaluation

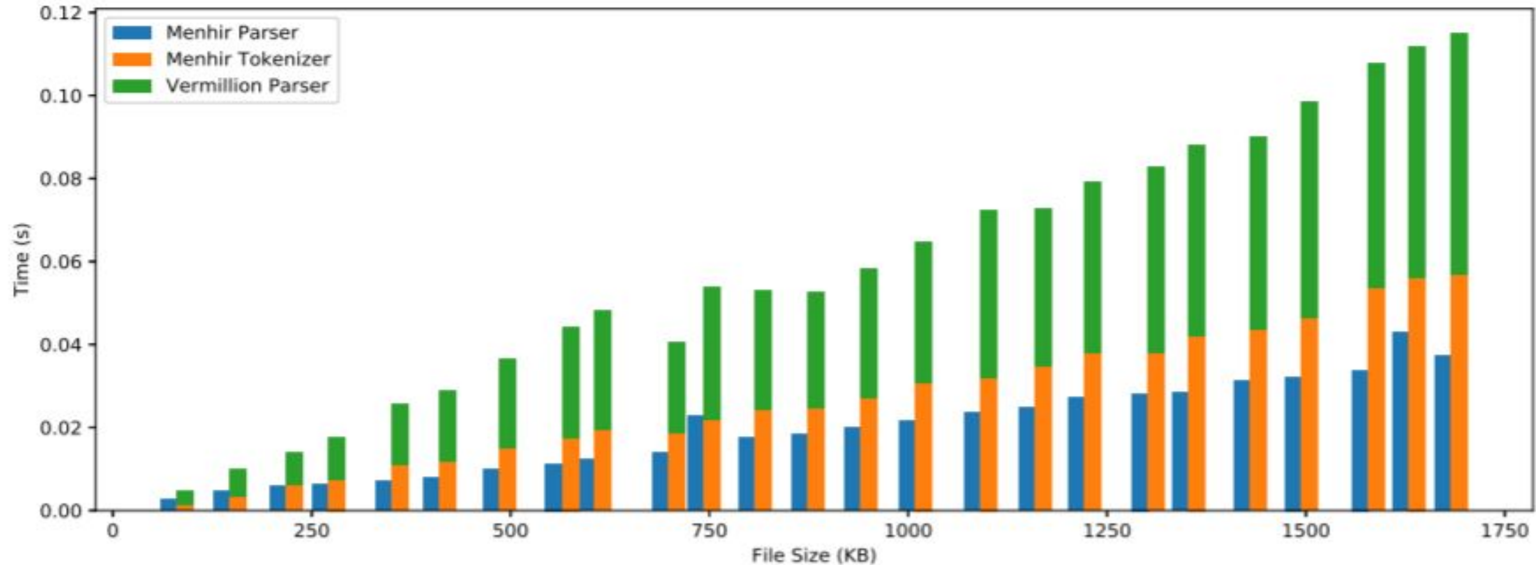


Figure 8 Average execution times of Menhir and Vermillion JSON parsers.

Conclusions

- Full verification of a (simple) parser generator is feasible
- Parser termination and correctness are closely intertwined
- Verification can bring clarity to a well-established field