CSE211: Compiler Design

Oct. 8, 2021

- **Topic**: Parsing regular expressions with derivatives
- **Questions**:
  - *How do you parse a regular expression?*
    How do you parse a context free grammar?

- δ_c *(re)*, where *re* is:
  - $re_{rhs} \cdot re_{lhs}$

    $\delta_c(re_{rhs}) \cdot re_{lhs}$ |

    if ε in re_{rhs} *then* $\delta_c(re_{lhs})$ *else {}*

Announcements

- Homework 1 is out
  - Due on the 18th
  - Get started early!
  - Today we will do parsing with derivatives

- Reading for today:
  - first 7 pages
  - https://www.ccs.neu.edu/home/turon/re-deriv.pdf
  - **Not optional! It will make part 2 of the homework much easier**

- End of module 1, starting module 2 next week

CSE211: Compiler Design

Oct. 8, 2021

- **Topic**: Parsing regular expressions with derivatives

- **Questions**:
  - *How do you parse a regular expression?*
    How do you parse a context free grammar?

- δ_c *(re)*, where *re* is:
  - re_{rhs} . re_{lhs}

    $\delta_c(re_{rhs})$. re_{lhs} |

    if ε in re_{rhs} *then* $\delta_c(re_{lhs})$ *else {}*

Parsing RE’s with Derivatives

- A simple regular expression parser implementation
  - Given an RE AST, you can parse with very few lines of code

- Think recursively!

Language Derivatives

- A language is a (potentially infinite) set of strings $\{s_1, s_2, s_3, s_4, \ldots\}$
- A language is regular if it can be captured using a regular expression
- Examples of regular languages:
  - *{"a"}, {"+"}, {"+", "-", "*", "\"}*
  - *{"1", "1+1", "1+1+1"}*
  - {""}, also called {ε}
  - {}

Subtle distinction between {} and {ε}

Language Derivatives

- The Derivative of language *L* with respect to character *c* (noted δ_c(L)) is:

for all *s* in *L*, if *s* begins with *c*, then *s[1:]* is in $\delta_c(L)$

- We'll go over some examples in the next slides

Language Derivatives Examples

- $L = \{\text{"}a\text{"}\}$

- $\delta_a(L) = \{\text{""}\}$

- $\delta_b(L) = \{\}$

Language Derivatives Examples

- $L = \{“+”, “-”, “*”, “/”\}$

- $\delta_{+}(L) = \{\varepsilon\}$

- $\delta_{\wedge}(L) = \{\}$

- $\delta_{*}(L) = \{\varepsilon\}$

Language Derivatives Examples

- *L = {"1", "1+1", "1+1+1", "1+1+1+1", …}*

- δ_{+} *(L) = {}*

- δ_{1} *(L) = {"" ,"+1", "+1+1", "+1+1+1", …}*

- δ_{1+} *(L) = {"1", "1+1", "1+1+1", ..} = L*

Language Derivatives Examples

- $L = \{\text{“aaa”}, \text{“ab”}, \text{“ba”}, \text{“bba”}\}$

- $\delta_a (L) = \{\text{“aa”}, \text{“b”}\}$

- $\delta_{aa} (L) = \{\text{“a”}\}$

- $\delta_b (L) = \{\text{“a”}, \text{“ba”}\}$

- $\delta_{ba} (L) = \{\text{“”}\}$

Regular Expressions

Recall we defined regular expressions recursively:

The three base cases: a character literal

- The RE for a character “a” is given by “a”. It matches only the character “a”
- The RE for the empty string is is given by “” or ε
- The RE for the empty set is given by {}

Regular Expressions

three recursive definitions

- The concatenation of two REs x and y is given by x.y and matches the strings of RE x **concatenated** with the strings of RE y
- The union of two REs x and y is given by x|y and matches the strings of RE x **or** the strings of RE y
- The Kleene star of an RE x is given by x* and matches the strings of RE x **repeated** 0 or more times

Regular expressions recursive definition

re =

    | {}
    | ""
    | *c* (single character)
    | re_{lhs} | re_{rhs}
    | re_{lhs} . re_{rhs}
    | $re_{starred}$ *

