CSE211: Compiler Design

Oct. 13, 2020

- **Topic**: PLY and parsing with derivatives
- **Questions**:

  How would you implement what we've talked about so far?

  Have you ever implemented a parser?

```
import ply.lex as lex
import ply.yacc as yacc

# All token names must be declared in this global variab
tokens = ['NUM', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'EX

t_NUM = r'[0-9]+'
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_EXP = r'\^'
t_LPAR = r'\('
t_RPAR = r'\)'

# Ignore spaces and tabs
t_ignore = ' '

# Required in case of a lexing error
def t_error(t):
    print("lexing error!")
    exit(0)

# Build the lexer
lexer = lex.lex()

def p_expression(p):
    'expr : term'
    p[0] = p[1]
```

Announcements

- End of module 1!
  - Module 2 starts next week
- Homework is released: Due in 2 weeks
  - Questions?
- References for today is an academic paper and code docs

CSE211: Compiler Design

Oct. 13, 2020

- **Topic**: PLY and parsing with derivatives
- **Questions**:

  How would you implement what we've talked about so far?

  Have you ever implemented a parser?

```
import ply.lex as lex
import ply.yacc as yacc

# All token names must be declared in this global variab
tokens = ['NUM', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'EX

t_NUM = r'[0-9]+'
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_EXP = r'\^'
t_LPAR = r'\('
t_RPAR = r'\)'

# Ignore spaces and tabs
t_ignore = ' '

# Required in case of a lexing error
def t_error(t):
    print("lexing error!")
    exit(0)

# Build the lexer
lexer = lex.lex()

def p_expression(p):
    'expr : term'
    p[0] = p[1]
```

Today

- Building a simple interpreter
- Parsing regular expressions with derivatives
- Homework overview

Reminder

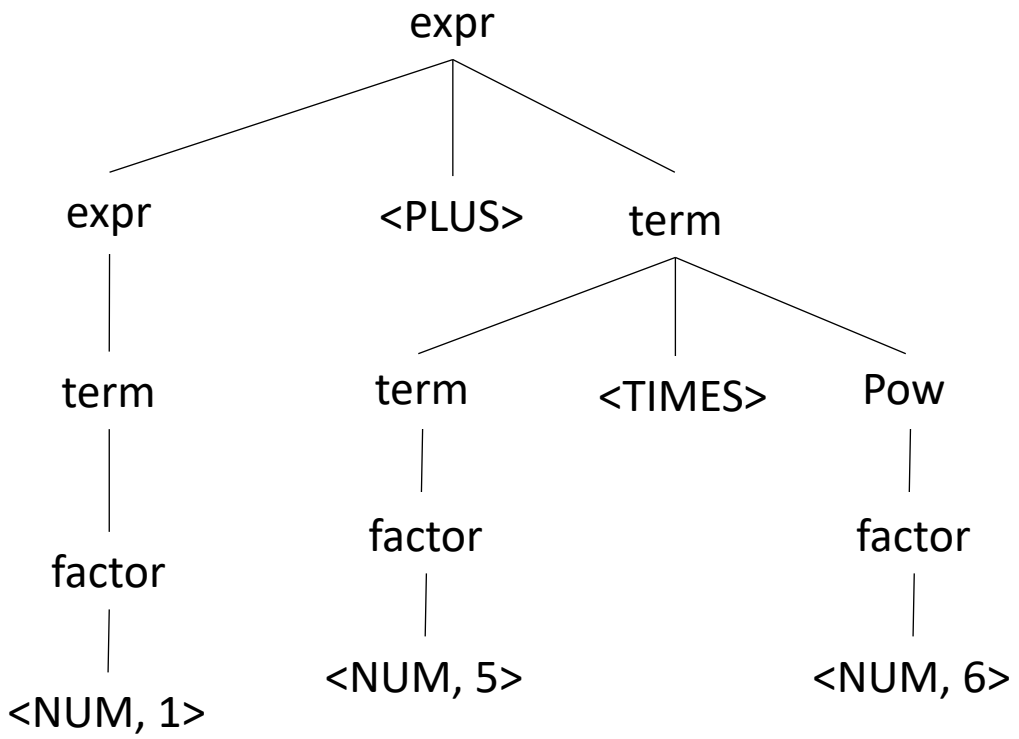
What are the levels for?

