CSE211: Compiler Design

Oct. 13, 2023

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

- **Questions**:
  - What is “scope”
  - *How do you parse a regular expression?*
  - *How do you parse a context free grammar?*

- $\delta_c\ (re)$, 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!
  - Please partner up if you haven't. If you don't have a partner you can make a private post on Piazza. Please do that in the next few days.
  - Failing to find a partner by Monday or else it will be a 20% deduction AND you will have to do the homework assignment by yourself.
  - Please record your partners in the shared spreadsheet. If you do not record, I will assume that you don't have a partner and you will get the deduction.
    - Please help keep up with organization
  - Use Piazza to find a partner

Announcements

- Homework 1 is out!
  - Where we are at now:
  - The homework has you using PLY to parse 2 languages
    - A calculator language
    - A regular expression language
  - You should be able to parse both languages now
  - By the end of today you should be able to do all of part 1
  - By the end of Monday you should be able to do all of part 2

Review

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?

New material

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 during parsing?

- 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 = “;”

statements are either a declaration or an increment

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

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

adding in scope

How to track scope?

- `SymbolTable ST;`

```
statement : variable_inc
          | declare_variable
          | LB statement_list RB

statement_list : statement statement_list
               | statement
```

How to track scope?

- `SymbolTable ST;`

statement : LB statement_list RB

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 : LB statement_list RB

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`

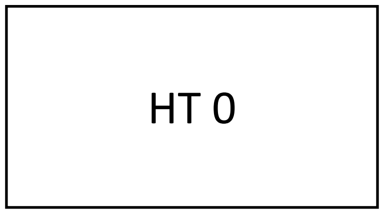
  - **`pop_scope()`** `: pop a scope from the symbol table`

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

base scope



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()`

HT 0

Stack of hash tables

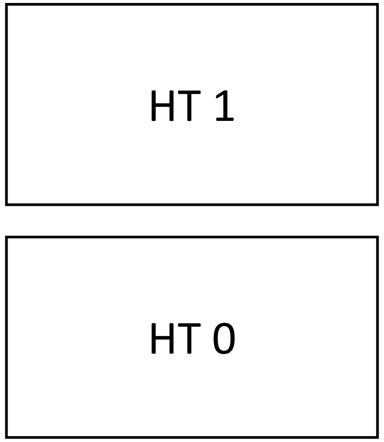
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

adds a new
Hash Table
to the top of the stack

`push_scope()`



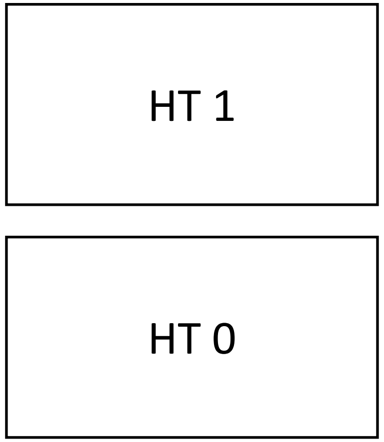
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`insert(id,data)`



Stack of hash tables

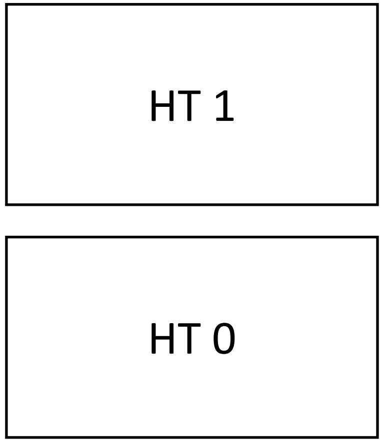
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

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

`insert(id,data)`



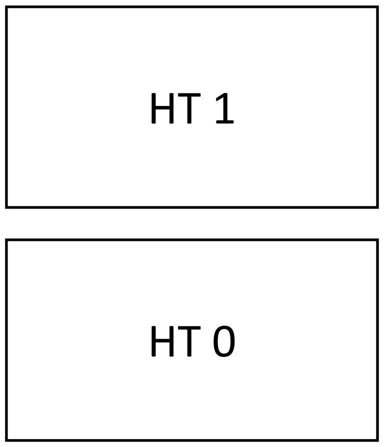
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`lookup(id)`