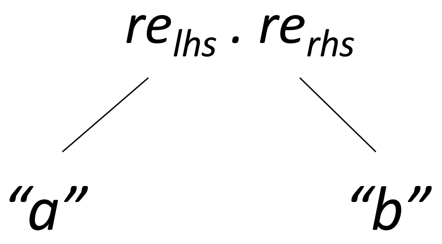
Regular expressions recursive definition

re =

- | {}
- | “”
- | *c* (single character)
- | re_{lhs} | re_{rhs}
- | re_{lhs} . re_{rhs}
- | $re_{starred}$ *

re = “a”.”b”

=



parse tree for a regular expression

`input: “a”.”b” | “c”*`

Operator	Name	Productions
|	union	: union PIPE concat | concat
.	concat	: concat CONCAT starred | starred
*	starred	: starred STAR | unit
	unit	: CHAR | “”

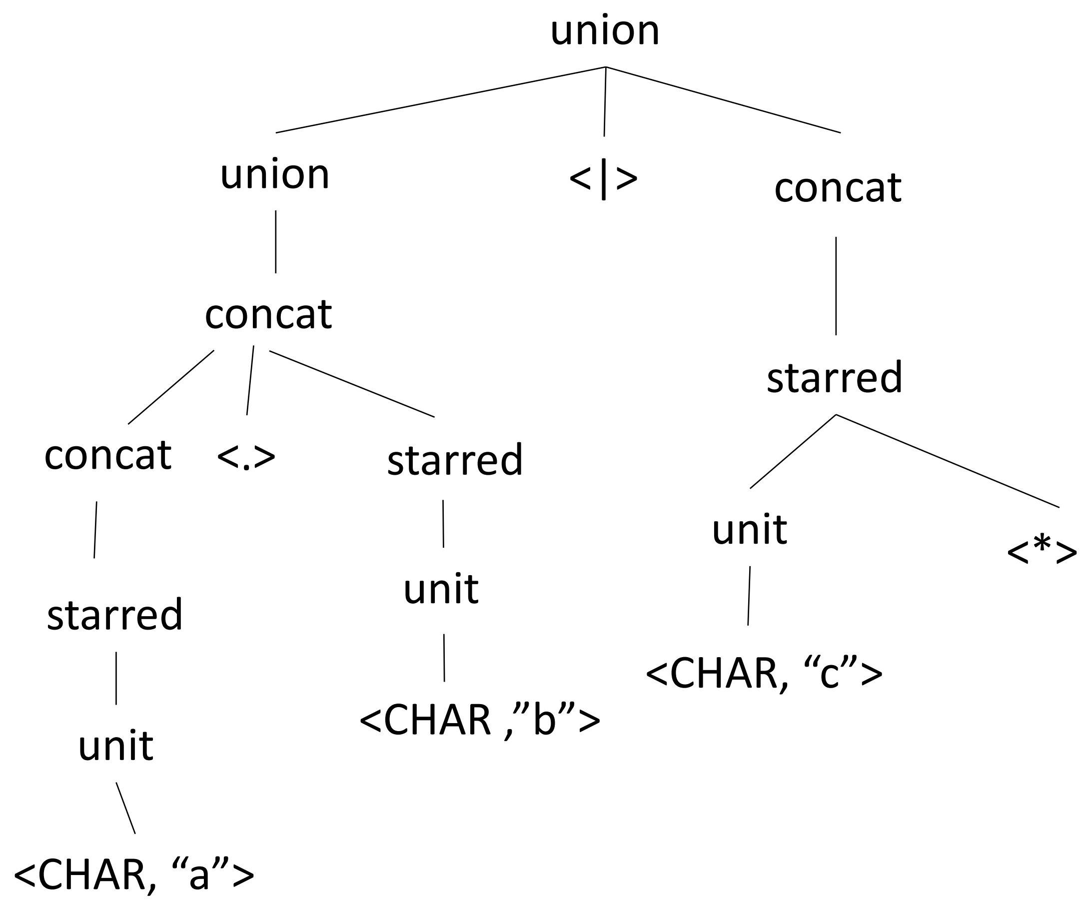
Excluding special cases for {}

parse tree for a regular expression

<table>
<tr><th>Operator</th><th>Name</th><th>Productions</th></tr>
<tr><td>|</td><td>union</td><td>: union PIPE concat
| concat</td></tr>
<tr><td>.</td><td>concat</td><td>: concat CONCAT starred
| starred</td></tr>
<tr><td>*</td><td>starred</td><td>: starred STAR
| unit</td></tr>
<tr><td></td><td>unit</td><td>: CHAR
| “”</td></tr>
</table>

