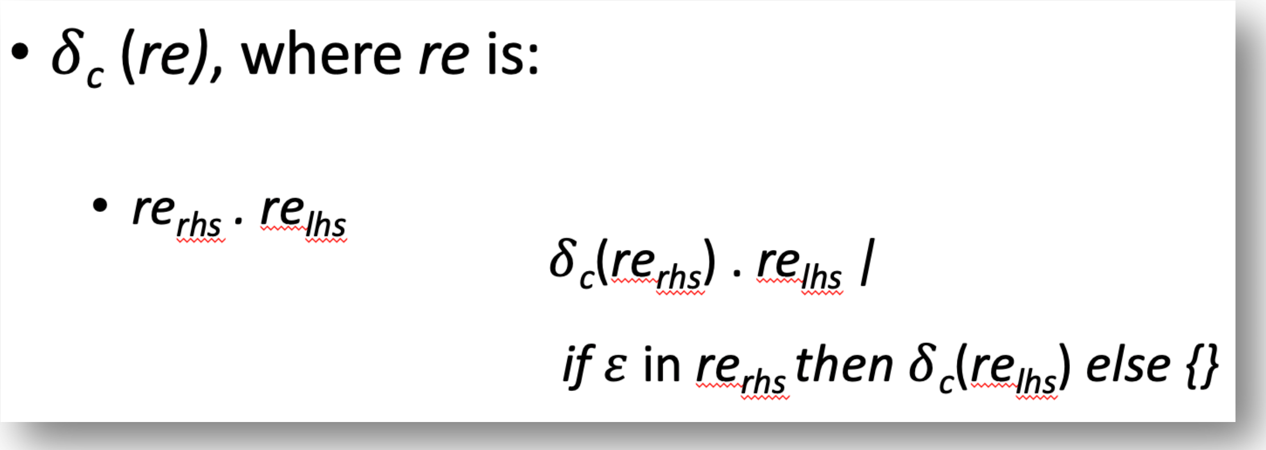
CSE211: Compiler Design

Oct. 6, 2022

- **Topic**:
  - Symbol tables
  - parsing with derivatives

- **Questions**:
  - What is “scope”
  - *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 {}*

Logistics

- Homework 1 is released

- Please find a partner ASAP
  - Someone will need to join as a third person. It is not fair to that team to have someone join late

  - Because of this, if you do not find a partner by the end of the day, I'll assign a partner.
    - But please try to self organize

  - Jeremy set up a class discord
    - I will not moderate the discord
    - Don't cheat and be nice to each other

Logistics

- Pair programming assignment:
  - Different from a group project
  - **Any work on the assignment must be done together!**
  - **Help each other with understanding!**

Logistics

- Office hours moved to Friday again this week so that you have a chance to get started on the HW
- Sign up sheet will be released at 11 AM on Friday
    - Look for a canvas announcement

Logistics

- Next week:
  - I will be in Chicago for PACT
  - Tuesdays lecture will be asynchronous
  - Office hours will move to Friday again.

- The week after:
  - I will be in Phoenix for the Khronos Group F2F
  - Thursdays lecture will be asynchronous
  - Office hours will be on Tuesday after class

- **That should be all my travel for the quarter**

I'll send out an announcement to remind you

Review

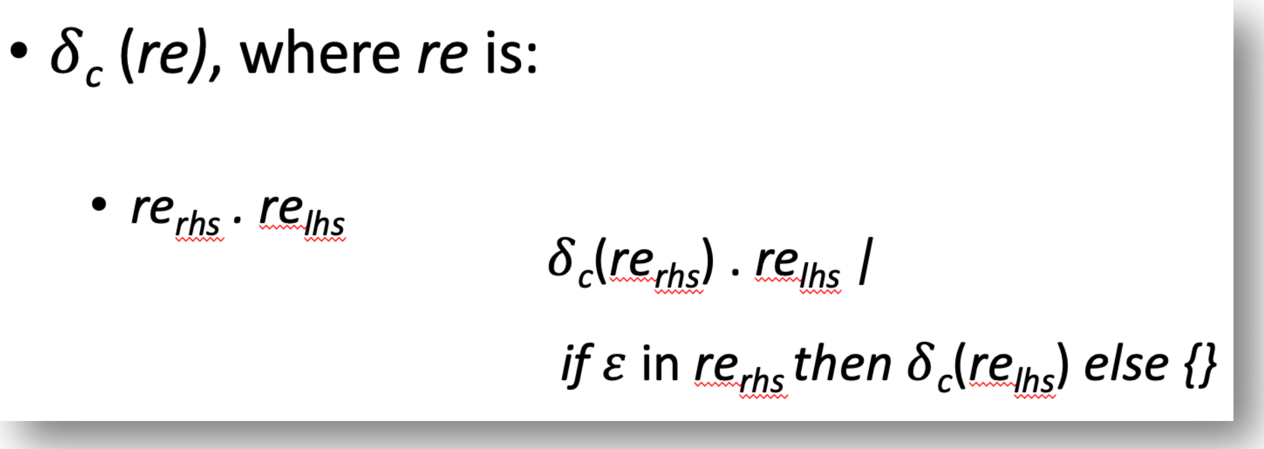
- What is a parser generator?
- How do you use a parser generator?
- What features do parser generators have that can make your life easier?
  - As a compiler writer?
  - As a compiler user?

