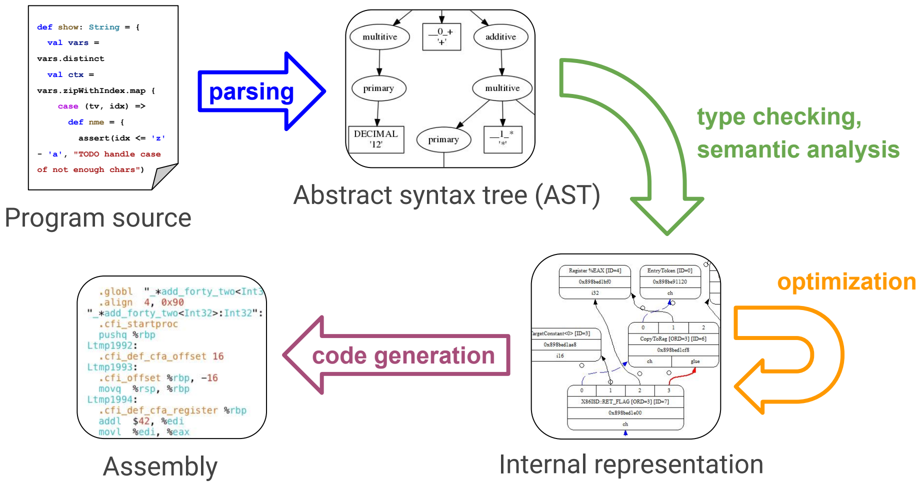
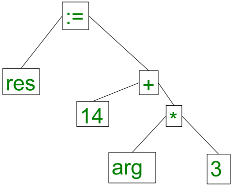
Recapitulation from Previous Lectures

Recap: Compilers



Recap: Compiler Phases

characters	`res = 14 + arg * 3`
↓ lexical analyzer	
words	`res` `=` `14` `+` `arg` `*` `3`
↓ parser	
trees	Assign(res, Plus(C(14), Times(V(arg),C(3))))
↓ name analyzer	
graphs	(variables mapped to declarations)
↓ type checker	
graphs	Assign(res:Int, Plus(C(14), Times(V(arg):Int,C(3)))):Unit
↓ intermediate code generator	
intermediate code	e.g. LLVM bitcode, JVM bytecode, Web Assembly
↓ JIT compiler or platform-specific back end	
machine code	e.g. x86, ARM, RISC-V



Recap: Transforming Text Into a Tree

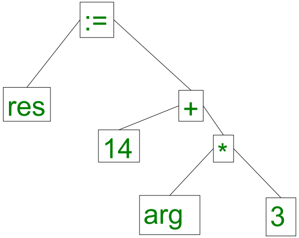
characters `res = 14 + arg * 3`

↓ lexical analyzer

words $\boxed{\texttt{res}}\ \boxed{\texttt{=}}\ \boxed{\texttt{14}}\ \boxed{\texttt{+}}\ \boxed{\texttt{arg}}\ \boxed{\texttt{*}}\ \boxed{\texttt{3}}$

↓ parser

trees Assign(res, Plus(C(14), Times(V(arg),C(3))))



First two phases:

1. lexical analyzer (lexer): sequence of characters → sequence of words
2. syntax analyzer (parser): sequence of words → tree

We will study *linear-time algorithms* for these problems.

We start with the underlying *theory of formal languages*.

Recap: Words

Let A be an alphabet $\{\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots\}$

We define words of length n, denoted A^n, as follows:

$A^0 = \{\varepsilon\}$ (only one word of length zero, always denoted ε)

For $n > 0$, $A^n = \{\, \mathrm{a}w \mid w \in A^{n-1} \,\}$

Notation '$\mathrm{a}\varepsilon$' abbreviated to 'a'.

Recap: Words

Let A be an alphabet $\{\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots\}$

We define words of length n, denoted A^n, as follows:

$A^0 = \{\varepsilon\}$ (only one word of length zero, always denoted ε)

For $n > 0$, $A^n = \{\, \mathrm{a}w \mid w \in A^{n-1} \,\}$

Notation 'aε' abbreviated to 'a'.

Concatenation: $u \cdot v$, also written uv, is associative.
We can decompose words arbitrarily, as in $w = u\mathrm{a}$ (i.e., $w = u \cdot \mathrm{a}\varepsilon$) if $|w| > 0$

Referring to letters by index: $w_{(0)} = \mathbf{1}$ $w_{(1)} = \mathbf{0}$ $w_{(2)} = \mathbf{1}$ $w_{(3)} = \mathbf{1}$

Recap: Words

Let A be an alphabet $\{\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots\}$

We define words of length n, denoted A^n, as follows:

$A^0 = \{\varepsilon\}$ (only one word of length zero, always denoted ε)

For $n > 0$, $A^n = \{\, \mathrm{a}w \mid w \in A^{n-1} \,\}$

Notation '$\mathrm{a}\varepsilon$' abbreviated to 'a'.

Concatenation: $u \cdot v$, also written uv, is associative.
We can decompose words arbitrarily, as in $w = u\mathrm{a}$ (i.e., $w = u \cdot \mathrm{a}\varepsilon$) if $|w| > 0$

Referring to letters by index: $w_{(0)} = \mathbf{1}$ $w_{(1)} = \mathbf{0}$ $w_{(2)} = \mathbf{1}$ $w_{(3)} = \mathbf{1}$

Set of all words: $A^* = \bigcup_{n \geq 0} A^n$

which means: $w \in A^*$ if and only iff there exists n such that $w \in A^n$.

Recap: Languages

A *language* over alphabet A is a set $L \subseteq A^*$.

Recap: Languages

A *language* over alphabet A is a set $L \subseteq A^*$.

Examples for $A = \{0,1\}$:

- empty language $\emptyset$;
- finite languages like $L = \{1, 10, 1001\}$;
- language L described by a characteristic function f, i.e., $L = \{\, w \in A^* \mid f(w) \,\}$

Recap: Languages

A *language* over alphabet A is a set $L \subseteq A^*$.