Excluding special cases for {}

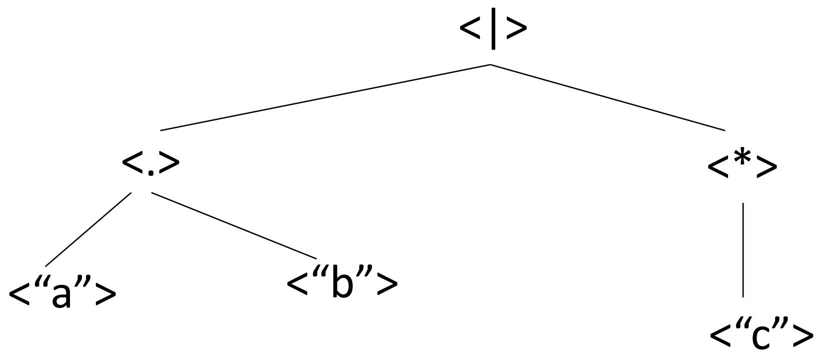
`input: “a”.”b” | “c”*`

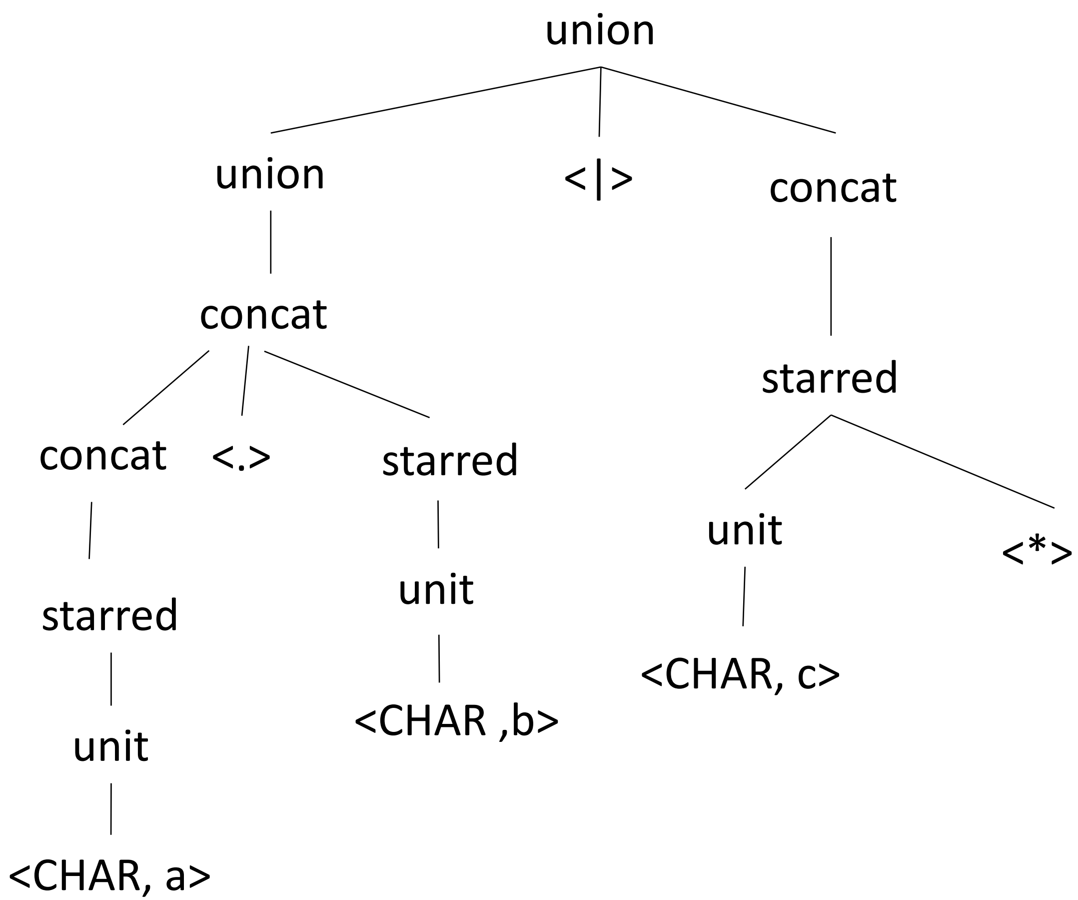


parse tree for a regular expression

`input: “a”.”b” | “c”*`

abstract syntax tree

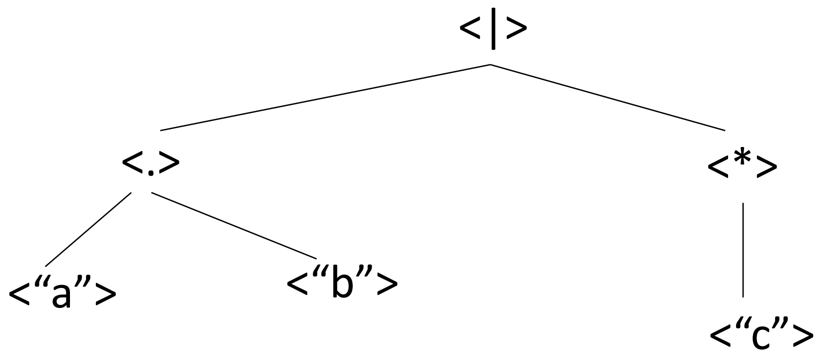




parse tree for a regular expression

`input: “a”.”b” | “c”*`

abstract syntax tree

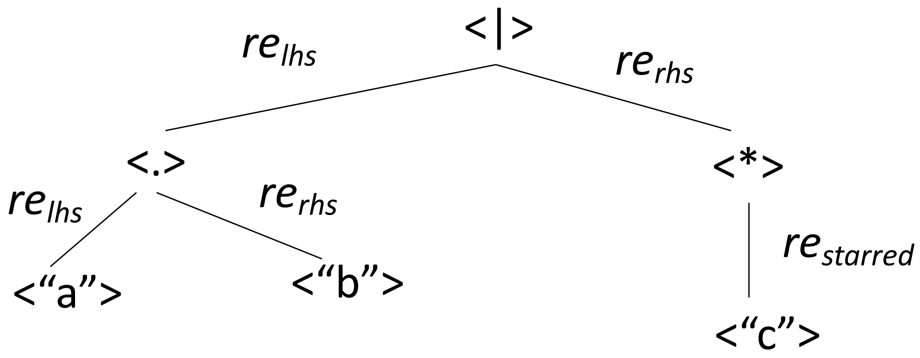


- re =
  - |{}
  - | “”
  - | a (single character)
