The background of the slide is a painting of the Grand Canyon. On the left, a large, gnarled tree with sparse green leaves stands prominently. In the center, the deep, layered rock walls of the canyon are visible, with snow-capped peaks in the distance. On the right, a rocky cliff face rises. The overall scene is a mix of earthy tones and bright highlights from the sunlight.

Secure and Efficient Parsing via Programming Language Theory

Neel Krishnaswami & Jeremy Yallop

parsing & security



types & algebras

$$\Gamma \vdash e : \tau$$

staging & speed

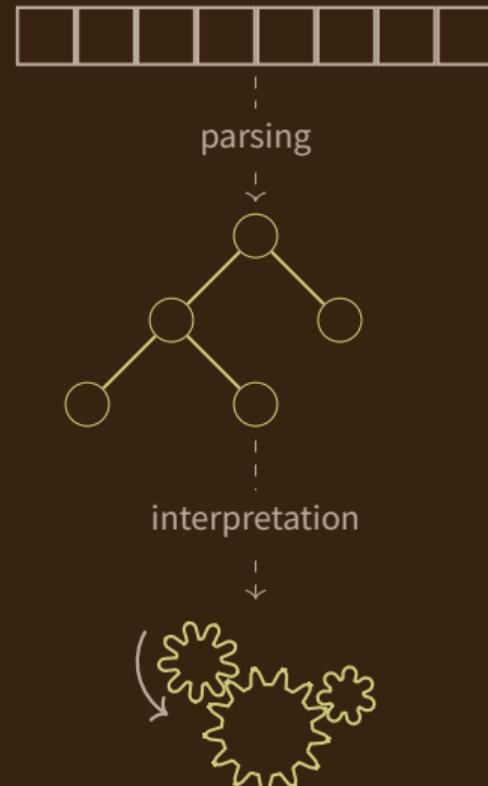
$$\ll e \gg$$

speed & correctness



Parsing and security

parsing &
security 6



Parser combinators: appeal

parsing &
security 6

declarative:
parsers resemble BNF

```
sexp = (lparen >>
         star sexp >>
         rparen)
      ⊕ atom

sexp ::= LPAREN sexp* RPAREN
      | ATOM
```

simplicity:
parsers are functions

```
star :: Parser a → Parser [a]
star p = ps ⊕ empty
where
  empty = return []
  ps = do x ← p
         xs ← star p
         return (x : xs)
```

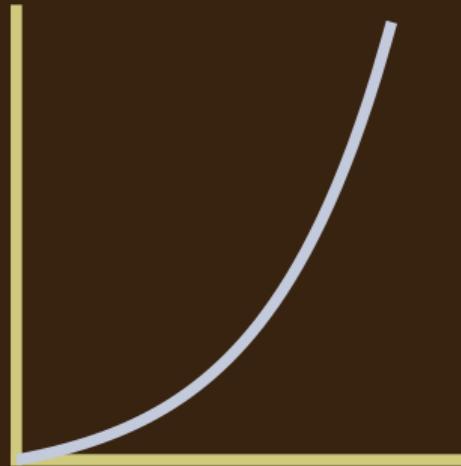
Parser combinators: pitfalls

parsing &
security 

declarative?
not in practice

complexity:
exponential (or worse)

$$p \oplus q \neq p \oplus q$$



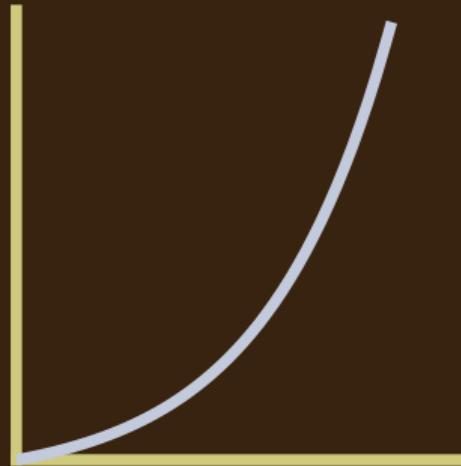
Parser combinators: pitfalls

parsing &
security 

declarative?
not in practice

complexity:
exponential (or worse)

$$p \oplus q \neq p \oplus q$$



(demonstration)

ASP and its aims

parsing &
security 

types & \vdash
algebras \equiv

using &
existing code

speed & /
scalability

asp: a combinator library with an unusual combination of features

Conventional
Interface

Unsurprising
Semantics

Guaranteed
Determinism

Competitive
Performance

parsing &
security 

types & \vdash
algebras \equiv

string &
charStream

speed &
parallelism

Abstract grammar interface (context-free expressions)

```
type α t
val chr: char → char t
val eps: unit t
val seq: α t → β t → (α * β) t
val bot: α t
val alt: α t → α t → α t
val fix: (α t → α t) → α t
val map: (α → β) → α t → β t
```

User-defined functions

```
let option r = alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also star, plus, infix, &c.)

Parsers from grammars

```
val parser: α t → (char Stream.t → α)
```

parsing &
security 

types & \vdash
algebras \equiv

string &
char types

speed & //
optimizations

Abstract grammar interface (context-free expressions)

```
type  $\alpha$  t
val chr: char → char t
val eps: unit t
val seq:  $\alpha$  t →  $\beta$  t → ( $\alpha$  *  $\beta$ ) t
val bot:  $\alpha$  t
val alt:  $\alpha$  t →  $\alpha$  t →  $\alpha$  t
val fix: ( $\alpha$  t →  $\alpha$  t) →  $\alpha$  t
val map: ( $\alpha$  →  $\beta$ ) →  $\alpha$  t →  $\beta$  t
```

Real interface: arbitrary tokens

User-defined functions

```
let option r = alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also star, plus, infix, &c.)

