Intrinsic Verification of Parsers and Formal Grammar Theory in Dependent Lambek Calculus

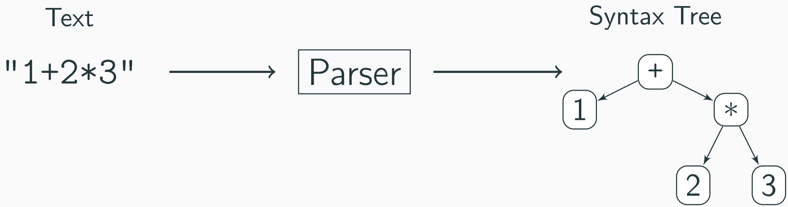
Steven Schaefer [1] Nathan Varner [1] Pedro H. Azevedo de Amorim [2]
Max S. New [1]

[1]University of Michigan

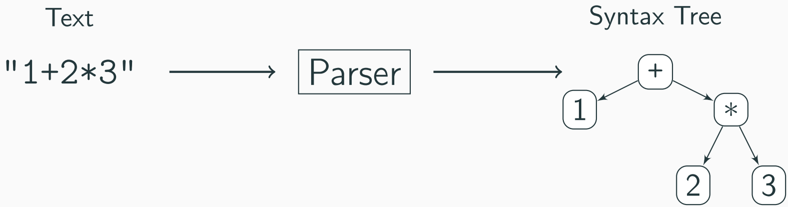
[2]University of Oxford

July 24, 2025

Parsing is Everywhere

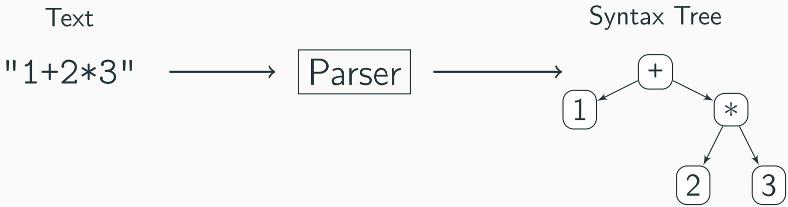


Parsing is Everywhere



The Cost of Incorrect Parsers

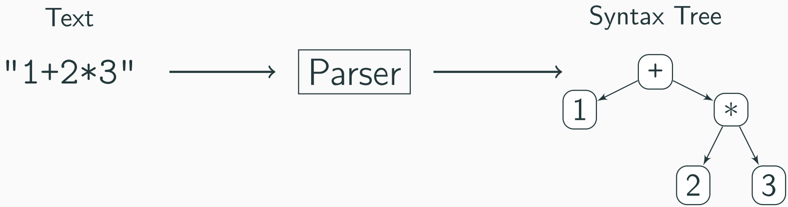
Parsing is Everywhere



The Cost of Incorrect Parsers

- Critical real-world vulnerabilities, including remote code execution **(Equifax, 2017)**

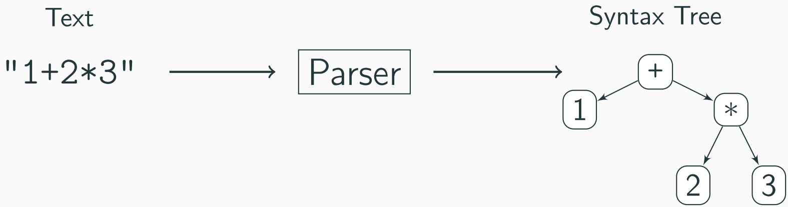
Parsing is Everywhere



The Cost of Incorrect Parsers

- Critical real-world vulnerabilities, including remote code execution **(Equifax, 2017)**
- The guarantees of other software verification can be invalidated

Parsing is Everywhere



The Cost of Incorrect Parsers

- Critical real-world vulnerabilities, including remote code execution **(Equifax, 2017)**
- The guarantees of other software verification can be invalidated
- Early versions of CompCert
  - 0 middle or backend bugs, 6 parser bugs **(Yang et al., 2011)**

Ensuring Parser Correctness

For a formal grammar $\mathtt{A}$, a parser is a function $\mathtt{f} : \textbf{String} \rightharpoonup \textbf{Parse}(\mathtt{A})$