  - | re_{lhs} | re_{rhs}
  - | re_{lhs} . re_{rhs}
  - | $re_{starred}$ *

parse tree for a regular expression

`input: “a”.”b” | “c”*`

abstract syntax tree



- re =

  |{}

  | “”

  | a (single character)

  | re_{lhs} | re_{rhs}

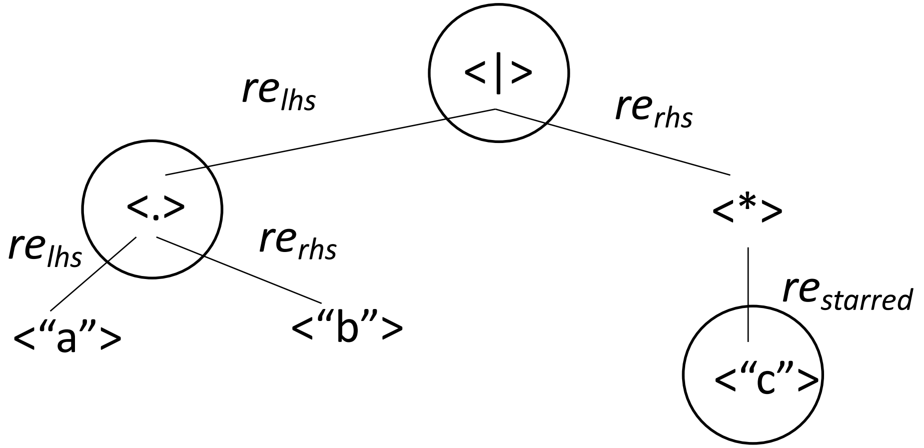
  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

parse tree for a regular expression

`input: “a”.”b” | “c”*`

abstract syntax tree



each node is
also a regular expression!