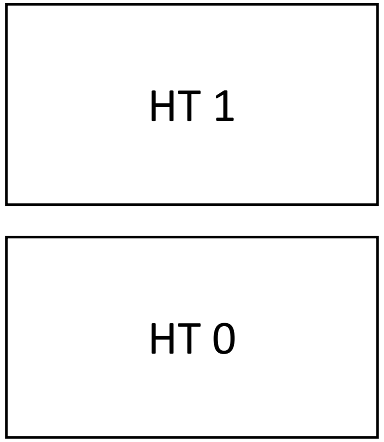
Stack of hash tables

How to implement a symbol table?

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

`lookup(id)`

check here first



Stack of hash tables

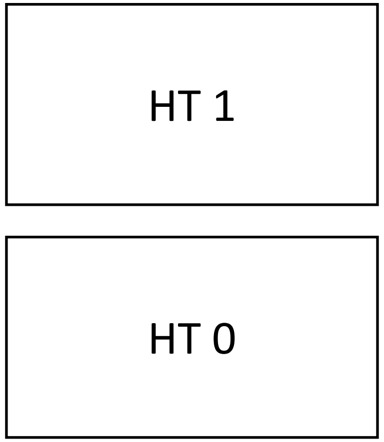
How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`lookup(id)`

then check
here



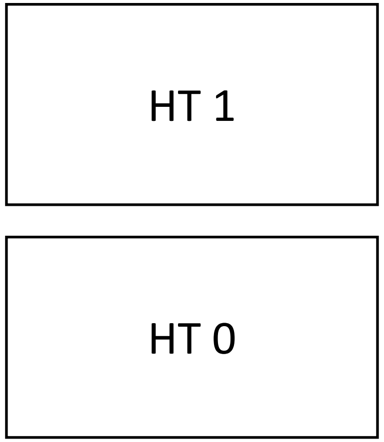
Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

`pop_scope()`

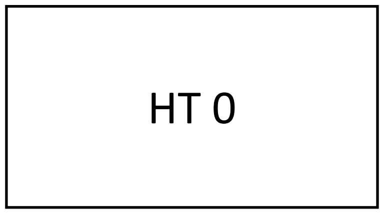


Stack of hash tables

How to implement a symbol table?

- Many ways to implement:

- A good way is a stack of hash tables:

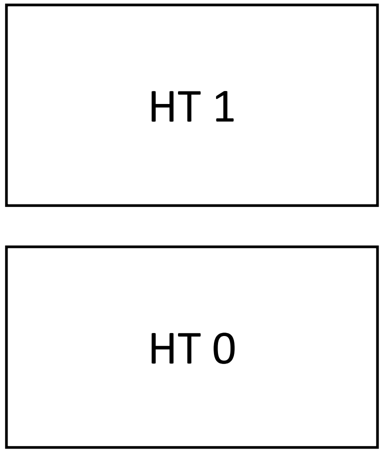


Stack of hash tables

How to implement a symbol table?

- Example

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



Stack of hash tables

Moving on

- Parsing with derivatives!

Matching 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 $\{\varepsilon\}$
  - {}

Subtle distinction between {} and $\{\varepsilon\}$

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

- $L = \{``+", ``-", ``*", ``/"\}$
- $\delta_+ (L) = \{``"\}$
- $\delta_\wedge (L) = \{\}$
- $\delta_* (L) = \{``"\}$

Language Derivatives Examples

- $L = \{\text{"1"}, \text{"1+1"}, \text{"1+1+1"}, \text{"1+1+1+1"}, \ldots\}$

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

- $\delta_{1}(L) = \{\text{""}, \text{"+1"}, \text{"+1+1"} \ldots\}$

- $\delta_{1+}(L) = L$

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 =

$\quad |\ \{\}$

$\quad |$ “”

$\quad |\ c$ (single character)