Parsers from grammars

```
val parser:  $\alpha$  t → (char Stream.t →  $\alpha$ )
```

parsing &
security 

types & \vdash
algebras \equiv

string &
tokens

speed &
parallelism

Abstract grammar interface (context-free expressions)

```
type  $\alpha$  t
val chr: char → char t
val eps: unit t
val seq:  $\alpha$  t →  $\beta$  t → ( $\alpha$  *  $\beta$ ) t
val bot:  $\alpha$  t
val alt:  $\alpha$  t →  $\alpha$  t →  $\alpha$  t
val fix: ( $\alpha$  t →  $\alpha$  t) →  $\alpha$  t
val map: ( $\alpha$  →  $\beta$ ) →  $\alpha$  t →  $\beta$  t
```

Real interface: arbitrary tokens

User-defined functions

```
let option r = alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also star, plus, infix, &c.)

Parsers from grammars

```
val parser:  $\alpha$  t → (char Stream.t →  $\alpha$ )
```

parsing &
security 

types & \vdash
algebras \equiv

using &
examples

speed &
optimization

Abstract grammar interface (context-free expressions)

```
type  $\alpha$  t
val chr: char → char t
val eps: unit t
val seq:  $\alpha$  t →  $\beta$  t → ( $\alpha$  *  $\beta$ ) t
val bot:  $\alpha$  t
val alt:  $\alpha$  t →  $\alpha$  t →  $\alpha$  t
val fix: ( $\alpha$  t →  $\alpha$  t) →  $\alpha$  t
val map: ( $\alpha$  →  $\beta$ ) →  $\alpha$  t →  $\beta$  t
```

Real interface: arbitrary tokens

User-defined functions

```
let option r = alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also star, plus, infix, &c.)

Parsers from grammars

```
val parser:  $\alpha$  t → (char Stream.t →  $\alpha$ )
```

parsing &
security 

types & \vdash
algebras \equiv

using &
examples

speed &
optimization

Abstract grammar interface (context-free expressions)

```
type  $\alpha$  t
val chr: char → char t
val eps: unit t
val seq:  $\alpha$  t →  $\beta$  t → ( $\alpha$  *  $\beta$ ) t
val bot:  $\alpha$  t
val alt:  $\alpha$  t →  $\alpha$  t →  $\alpha$  t
val fix: ( $\alpha$  t →  $\alpha$  t) →  $\alpha$  t
val map: ( $\alpha$  →  $\beta$ ) →  $\alpha$  t →  $\beta$  t
```

Real interface: arbitrary tokens

User-defined functions

```
let option r = alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also star, plus, infix, &c.)

Parsers from grammars

Imperative stream

```
val parser:  $\alpha$  t → (char Stream.t →  $\alpha$ )
```

parsing &
security 

types & \vdash
algebras \equiv

binding &
context

speed &
optimization

Abstract grammar interface (context-free expressions)

```
type  $\alpha$  t
val chr: char → char t
val eps: unit t
val seq:  $\alpha$  t →  $\beta$  t → ( $\alpha$  *  $\beta$ ) t
val bot:  $\alpha$  t
val alt:  $\alpha$  t →  $\alpha$  t →  $\alpha$  t
val fix: ( $\alpha$  t →  $\alpha$  t) →  $\alpha$  t
val map: ( $\alpha$  →  $\beta$ ) →  $\alpha$  t →  $\beta$  t
```

Real interface: arbitrary tokens

User-defined functions

```
let option r = alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also star, plus, infix, &c.)

May fail!

Parsers from grammars

```
val parser:  $\alpha$  t → (char Stream.t →  $\alpha$ )
```

Imperative stream

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))  
  (map (fun _ → 2) (chr 'b'))
```

```
alt (map (fun _ → 1) (chr 'a'))  
  (map (fun _ → 2) (chr 'a'))
```

```
seq (chr 'a')  
  (option (chr 'b'))
```

```
seq (option (chr 'a'))  
  (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))  
  (map (fun _ → 2) (chr 'b'))
```



```
alt (map (fun _ → 1) (chr 'a'))  
  (map (fun _ → 2) (chr 'a'))
```

```
seq (chr 'a')  
  (option (chr 'b'))
```

```
seq (option (chr 'a'))  
  (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))  
  (map (fun _ → 2) (chr 'b'))
```



```
alt (map (fun _ → 1) (chr 'a'))  
  (map (fun _ → 2) (chr 'a'))
```

```
seq (chr 'a')  
  (option (chr 'b'))
```

```
seq (option (chr 'a'))  
  (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'b'))
```



```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism X

```
seq (chr 'a')  
    (option (chr 'b'))
```

```
seq (option (chr 'a'))  
    (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'b'))
```



```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism X

```
seq (chr 'a')  
    (option (chr 'b'))
```

```
seq (option (chr 'a'))  
    (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'b'))
```



```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism X

```
seq (chr 'a')  
    (option (chr 'b'))
```



```
seq (option (chr 'a'))  
    (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'b'))
```



```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism X

```
seq (chr 'a')  
    (option (chr 'b'))
```



```
seq (option (chr 'a'))  
    (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'b'))
```



```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism X

```
seq (chr 'a')  
    (option (chr 'b'))
```



```
seq (option (chr 'a'))  
    (option (chr 'a'))
```

sequential non-determinism X

(also reject: left recursion, non-left-factored)

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'b'))
```



```
alt (map (fun _ → 1) (chr 'a'))  
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism X

```
seq (chr 'a')  
    (option (chr 'b'))
```



```
seq (option (chr 'a'))  
    (option (chr 'a'))
```

sequential non-determinism X

(also reject: left recursion, non-left-factored)

Plan: use a **type system** to decide

context-free expressions

Context-free expressions (CFEs)

parsing &
security 

types & 
algebras 

using &
rewriting

fixed point
semantics

$$g ::= \perp \mid g \vee g' \mid \epsilon \mid c \mid g \cdot g' \mid x \mid \mu x. g$$