- re =

  |{}

  | “”

  | a (single character)

  | re_{lhs} | re_{rhs}

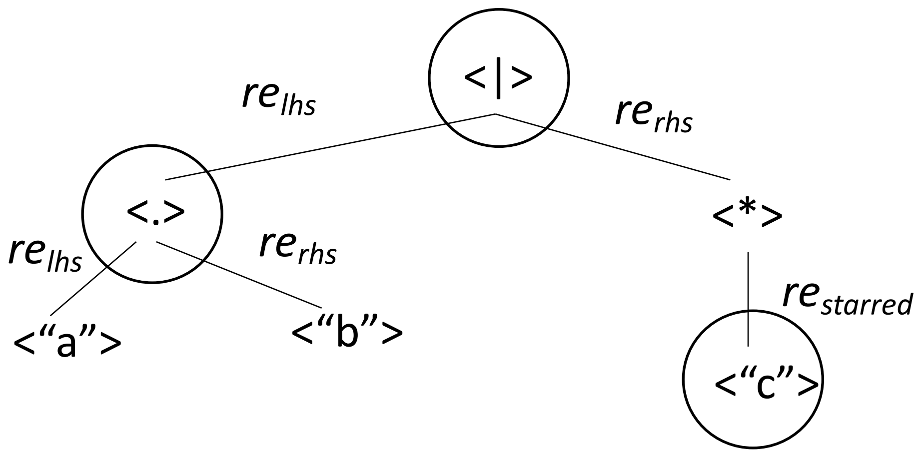
  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

parse tree for a regular expression

```
input: “a”.”b” | “c”*
```

abstract syntax tree



each node is
also a regular expression!

- *Check homework code to see AST construction*
- *Question: given a regular expression AST, how check if a string is in the language?*
- *parsing with derivatives!*

Regular expressions are closed under derivatives

- Given a regular expression *re*, any derivative of *re* is also a regular expression

- *Let’s try some!*

Regular expressions are closed under derivatives

- *re* = *"a"*
- {"a"}
- $\delta_a(re)$ = *{""} = re("")*
- $\delta_b(re)$ = *{}*

Regular expressions are closed under derivatives

- $re = \text{“}a\text{”}$
- $L(re) = \{\text{“}a\text{”}\}$
- $\delta_a(re) = \text{“”}$
- $\delta_b(re) = \{\}$

Regular expressions are closed under derivatives

- *re = “a” | “b”*
- *{“a”,”b”}*
- δ_a*(re) = {””}*
- δ_b*(re) = {“”}*

Regular expressions are closed under derivatives

- *re = “a” | “b”*
- *L(re) = {“a”, “b”}*
- δ_a*(re) = “”*
- δ_b*(re) = “”*

Regular expressions are closed under derivatives

- *re = “a”.”a” | “a”.”b”*

{”aa”, “ab”}

- δ_a*(re) = {“a”,”b”} = “a” | “b”*

- δ_b*(re) = {}*

Regular expressions are closed under derivatives

- *re = “a”.”a” | “a”.”b”*

- *L = {“aa”, “ab”}*

- $\delta_a(re)$ *= ??*

- $\delta_b(re)$ *= ??*

Regular expressions are closed under derivatives

- *re = “a”.”a” | “a”.”b”*

- *L = {“aa”, “ab”}*

- δ_a*(re) = {“a”, “b”} = ??*

- δ_b*(re) = {}*

Regular expressions are closed under derivatives

- *re = "a"."a" | "a"."b"*
- *L = {"aa", "ab"}*
- δ_a*(re) = {"a", "b"} = "a" | "b"*
- δ_b*(re) = {}*

Regular expressions are closed under derivatives

- *re = (“a”.”b”.”c”)**
- *{“”, “abc”, “abcabc”, “abcabcabc”, ...}*
- δ_a*(re) = {“bc”, “bcacb”, “bcabcabc” ...} = “b”.”c”.(“a”.”b”.”c”)**

Regular expressions are closed under derivatives

- *re = (“a”.”b”.”c”)**
- *L = {“”, ”abc”, “abcabc”, “abcabcabc” …}*
- $\delta_a(re)$ = ??