Examples for $A = \{0, 1\}$:

- empty language $\varnothing$;
- finite languages like $L = \{1, 10, 1001\}$;
- language L described by a characteristic function f, i.e., $L = \{\, w \in A^* \mid f(w) \,\}$

Empty language $\varnothing$ and language of the empty word $\{\varepsilon\}$ are *very different*.
$\forall L.$ we have $L\varnothing = \varnothing$ but $L\{\varepsilon\} = L$

Recap: Languages

A *language* over alphabet A is a set $L \subseteq A^*$.

Examples for $A = \{0,1\}$:

- empty language $\varnothing$;
- finite languages like $L = \{1, 10, 1001\}$;
- language L described by a characteristic function f, i.e., $L = \{\, w \in A^* \mid f(w) \,\}$

Empty language $\varnothing$ and language of the empty word $\{\varepsilon\}$ are *very different*.
$\forall L$. we have $L\varnothing = \varnothing$ but $L\{\varepsilon\} = L$

Operations on languages:

- Set operations: union ($\cup$), intersection ($\cap$), difference ($\setminus$), etc.
- Language concatenation: $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}$, also written $L_1 L_2$
- Language exponentiation: $L^0 = \{\varepsilon\}$ and $L^{n+1} = L \cdot L^n$

Recap: Regular Expressions

Syntax of regular expressions: $e ::= \varnothing \mid \varepsilon \mid c \mid (e_1 \,|\, e_2) \mid e_1 e_2 \mid e^*$

Recap: Regular Expressions

Syntax of regular expressions: $e ::= \varnothing \mid \varepsilon \mid c \mid (e_1 \,|\, e_2) \mid e_1 e_2 \mid e^*$

The *semantics* of regular expression e is the language $L(e)$ – also written L_e – where:

$$
\begin{aligned}
L(\varnothing) &= \varnothing \\
L(\varepsilon) &= \{\varepsilon\} \\
L(c) &= \{c\} \qquad\qquad (c \in A) \\
L(e_1 \,|\, e_2) &= L(e_1) \cup L(e_2) \\
L(e_1 e_2) &= L(e_1) \cdot L(e_2) \\
L(e^*) &= L(e)^*
\end{aligned}
$$

Example: $\mathit{letter}(\mathit{letter} \,|\, \mathit{digit})^*$ where $\mathit{letter} = \mathrm{a} \,|\, \mathrm{b} \,|\, \mathrm{c} \,|\, \ldots$ and $\mathit{digit} = 0 \,|\, 1 \,|\, 2 \ldots |\, 9$

Recap: Closed Operations on Regular Expressions

Regular languages/expressions are closed under these operations:

- Shorthands for finite languages, such as $[\mathrm{a..z}] = \mathrm{a} \,|\, \mathrm{b} \,|\, ... \,|\, \mathrm{z}$
- Optionality $e^{?} = e \,|\, \varepsilon$
- Repeating at least once $e^{+} = ee^{*}$
- Other repetitions $e^{k..*} = e^{k}e^{*}$ and $e^{p..q} = e^{p}(e^{?})^{q-p}$
- Complementation: $!e$, denoting $A^{*} \setminus L(e)$ — Q: how to translate?
- Intersection $e_1 \,\&\, e_2 = !(!e \,|\, !e)$, denoting $L(e_1) \cap L(e_2)$

Recap: First and Nullable (Definitions)

Formal definitions of *first* and *nullable*:

Recap: First and Nullable (Definitions)

Formal definitions of *first* and *nullable*:

$$\mathit{first}(e) \;=\; \{\, c \mid \exists w.\, cw \in L(e) \,\}$$

Recap: First and Nullable (Definitions)

Formal definitions of *first* and *nullable*:

$$\mathit{first}(e) \;=\; \{\, c \mid \exists w.\, cw \in L(e) \,\}$$

$$\mathit{nullable}(e) \;=\; \varepsilon \in L(e)$$

Recap: First and Nullable (Definitions)

Formal definitions of *first* and *nullable*:

$$\mathit{first}(e) \;=\; \{\, c \mid \exists w.\, cw \in L(e) \,\}$$

$$\mathit{nullable}(e) \;=\; \varepsilon \in L(e)$$

Examples:

$$\begin{aligned}
\mathit{first}(\mathrm{ab}^*) &= \{\mathrm{a}\} \\
\mathit{first}(\mathrm{ab}^* \,|\, \mathrm{c}) &= \{\mathrm{a}, \mathrm{c}\} \\
\mathit{first}(\mathrm{a}^*\mathrm{b}^*\mathrm{c}) &= \{\mathrm{a}, \mathrm{b}, \mathrm{c}\} \\
\mathit{first}((\mathrm{cb} \,|\, \mathrm{a}^*\mathrm{c}^*)\mathrm{d}^* e)) &= \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}
\end{aligned}$$

Recap: First and Nullable (Algorithms)

Algorithms for computing *first* and *nullable*:

$$
\begin{aligned}
\mathit{nullable}(\emptyset) &= \mathit{false} \\
\mathit{nullable}(\varepsilon) &= \mathit{true} \\
\mathit{nullable}(a) &= \mathit{false} \\
\mathit{nullable}(e_1|e_2) &= \mathit{nullable}(e_1) \vee \mathit{nullable}(e_2) \\
\mathit{nullable}(e^*) &= \mathit{true} \\
\mathit{nullable}(e_1 e_2) &= \mathit{nullable}(e_1) \wedge \mathit{nullable}(e_2)
\end{aligned}
$$

Recap: First and Nullable (Algorithms)

Algorithms for computing *first* and *nullable*:

$$
\begin{aligned}
nullable(\emptyset) &= false \\
nullable(\varepsilon) &= true \\
nullable(a) &= false \\
nullable(e_1|e_2) &= nullable(e_1) \vee nullable(e_2) \\
nullable(e^*) &= true \\
nullable(e_1 e_2) &= nullable(e_1) \wedge nullable(e_2)
\end{aligned}
$$

$$
\begin{aligned}
first(\emptyset) &= \emptyset \\
first(\varepsilon) &= \emptyset \\
first(a) &= \{a\}, \text{ for } a \in A \\
first(e_1|e_2) &= first(e_1) \cup first(e_2) \\
first(e*) &= first(e) \\
first(e_1 e_2) &= if\,(nullable(e_1))\ then\ first(e_1) \cup first(e_2) \\
&\ else\ first(e_1)
\end{aligned}
$$