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

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

$\quad |\ re_{starred}$ *

Regular expressions recursive definition

re =

$\quad$ | {}

$\quad$ | “”

$\quad$ | *c* (single character)

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

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

$\quad$ | $re_{starred}$ *

re = a.b

=

re_{lhs} . re_{rhs}

“a” *“b”*

parse tree for a regular expression

`input: a.b |c*`

<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>

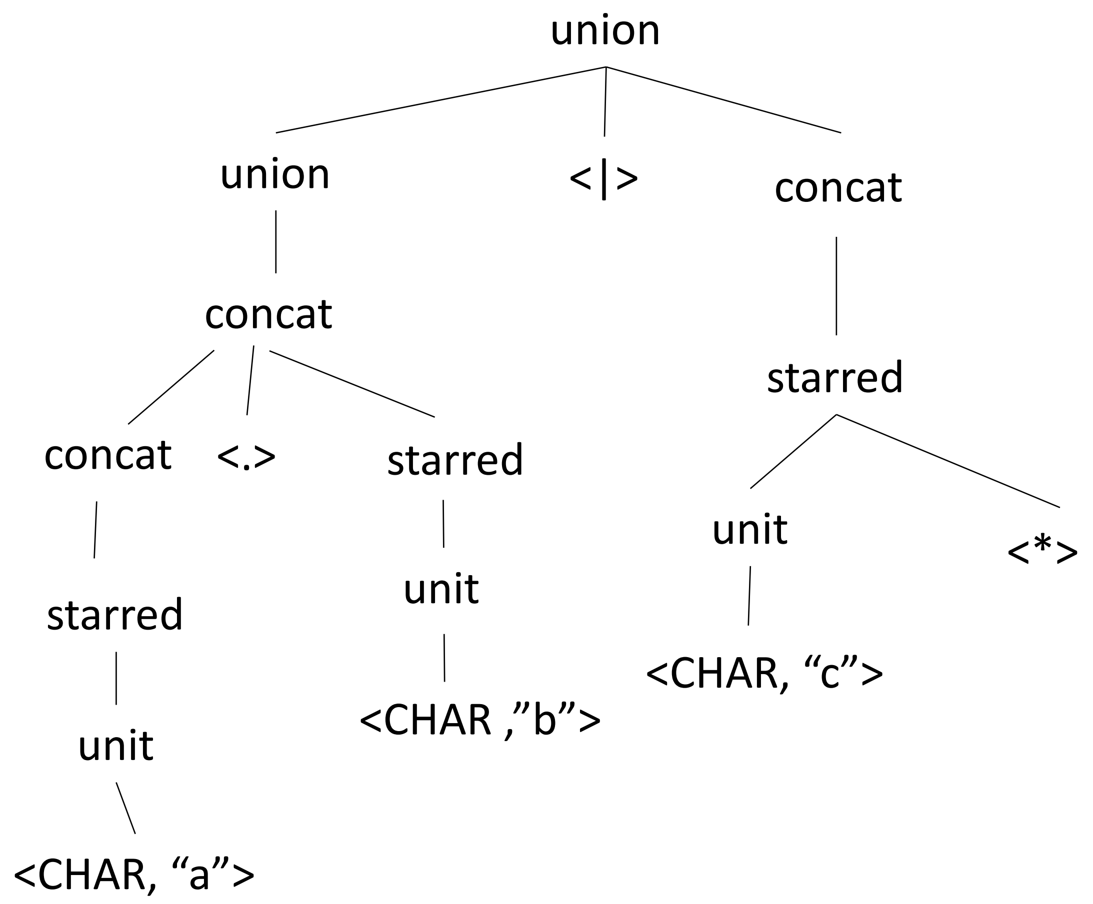
Excluding special cases for {}

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 {}

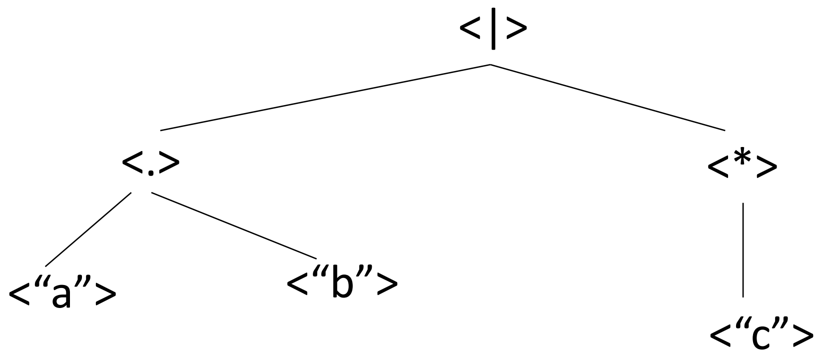
```
input: a.b |c*
```

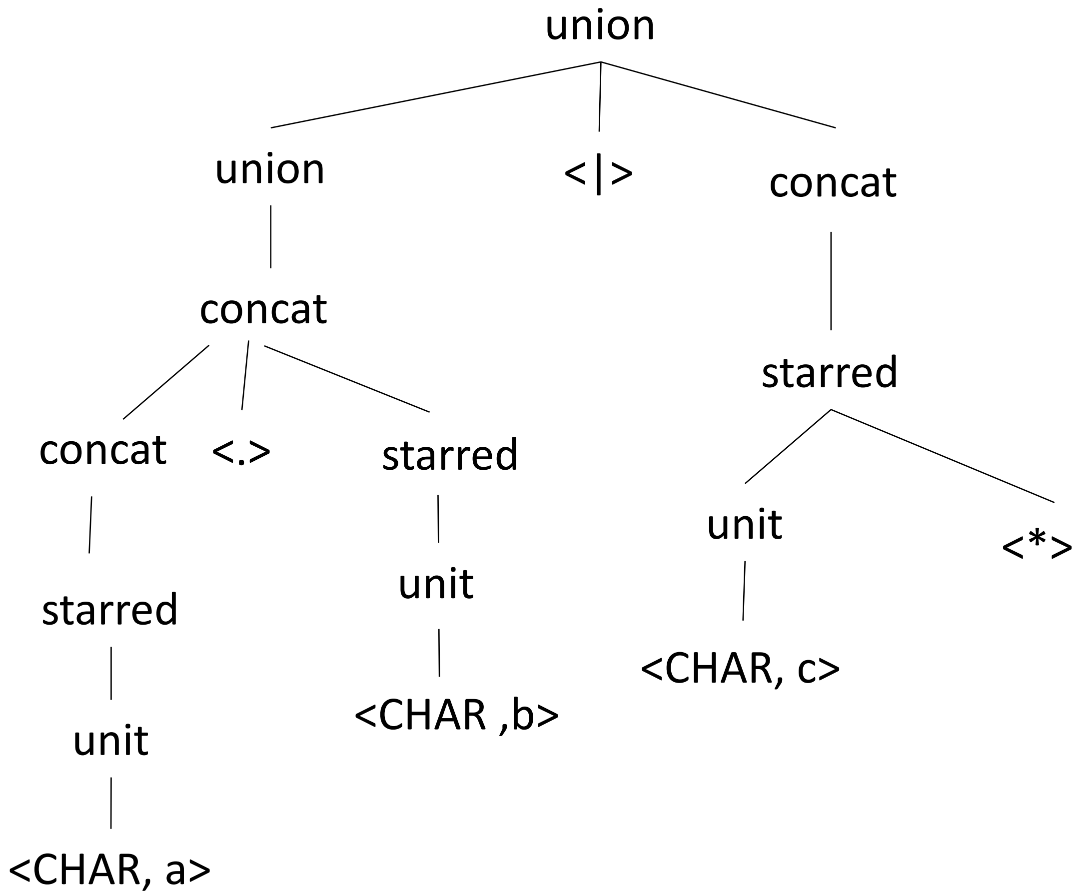


parse tree for a regular expression

`input: a.b |c*`

abstract syntax tree

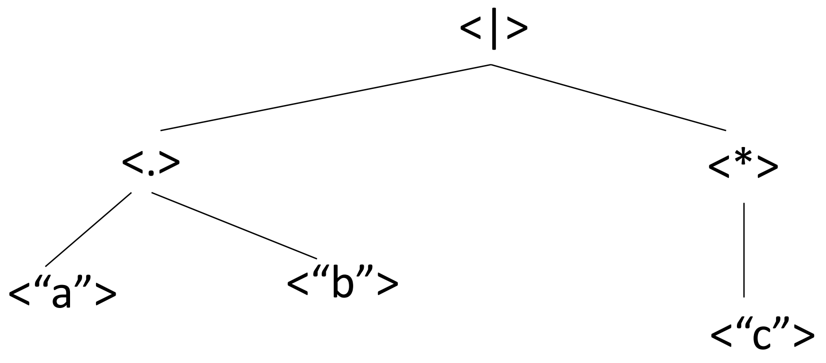




parse tree for a regular expression