Semantics of CFEs

$$\begin{aligned}\llbracket \perp \rrbracket \gamma &= \emptyset \\ \llbracket g \vee g' \rrbracket \gamma &= \llbracket g \rrbracket \gamma \cup \llbracket g' \rrbracket \gamma \\ \llbracket \epsilon \rrbracket \gamma &= \{\epsilon\} \\ \llbracket c \rrbracket \gamma &= \{c\} \\ \llbracket g \cdot g' \rrbracket \gamma &= \{w \cdot w' \mid w \in \llbracket g \rrbracket \gamma \wedge w' \in \llbracket g' \rrbracket \gamma\} \\ \llbracket x \rrbracket \gamma &= \gamma(x) \\ \llbracket \mu x. g \rrbracket \gamma &= fix(\lambda X. \llbracket g \rrbracket (\gamma, X/x))\end{aligned}$$

$$fix(f) = \bigcup_{i \in \mathbb{N}} L_i \text{ where } \begin{array}{lcl} L_0 & = & \emptyset \\ L_{n+1} & = & f(L_n) \end{array}$$

parsing &
security 

types & 
algebras 

using &
rewriting

speed &
parallelism

CFEs form an *idempotent semiring*

$$\begin{aligned} g_1 \vee (g_2 \vee g_3) &= (g_1 \vee g_2) \vee g_3 \\ g \vee g' &= g' \vee g \\ g \vee \perp &= g \\ g \vee g &= g \\ g_1 \cdot (g_2 \cdot g_3) &= (g_1 \cdot g_2) \cdot g_3 \\ g \cdot \epsilon &= g \\ (g_1 \vee g_2) \cdot g &= (g_1 \cdot g) \vee (g_2 \cdot g) \\ g \cdot (g_1 \vee g_2) &= (g \cdot g_1) \vee (g \cdot g_2) \\ g \cdot \perp &= \perp \\ \perp \cdot g &= \perp \end{aligned}$$

(along with some equations for μ)

parsing &
security 

types & 
algebras \equiv

Types for languages

Types $\tau \in \{ \text{NULL} : 2; \text{FIRST} : \mathcal{P}(\Sigma); \text{FLAST} : \mathcal{P}(\Sigma) \}$

$$\tau_1 \vee \tau_2 = \begin{cases} \text{NULL} &= \tau_1.\text{NULL} \vee \tau_2.\text{NULL} \\ \text{FIRST} &= \tau_1.\text{FIRST} \cup \tau_2.\text{FIRST} \\ \text{FLAST} &= \tau_1.\text{FLAST} \cup \tau_2.\text{FLAST} \end{cases}$$

Type predicates

$$\tau_1 \# \tau_2 \triangleq (\tau_1.\text{FIRST} \cap \tau_2.\text{FIRST} = \emptyset) \wedge \neg(\tau_1.\text{NULL} \wedge \tau_2.\text{NULL})$$

Properties of types

If $L \models \tau$ and $M \models \tau'$ and $\tau \# \tau'$, then $L \cup M \models \tau \vee \tau'$.

parsing &
security 

types & 
algebras 

Syntactic type system for LL(1) grammars

$$\frac{\Gamma; \Delta \vdash g : \tau \quad \Gamma; \Delta \vdash g' : \tau' \quad \tau \neq \tau'}{\Gamma; \Delta \vdash g \vee g' : \tau \vee \tau'}$$

Semantic soundness

If $\Gamma; \Delta \vdash g : \tau$ and $\gamma \models \Gamma$ and $\delta \models \Delta$ then $\llbracket g \rrbracket (\gamma, \delta) \models \tau$

Type inference

No type annotations needed (even for fixed points)

parsing &
security 

types & \vdash
algebras \equiv

parsing &
semantics

sound &
completeness

A simple parsing algorithm

$$\mathcal{P}(\Gamma; \Delta \vdash g : \tau) \in \text{Env}(\Gamma) \rightarrow \text{Env}(\Delta) \rightarrow \Sigma^* \rightarrow \Sigma^*$$

$$\mathcal{P}(\Gamma; \Delta \vdash g \vee g' : \tau \vee \tau') \ \hat{\gamma} \ \hat{\delta} \ [] =$$

$$\begin{cases} [] & \text{when } (\tau \vee \tau').\text{NULL} \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\mathcal{P}(\Gamma; \Delta \vdash g \vee g' : \tau \vee \tau') \ \hat{\gamma} \ \hat{\delta} \ ((c :: _) \text{ as } s) =$$

$$\begin{cases} \mathcal{P}(\Gamma; \Delta \vdash g : \tau) \ \hat{\gamma} \ \hat{\delta} \ s & \text{when } c \in \tau.\text{FIRST} \\ & \text{or } \tau.\text{NULL} \wedge c \notin (\tau \vee \tau').\text{FIRST} \\ \mathcal{P}(\Gamma; \Delta \vdash g' : \tau') \ \hat{\gamma} \ \hat{\delta} \ s & \text{when } c \in \tau'.\text{FIRST} \\ & \text{or } \tau'.\text{NULL} \wedge c \notin (\tau \vee \tau').\text{FIRST} \\ \text{fail} & \text{otherwise} \end{cases}$$

The parsing algorithm is **sound** and **complete**
i.e. it parses exactly the words of the language.