Ensuring Parser Correctness

For a formal grammar $\mathtt{A}$, a parser is a function $\mathtt{f} : \textbf{String} \rightharpoonup \textbf{Parse}(\mathtt{A})$

Soundness For a string $\mathtt{w}$, $\mathtt{f(w)}$ is a valid parse tree for the grammar $\mathtt{A}$ of $\mathtt{w}$

Ensuring Parser Correctness

For a formal grammar $\mathtt{A}$, a parser is a function $\mathtt{f} : \textbf{String} \rightharpoonup \textbf{Parse}(\mathtt{A})$

Soundness For a string $\mathtt{w}$, $\mathtt{f(w)}$ is a valid parse tree for the grammar $\mathtt{A}$ of $\mathtt{w}$

Completeness Parser returns a parse tree whenever one exists

Verified Parsing Implementations

Existing verified parsers often implement a single formalism/algorithm

Verified Parsing Implementations

Existing verified parsers often implement a single formalism/algorithm

- **LL**, **LR**, **CYK**, parser combinators, Brzozowski derivatives, etc

Verified Parsing Implementations

Existing verified parsers often implement a single formalism/algorithm

- **LL**, **LR**, **CYK**, parser combinators, Brzozowski derivatives, etc
- Each formalism requires its own implementation

Dependent Lambek Calculus (Lambek$^{\mathbb{D}}$)

Lambek$^{\mathbb{D}}$: DSL for *intrinsically* verified parsing

Dependent Lambek Calculus (Lambek$^{\mathtt{D}}$)

Lambek$^{\mathtt{D}}$: DSL for *intrinsically* verified parsing

- Functional programming language for writing verified parsers

Dependent Lambek Calculus (Lambek$^{\mathtt{D}}$)

Lambek$^{\mathtt{D}}$: DSL for *intrinsically* verified parsing

- Functional programming language for writing verified parsers
- Ordered, linear type system guarantees parser soundness for free

Dependent Lambek Calculus (Lambek$^{\mathtt{D}}$)

Lambek$^{\mathtt{D}}$: DSL for *intrinsically* verified parsing

- Functional programming language for writing verified parsers
- Ordered, linear type system guarantees parser soundness for free
  - Completeness with a little effort

Dependent Lambek Calculus (Lambek$^{\mathbb{D}}$)

Lambek$^{\mathbb{D}}$: DSL for *intrinsically* verified parsing

- Functional programming language for writing verified parsers
- Ordered, linear type system guarantees parser soundness for free
  - Completeness with a little effort
- Extensible framework for implementing grammar formalisms, their parsers, and their parser generators

Dependent Lambek Calculus (Lambek$^{\mathbb{D}}$)

Lambek$^{\mathbb{D}}$: DSL for *intrinsically* verified parsing

- Functional programming language for writing verified parsers
- Ordered, linear type system guarantees parser soundness for free
  - Completeness with a little effort
- Extensible framework for implementing grammar formalisms, their parsers, and their parser generators

Universal formalism that expresses grammars as types

Dependent Lambek Calculus (Lambek$^{\mathtt{D}}$)

Lambek$^{\mathtt{D}}$: DSL for *intrinsically* verified parsing

- Functional programming language for writing verified parsers
- Ordered, linear type system guarantees parser soundness for free
  - Completeness with a little effort
- Extensible framework for implementing grammar formalisms, their parsers, and their parser generators

Universal formalism that expresses grammars as types

Implemented as an embedded DSL in Cubical Agda

- Verified parsers for selected context-free grammars
- Verified parser generator for regular expressions

1. Lambek$^{\mathtt{D}}$

2. Dyck Grammar: An Extended Example

3. Conclusion

1. Lambek$^{\mathtt{D}}$

2. Dyck Grammar: An Extended Example

3. Conclusion

Lambek Calculus: The Basis for Lambek$^{\mathbb{D}}$

Parsing is a deductive problem

$$\dfrac{\dfrac{\cdots}{\texttt{"a"} \vdash \texttt{'a'} \oplus \texttt{'b'}} \qquad \dfrac{\cdots}{\texttt{"b"} \vdash (\texttt{'a'} \oplus \texttt{'b'})^*}}{\texttt{"ab"} \vdash (\texttt{'a'} \oplus \texttt{'b'})^*}$$