CSE211: Compiler Design

Oct. 6, 2022

- **Topic**:
  - Symbol tables
  - parsing with derivatives

- **Questions**:
  - What is “scope”
  - *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* {}

First topic of today: Scope

- What is scope?
- Can it be determined at compile time? Can it be determined at runtime?
- C vs. Python
- Anyone have any interesting scoping rules they know of?

One consideration: Scope

- Lexical scope example

```
int x = 0;
int y = 0;
{
  int y = 0;
  x+=1;
  y+=1;
}
x+=1;
y+=1;
```

What are the final values in x and y?

How to track scope?

- Symbol table
- Global object, accessible (and mutable) by all production actions

- two methods:
  - **`lookup(id) :`** `lookup an id in the symbol table. Returns None if the id is not in the symbol table.`
  - **`insert(id,info) :`** `insert a new id (or overwrite an existing id) into the symbol table along with a set of information about the id.`

What information might we store about an id?

a very simple programming language

VARIABLE_NAME = “[a-z]+”

INCREMENT = “\+\+”

TYPE = “int”

LB = “{“

RB = “}”

SEMI = “;”

```
int x;
x++;
int y;
y++;
```

statements are either a declaration or an increment

a very simple programming language

VARIABLE_NAME = “[a-z]+”

INCREMENT = “\+\+”

TYPE = “int”

LB = “{“

RB = “}”

SEMI = “;”

```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

statements are either a declaration or an increment

a very simple programming language

VARIABLE_NAME = “[a-z]+”
INCREMENT = “\+\+”
TYPE = “int”
LB = “{“
RB = “}”
SEMI = “;”

```
int x;
{
  int y;
  x++;
  y++;
}
y++;
```

statements are either a declaration or an increment

How to track scope?

- `SymbolTable ST;`

declare_variable: TYPE VARIABLE_NAME SEMI

`{}`

Say we are matched string:

`int x;`

`lookup(id)` `: lookup an id in the symbol table. Returns None if the id is not in the symbol table.`

`insert(id,info)` `: insert a new id (or overwrite an existing id) into the symbol table along with a set of information about the id.`

How to track scope?

- `SymbolTable ST;`

declare_variable: TYPE VARIABLE_NAME SEMI

`{ST.insert(C[1],C[0])}`

Say we are matched string:

`int x;`

In this example we are storing a type

How to track scope?

- `SymbolTable ST;`

Say we are matched string:
`x++;`

variable_inc: VARIABLE_NAME INCREMENT SEMI

`{}`

lookup(id) : lookup an id in the symbol table. Returns None if the id is not in the symbol table.

insert(id,info) : insert a new id (or overwrite an existing id) into the symbol table along with a set of information about the id.

How to track scope?

- `SymbolTable ST;`

Say we are matched string:
`x++;`

variable_inc: VARIABLE_NAME INCREMENT SEMI

```
{if not ST.lookup(x):
    raise SymbolTableException;
 else:
    ... // continue}
```

How to track scope?

- `SymbolTable ST;`

```
statement : variable_inc
          | declare_variable

statement_list : statement statement_list
               | statement
```

How to track scope?

- `SymbolTable ST;`

```
statement : variable_inc
          | declare_variable

statement_list : statement statement_list
               | statement
```

adding in scope

How to track scope?

- `SymbolTable ST;`

```
statement : variable_inc
          | declare_variable
          | LBAR statement_list RBAR