Recap: Lexers

Definition of a *lexer* (aka *tokenizer*):
an ordered set of n labelled token definitions

$$\langle\ Token_1 := e_1\ ;\ Token_2 := e_2\ ;\ \ldots\ ;\ Token_n := e_n\ \rangle$$

Disambiguation rules:

Recap: Lexers

Definition of a *lexer* (aka *tokenizer*):

an ordered set of n labelled token definitions

$$\langle\ Token_1 := e_1\ ;\ Token_2 := e_2\ ;\ \ldots\ ;\ Token_n := e_n\ \rangle$$

Disambiguation rules:

- Longest-match

Recap: Lexers

Definition of a *lexer* (aka *tokenizer*):

an ordered set of n labelled token definitions

$$\langle\, Token_1 := e_1\,;\; Token_2 := e_2\,;\; \ldots\,;\; Token_n := e_n \,\rangle$$

Disambiguation rules:

- Longest-match
- First-match

Trickiness of Lexing

Consider a lexer for these two classes of tokens:

$$\left\langle \mathsf{Token}_1 := \mathsf{aaa}\,;\ \mathsf{Token}_2 := \mathsf{a}^*\mathsf{b} \right\rangle$$

Example run on aaaab:

Trickiness of Lexing

Consider a lexer for these two classes of tokens:

$$\left\langle \text{Token}_1 := \text{aaa}\,;\ \text{Token}_2 := \text{a}^*\text{b} \right\rangle$$

Example run on aaaab:

{}aaaab Token$_1$ or Token$_2$?

Trickiness of Lexing

Consider a lexer for these two classes of tokens:

$$\left\langle \mathsf{Token}_1 := \mathsf{aaa}\,;\ \mathsf{Token}_2 := \mathsf{a}^*\mathsf{b} \right\rangle$$

Example run on aaaab:

{}aaaab	Token_1 or Token_2 ?
{a}aaab	Token_1 or Token_2 ?

Trickiness of Lexing

Consider a lexer for these two classes of tokens:

$$\left\langle \mathsf{Token}_1 := \mathsf{aaa}\,;\ \mathsf{Token}_2 := \mathsf{a}^*\mathsf{b} \right\rangle$$

Example run on aaaab:

{}aaaab	Token_1 or Token_2 ?
{a}aaab	Token_1 or Token_2 ?
{aa}aab	Token_1 or Token_2 ?

Trickiness of Lexing

Consider a lexer for these two classes of tokens:

$$\left\langle \mathsf{Token}_1 := \mathsf{aaa}\,;\ \mathsf{Token}_2 := \mathsf{a}^*\mathsf{b} \right\rangle$$

Example run on aaaab:

{}aaaab	Token_1 or Token_2 ?
{a}aaab	Token_1 or Token_2 ?
{aa}aab	Token_1 or Token_2 ?
{aaa}ab	$\mathbf{Token}_1$ or Token_2 ?

Trickiness of Lexing

Consider a lexer for these two classes of tokens:

$$\left\langle\, \mathsf{Token}_1 := \mathsf{aaa}\,;\; \mathsf{Token}_2 := \mathsf{a}^*\mathsf{b} \,\right\rangle$$

Example run on aaaab:

{}aaaab	Token_1 or Token_2 ?
{a}aaab	Token_1 or Token_2 ?
{aa}aab	Token_1 or Token_2 ?
{aaa}ab	$\mathbf{Token}_1$ or Token_2 ?
{aaaa}b	Token_1 then (Token_1 or Token_2) or still Token_2 ?

Trickiness of Lexing

Consider a lexer for these two classes of tokens:

$$\left\langle \text{Token}_1 := \text{aaa}\,;\ \text{Token}_2 := \text{a}^*\text{b} \right\rangle$$

Example run on aaaab:

{}aaaab	Token_1 or Token_2 ?
{a}aaab	Token_1 or Token_2 ?
{aa}aab	Token_1 or Token_2 ?
{aaa}ab	$\textbf{Token}_1$ or Token_2 ?
{aaaa}b	Token_1 then (Token_1 or Token_2) or still Token_2 ?
{aaaab}	~~Token_1 then (Token_1 or Token_2) or still~~ $\textbf{Token}_2$!

Trickiness of Lexing

Consider a lexer for these two classes of tokens:

$$\left\langle\, \mathsf{Token}_1 := \mathsf{aaa}\,;\ \mathsf{Token}_2 := \mathsf{a}^*\mathsf{b} \,\right\rangle$$

Example run on aaaab:

{}aaaab	Token_1 or Token_2 ?
{a}aaab	Token_1 or Token_2 ?
{aa}aab	Token_1 or Token_2 ?
{aaa}ab	$\mathbf{Token}_1$ or Token_2 ?
{aaaa}b	Token_1 then (Token_1 or Token_2) or still Token_2 ?
{aaaab}	~~Token_1 then (Token_1 or Token_2) or still~~ $\mathbf{Token}_2$!

How about a run on aaaaaa? And aaaaaaaaa? And aaaaaaaab?

General Approach to Automatic Lexing

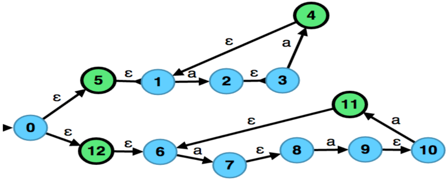
Compile regexps into some sort of *automata* with *transitions* between *states*.

General Approach to Automatic Lexing

Compile regexps into some sort of *automata* with *transitions* between *states*.

Traditional approach:

1. convert to nondeterministic finite-state automaton;
2. perform determinization (can be expensive);
3. run the resulting automaton on input (linear in the input size).