Lambek Calculus: The Basis for Lambek$^{\mathbb{D}}$

Parsing is a deductive problem

- User provides a type-level specification

$$\dfrac{\dfrac{\cdots}{\texttt{"a"} \vdash \texttt{'a'} \oplus \texttt{'b'}} \qquad \dfrac{\cdots}{\texttt{"b"} \vdash (\texttt{'a'} \oplus \texttt{'b'})^*}}{\texttt{"ab"} \vdash (\texttt{'a'} \oplus \texttt{'b'})^*}$$

Lambek Calculus: The Basis for Lambek$^{\mathbb{D}}$

Parsing is a deductive problem

- User provides a type-level specification
- Parse trees are synthesized via logic programming

$$\dfrac{\dfrac{\cdots}{\texttt{"a"} \vdash \texttt{'a'} \oplus \texttt{'b'}} \qquad \dfrac{\cdots}{\texttt{"b"} \vdash (\texttt{'a'} \oplus \texttt{'b'})^*}}{\texttt{"ab"} \vdash (\texttt{'a'} \oplus \texttt{'b'})^*}$$

Lambek Calculus: The Basis for Lambek$^{\mathtt{D}}$

Parsing is a deductive problem

- User provides a type-level specification
- Parse trees are synthesized via logic programming

$$\dfrac{\dfrac{\cdots}{\texttt{"a"} \vdash \texttt{'a'} \oplus \texttt{'b'}} \qquad \dfrac{\cdots}{\texttt{"b"} \vdash (\texttt{'a'} \oplus \texttt{'b'})^*}}{\texttt{"ab"} \vdash (\texttt{'a'} \oplus \texttt{'b'})^*}$$

Lambek$^{\mathtt{D}}$ extends the expressivity of Lambek Calculus by adding dependent types

Types in Lambek$^{\mathbb{D}}$

Types in Lambek$^{\mathbb{D}}$

- Characters `'c'`

Types in Lambek$^{\mathtt{D}}$

- Characters `'c'`
- Empty string `I`

Types in Lambek$^{\mathbb{D}}$

- Characters `'c'`
- Empty string `I`
- Concatenation $\otimes$

Types in Lambek$^{\mathbb{D}}$

- Characters `'c'`
- Empty string `I`
- Concatenation $\otimes$
- Disjunction ($\oplus$, `0`)

Types in $\mathrm{Lambek}^{\mathbb{D}}$

- Characters `'c'`
- Empty string `I`
- Concatenation $\otimes$
- Disjunction ($\oplus$, `0`)
- Conjunction (`&`, $\top$)

Types in Lambek$^{\mathbb{D}}$

- Characters `'c'`
- Empty string `I`
- Concatenation $\otimes$
- Disjunction ($\oplus$, `0`)
- Conjunction (`&`, $\top$)
- Kleene Star $\cdot^*$

Types in Lambek$^{\mathbb{D}}$

- Characters `'c'`
- Empty string $\mathtt{I}$
- Concatenation $\otimes$
- Disjunction $(\oplus, 0)$
- Conjunction $(\&, \top)$
- Kleene Star $\cdot^*$
  - Inductive linear type of lists

```
data A* : LinTy where
  nil  :  I ⊸ A*
  cons : A ⊸ A* ⊸ A*
```

Types in Lambek$^{\mathrm{D}}$

- Characters `'c'`
- Empty string `I`
- Concatenation $\otimes$
- Disjunction ($\oplus$, 0)
- Conjunction ($\&$, $\top$)
- Kleene Star $\cdot^*$
  - Inductive linear type of lists
  - Arbitrary inductive types generalize to context-free grammars and beyond

```
data A* : LinTy where
  nil  :  I ⊸ A*
  cons : A ⊸ A* ⊸ A*
```

A Linear, Ordered Theory

$\boxed{\text{Parse trees of "cat"}}$

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash (x, (y, z)) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'} \quad \checkmark$$

A Linear, Ordered Theory

$\boxed{\text{Parse trees of } \texttt{"cat"}}$

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash (x, (y, z)) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'} \quad ✓$$

Variables used in order (no exchange)

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \nvdash (y, (x, z)) : \texttt{'a'} \otimes \texttt{'c'} \otimes \texttt{'t'} \quad ✗$$