```
input: a.b |c*
```

abstract syntax tree

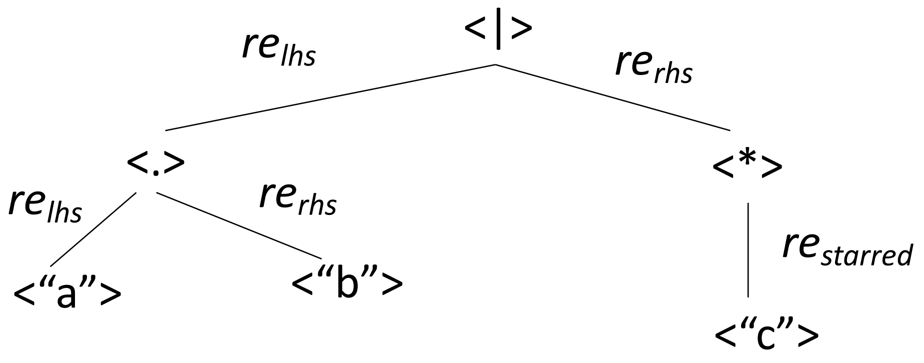


- re =
  - |{}
  - | “”
  - | a (single character)
  - | re_{lhs} | re_{rhs}
  - | re_{lhs} . re_{rhs}
  - | $re_{starred}$ *

parse tree for a regular expression

`input: a.b |c*`

abstract syntax tree

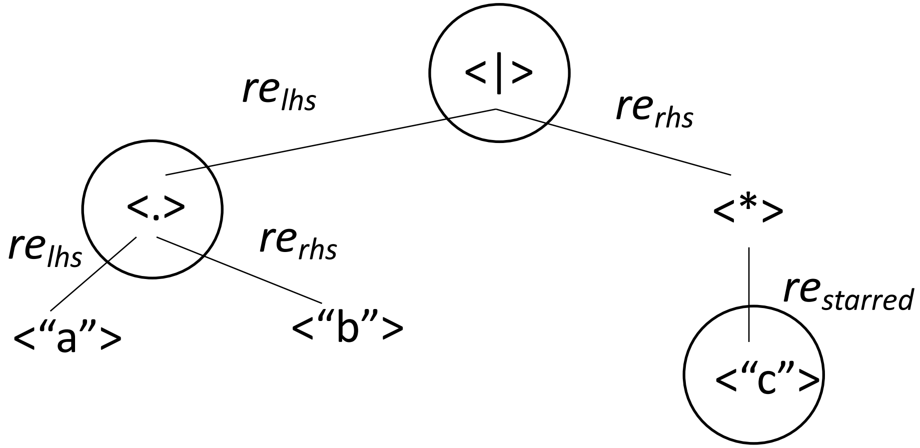


- re =
  - |{}
  - | “”
  - | a (single character)
  - | re_{lhs} | re_{rhs}
  - | re_{lhs} . re_{rhs}
  - | $re_{starred}$ *