statement_list : statement statement_list
               | statement
```

How to track scope?

- `SymbolTable ST;`

statement : LBAR statement_list RBAR

start a new scope S

remove the scope S

How to track scope?

- Symbol table
- four methods:
  - **`lookup(id)`** `: lookup an id in the symbol table. Returns None if the id is not in the symbol table.`
  - **`insert(id,info)`** `: insert a new id into the symbol table along with a set of information about the id.`
  - **`push_scope()`** `: push a new scope to the symbol table`
  - **`pop_scope()`** `: pop a scope from the symbol table`

How to track scope?

- `SymbolTable ST;`

statement : LBAR statement_list RBAR

start a new scope S

remove the scope S

Think about how to solve with production rules

How to implement a symbol table?

- Thoughts? What data structures are good at mapping strings?

- Symbol table
- four methods:
  - **`lookup(id)`** `: lookup an id in the symbol table. Returns None if the id is not in the symbol table.`

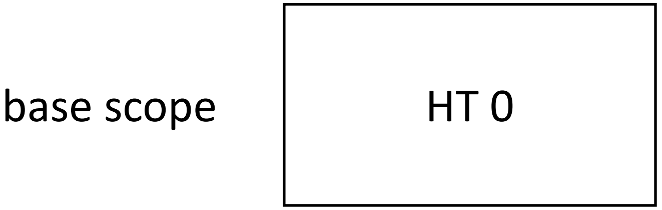
  - **`insert(id,info)`** `: insert a new id into the symbol table along with a set of information about the id.`

  - **`push_scope()`** `: push a new scope to the symbol table`

  - **`pop_scope()`** `: pop a scope from the symbol table`

How to implement a symbol table?

- Many ways to implement:
- A good way is a stack of hash tables:

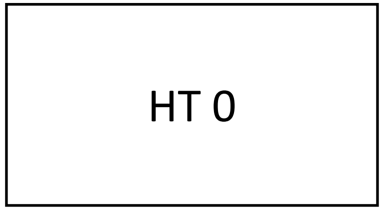


Stack of hash tables

How to implement a symbol table?

- Many ways to implement:
- A good way is a stack of hash tables:

`push_scope()`

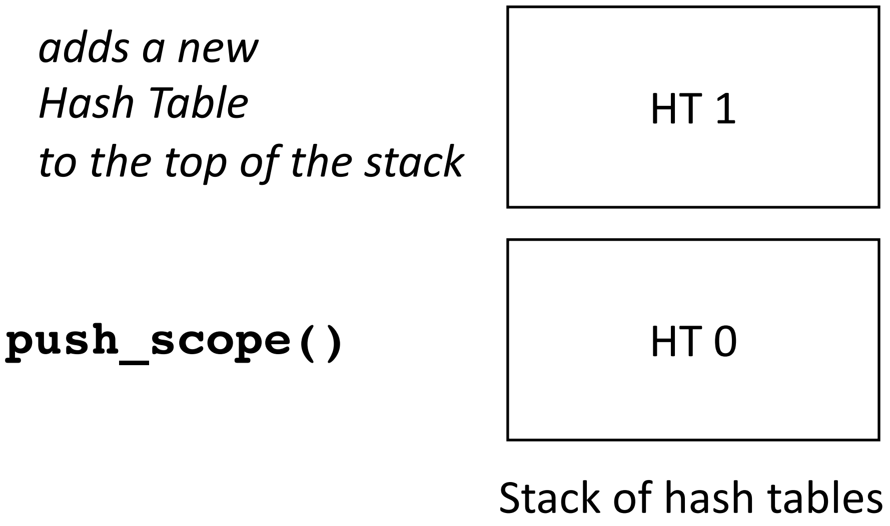


Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:



adds a new
Hash Table
to the top of the stack

`push_scope()`

HT 1

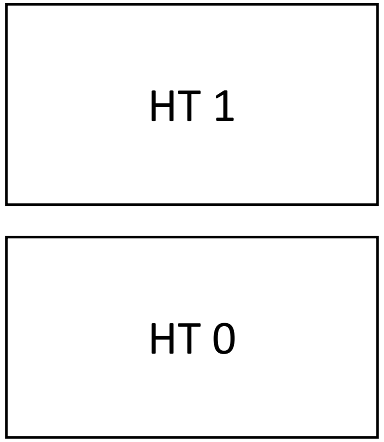
HT 0

Stack of hash tables

How to implement a symbol table?

- Many ways to implement:
- A good way is a stack of hash tables:

`insert(id,data)`



Stack of hash tables

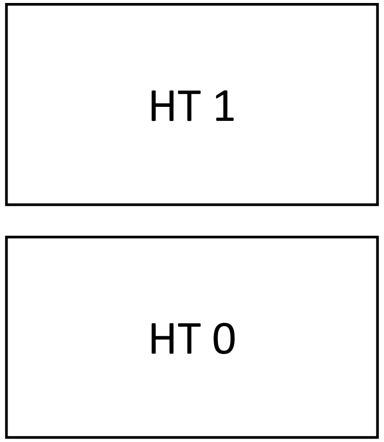
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`insert(id,data)`

insert `(id -> data)` at top hash table

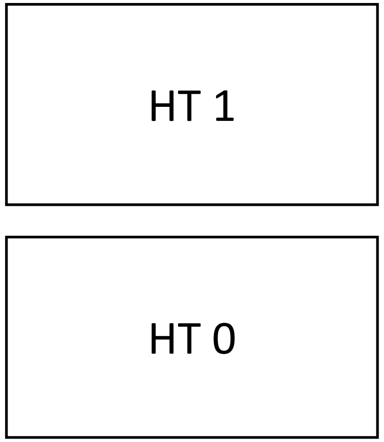


Stack of hash tables

How to implement a symbol table?

- Many ways to implement:
- A good way is a stack of hash tables:

`lookup(id)`



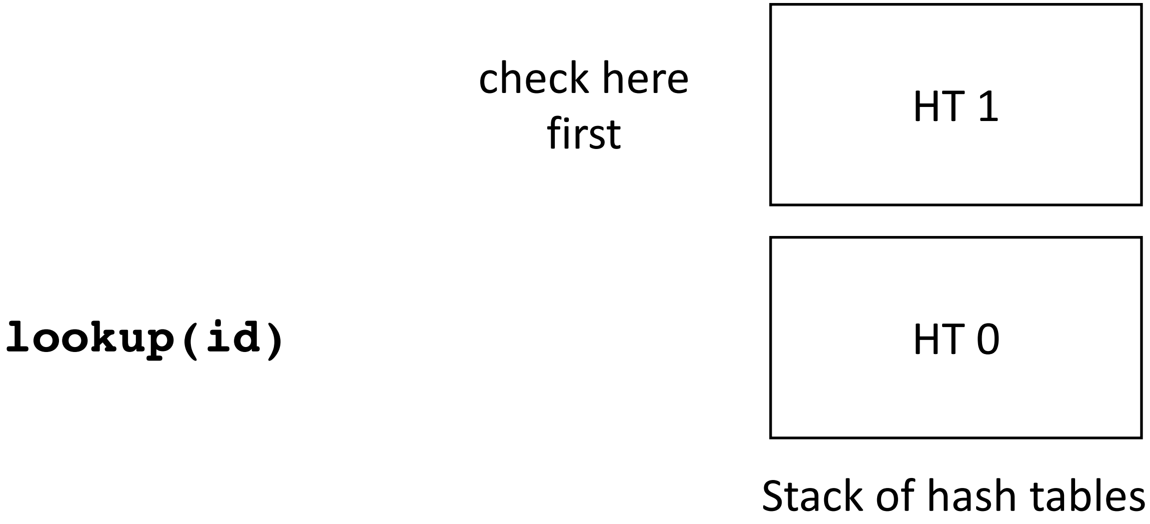
HT 1

HT 0

Stack of hash tables

How to implement a symbol table?

- Many ways to implement:
- A good way is a stack of hash tables:



check here first

HT 1

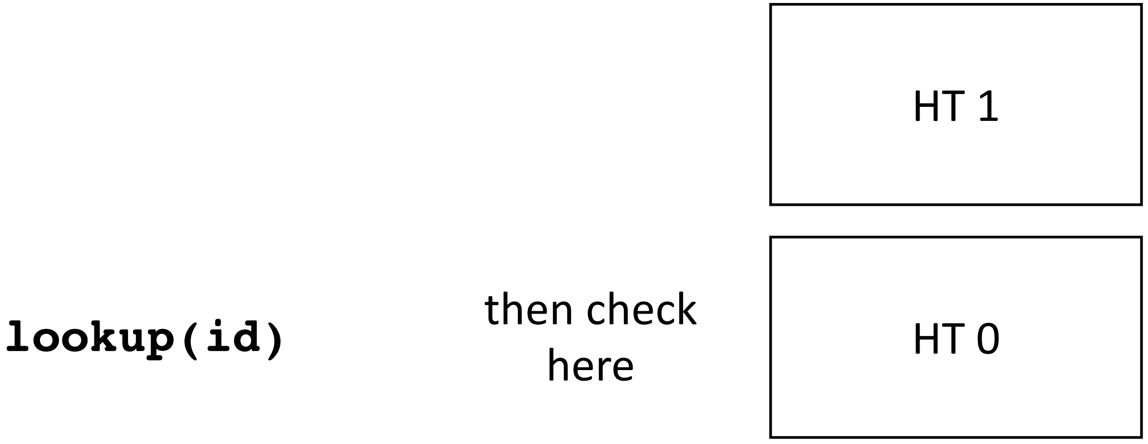
`lookup(id)`

HT 0

Stack of hash tables

How to implement a symbol table?

- Many ways to implement:
- A good way is a stack of hash tables:



`lookup(id)`

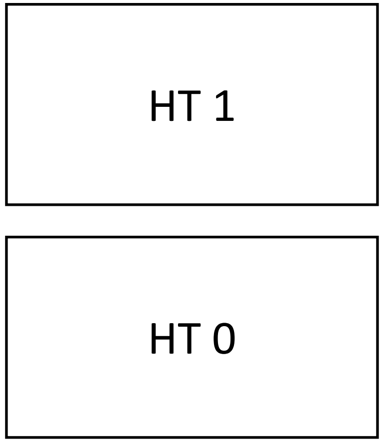
then check here

Stack of hash tables

How to implement a symbol table?

- Many ways to implement:
- A good way is a stack of hash tables:

`pop_scope()`



HT 1

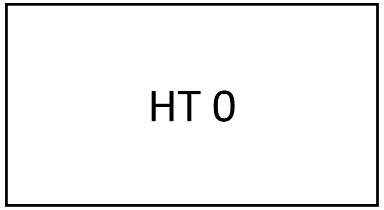
HT 0

Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:



HT 0

Stack of hash tables

How to implement a symbol table?

- Example