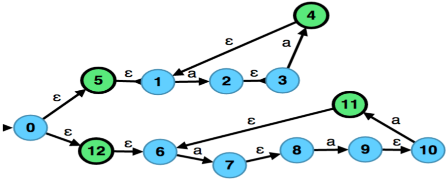


General Approach to Automatic Lexing

Compile regexps into some sort of *automata* with *transitions* between *states*.

Traditional approach:

1. convert to nondeterministic finite-state automaton;
2. perform determinization (can be expensive);
3. run the resulting automaton on input (linear in the input size).



In this course: we'll look at a more straightforward approach based on *derivatives*

Automatically Recognizing Regular Languages

Regular Expression Matcher: First Trial

$$accepts : (e, w) \rightarrow \{\textbf{true}, \textbf{false}\}$$

Regular Expression Matcher: First Trial

$$accepts : \ (e, w) \ \rightarrow \ \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) &= \\ accepts(e, cu) &= \end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : \ (e, w) \ \rightarrow \ \{\mathbf{true}, \mathbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) \ &= \ nullable(e) \\ accepts(e, cu) \ &= \end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : \ (e, w) \ \rightarrow \ \{\mathbf{true}, \mathbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) \ &= \ nullable(e) \\ accepts(e, cu) \ &= \\ &\quad \mathbf{if} \ \ e = \emptyset, \end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : (e, w) \rightarrow \{\mathbf{true}, \mathbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) &= nullable(e) \\ accepts(e, cu) &= \\ &\quad \mathbf{if}\ e = \varnothing,\ \mathbf{false} \end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : \ (e, w) \ \rightarrow \ \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) \ &= \ nullable(e) \\ accepts(e, cu) \ &= \\ &\quad \textbf{if} \ \ e = \varnothing, \ \textbf{false} \\ &\quad \textbf{if} \ \ e = \varepsilon, \end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : \ (e, w) \ \rightarrow \ \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) \ &= \ nullable(e) \\ accepts(e, cu) \ &= \\ &\textbf{if} \ \ e = \emptyset, \ \textbf{false} \\ &\textbf{if} \ \ e = \varepsilon, \ \textbf{false} \end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : \; (e, w) \; \rightarrow \; \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) \; &= \; nullable(e) \\ accepts(e, cu) \; &= \\ & \quad \textbf{if} \;\; e = \emptyset, \;\; \textbf{false} \\ & \quad \textbf{if} \;\; e = \varepsilon, \;\; \textbf{false} \\ & \quad \textbf{if} \;\; e = c, \end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts: (e, w) \rightarrow \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned}
accepts(e, \varepsilon) &= nullable(e) \\
accepts(e, cu) &= \\
&\quad \textbf{if } e = \emptyset,\ \textbf{false} \\
&\quad \textbf{if } e = \varepsilon,\ \textbf{false} \\
&\quad \textbf{if } e = c,\ \textbf{true if } u = \varepsilon,\ \textbf{false otherwise}
\end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : (e, w) \rightarrow \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned}
accepts(e, \varepsilon) &= nullable(e) \\
accepts(e, cu) &= \\
&\quad \textbf{if } e = \emptyset, \ \textbf{false} \\
&\quad \textbf{if } e = \varepsilon, \ \textbf{false} \\
&\quad \textbf{if } e = c, \ \textbf{true if } u = \varepsilon, \ \textbf{false otherwise} \\
&\quad \textbf{if } e = d \ (d \neq c),
\end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : (e, w) \rightarrow \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned}
accepts(e, \varepsilon) &= nullable(e) \\
accepts(e, cu) &= \\
&\qquad \textbf{if } e = \varnothing,\ \textbf{false} \\
&\qquad \textbf{if } e = \varepsilon,\ \textbf{false} \\
&\qquad \textbf{if } e = c,\ \textbf{true if } u = \varepsilon,\ \textbf{false otherwise} \\
&\qquad \textbf{if } e = d\ (d \neq c),\ \textbf{false}
\end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : \ (e, w) \ \rightarrow \ \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) \ &= \ nullable(e) \\ accepts(e, cu) \ &= \\ & \qquad \textbf{if} \ \ e = \varnothing, \ \textbf{false} \\ & \qquad \textbf{if} \ \ e = \varepsilon, \ \textbf{false} \\ & \qquad \textbf{if} \ \ e = c, \ \textbf{true if} \ \ u = \varepsilon, \ \textbf{false otherwise} \\ & \qquad \textbf{if} \ \ e = d \ \ (d \neq c), \ \textbf{false} \\ & \qquad \textbf{if} \ \ e = e_1 \,|\, e_2, \end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : \ (e, w) \ \rightarrow \ \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned}
accepts(e, \varepsilon) \ &= \ nullable(e) \\
accepts(e, cu) \ &= \\
& \qquad \textbf{if} \ \ e = \varnothing, \ \ \textbf{false} \\
& \qquad \textbf{if} \ \ e = \varepsilon, \ \ \textbf{false} \\
& \qquad \textbf{if} \ \ e = c, \ \ \textbf{true if} \ \ u = \varepsilon, \ \ \textbf{false otherwise} \\
& \qquad \textbf{if} \ \ e = d \ \ (d \neq c), \ \ \textbf{false} \\
& \qquad \textbf{if} \ \ e = e_1 \mid e_2, \ \ accepts(e_1, w) \vee accepts(e_2, w)
\end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : \; (e, w) \; \rightarrow \; \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned}
accepts(e, \varepsilon) \; &= \; nullable(e) \\
accepts(e, cu) \; &= \\
& \quad \textbf{if} \;\; e = \emptyset, \;\; \textbf{false} \\
& \quad \textbf{if} \;\; e = \varepsilon, \;\; \textbf{false} \\
& \quad \textbf{if} \;\; e = c, \;\; \textbf{true if} \;\; u = \varepsilon, \;\; \textbf{false otherwise} \\
& \quad \textbf{if} \;\; e = d \;\; (d \neq c), \;\; \textbf{false} \\
& \quad \textbf{if} \;\; e = e_1 \,|\, e_2, \;\; accepts(e_1, w) \lor accepts(e_2, w) \\
& \quad \textbf{if} \;\; e = e_1 e_2,
\end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : (e, w) \rightarrow \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned}
accepts(e, \varepsilon) &= nullable(e) \\
accepts(e, cu) &= \\
&\quad \textbf{if } e = \emptyset,\ \textbf{false} \\
&\quad \textbf{if } e = \varepsilon,\ \textbf{false} \\
&\quad \textbf{if } e = c,\ \textbf{true if } u = \varepsilon,\ \textbf{false otherwise} \\
&\quad \textbf{if } e = d\ (d \neq c),\ \textbf{false} \\
&\quad \textbf{if } e = e_1 \mid e_2,\ accepts(e_1, w) \vee accepts(e_2, w) \\
&\quad \textbf{if } e = e_1 e_2,\ \ldots\text{what to do? try all splits of } w\text{? (expensive)}
\end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : (e, w) \rightarrow \{\textbf{true}, \textbf{false}\}$$