A Linear, Ordered Theory

$\boxed{\text{Parse trees of } \texttt{"cat"}}$

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash (x, (y, z)) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'} \quad ✓$$

Variables used in order (no exchange)

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \nvdash (y, (x, z)) : \texttt{'a'} \otimes \texttt{'c'} \otimes \texttt{'t'} \quad ✗$$

At least once (no weakening)

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \nvdash (y, z) : \texttt{'a'} \otimes \texttt{'t'} \quad ✗$$

A Linear, Ordered Theory

$\boxed{\text{Parse trees of } \texttt{"cat"}}$

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash (x, (y, z)) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'} \quad ✓$$

Variables used in order (no exchange)

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \nvdash (y, (x, z)) : \texttt{'a'} \otimes \texttt{'c'} \otimes \texttt{'t'} \quad ✗$$

At least once (no weakening)

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \nvdash (y, z) : \texttt{'a'} \otimes \texttt{'t'} \quad ✗$$

At most once (no contraction)

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \nvdash (x, (y, (z, z))) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'} \otimes \texttt{'t'} \quad ✗$$

A Linear, Ordered Theory

$\boxed{\text{Parse trees of } \texttt{"cat"}}$

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \vdash (x, (y, z)) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'} \quad ✓$$

Variables used in order (no exchange)

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \nvdash (y, (x, z)) : \texttt{'a'} \otimes \texttt{'c'} \otimes \texttt{'t'} \quad ✗$$

At least once (no weakening)

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \nvdash (y, z) : \texttt{'a'} \otimes \texttt{'t'} \quad ✗$$

At most once (no contraction)

$$x : \texttt{'c'}, y : \texttt{'a'}, z : \texttt{'t'} \nvdash (x, (y, (z, z))) : \texttt{'c'} \otimes \texttt{'a'} \otimes \texttt{'t'} \otimes \texttt{'t'} \quad ✗$$

These restrictions ensure parsers are sound-by-construction

Terms as Grammar Reductions

Terms as Grammar Reductions

All strings parsed by $\mathtt{(A} \otimes \mathtt{A)}^*$ are also parsed by $\mathtt{A}^*$

$$aas : (\mathtt{A} \otimes \mathtt{A})^* \vdash \mathtt{flatten}\ aas : \mathtt{A}^*$$

Terms as Grammar Reductions

All strings parsed by $\mathtt{(A} \otimes \mathtt{A)}^*$ are also parsed by $\mathtt{A}^*$

$$aas : \mathtt{(A} \otimes \mathtt{A)}^* \vdash \mathtt{flatten}\ aas : \mathtt{A}^*$$

```
flatten : (A ⊗ A)* ⊸ A*
flatten nil = nil
flatten (cons (a , a') aas) =
   cons a (cons a' (flatten aas))
```

Dyck Grammar as a Type in Lambek$^{\mathrm{D}}$

Context-free grammar of balanced parentheses

$$D \to \varepsilon \mid \mathtt{(} D \mathtt{)} D$$

Dyck Grammar as a Type in Lambek$^{\mathbb{D}}$

Context-free grammar of balanced parentheses

$$D \to \varepsilon \mid (D)\,D$$

```
data Dyck : LinTy where
  nil :  I ⊸ Dyck
  bal : '(' ⊸ Dyck ⊸ ')' ⊸ Dyck ⊸ Dyck
```

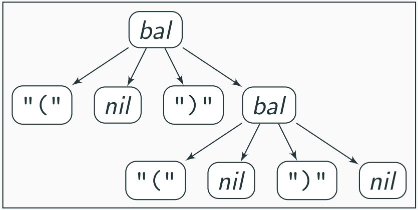
Dyck Grammar as a Type in Lambek$^{\mathbb{D}}$

Context-free grammar of balanced parentheses

$$D \to \varepsilon \mid (D)D$$

```
data Dyck : LinTy where
  nil :  I ⊸ Dyck
  bal : '(' ⊸ Dyck ⊸ ')' ⊸ Dyck ⊸ Dyck
```

$$l : \texttt{'('}, r : \texttt{')'}, l' : \texttt{'('}, r' : \texttt{')'} \vdash bal\ l\ nil\ r\ (bal\ l'\ nil\ r') : \texttt{Dyck}$$