ASP: linear-time guarantee

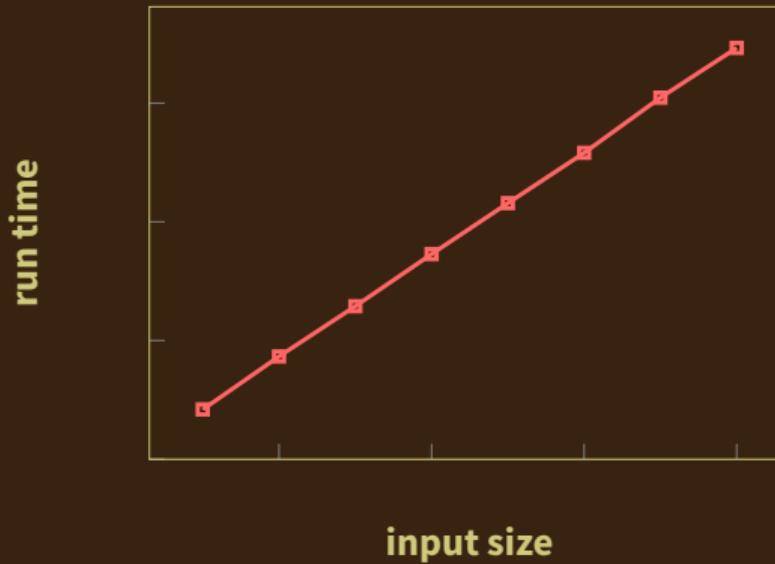
parsing &
security 

types & 
algebras 

design &
implementation

speed &
optimization

Guarantee: well-typed parsers don't back-track

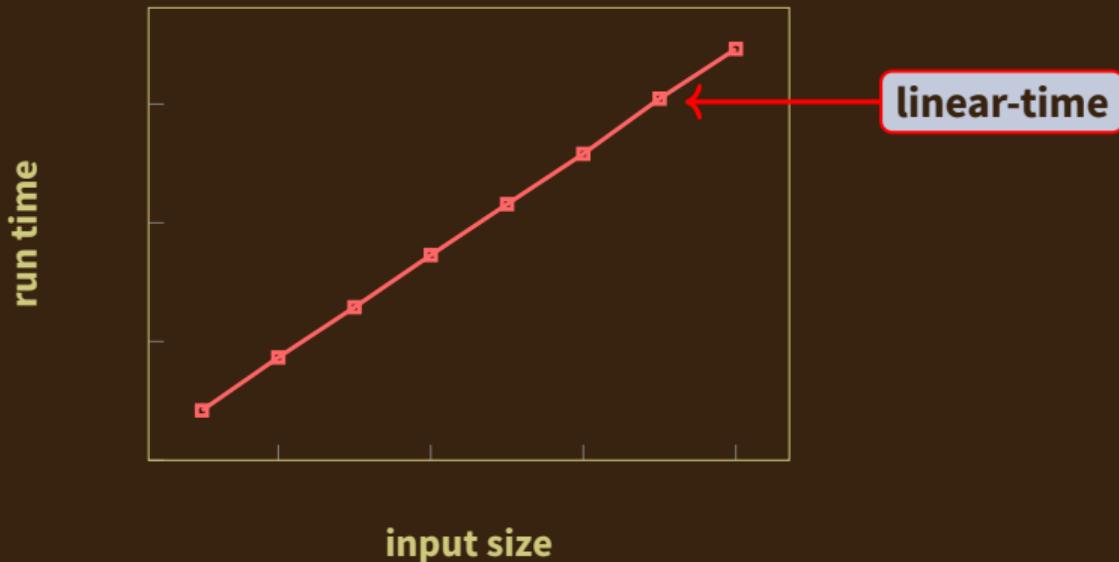


ASP: linear-time guarantee

parsing &
security 

types & 
algebras 

Guarantee: well-typed parsers don't back-track



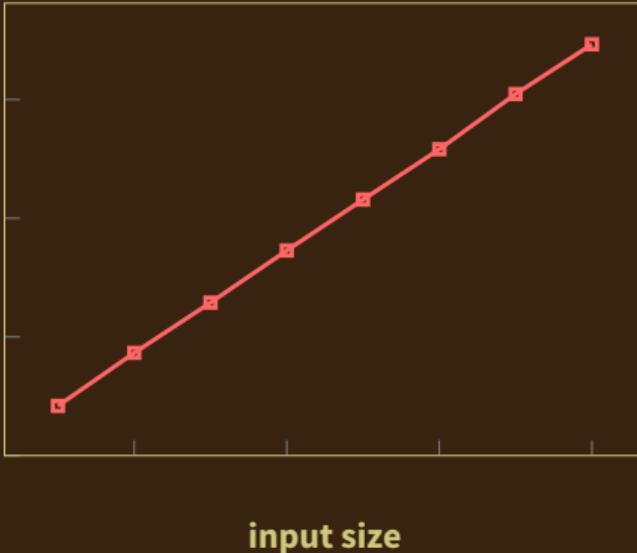
Speed: combinators vs yacc

parsing &
security 

types & 
algebras 

staging &
speed <<e>>

run time



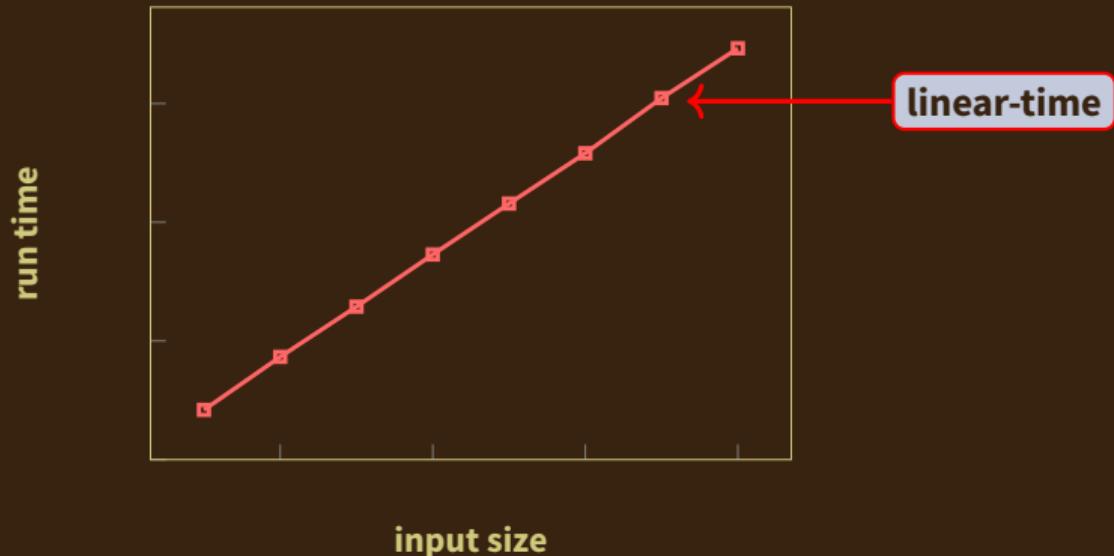
speed <<e>>

Speed: combinators vs yacc

parsing &
security 

types & 
algebras 

staging &
speed <<e>>

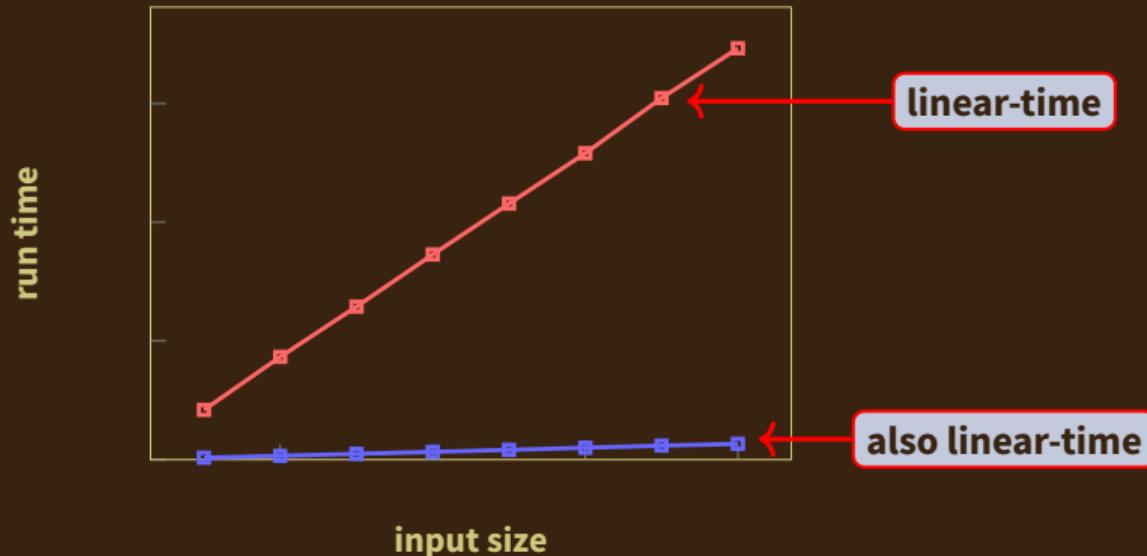


Speed: combinators vs yacc

parsing &
security 

types & 
algebras 

staging &
speed <<e>>

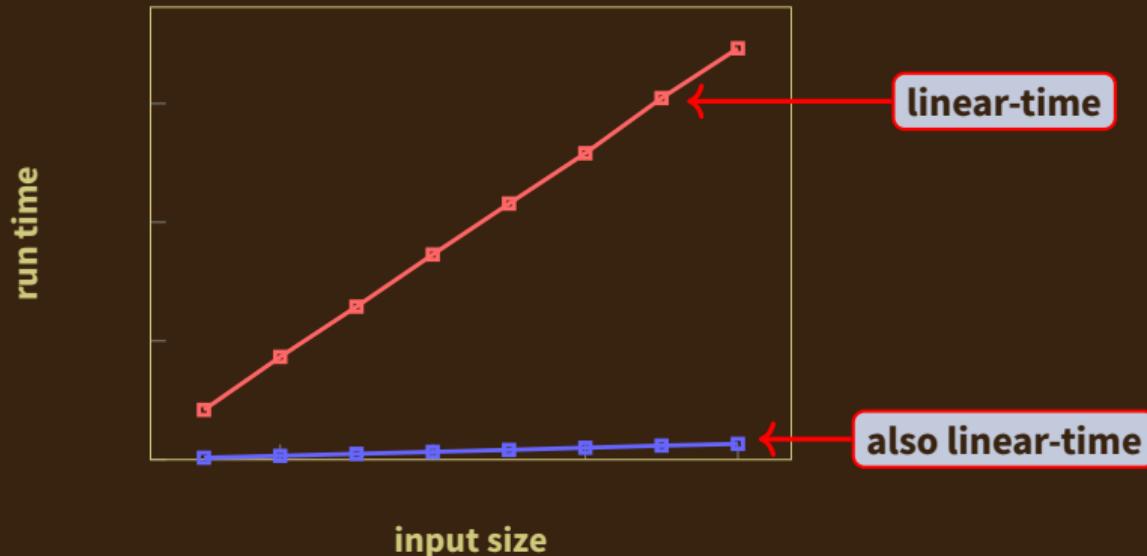


Speed: combinators vs yacc

parsing &
security 

types & 
algebras 

staging &
speed <<e>>



How can we close the gap? **Staging**

staging removes overhead

parsing &
security 

types & 
algebras 

staging &
speed  

speed  
dependencies

abstraction \implies overhead

Parser **combinators abstract** over the grammar

Abstraction carries a **performance penalty**

abstraction + staging \rightarrow no overhead

Use **staging** to specialize code once grammar is known

Delay (quote) code that accesses the input stream
`<< peek stream == 'a' >>`

Evaluate code that depends only on the grammar

staging removes overhead

parsing &
security 

types & 
algebras 

staging &
speed 

speed is all
you need

abstraction \implies overhead

Parser **combinators abstract** over the grammar

Abstraction carries a **performance penalty**

abstraction + staging \implies no overhead

Use **staging** to specialize code once grammar is known

Delay (quote) code that accesses the input stream
`<< peek stream == 'a' >>`

Evaluate code that depends only on the grammar

less-naive staging

parsing &
security 

types & \vdash
algebras \equiv

staging &
speed $\ll e \gg$

speed & $\ll e \gg$

Binding-time improvements turn **dynamic terms static**

```
if peek stream == 'a' then e
```