Operator	Name	Productions
+,-	Expr	: Expr + Term | Expr - Term | Term
*,/	Term	: Term * Pow : Term / Pow | Pow
^	Pow	: Factor ^ Pow | Factor
()	Factor	: (Expr) | NUM

Reminder

Operator	Name	Productions
+,-	Expr	: Expr + Term | Expr - Term | Term
*,/	Term	: Term * Pow : Term / Pow | Pow
^	Pow	: Factor ^ Pow | Factor
()	Factor	: (Expr) | NUM

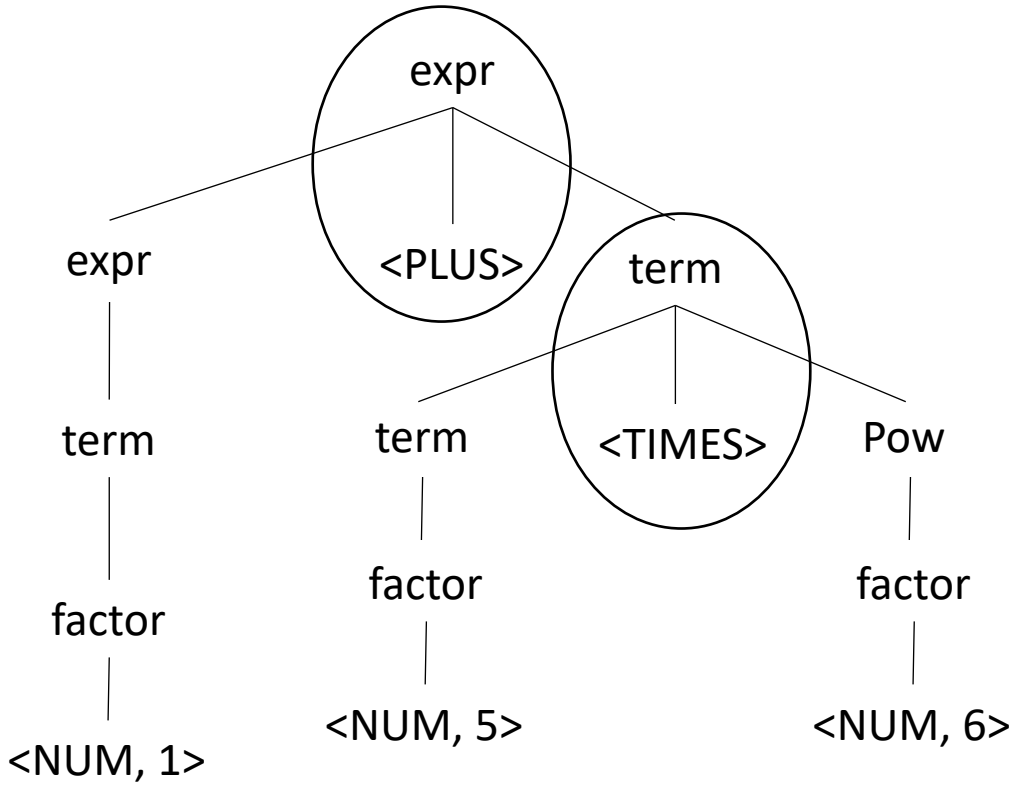
`input: 1+5*6`



Reminder

Operator	Name	Productions
+,-	Expr	: Expr + Term | Expr - Term | Term
*,/	Term	: Term * Pow : Term / Pow | Pow
^	Pow	: Factor ^ Pow | Factor
()	Factor	: (Expr) | NUM