A Parser for the Dyck Grammar

Dyck Parser

A parser for $\mathtt{Dyck}$ is a function

$$\mathtt{String} \multimap \mathtt{Dyck} \oplus \top$$

Dyck Parser

A parser for $\texttt{Dyck}$ is a function

$$\texttt{String} \multimap \texttt{Dyck} \oplus \top$$

Soundness Guaranteed for free by the type system

A Parser for the Dyck Grammar

Dyck Parser

A parser for $\texttt{Dyck}$ is a function

$$\texttt{String} \multimap \texttt{Dyck} \oplus \top$$

Soundness Guaranteed for free by the type system

Completeness Not guaranteed, as the parser can choose to always fail

A Parser for the Dyck Grammar

Complete Dyck Parser

A *complete* parser for $\texttt{Dyck}$ is a choice of type $\texttt{Dyck}_{\neg}$ and function

$$\texttt{String} \multimap \texttt{Dyck} \oplus \texttt{Dyck}_{\neg}$$

where $\texttt{Dyck} \mathbin{\&} \texttt{Dyck}_{\neg} \cong \texttt{0}$

A Parser for the Dyck Grammar

Complete Dyck Parser

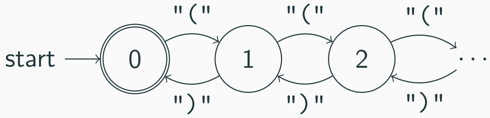
A *complete* parser for $\mathtt{Dyck}$ is a choice of type $\mathtt{Dyck}_\neg$ and function

$$\mathtt{String} \multimap \mathtt{Dyck} \oplus \mathtt{Dyck}_\neg$$

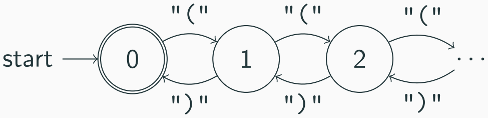
where $\mathtt{Dyck} \mathbin{\&} \mathtt{Dyck}_\neg \cong 0$

To define the parser we will use an automaton

Deterministic Automaton for the Dyck Grammar



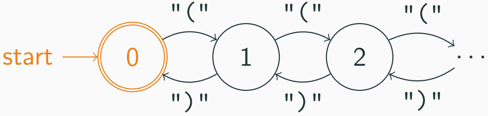
Deterministic Automaton for the Dyck Grammar




```
data Trace : Bool → ℕ → LinTy where
 eof : Trace true 0
 leftovers : ∀ n → Trace false (suc n)
 push : ∀ b n → '(' ⊸ Trace b (suc n) ⊸ Trace b n
 pop : ∀ b n → ')' ⊸ Trace b n ⊸ Trace b (suc n)
 unexpected : ')' ⊸ ⊤ ⊸ Trace false 0
```

Trace b n denotes a path through the automaton from state n ending in some state with acceptance criteria b

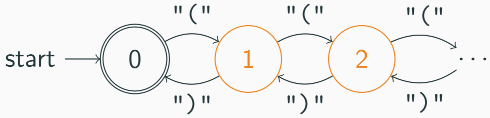
Deterministic Automaton for the Dyck Grammar




```
data Trace : Bool → ℕ → LinTy where
 eof : Trace true 0
 leftovers : ∀ n → Trace false (suc n)
 push : ∀ b n → '(' ⊸ Trace b (suc n) ⊸ Trace b n
 pop : ∀ b n → ')' ⊸ Trace b n ⊸ Trace b (suc n)
 unexpected : ')' ⊸ ⊤ ⊸ Trace false 0
```

Trace b n denotes a path through the automaton from state n ending in some state with acceptance criteria b

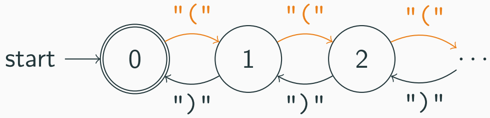
Deterministic Automaton for the Dyck Grammar




```
data Trace : Bool → ℕ → LinTy where
 eof : Trace true 0
 leftovers : ∀ n → Trace false (suc n)
 push : ∀ b n → '(' ⊸ Trace b (suc n) ⊸ Trace b n
 pop : ∀ b n → ')' ⊸ Trace b n ⊸ Trace b (suc n)
 unexpected : ')' ⊸ ⊤ ⊸ Trace false 0
```