↓
CPS-convert

```
peek (fun c → if c == 'a' then e) stream
```



less-naive staging

parsing &
security 

types & \vdash
algebras \equiv

staging &
speed $\ll e \gg$

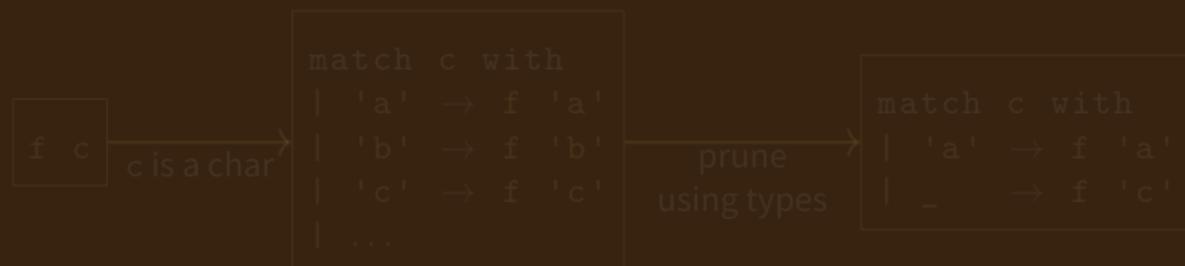
speed & //
expressiveness

Binding-time improvements turn **dynamic terms static**

if `peek stream == 'a' then e`

↓
CPS-convert

`peek (fun c → if c == 'a' then e) stream`



less-naive staging

parsing &
security 🔒

types & ⊢
algebras ≡

staging &
speed <<e>>

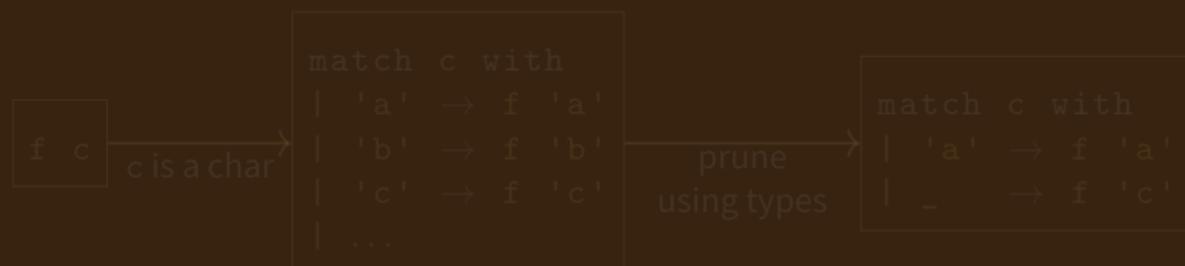
speed & //
expressiveness

Binding-time improvements turn **dynamic terms static**

```
if peek stream == 'a' then e
```

↓
CPS-convert

```
peek (fun c → if c == 'a' then e) stream
```



less-naive staging

parsing &
security 

types & \vdash
algebras \equiv

staging &
speed $\ll e \gg$

speed & //
expressiveness

Binding-time improvements turn **dynamic terms static**

if `peek stream == 'a' then e`

↓
CPS-convert

`peek (fun c → if c == 'a' then e) stream`



less-naive staging

parsing &
security 

types & \vdash
algebras \equiv

staging &
speed $\ll e \gg$

speed & $\ll e \gg$

Binding-time improvements turn **dynamic terms static**

```
if peek stream == 'a' then e
```

↓
CPS-convert

```
peek (fun c → if c == 'a' then e) stream
```



less-naive staging

parsing &
security 

types & \vdash
algebras \equiv

staging &
speed $\ll e \gg$

Binding-time improvements turn **dynamic terms static**

```
if peek stream == 'a' then e
```

↓
CPS-convert

```
peek (fun c → if c == 'a' then e) stream
```

f c
c is a char

```
match c with
| 'a' → f 'a'
| 'b' → f 'b'
| 'c' → f 'c'
| ...
```

prune
using types

```
match c with
| 'a' → f 'a'
| _   → f 'c'
```

less-naive staging

parsing &
security 

types & \vdash
algebras \equiv

staging &
speed $\ll e \gg$

Binding-time improvements turn **dynamic terms static**

```
if peek stream == 'a' then e
```

↓
CPS-convert

```
peek (fun c → if c == 'a' then e) stream
```



less-naive staging

parsing &
security 

types &  algebras 

staging &
speed  <<e>>

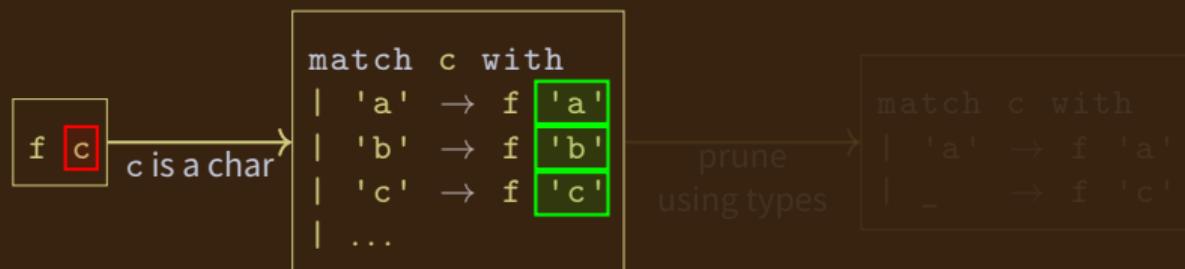
speed &  performance

Binding-time improvements turn **dynamic terms static**

```
if peek stream == 'a' then e
```

↓
CPS-convert

```
peek (fun c → if c == 'a' then e) stream
```



less-naive staging

parsing &
security 

types & 
algebras 

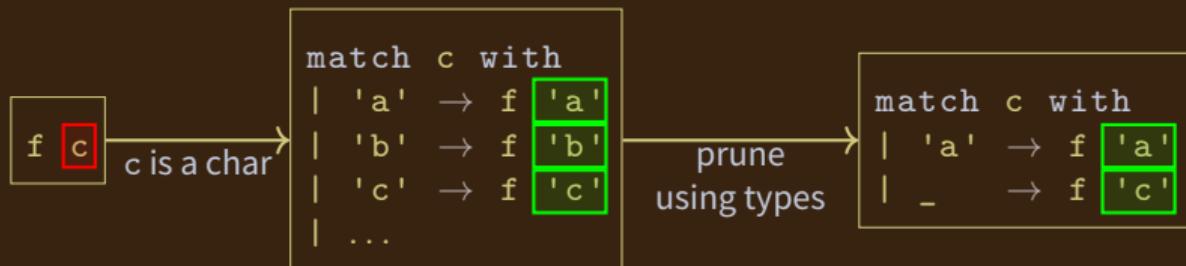
staging &
speed 

Binding-time improvements turn **dynamic terms static**

```
if peek stream == 'a' then e
```