```
int x = 0;
int y = 0;
{
  int y = 0;
  x++;
  y++;
}
x++;
y++;
```

HT 0

Stack of hash tables

Moving on

- Parsing with derivatives!

Parsing RE’s with Derivatives

- A simple regular expression matcher implementation
    - Given an RE AST, you can check matches 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

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

Language Derivatives Examples

- $L = \{\text{“}+\text{”}, \text{“}-\text{”}, \text{“}*\text{”}, \text{“}/\text{”}\}$
- $\delta_{+}(L) = ?$
- $\delta_{\wedge}(L) = ?$
- $\delta_{*}(L) = ?$

Language Derivatives Examples

- $L = \{\text{“1”}, \text{“1+1”}, \text{“1+1+1”}, \text{“1+1+1+1”}, \ldots\}$
- $\delta_{+}(L) = \{?\}$
- $\delta_{1}(L) = \{?\}$
- $\delta_{1+}(L) = \{?\}$

Language Derivatives Examples

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

- $\delta_a (L) = \{?\}$

- $\delta_{aa} (L) = \{?\}$

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

- $\delta_{ba} (L) = \{?\}$

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

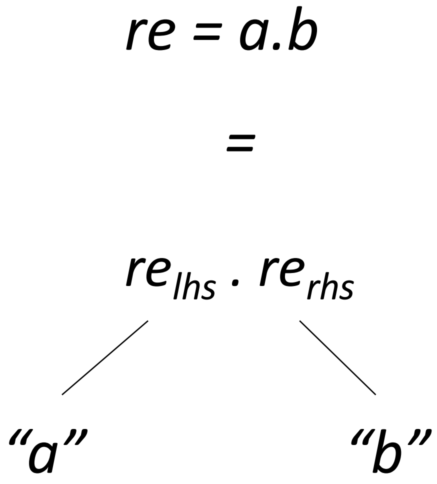
- The concatenation of two REs x and y is given by x.y and matches the strings of RE x **concatenated** with the strings of RE y
- The union of two REs x and y is given by x|y and matches the strings of RE x **or** the strings of RE y
- The Kleene star of an RE x is given by x* and matches the strings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

=

$re_{lhs} . re_{rhs}$

“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 | “”

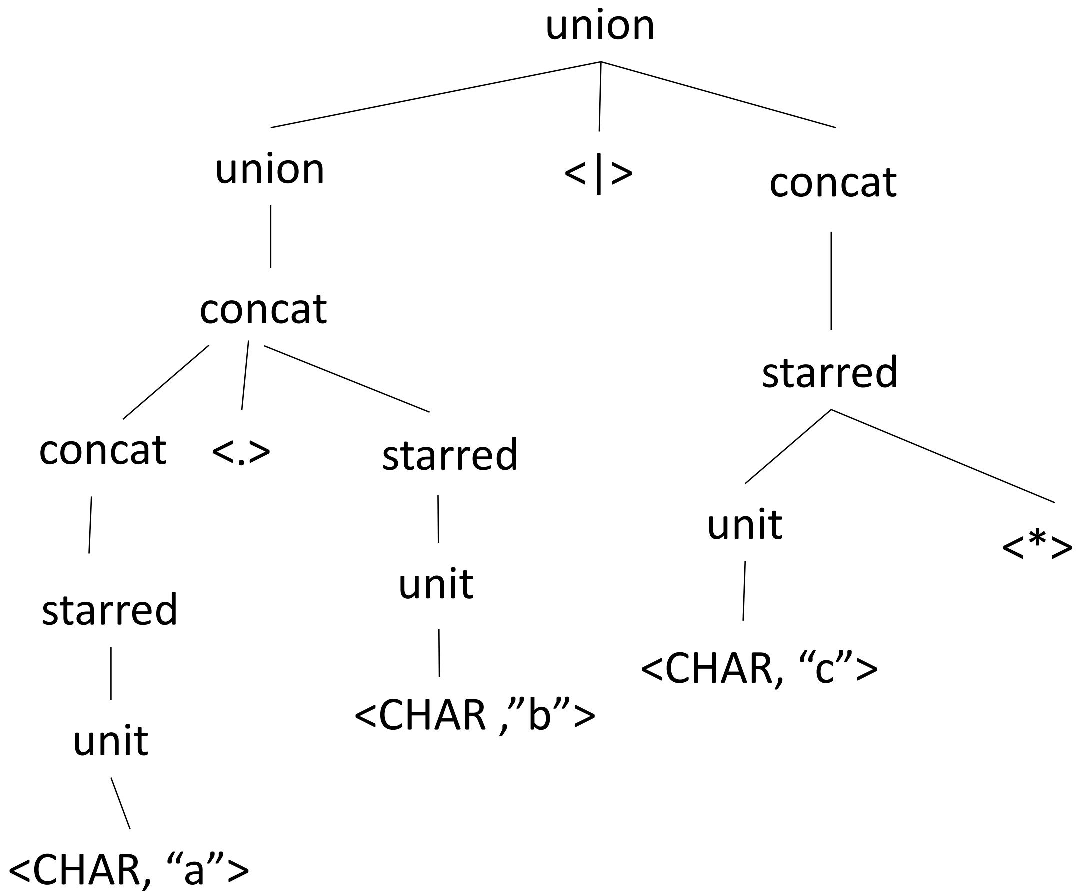
Excluding special cases for {}

parse tree for a regular expression

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

Excluding special cases for {}

`input: a.b |c*`



union

union

<|>

concat

concat

starred

concat

<.>

starred

unit

<*>

starred

unit

<CHAR, “c”>

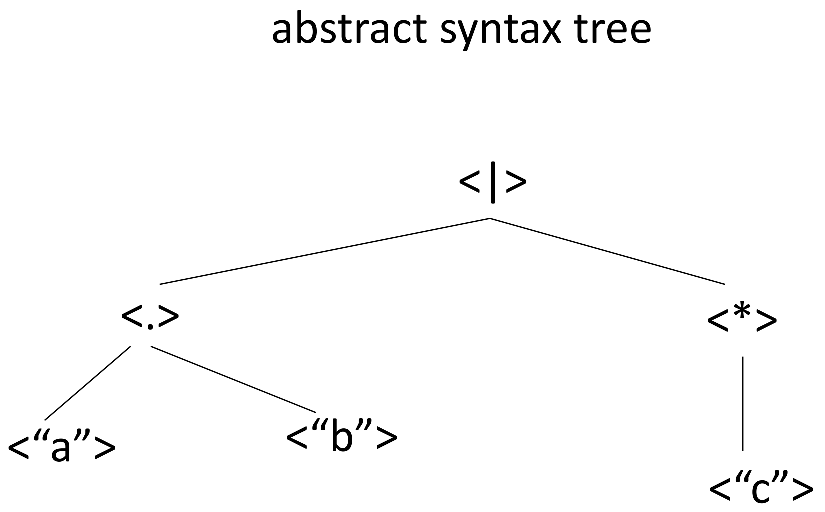
<CHAR ,”b”>