Regular expressions are closed under derivatives

- *re = (“a”.”b”.”c”)**

- *L = {“”, ”abc”, “abcabc”, “abcabcabc” …}*

- δ_a*(re) = {“bc”, “bcabc”, “bcabcabc”, ...} = ??*

Regular expressions are closed under derivatives

- *re = (“a”.”b”.”c”)**

- *L = {“”, ”abc”, “abcabc”, “abcabcabc” …}*

- δ_a*(re) = {“bc”, “bcabc”, “bcabcabc”, ...} = “b”.”c”.(”a”.”b”.”c”)**

What is a method for computing the derivative?

Consider the base cases

- δ_c *(re)* = match re with:
  - {}
    return {}
  - “”
    return {}
  - *a* (single character)
    if a == c then return {ε}
    else return {}

- re =
  | {}
  | ε
  | a (single character)
  | re_{lhs} | re_{rhs}
  | re_{lhs} . re_{rhs}
  | $re_{starred}$ *

Derivative Recursive Cases

Consider the recursive cases:

- $\delta_c(re)$ = match re with:
  - $re_{lhs} \mid re_{rhs}$

    return ? ?
  - $re_{starred}{}^*$

    return ? ?
  - $re_{lhs} . re_{rhs}$

    return ? ?

- re =
  - | {}
  - | ε
  - | a (single character)
  - | re_{lhs} | re_{rhs}
  - | re_{lhs} . re_{rhs}
  - | $re_{starred}$ *

Regular expressions are closed under derivatives

- *re = “a”.”a” | “a”.”b”*
- *L = {“aa”, “ab”}*
- δ_a*(re) = {“a”, “b”} = “a” | “b”*
- δ_b*(re) = {}*

Derivative Recursive Cases

Consider the recursive cases:

- $\delta_c\,(re)$ = match re with:
  - re_{lhs} | re_{rhs}

    return ? ?

  - $re_{starred}$*

    return ? ?

  - re_{lhs} . re_{rhs}

    return ? ?

- re =

  |{}

  | ε

  | a (single character)

  | re_{lhs} | re_{rhs}

  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

Derivative Recursive Cases

Consider the recursive cases:

- $\delta_c\,(re)$ = match re with:
  - $re_{lhs} \mid re_{rhs}$

    return $\delta_c(re_{lhs}) \mid \delta_c\,(re_{rhs})$
  - $re_{starred}{}^*$

    return ? ?
  - $re_{lhs} \,.\, re_{rhs}$

    return ? ?

- re =
  - | {}
  - | ε
  - | a (single character)
  - | $re_{lhs} \mid re_{rhs}$
  - | $re_{lhs} \,.\, re_{rhs}$
  - | $re_{starred}$ *

Regular expressions are closed under derivatives

- *re = (“a”.”b”.”c”)**

- *L = {“”, ”abc”, “abcabc”, “abcabcabc” …}*

- δ_a*(re) = {“bc”, “bcabc”, “bcabcabc”, ...} = “b”.”c”.(”a”.”b”.”c”)**

Derivative Recursive Cases

Consider the recursive cases:

- $\delta_c\,(re)$ = match re with:
  - $re_{lhs}\ |\ re_{rhs}$

    return $\delta_c(re_{lhs})\ |\ \delta_c\,(re_{rhs})$
  - $re_{starred}{}^*$

    return ? ?
  - $re_{lhs}\ .\ re_{rhs}$

    return ? ?

- re =

  |{}

  | ε

  | a (single character)

  | re_{lhs} | re_{rhs}

  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

Derivative Recursive Cases

Consider the recursive cases:

- $\delta_c\,(re)$ = match re with:
  - $re_{lhs}\mid re_{rhs}$

    return $\delta_c(re_{lhs}) \mid \delta_c\,(re_{rhs})$
  - $re_{starred}{}^*$

    return $\delta_c(re_{starred})\;.\;re_{starred}{}^*$
  - $re_{lhs}\;.\;re_{rhs}$

    return ??