$$
\begin{aligned}
accepts(e, \varepsilon) &= nullable(e) \\
accepts(e, cu) &= \\
&\quad \textbf{if } e = \emptyset,\ \textbf{false} \\
&\quad \textbf{if } e = \varepsilon,\ \textbf{false} \\
&\quad \textbf{if } e = c,\ \textbf{true if } u = \varepsilon,\ \textbf{false otherwise} \\
&\quad \textbf{if } e = d\ (d \neq c),\ \textbf{false} \\
&\quad \textbf{if } e = e_1 \mid e_2,\ accepts(e_1, w) \vee accepts(e_2, w) \\
&\quad \textbf{if } e = e_1 e_2,\ \ldots\text{what to do? try all splits of } w\text{? (expensive)} \\
&\quad \textbf{if } e = e_1^*,
\end{aligned}
$$

Regular Expression Matcher: First Trial

$$accepts : \ (e, w) \ \rightarrow \ \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned}
accepts(e, \varepsilon) \ &= \ nullable(e) \\
accepts(e, cu) \ &= \\
&\qquad \textbf{if} \ e = \varnothing, \ \textbf{false} \\
&\qquad \textbf{if} \ e = \varepsilon, \ \textbf{false} \\
&\qquad \textbf{if} \ e = c, \ \textbf{true if} \ u = \varepsilon, \ \textbf{false otherwise} \\
&\qquad \textbf{if} \ e = d \ (d \neq c), \ \textbf{false} \\
&\qquad \textbf{if} \ e = e_1 \mid e_2, \ accepts(e_1, w) \vee accepts(e_2, w) \\
&\qquad \textbf{if} \ e = e_1 e_2, \ \ldots \text{what to do? try all splits of } w\text{? (expensive)} \\
&\qquad \textbf{if} \ e = e_1^*, \ \ldots \text{similar problem here}
\end{aligned}$$

Regular Expression Matcher: First Trial

$$accepts : \ (e, w) \ \rightarrow \ \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) \ &= \ nullable(e) \\ accepts(e, cu) \ &= \end{aligned}$$

- **if** $e = \emptyset$, **false**
- **if** $e = \varepsilon$, **false**
- **if** $e = c$, **true if** $u = \varepsilon$, **false otherwise**
- **if** $e = d \ (d \neq c)$, **false**
- **if** $e = e_1 \mid e_2$, $accepts(e_1, w) \vee accepts(e_2, w)$
- **if** $e = e_1 e_2$, ...what to do? try all splits of w? (expensive)
- **if** $e = e_1^*$, ...similar problem here

Regular Expression Matcher: First Trial

$$accepts : \ (e, w) \ \rightarrow \ \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned}
accepts(e, \varepsilon) \ &= \ nullable(e) \\
accepts(e, cu) \ &= \\
& \quad \textbf{if } \ e = \emptyset, \ \textbf{false} \\
& \quad \textbf{if } \ e = \varepsilon, \ \textbf{false} \\
& \quad \textbf{if } \ e = c, \ \textbf{true if } \ u = \varepsilon, \ \textbf{false otherwise} \\
& \quad \textbf{if } \ e = d \ (d \neq c), \ \textbf{false} \\
& \quad \textbf{if } \ e = e_1 \mid e_2, \ accepts(e_1, w) \vee accepts(e_2, w) \\
& \quad \textbf{if } \ e = e_1 e_2, \ \ldots \text{what to do? try all splits of } w\text{? (expensive)} \\
& \quad \textbf{if } \ e = e_1^*, \ \ldots \text{similar problem here}
\end{aligned}$$

We need a concept of "the rest of the regexp after consuming a given letter."

Brzozowskis Derivatives, Definition

We need a concept of “the rest of the regexp after consuming a given letter.”

Brzozowskis Derivatives, Definition

We need a concept of "the rest of the regexp after consuming a given letter."

Definition (Brzozowski Derivative)

The *derivative* of regexp e with respect to letter c, written $\delta^c(e)$, is defined as:

$$L(\delta^c(e)) = \{ w \mid cw \in L(e) \}$$

Brzozowskis Derivatives, Definition

We need a concept of "the rest of the regexp after consuming a given letter."

Definition (Brzozowski Derivative)

The *derivative* of regexp e with respect to letter c, written $\delta^c(e)$, is defined as:

$$L(\delta^c(e)) = \{ w \mid cw \in L(e) \}$$

Theorem

The derivative of a regexp is a regexp.

Brzozowskis Derivatives, Definition

We need a concept of "the rest of the regexp after consuming a given letter."

Definition (Brzozowski Derivative)

The *derivative* of regexp e with respect to letter c, written $\delta^c(e)$, is defined as:

$$L(\delta^c(e)) = \{ w \mid cw \in L(e) \}$$

Theorem

The derivative of a regexp is a regexp.

Proof.