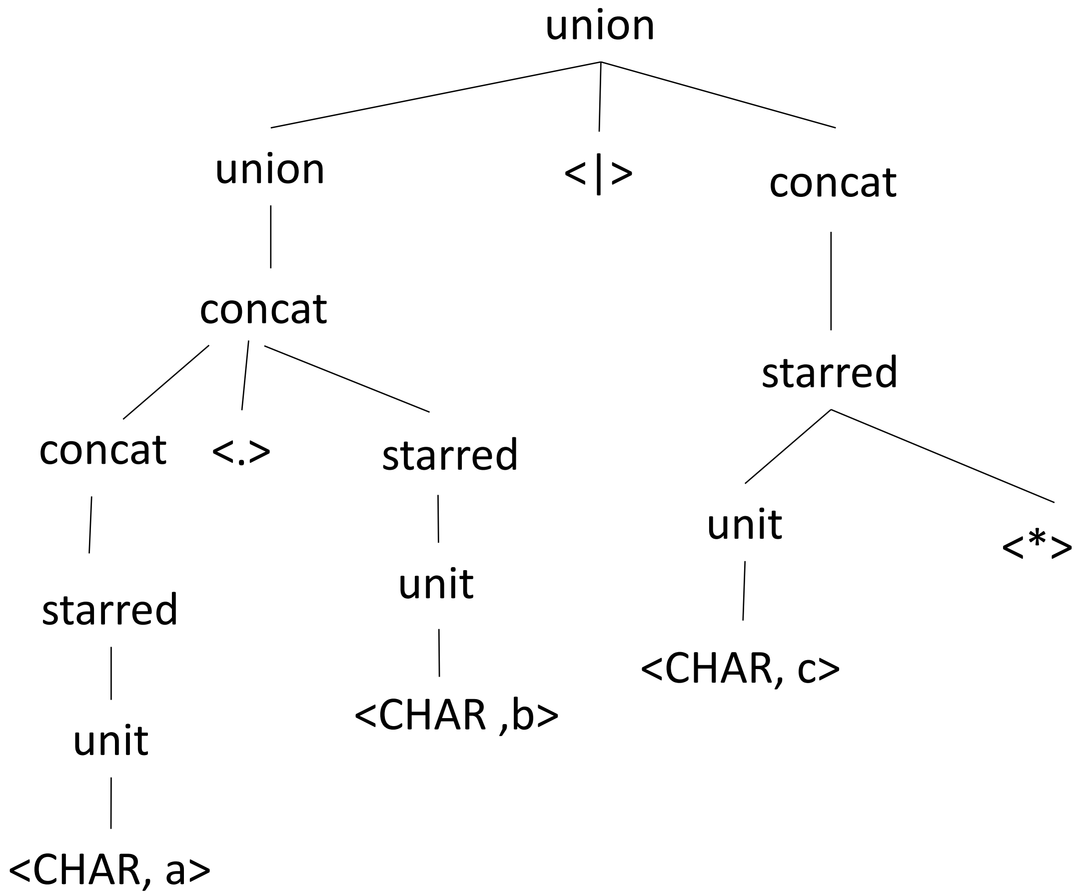
unit

<CHAR, “a”>

parse tree for a regular expression

`input: a.b |c*`

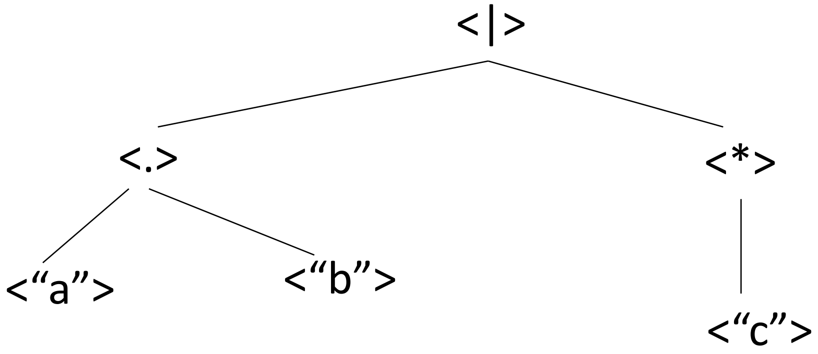




parse tree for a regular expression

`input: a.b |c*`

abstract syntax tree


<|>

<.>

<*>

<“a”>

<“b”>

<“c”>

- re =

  |{}

  | “”

  | a (single character)

  | re_{lhs} | re_{rhs}

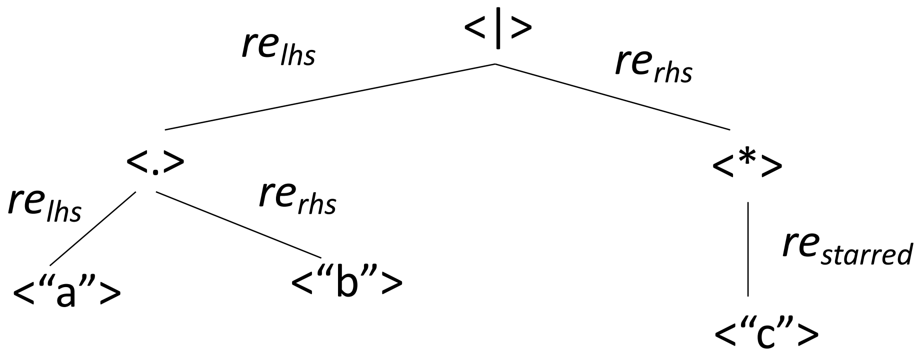
  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

parse tree for a regular expression

`input: a.b |c*`

abstract syntax tree

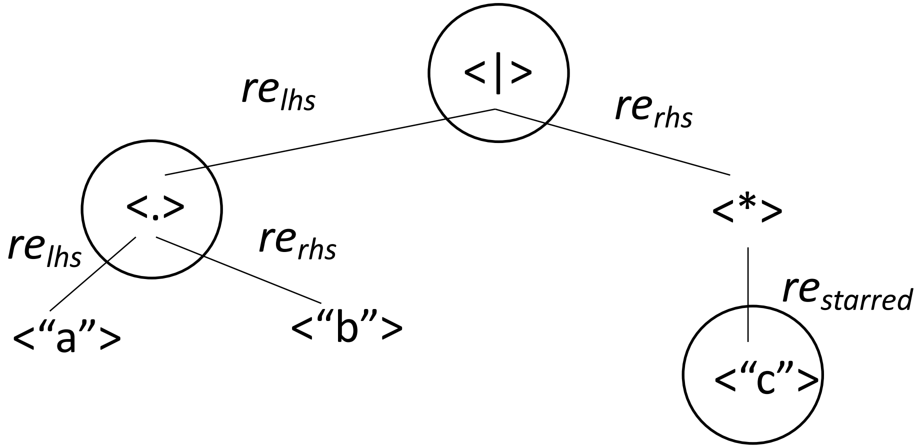


- re =
  |{}
  | “”
  | a (single character)
  | re_{lhs} | re_{rhs}
  | re_{lhs} . re_{rhs}
  | $re_{starred}$ *

parse tree for a regular expression

`input: a.b |c*`

abstract syntax tree



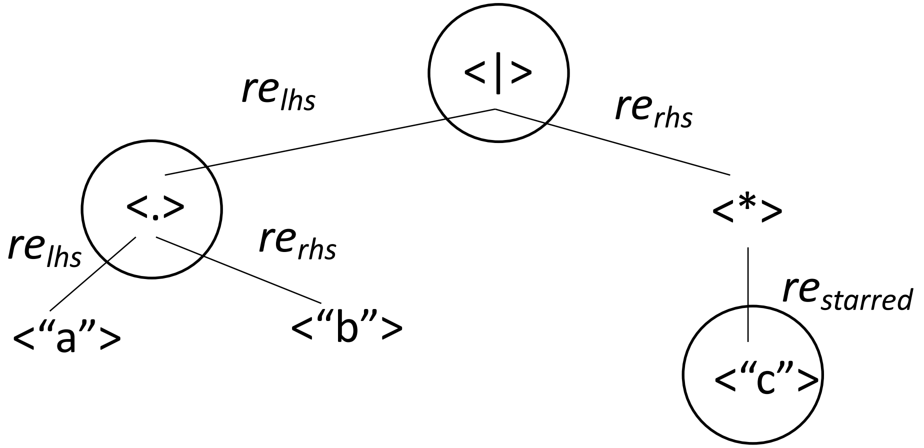
each node is
also a regular expression!

- 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!

- *In your homework you will need to generate an RE AST using production rules*
- *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 language L, any derivative of L is also a regular language.

- *Let’s try some!*

Regular expressions are closed under derivatives

- $re = a$
- L = ?
- $\delta_a(L) = ?$
- $\delta_a(re) = ?$
- $\delta_b(re) = ?$

Regular expressions are closed under derivatives

- $re = a \mid b$
- $L = ?$
- $\delta_a(re) = ?$
- $\delta_b(re) = ?$

Regular expressions are closed under derivatives

- $re = a.a \mid a.b$
- $L = ?$
- $\delta_a(re) = ?$
- $\delta_b(re) = ?$

Regular expressions are closed under derivatives

- $re = (a.b.c)^*$

- $L = ?$

- $\delta_a(re) = ?$

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}
  | $\text{re}_{\text{starred}}$ *

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}

  | $\text{re}_{\text{lhs}} . \text{re}_{\text{rhs}}$

  | $\text{re}_{\text{starred}}$ *

Regular expressions are closed under derivatives