- re =
  - |{}
  - | ε
  - | a (single character)
  - | re_{lhs} | re_{rhs}
  - | re_{lhs} . re_{rhs}
  - | $re_{starred}$ *

Derivative Recursive Cases

Let's look at concatenation:

- δ_c (*re*) = match re with:
  - re_{lhs} . re_{rhs}

return ??

Example:

re = *"a"."b"*

δ_a(*re*) = *"b"*

Derivative Recursive Cases

Let's look at concatenation:

- $\delta_c\,(re)$ = match re with:
  - $re_{lhs}\,.\,re_{rhs}$

    return $\delta_c(re_{lhs})\,.\,re_{rhs}$

Example:

re = "a"."b"

$\delta_a(re)$ = "b"

Derivative Recursive Cases

Let's look at concatenation:

- δ_c *(re)* = match re with:
  - re_{lhs} . re_{rhs}

    return $\delta_c(re_{lhs})$. re_{rhs}

What about?

Example:

re = "c"."a"."b"*

δ_a*(re) = "b"*

Derivative Recursive Cases

Let's look at concatenation:

- δ_c (*re*) = match re with:
  - $re_{lhs} . re_{rhs}$

return $\delta_c(re_{lhs}) . re_{rhs}$ |

if "" in re_{lhs} *then* $\delta_c(re_{rhs})$ *else* {}

Example:

re = "c"*."a"."b"

$\delta_a(re)$ = "b"

Derivative Recursive Cases

Consider the recursive cases:

- δ_c (*re*) = match re with:
  - re_{lhs} | re_{rhs}

    return $\delta_c(re_{lhs})$ | δ_c (re_{rhs})
  - $re_{starred}$*

    return $\delta_c(re_{starred})$. $re_{starred}$*
  - re_{lhs} . re_{rhs}

    return $\delta_c(re_{lhs})$. re_{rhs} |

    if "" in re_{lhs} *then* $\delta_c(re_{rhs})$ *else {}*

- re =

  |{}

  | ε

  | a (single character)

  | re_{lhs} | re_{rhs}

  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

Nullable operator

- NULL(re) =

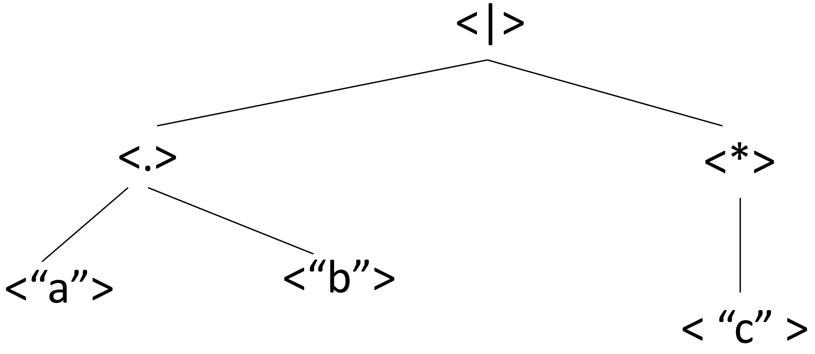
  if “” $\in re$ then: “”
  else: {}

Nullable operator

- NULL(re) =

  if “” ∈ re then: *“”*
  else: {}

implement over a RE abstract syntax tree



- re =

  | {}
  | “”
  | a (single character)
  | re_{lhs} | re_{rhs}
  | re_{lhs} . re_{rhs}
  | $re_{starred}$ *

What is a method for computing NULL?

Consider the base cases

- NULL(*re*) = match re with:
  - {}
    return {}
  - “”
    return “”
  - *a* (single character)
    return {}

- re =
  | {}
  | “”
  | a (single character)
  | re_{lhs} | re_{rhs}
  | re_{lhs} . re_{rhs}
  | $re_{starred}$ *

What is a method for computing NULL?

Consider the recursive cases:

- NULL(*re*) = match re with:
  - re_{lhs} | re_{rhs}

    return NULL(re_lhs) | NULL(re_rhs)
  - $re_{starred}$*

    return “”
  - re_{lhs} . re_{rhs}

    return NULL(re_lhs) . NULL(re_rhs)

- re =

  |{}

  | ε

  | a (single character)

  | re_{lhs} | re_{rhs}

  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

What is a method for computing NULL?