`input: 1+5*6`



Higher precedence is lower level

Reminder

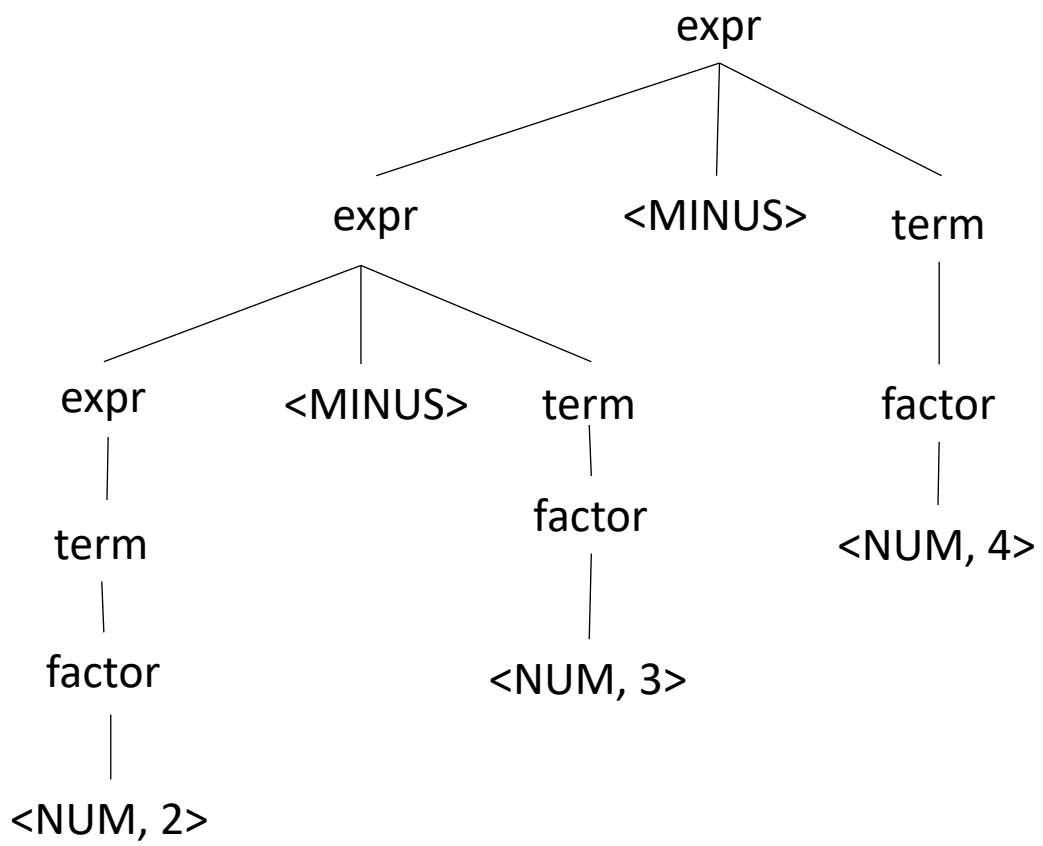
What are the recursion choices?

Operator	Name	Productions
+,-	Expr	: Expr + Term | Expr - Term | Term
*,/	Term	: Term * Pow : Term / Pow | Pow
^	Pow	: Factor ^ Pow | Factor
()	Factor	: (Expr) | NUM

Reminder

Operator	Name	Productions
+,-	Expr	: Expr + Term | Expr - Term | Term
*,/	Term	: Term * Pow : Term / Pow | Pow
^	Pow	: Factor ^ Pow | Factor
()	Factor	: (Expr) | NUM

`input: 2-3-4`



Production actions

Operator	Name	Productions	Action
+,-	Expr	: Expr + Term | Expr - Term | Term	
*,/	Term	: Term * Pow : Term / Pow | Pow	
^	Pow	: Pow ^ Factor | Factor	
()	Factor	: (Expr) | NUM	

One action per row.

a struct p with values for each production element, with p[0] as the return value

For example:

”(5)” will eventually execute the production rule:

Factor: (Expr)