- $re = a.a \mid a.b$
- $L = \{\text{“aa”}, \text{“ab”}\}$
- $\delta_a(re) = \{a, b\} = a \mid b$
- $\delta_b(re) = \{\}$

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)

  | $\text{re}_{\text{lhs}} \mid \text{re}_{\text{rhs}}$

  | $\text{re}_{\text{lhs}} \,.\, \text{re}_{\text{rhs}}$

  | $\text{re}_{\text{starred}}{}^{*}$

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}

  | $re_{starred}$ *

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)*

How do certain regular expressions combine?

- a | “” = ?
- a | {} = ?

- a . “” = ?
- a . {} = ?

- “” * = ?
- {} * = ?

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:

- δ_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)
  - | $re_{lhs} \mid re_{rhs}$
  - | $re_{lhs}\ .\ re_{rhs}$
  - | $re_{starred}{}^{*}$

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

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:

$re = c^{*}.a.b$

$\delta_a(re) = ?$

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:

$re = c^{*}.a.b$

$\delta_a(re) = ?$

Derivative Recursive Cases

Consider the recursive cases:

- δ_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 $\delta_c(re_{lhs}) \,.\, re_{rhs} \mid$

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

- re =

  | {}

  | ε

  | a (single character)

  | $\text{re}_{\text{lhs}} \mid \text{re}_{\text{rhs}}$

  | $\text{re}_{\text{lhs}} \,.\, \text{re}_{\text{rhs}}$

  | $\text{re}_{\text{starred}}{}^*$

Nullable operator

- NULL(re) =

  if “” $\in re$ then: *“”*

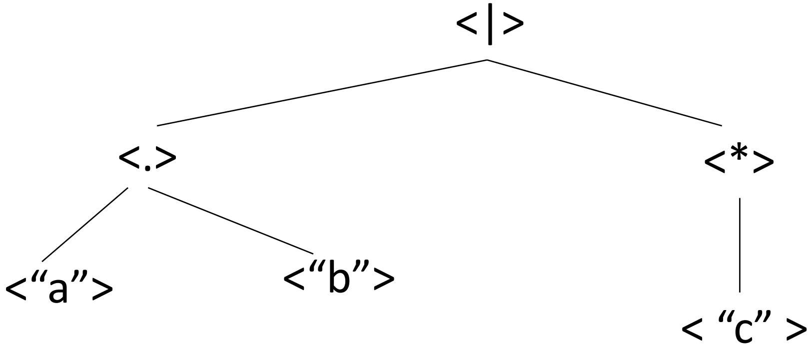
  else: {}

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:
  - $re_{lhs} \mid re_{rhs}$

    return ??
  - $re_{starred}*$

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

    return ??

- re =

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

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:

- $\delta_c\,(re)$ = 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 $\delta_c(re_{lhs}) \,.\, re_{rhs} \mid$

    $\text{if } \varepsilon \text{ in } re_{lhs} \text{ then } \delta_c(re_{rhs}) \text{ else } \{\}$

- re =

  | {}

  | ε

  | a (single character)

  | $re_{lhs} \mid re_{rhs}$

  | $re_{lhs} \,.\, re_{rhs}$

  | $re_{starred}{}^*$

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

    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}

  | $\text{re}_{\text{starred}}$ *

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*?

$\delta_{c1}\ (re)$

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

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

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*?

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

$\delta_{c1}(re)$

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

$\delta_{c2}(\delta_{c1}(\mathrm{re})) = \delta_{c1,c2}(re)$

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

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*?

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

$\delta_{c1} (re)$

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

$\delta_{c2} (\delta_{c1} (\mathrm{re})) = \delta_{c1,c2} (re)$

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

$\delta_s(re)$

$\mathrm{L}(\delta_s(re)) = \{.. \varepsilon ..\}$

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

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

$\delta_{c1} (re)$

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

$\delta_{c2} (\delta_{c1} (\mathrm{re}))\ = \delta_{c1,c2} (re)$

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

$\delta_s(re)$

$L(\delta_s(re)) = \{.. “” ..\}$

$NULL(\delta_s(re)) == “”$

Have a good weekend!

Take a look at part 2 of the homework, you will be implementing a parsing with derivative matcher.

Next week we start module 2!