↓
CPS-convert

```
peek (fun c → if c == 'a' then e) stream
```

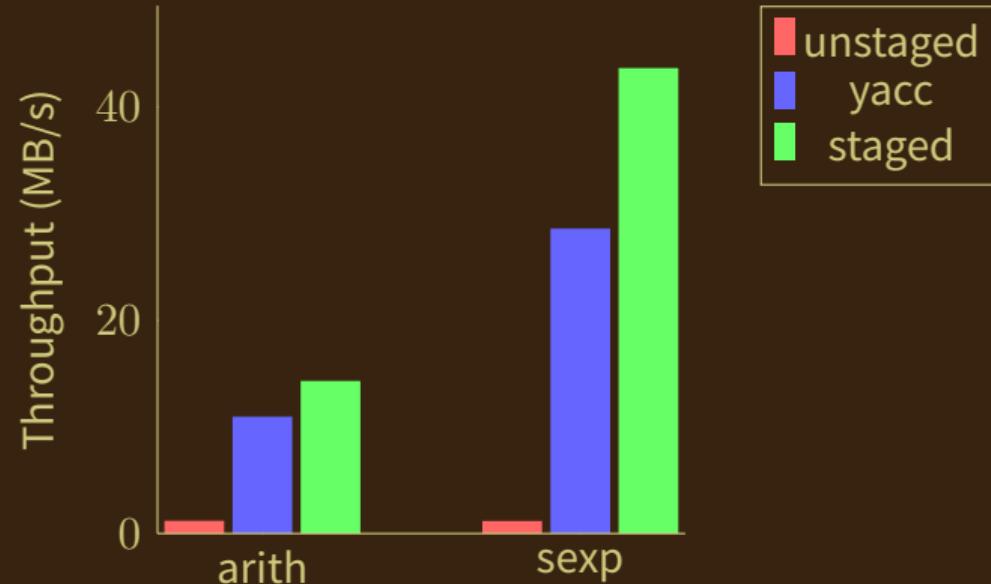


faster than yacc!