at that point:
p = {‘(’, 5, ‘)’}

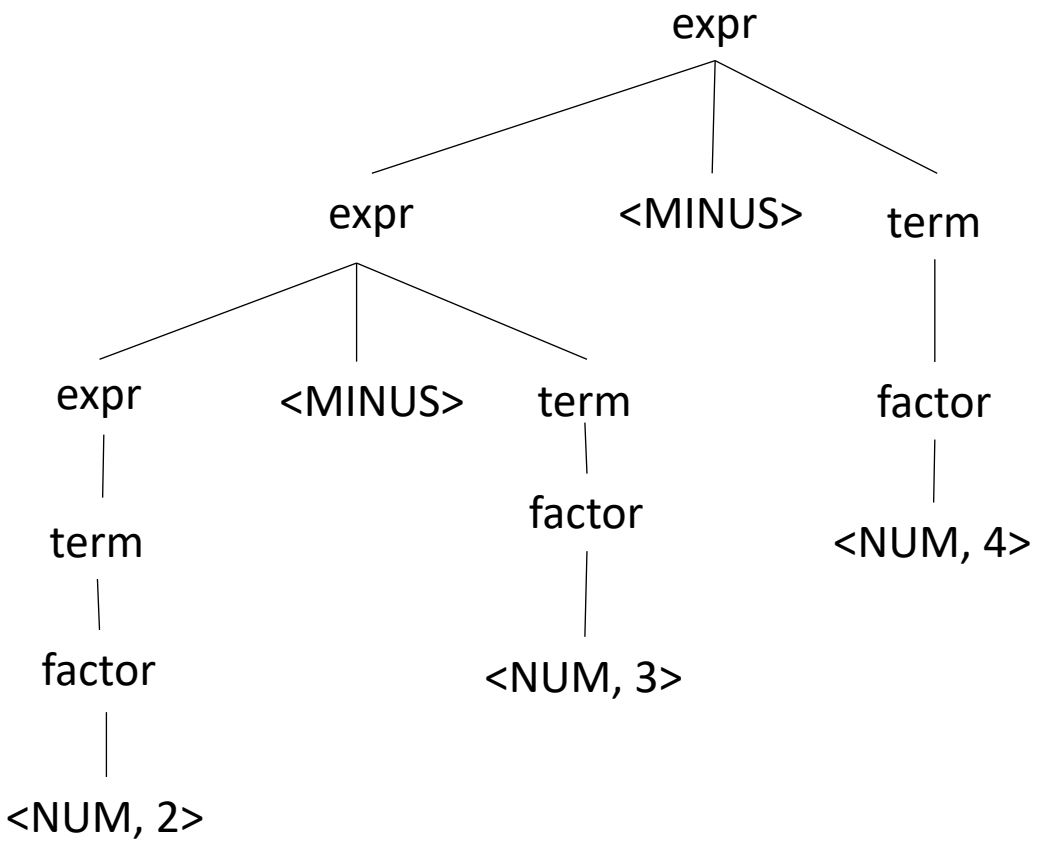
Production actions

Operator	Name	Productions	Action

+,-	Expr	: Expr + Term
		| Expr - Term
		| Term
			`{p[0] = p[1] + p[3]}`
			`{p[0] = p[1] – p[3]}`
			`{p[0] = p[1]}`

*,/	Term	: Term * Pow
		: Term / Pow
		| Pow
			`{p[0] = p[1] * p[3]}`
			`{p[0] = p[1] / p[3]}`
			`{p[0] = p[1]}`

^	Pow	: Pow ^ Factor | Factor	`{p[0] = p[1] ^ p[3]}` `{p[0] = p[1]}`
()	Factor	: (Expr) | NUM	`{p[0] = p[2]}` `{p[0] = int(p[1])}`



Let’s implement it!

- Related to the PLY calc example:
  - https://github.com/dabeaz/ply/blob/master/example/calc/calc.py
- You can also reference the PLY documentation:
  - https://www.dabeaz.com/ply/ply.html
- I will provide the code examples on Canvas

Lecture Part 2: Parsing RE's with Derivatives

- A simple, functional style regular expression parser implementation
- Uses an regular expression AST

Language Derivatives

- A language is a (potentially infinite) set of strings *{s_1, s_2, s_3, s_4, …}*
- 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 δ_c(*L*)

- Examples:

Language Derivatives Examples

- $L = \{\text{“}a\text{”}\}$

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

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

Language Derivatives Examples

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

Language Derivatives Examples

- $L = \{\text{“aaa”}, \text{“ab”}, \text{“ba”}, \text{“bba”}\}$
- $\delta_a (L) = \{\text{“aa”}, \text{“b”}\}$
- $\delta_{aa} (L) = \{\text{“a”}\}$
- $\delta_b (L) = \{\text{“a”}, \text{“ba”}\}$
- $\delta_{ba} (L) = \{\varepsilon\}$

Language Derivatives Examples

- *L = {“1”, “1+1”, “1+1+1”, “1+1+1+1”, …}*
- $\delta_+ (L) = \{\}$
- $\delta_1 (L) = \{\varepsilon$ *,”+1”, “+1+1”, “+1+1+1”, …}*
- $\delta_{1+} (L) =$ *{“1”, “1+1”, “1+1”, ..} = L*

Regular expressions are closed under derivatives

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

- *regular expression =*
  - *|{}*
  - *| {ε}*
  - *| ”a” (single character)*
  - *| re_{lhs} \| re_{rhs}*
  - *| re_{lhs} . re_{rhs}*
  - *| $re_{starred}$ **

Derivative Base Cases

- δ_c (*re)*, where *re* is:
  - {}

    {}
  - {ε}

    {}
  - *“a”* (single character)

Derivative Base Cases

- δ_c (*re)*, where *re* is:
  - {}

    return {}
  - $\{\varepsilon\}$

    return {}
  - *“a”* (single character)

    return if “a” == c then {ε}, else {}

Derivative Recursive Cases

- δ_c (*re)*, where *re* is:
  - 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 {}*