Trace b n denotes a path through the automaton from state n ending in some state with acceptance criteria b

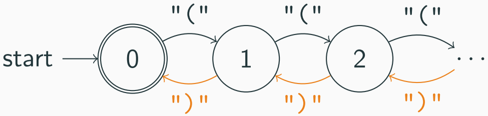
Deterministic Automaton for the Dyck Grammar




```
data Trace : Bool → ℕ → LinTy where
 eof : Trace true 0
 leftovers : ∀ n → Trace false (suc n)
 push : ∀ b n → '(' ⊸ Trace b (suc n) ⊸ Trace b n
 pop : ∀ b n → ')' ⊸ Trace b n ⊸ Trace b (suc n)
 unexpected : ')' ⊸ ⊤ ⊸ Trace false 0
```

Trace b n denotes a path through the automaton from state n ending in some state with acceptance criteria b

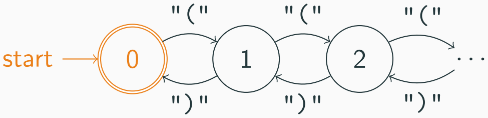
Deterministic Automaton for the Dyck Grammar




```
data Trace : Bool → ℕ → LinTy where
 eof : Trace true 0
 leftovers : ∀ n → Trace false (suc n)
 push : ∀ b n → '(' ⊸ Trace b (suc n) ⊸ Trace b n
 pop : ∀ b n → ')' ⊸ Trace b n ⊸ Trace b (suc n)
 unexpected : ')' ⊸ ⊤ ⊸ Trace false 0
```

Trace b n denotes a path through the automaton from state n ending in some state with acceptance criteria b

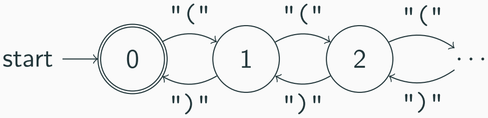
Deterministic Automaton for the Dyck Grammar




```
data Trace : Bool → ℕ → LinTy where
 eof : Trace true 0
 leftovers : ∀ n → Trace false (suc n)
 push : ∀ b n → '(' ⊸ Trace b (suc n) ⊸ Trace b n
 pop : ∀ b n → ')' ⊸ Trace b n ⊸ Trace b (suc n)
 unexpected : ')' ⊸ ⊤ ⊸ Trace false 0
```

Trace b n denotes a path through the automaton from state n ending in some state with acceptance criteria b

Deterministic Automaton for the Dyck Grammar




```
data Trace : Bool → ℕ → LinTy where
 eof : Trace true 0
 leftovers : ∀ n → Trace false (suc n)
 push : ∀ b n → '(' ⊸ Trace b (suc n) ⊸ Trace b n
 pop : ∀ b n → ')' ⊸ Trace b n ⊸ Trace b (suc n)
 unexpected : ')' ⊸ ⊤ ⊸ Trace false 0
```

Trace b n denotes a path through the automaton from state n ending in some state with acceptance criteria b

A Parser for Dyck Traces

Looks like an ordinary functional program

```
parse : String ⊸ &[ n ∈ ℕ ] (Trace true n ⊕ Trace false n)
parse nil zero = σ true eof
parse nil (suc n) = σ false leftovers
parse (cons (σ '(' a) w) n =
  let σ b tr = parse w (suc n) in
  σ b (push a tr)
parse (cons (σ ')' a) w) zero =
  σ false (unexpected a _)
parse (cons (σ ')' a) w) (suc n) =
  let σ b tr = parse w n in
  σ b (pop a tr)
```

A Parser for the Dyck Grammar

Deterministic Automaton

$$\texttt{String} \cong \texttt{Trace}\ \texttt{true}\ 0 \oplus \texttt{Trace}\ \texttt{false}\ 0$$

A Parser for the Dyck Grammar

Deterministic Automaton

$$\mathtt{String} \cong \mathtt{Trace}\ \mathtt{true}\ 0 \oplus \mathtt{Trace}\ \mathtt{false}\ 0$$

The Dyck Grammar is Recognized by the Automaton

$$\mathtt{Dyck} \cong \mathtt{Trace}\ \mathtt{true}\ 0$$

A Parser for the Dyck Grammar

Deterministic Automaton

$$\mathtt{String} \cong \mathtt{Trace}\ \mathtt{true}\ 0 \oplus \mathtt{Trace}\ \mathtt{false}\ 0$$