parsing &
security 

types & 
algebras 

staging &
speed <<e>>



Future: heterogeneous staging

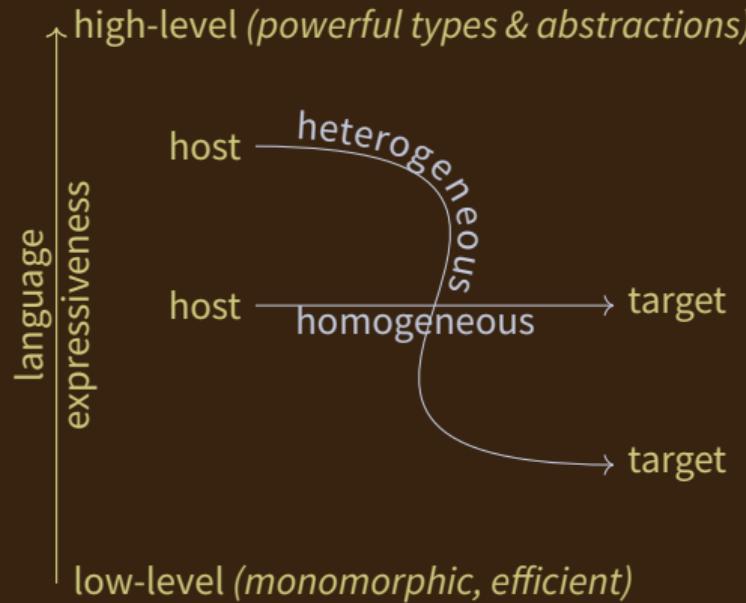
parsing &
security 

types & 
algebras 

staging &
speed <<e>>

speed & 
correctness

Homogeneity limits both guarantees and performance



Future: verified staging

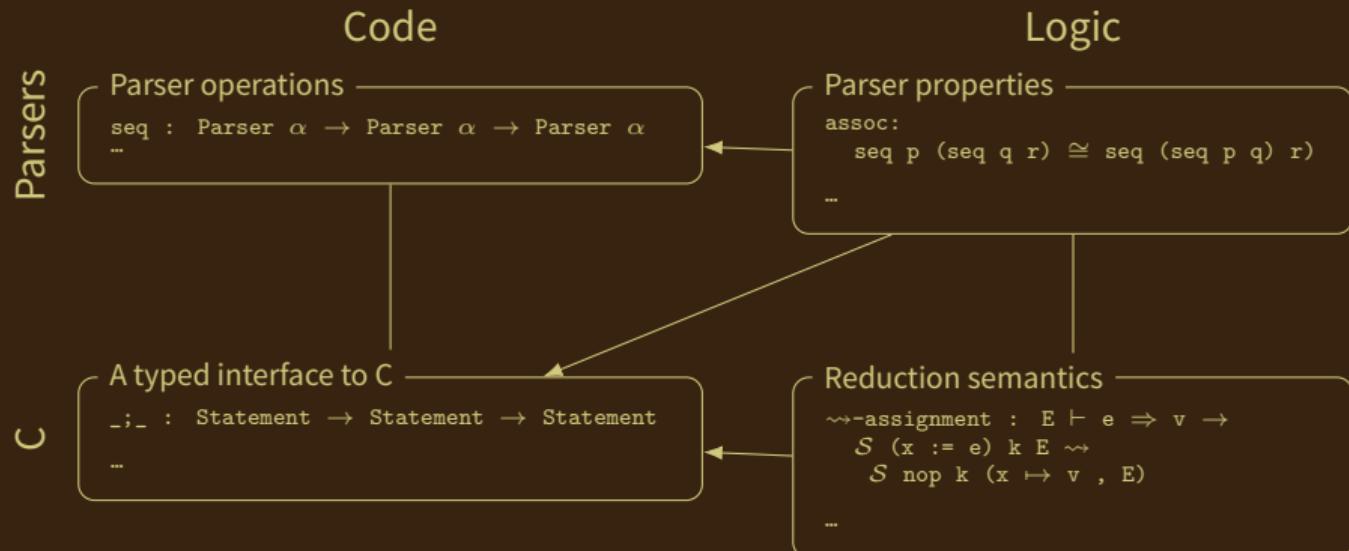
parsing &
security 

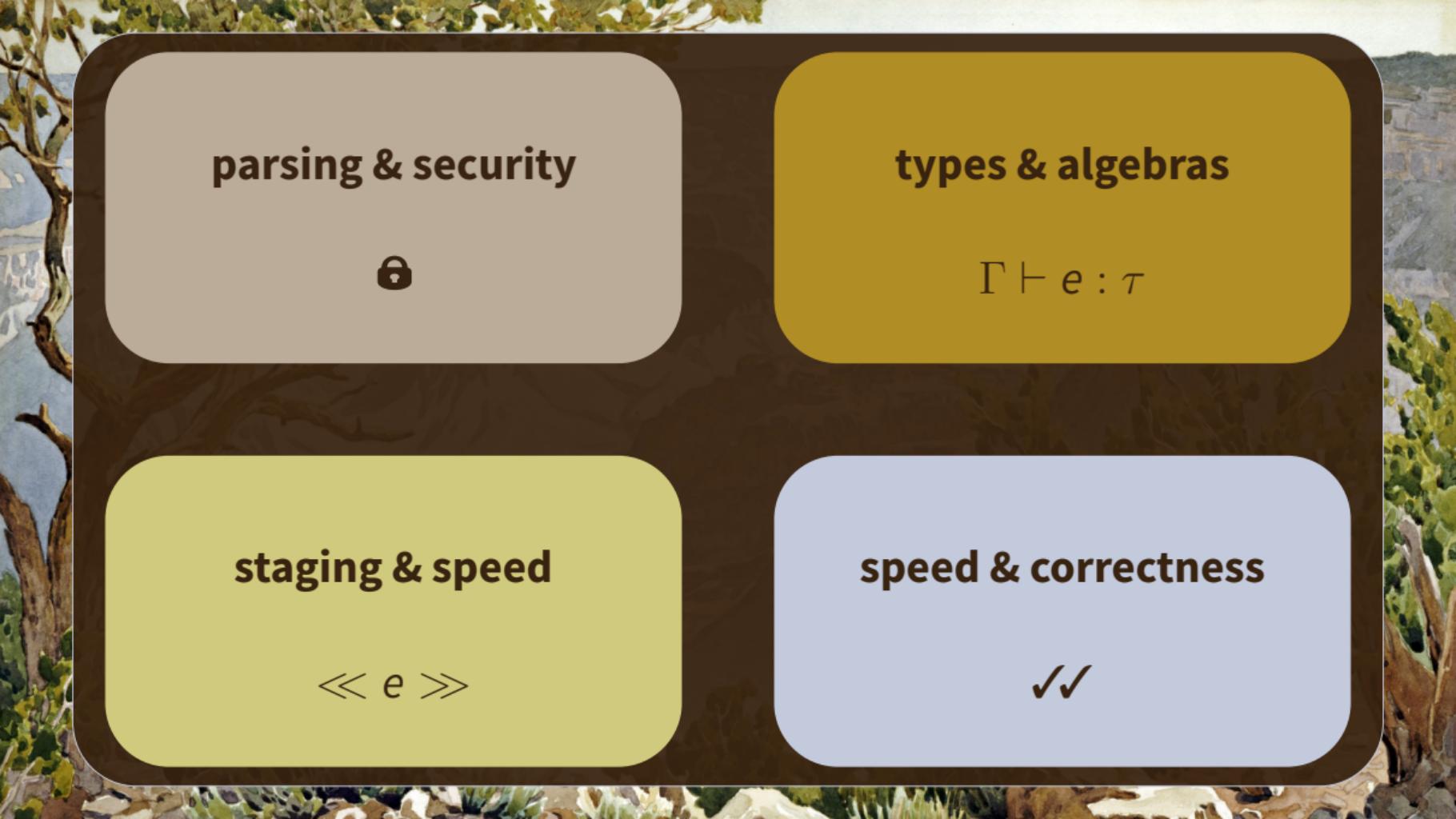
types & \vdash
algebras \equiv

staging &
speed $\ll e \gg$

speed & 
correctness

Challenge: verifying a staged program





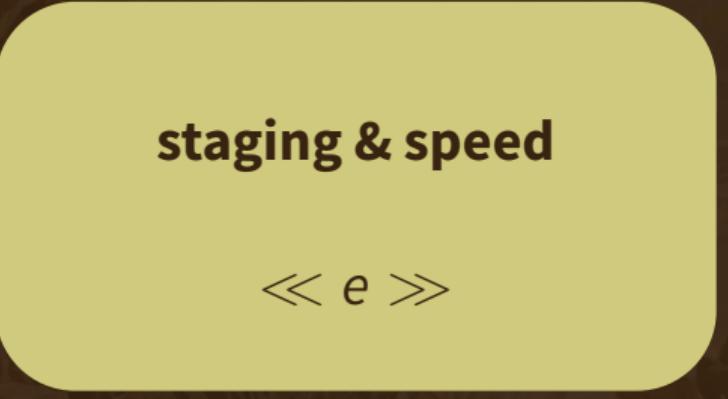
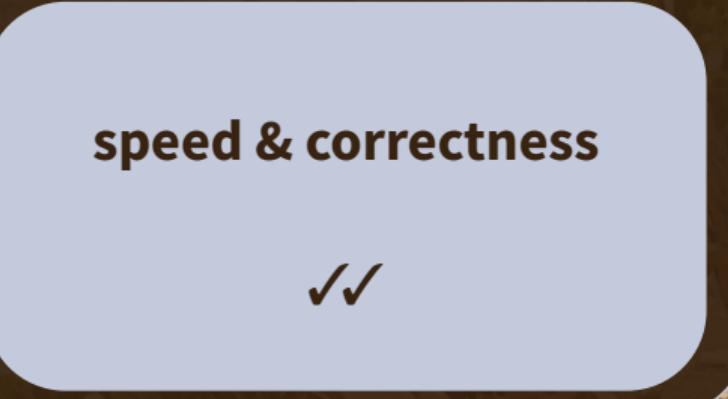
parsing & security



types & algebras

$$\Gamma \vdash e : \tau$$

staging & speed


$$\ll e \gg$$


speed & correctness