Consider the recursive cases:

- NULL(*re*) = match re with:
  - re_{lhs} | re_{rhs}

    return NULL(re_{lhs}) | NULL(re_{rhs})
  - $re_{starred}$*

    return “”
  - re_{lhs} . re_{rhs}

    return NULL(re_{lhs}) . NULL(re_{rhs})

- re =

  |{}

  | ε

  | a (single character)

  | re_{lhs} | re_{rhs}

  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

Derivative Recursive Cases

Consider the recursive cases:

- δ_c *(re)* = match re with:
  - re_{lhs} | re_{rhs}

    return $\delta_c(re_{lhs})$ | δ_c (re_{rhs})
  - $re_{starred}$*

    return $\delta_c(re_{starred})$. $re_{starred}$*
  - re_{lhs} . re_{rhs}

    return $\delta_c(re_{lhs})$. re_{rhs} |

    if ε in re_{lhs} *then* $\delta_c(re_{rhs})$ *else* {}

- re =

  |{}

  | ε

  | a (single character)

  | re_{lhs} | re_{rhs}

  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

Derivative Recursive Cases

Consider the recursive cases:

- δ_c (*re*) = match re with:
  - re_{lhs} / re_{rhs}

    return $\delta_c(re_{lhs})$ | δ_c (re_{rhs})
  - $re_{starred}$*

    return $\delta_c(re_{starred})$. $re_{starred}$*
  - re_{lhs} . re_{rhs}

    return $\delta_c(re_{lhs})$. re_{rhs} /

    $NULL(re_{lhs})$. $\delta_c(re_{rhs})$

- re =

  |{}

  | ε

  | a (single character)

  | re_{lhs} | re_{rhs}

  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 \,.\, c_2 \,.\, c_3 \ldots$ (concat of characters)

Can we check if *re* matches *s*?

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 . c_2 . c_3 \ldots$ (concat of characters)

Can we check if *re* matches *s*?

L(re) = {.. s ..}

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 . c_2 . c_3 \ldots$ (concat of characters)

Can we check if *re* matches *s*?

δ_{c1} *(re)*

L(re) = {.. s ..}

$L(\delta_{c1}$ (re)$) = \{.. s[1:] ..\}$

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 . c_2 . c_3 \ldots$ (concat of characters)

Can we check if *re* matches *s*?

	δ_{c1} *(re)*	δ_{c2} (δ_{c1} (re)) = $\delta_{c1,c2}$ *(re)*
L(re) = {.. s ..}		
	L(δ_{c1} (re)*) = {.. s[1:] ..}*	*L(*$\delta_{c1,c2}$ (re)*) = {.. s[2:] ..}*

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 . c_2 . c_3 \ldots$ (concat of characters)

Can we check if *re* matches *s*?

	$\delta_{c1}(re)$	$\delta_{c2}(\delta_{c1}(\text{re})) = \delta_{c1,c2}(re)$	$\delta_s(re)$
$L(re) = \{.. s ..\}$			
	$L(\delta_{c1}(\text{re})) = \{.. s[1:] ..\}$	$L(\delta_{c1,c2}(\text{re})) = \{.. s[2:] ..\}$	$\text{L}(\delta_s(re)) = \{.. \varepsilon ..\}$

Parsing REs with derivative

given a function δ_c to compute the derivative of an RE, the NULL function, an RE *re*, and a string $s = c_1 . c_2 . c_3 \ldots$ (concat of characters)

Can we check if *re* matches *s*?

If this is true,
Then *re matches* s

	$\delta_{c1}(re)$	$\delta_{c2}(\delta_{c1}(\mathrm{re})) = \delta_{c1,c2}(re)$	$\delta_s(re)$	
$L(re) = \{.. s ..\}$				$NULL(\delta_s(re)) ==$ “”
	$L(\delta_{c1}(\mathrm{re})) = \{.. s[1:] ..\}$	$L(\delta_{c1,c2}(\mathrm{re})) = \{.. s[2:] ..\}$	$L(\delta_s(re)) = \{..$ “” $..\}$	

Have a good weekend!

Take a look at part 2 of the homework, everything we discussed today is implemented there, with a few missing pieces for you to implement!

Next week we start module 2!