We'll give an algorithm for computing derivatives, which only yields regexps. $\square$

Brzozowskis Derivatives, Examples

Here are a few derivative examples:

$\delta^{\mathsf{a}}(\mathsf{aaa}) =$

Brzozowskis Derivatives, Examples

Here are a few derivative examples:

$$\delta^{\mathsf{a}}(\mathsf{aaa}) \;=\; \mathsf{aa}$$

$$\delta^{\mathsf{a}}(\mathsf{ab}\,|\,\mathsf{ac}\,|\,\mathsf{da}) \;=$$

Brzozowskis Derivatives, Examples

Here are a few derivative examples:

$$\delta^{\mathsf{a}}(\mathsf{aaa}) \;=\; \mathsf{aa}$$

$$\delta^{\mathsf{a}}(\mathsf{ab}\,|\,\mathsf{ac}\,|\,\mathsf{da}) \;=\; \mathsf{b}\,|\,\mathsf{c}$$

$$\delta^{\mathsf{a}}((\mathsf{ab})^{*}) \;=$$

Brzozowskis Derivatives, Examples

Here are a few derivative examples:

$$\delta^{\mathsf{a}}(\mathsf{aaa}) = \mathsf{aa}$$

$$\delta^{\mathsf{a}}(\mathsf{ab} \,|\, \mathsf{ac} \,|\, \mathsf{da}) = \mathsf{b} \,|\, \mathsf{c}$$

$$\delta^{\mathsf{a}}((\mathsf{ab})^*) = \mathsf{b}(\mathsf{ab})^*$$

$$\delta^{\mathsf{a}}((\mathsf{ab}|\mathsf{c})^*\mathsf{ad}) =$$

Brzozowskis Derivatives, Examples

Here are a few derivative examples:

$$\delta^{\mathsf{a}}(\mathsf{aaa}) \;=\; \mathsf{aa}$$

$$\delta^{\mathsf{a}}(\mathsf{ab}\,|\,\mathsf{ac}\,|\,\mathsf{da}) \;=\; \mathsf{b}\,|\,\mathsf{c}$$

$$\delta^{\mathsf{a}}((\mathsf{ab})^*) \;=\; \mathsf{b}(\mathsf{ab})^*$$

$$\delta^{\mathsf{a}}((\mathsf{ab}|\mathsf{c})^*\mathsf{ad}) \;=\; \mathsf{b}(\mathsf{ab}|\mathsf{c})^*\mathsf{ad}\,|\,\mathsf{d}$$

Brzozowskis Derivatives, Algorithm

Derivative of a regexp e with respect to letter c, written $\delta^c(e)$, can be computed as:

$$
\begin{aligned}
\delta^c(\varnothing) &= \varnothing \\
\delta^c(\varepsilon) &= \\
\delta^c(d) &= \\
\delta^c(e_1 \,|\, e_2) &= \\
\delta^c(e_1\, e_2) &= \\
\delta^c(e_1^*) &=
\end{aligned}
$$

Brzozowskis Derivatives, Algorithm

Derivative of a regexp e with respect to letter c, written $\delta^c(e)$, can be computed as:

$$\begin{aligned}
\delta^c(\emptyset) &= \emptyset \\
\delta^c(\varepsilon) &= \emptyset \\
\delta^c(d) &= \\
\delta^c(e_1 \,|\, e_2) &= \\
\delta^c(e_1\, e_2) &= \\
\delta^c(e_1^*) &=
\end{aligned}$$

Brzozowskis Derivatives, Algorithm

Derivative of a regexp e with respect to letter c, written $\delta^c(e)$, can be computed as:

$$
\begin{aligned}
\delta^c(\varnothing) &= \varnothing \\
\delta^c(\varepsilon) &= \varnothing \\
\delta^c(d) &= \begin{cases} \varepsilon & \textbf{if } d = c \\ \varnothing & \textbf{if } d \neq c \end{cases} \\
\delta^c(e_1 \mid e_2) &= \\
\delta^c(e_1\, e_2) &= \\
\delta^c(e_1^*) &=
\end{aligned}
$$

Brzozowskis Derivatives, Algorithm

Derivative of a regexp e with respect to letter c, written $\delta^c(e)$, can be computed as:

$$
\begin{aligned}
\delta^c(\varnothing) &= \varnothing \\
\delta^c(\varepsilon) &= \varnothing \\
\delta^c(d) &= \begin{cases} \varepsilon & \textbf{if } d = c \\ \varnothing & \textbf{if } d \neq c \end{cases} \\
\delta^c(e_1 \,|\, e_2) &= \delta^c(e_1) \,|\, \delta^c(e_2) \\
\delta^c(e_1\, e_2) &= \\
\delta^c(e_1^*) &=
\end{aligned}
$$

Brzozowskis Derivatives, Algorithm

Derivative of a regexp e with respect to letter c, written $\delta^c(e)$, can be computed as:

$$
\begin{aligned}
\delta^c(\varnothing) &= \varnothing \\
\delta^c(\varepsilon) &= \varnothing \\
\delta^c(d) &= \begin{cases} \varepsilon & \textbf{if } d = c \\ \varnothing & \textbf{if } d \neq c \end{cases} \\
\delta^c(e_1 \mid e_2) &= \delta^c(e_1) \mid \delta^c(e_2) \\
\delta^c(e_1\, e_2) &= \begin{cases} \delta^c(e_1)\, e_2 \mid \delta^c(e_2) & \textbf{if } \mathit{nullable}(e_1) \\ \delta^c(e_1)\, e_2 & \textbf{otherwise} \end{cases} \\
\delta^c(e_1^*) &=
\end{aligned}
$$

Brzozowskis Derivatives, Algorithm

Derivative of a regexp e with respect to letter c, written $\delta^c(e)$, can be computed as:

$$
\begin{aligned}
\delta^c(\varnothing) &= \varnothing \\
\delta^c(\varepsilon) &= \varnothing \\
\delta^c(d) &= \begin{cases} \varepsilon & \textbf{if } d = c \\ \varnothing & \textbf{if } d \neq c \end{cases} \\
\delta^c(e_1 \,|\, e_2) &= \delta^c(e_1) \mid \delta^c(e_2) \\
\delta^c(e_1\, e_2) &= \begin{cases} \delta^c(e_1)\, e_2 \mid \delta^c(e_2) & \textbf{if } \mathit{nullable}(e_1) \\ \delta^c(e_1)\, e_2 & \textbf{otherwise} \end{cases} \\
\delta^c(e_1^*) &= \delta^c(e_1)\, e_1^*
\end{aligned}
$$