The Dyck Grammar is Recognized by the Automaton

$$\mathtt{Dyck} \cong \mathtt{Trace}\ \mathtt{true}\ 0$$

$$\begin{aligned} \mathtt{String} &\multimap \mathtt{Trace}\ \mathtt{true}\ 0 \oplus \mathtt{Trace}\ \mathtt{false}\ 0 \\ &\multimap \mathtt{Dyck} \oplus \mathtt{Trace}\ \mathtt{false}\ 0 \end{aligned}$$

Strategy for Building a Parser

1. Describe a grammar A and the traces of an automaton each as *linear types*
2. In the equational theory of $\text{Lambek}^{\mathtt{D}}$, prove A is equivalent to the type of traces
3. Use the equivalence to create a parser for A

Strategy for Building a Parser

1. Describe a grammar A and the traces of an automaton each as *linear types*
2. In the equational theory of Lambek$^{\mathtt{D}}$, prove A is equivalent to the type of traces
3. Use the equivalence to create a parser for A

Instantiated the same strategy in

- Parser for an $\mathbf{LL}(1)$ grammar of arithmetic expressions and a lookahead automaton

Strategy for Building a Parser

1. Describe a grammar A and the traces of an automaton each as *linear types*
2. In the equational theory of Lambek$^{\mathtt{D}}$, prove A is equivalent to the type of traces
3. Use the equivalence to create a parser for A

Instantiated the same strategy in

- Parser for an $\mathbf{LL}(1)$ grammar of arithmetic expressions and a lookahead automaton
- *Parser generator* for regular expressions and finite automata
  - Thompson's construction and Powerset construction

1. Lambek$^{\mathbb{D}}$

2. Dyck Grammar: An Extended Example

3. Conclusion

In the Paper

Proving unambiguity of grammars in $\mathsf{Lambek}^{\mathtt{D}}$

In the Paper

Proving unambiguity of grammars in $\text{Lambek}^{\texttt{D}}$

Weak equivalence v. strong equivalence of grammars

In the Paper

Proving unambiguity of grammars in $\text{Lambek}^{\texttt{D}}$

Weak equivalence v. strong equivalence of grammars

Linear-Non-Linear Dependent Types

$$\bigoplus_{\texttt{x:X}} \texttt{A x} \qquad\qquad \mathop{\&}_{\texttt{x:X}} \texttt{A x}$$

Key to describing grammars of unrestricted complexity

In the Paper

Proving unambiguity of grammars in $\mathtt{Lambek}^{\mathtt{D}}$

Weak equivalence v. strong equivalence of grammars

Linear-Non-Linear Dependent Types

$$\bigoplus_{\mathtt{x:X}} \mathtt{A}\ \mathtt{x} \qquad\qquad \mathop{\&}_{\mathtt{x:X}} \mathtt{A}\ \mathtt{x}$$

Key to describing grammars of unrestricted complexity

Denotational Semantics

Interpret grammars in the category **String** $\to$ **Set**

Ongoing and Future Work

- Encode richer parsing algorithms (e.g. **LL**, **LR**, and forms of context-sensitivity)

Ongoing and Future Work

- Encode richer parsing algorithms (e.g. **LL**, **LR**, and forms of context-sensitivity)
- Verified semantic actions to integrate with language backends

Ongoing and Future Work

- Encode richer parsing algorithms (e.g. **LL**, **LR**, and forms of context-sensitivity)
- Verified semantic actions to integrate with language backends
- Verified frontends (typechecking and semantic analysis) in a modified version of Lambek$^{\mathtt{D}}$

Dependent Lambek Calculus (Lambek$^{\mathtt{D}}$)

Lambek$^{\mathtt{D}}$: DSL for *intrinsically* verified parsing

- Functional programming language for writing verified parsers
- Ordered, linear type system guarantees parser soundness for free
  - Completeness with a little effort
- Extensible framework for implementing grammar formalisms, their parsers, and their parser generators

Universal formalism that expresses grammars as types

Implemented as an embedded DSL in Cubical Agda

- Verified parsers for selected context-free grammars
- Verified parser generator for regular expressions