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}

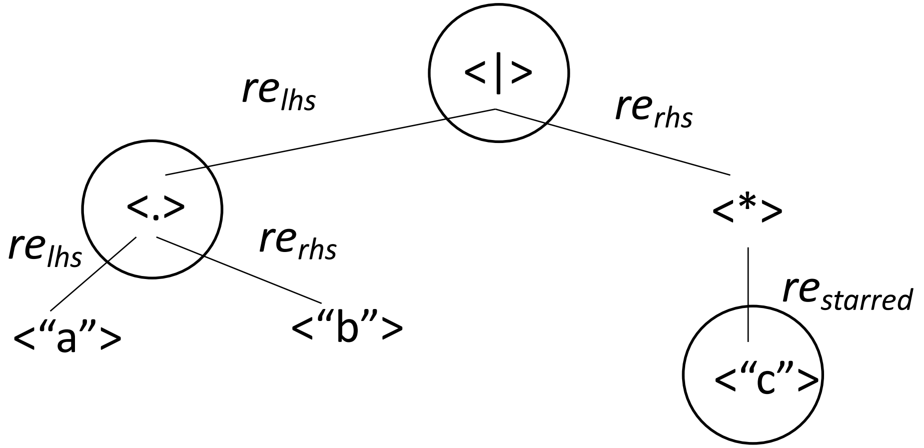
  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

parse tree for a regular expression

`input: a.b |c*`

abstract syntax tree



each node is
also a regular expression!

- *In your homework you will need to generate an RE AST using production rules*
- *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 = {“a”}
- $\delta_a(L) = \{“”\}$
- $\delta_a(re) = “”$
- $\delta_b(re) = \{\}$

Regular expressions are closed under derivatives

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

Regular expressions are closed under derivatives

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

Regular expressions are closed under derivatives

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

- *L = {"", "abc", "abcabc", ...}*
- $\delta_a (L) = \{$"bc", "bcabc", "bcabc", ...$\}$
- $\delta_a(re)$ = *b.c.(a.b.c)**

What is a method for computing the derivative?

Consider the base cases

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

    return ? ?

  - $re_{starred}{}^*$

    return ? ?

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

    return ? ?

- re =

  |{}

  | ε

  | a (single character)

  | $\mathrm{re}_{\mathrm{lhs}}$ | $\mathrm{re}_{\mathrm{rhs}}$

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

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

Regular expressions are closed under derivatives

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

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

    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} | 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”, ...}*

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

    return ? ?

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

    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)

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

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

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

Some properties/optimizations

How do certain regular expressions combine?

- a | “” = a
- a | {} = a

- a . “” = a
- a . {} = {}

- “” * = “”
- {} * = {}

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*) = ?

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}

Example:

re = a.b

$\delta_a(re) = b$

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) = ?*

Derivative Recursive Cases

Let's look at concatenation:

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

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

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)

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

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

  | $re_{starred}$ *

Nullable operator

- NULL(re) =

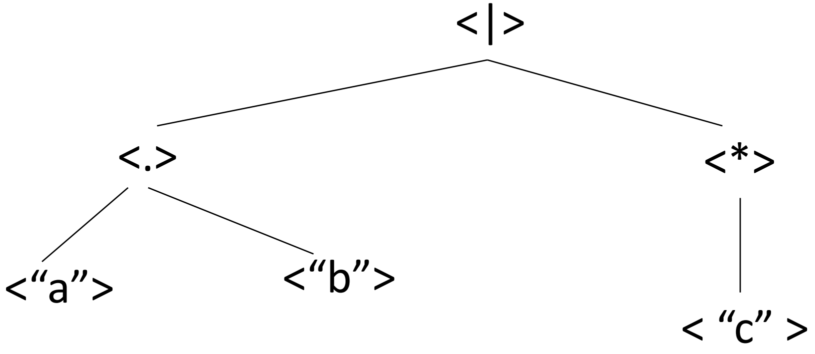
  if "" $\in re$ then: ""
  else: {}

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

    return ??

  - $re_{starred}{}^*$

    return ??

  - $re_{lhs} \cdot re_{rhs}$

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

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

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

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

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

    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*?

$\delta_{c1}\ (re)$

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

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

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*?

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

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*?

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

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

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