Matching Regular Expressions by Derivation

Idea: given a **regexp** e and a **word** w, to know whether e **accepts** w,

simply ***derive*** e ***for all letters of*** w*, then check whether the* ***result is nullable****!*

$$accepts : (e, w) \rightarrow \{\textbf{true}, \textbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) &= nullable(e) \\ accepts(e, cw) &= accepts(\delta^c(e), w) \end{aligned}$$

Matching Regular Expressions by Derivation

Idea: given a **regexp** e and a **word** w, to know whether e **accepts** w,

*simply **derive** e **for all letters of** w, then check whether the **result is nullable**!*

$$accepts : (e, w) \rightarrow \{\mathbf{true}, \mathbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) &= nullable(e) \\ accepts(e, cw) &= accepts(\delta^c(e), w) \end{aligned}$$

Matching Regular Expressions by Derivation

Idea: given a **regexp** e and a **word** w, to know whether e **accepts** w,

simply ***derive*** *e* ***for all letters of*** *w, then check whether the* ***result is nullable****!*

$$accepts : (e, w) \rightarrow \{\mathbf{true}, \mathbf{false}\}$$

$$\begin{aligned} accepts(e, \varepsilon) &= nullable(e) \\ accepts(e, cw) &= accepts(\delta^c(e), w) \end{aligned}$$

Important: need to *cache* each intermediate result to avoid duplicating work.

Regular Expressions in Scala

Regexp Data Type in Scala

```scala
enum RegExp:
  // empty language ∅
  case Failure
  // empty word ε
  case EmptyStr
  // character a such that predicate(a)
  case CharWhere(predicate: Character => Boolean)
  // union left | right
  case Union(left: RegExp, right: RegExp)
  // concatenation first | second
  case Concat(first: RegExp, second: RegExp)
  // Kleene star underlying*
  case Star(underlying: RegExp)
```

Regexp Methods

```
enum RegExp:
  case ...

  // is this regexp nullable?
  def acceptsEmpty: Boolean = ...

  // can this regexp possibly accept some words?
  def isProductive: Boolean = this match
    case Failure => false
    case EmptyStr | CharWhere(_) | Star(_) => true // approx.
    case Union(l, r) => l.isProductive || r.isProductive
    case Concat(l, r) => l.isProductive && r.isProductive
```

Regexp Combinators 1 (extension methods)

```
extension (expr: RegExpr):
  def ~ (that: RegExpr): RegExpr = Concat(expr, that)
  def | (that: RegExpr): RegExpr = Union(expr, that)
  def * : RegExpr = Star(expr)
  def ? : RegExpr = expr | EmptyStr
  def + : RegExpr = expr ~ expr.*
  def times(n: Int): RegExpr =
    if (n <= 0) EmptyStr else expr ~ expr.times(n - 1) }
```

Examples:

```
e1 ~ e2 ~ e3 | e4
e1.* | e2.+
```

Regexp Combinators 2 (top-level definitions)

```scala
def elem(pred: Char => Boolean): RegExpr = CharWhere(pred)

def elem(char: Char): RegExpr = CharWhere(_ == char)

def elem(chars: Iterable[Char]): RegExpr =
  chars.map(elem).foldLeft[RegExpr](Failure)(_ | _)

def word(chars: Iterable[Char]): RegExpr =
  chars.map(elem).foldLeft[RegExpr](EmptyStr)(_ ~ _)

def inRange(low: Char, high: Char): RegExpr =
  elem(c => c >= low && c <= high)

// Example:
elem(_.isLetter) ~ (elem(_.isLetter) | elem(_.isDigit)).*
```

Regexp Derivation in Scala

```scala
def derive(char: Character): RegExp =
   def work(expr: RegExp): RegExp = expr match
      case Failure | EmptyStr ⇒ Failure
      case CharWhere(pred) ⇒
         if pred(char) then EmptyStr else Failure
      case Union(left, right) ⇒ work(left) | work(right)
      case Concat(left, right) ⇒
         val w = work(left) ~ right
         if left.acceptsEmpty then w | work(right) else w
      case Star(inner) ⇒ work(inner) ~ expr
   work(this)
```

Efficiently Recognizing Regular Languages

Problem of Naive Derivation: Inefficiency

Pathological cases easily arise.

Problem of Naive Derivation: Inefficiency

Pathological cases easily arise.

Example: consider a*

Problem of Naive Derivation: Inefficiency

Pathological cases easily arise.

Example: consider a^*
$\delta^{\mathsf{a}}(\mathsf{a}^*) = \varepsilon\,\mathsf{a}^*$

Problem of Naive Derivation: Inefficiency

Pathological cases easily arise.

Example: consider a^*

$\delta^{\mathsf{a}}(\mathsf{a}^*) = \varepsilon\,\mathsf{a}^*$

$\delta^{\mathsf{aa}}(\mathsf{a}^*) = \varnothing\mathsf{a}^* \mid \varepsilon\,\mathsf{a}^*$

Problem of Naive Derivation: Inefficiency

Pathological cases easily arise.

Example: consider a^*

$\boldsymbol{\delta}^{\mathsf{a}}(\mathsf{a}^*) = \boldsymbol{\varepsilon}\,\mathsf{a}^*$

$\boldsymbol{\delta}^{\mathsf{aa}}(\mathsf{a}^*) = \varnothing\mathsf{a}^* \mid \boldsymbol{\varepsilon}\,\mathsf{a}^*$

$\boldsymbol{\delta}^{\mathsf{aaa}}(\mathsf{a}^*) = \varnothing\mathsf{a}^* \mid \varnothing\mathsf{a}^* \mid \boldsymbol{\varepsilon}\,\mathsf{a}^*$

...

Every step needs to go through the list of $\varnothing\mathsf{a}^*$, which grows without limit:

$\Rightarrow$ *unbounded number of "states" and wrong complexity*

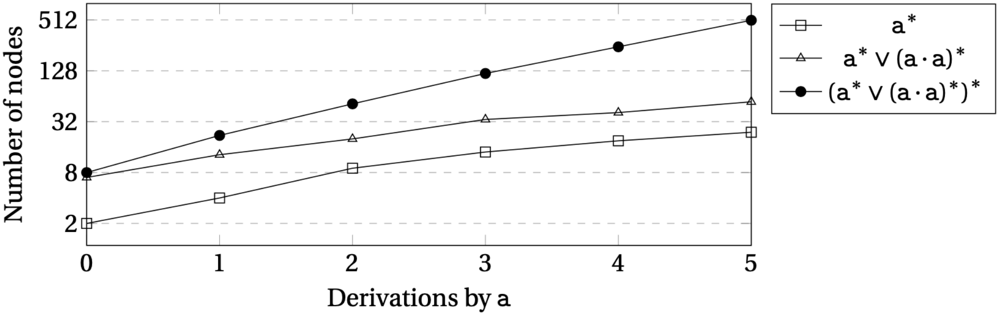
This is a simple case;

successive derivations are not always possible to "compact" on the fly...

– consider $(\mathsf{a}^* \mid (\mathsf{aa})^*)^*$

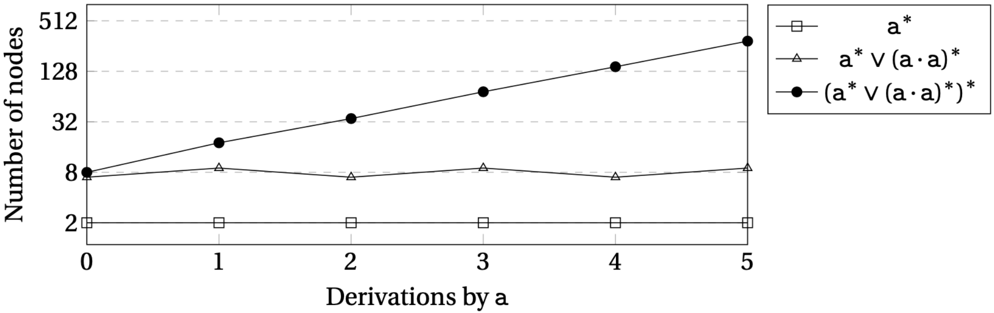
Problem of Naive Derivation: Inefficiency

Pathological cases easily arise.



Problem of Naive Derivation: Inefficiency

Not always possible to “compact” on the fly. Here it is with auto-compaction:



Idea: On-the-fly Normalization

To avoid getting into pathological cases,

it is sufficient to *normalize* the part of the expression we derive

Idea: On-the-fly Normalization

To avoid getting into pathological cases,

it is sufficient to *normalize* the part of the expression we derive

Normalization approach:

- associate all concatenations to the right: $(e_1 e_2) e_3 \Rightarrow e_1 (e_2 e_3)$
- avoid repetitions in unions: $(e_1 \mid e_2) \mid (e_3 \mid e_1) \Rightarrow e_1 \mid e_2 \mid e_3$

Normalizing Regexp Derivation in Scala

```
def deriveNorm(char: Character): RegExp =
  val disjuncted = collection.mutable.SortedSet[RegExp]()
  def work(expr: RegExp, rest: RegExp): Unit = expr match
    case CharWhere(pred) ⇒ if pred(char) then disjuncted += rest
    case Union(left, right) ⇒ work(left, rest); work(right, rest)
    case Concat(left, right) ⇒
      work(left, right ~ rest)
      if left.acceptsEmpty then work(right, rest)
    case Star(inner) ⇒ work(inner, expr ~ rest)
    case Failure | EmptyStr ⇒ ()
  work(this, EmptyStr) // register unions into `disjuncted`
  disjuncted.foldLeft[RegExp](Failure)(_ | _) // rebuild regexp
```

The Pumping Lemma

What is the Essence of Regular Expressions/Languages?

Although these languages are usually infinite,
they contain, at there core, *simple repeating patterns*.

Pumping Lemma

Recall: $w^n = \underbrace{w\,w \ldots w}_{n \ times}$

Pumping Lemma

Recall: $w^n = \underbrace{w\,w\ldots w}_{n\ times}$

Lemma (Pumping)

*For **all regexp** e, there **exists a constant** $p \geq 1$ such that*
for all $w \in L(e)$ of length at least $|w| \geq p$,
we have $w = w_0\,w_1\,w_2$ such that:

Pumping Lemma

Recall: $w^n = \underbrace{w\,w\ldots w}_{n\ \ times}$

Lemma (Pumping)

*For **all regexp** e, there **exists a constant** $p \geq 1$ such that*
for all $w \in L(e)$ of length at least $|w| \geq p$,
we have $w = w_0\,w_1\,w_2$ such that:

1. $w_1 \neq \varepsilon$
2. $|w_0\,w_1| \leq p$
3. $\forall n.\ w_0\,(w_1)^n\,w_2 \in L(e)$

Pumping Lemma

Recall: $w^n = \underbrace{w\,w \ldots w}_{n \ \textit{times}}$

Lemma (Pumping)

*For **all regexp** e, there **exists a constant** $p \geq 1$ such that*
for all $w \in L(e)$ of length at least $|w| \geq p$,
we have $w = w_0\, w_1\, w_2$ such that:

1. $w_1 \neq \varepsilon$
2. $|w_0\, w_1| \leq p$
3. $\forall n.\ w_0\,(w_1)^n\, w_2 \in L(e)$

Proof?

Proving the Pumping Lemma

Highly non-trivial.

By structural induction on e. Concatenation and Kleene star require careful case analysis on all possible length distributions.

Proving the Pumping Lemma

Highly non-trivial.

By structural induction on e. Concatenation and Kleene star require careful case analysis on all possible length distributions.

See §2.7 of Edelmann, Romain. Efficient Parsing with Derivatives and Zippers. PhD Thesis (EPFL). 2021.