Nullable operator

- NULL(re) =

  if $\epsilon \in re$ then: $\{\varepsilon\}$
  else: {}

Nullable operator

- NULL(re) =

  if $\epsilon \in re$ then: $\{\varepsilon\}$
  else: {}

- 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 {}*

Nullable operator

- NULL(re) =

  if $\epsilon \in re$ then: $\{\varepsilon\}$
  else: {}

- δ_c (*re)*, where *re* is:

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

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

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

Nullable Base Case

- *NULL(re)*, where *re* is:
  - {}

    return {}
  - $\{\varepsilon\}$

    return $\{\varepsilon\}$
  - “a” (single character)

    return {}

Nullable Recursive Cases

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

    $NULL(re_{lhs}) \mid NULL(re_{rhs})$

  - $re_{starred}$ *

    $\{\varepsilon\}$

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

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

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*?

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

δ_{c1} *(re)*
L(δ_{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*?

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

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

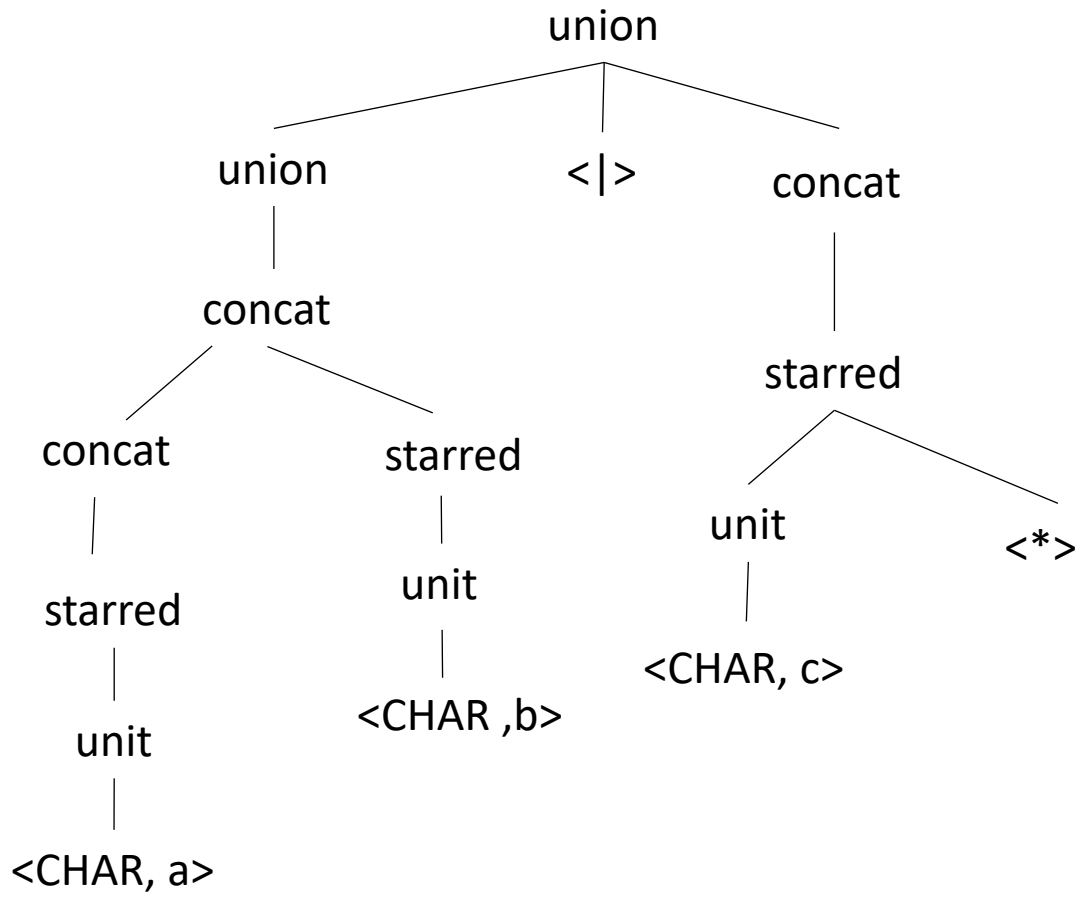
	δ_{c1} *(re)*	δ_{c2} (δ_{c1} (re)) = $\delta_{c1,c2}$ *(re)*	δ_s*(re)*
L(re) = {.. s ..}			*NULL*(δ_s*(re))* == {ε}
	L(δ_{c1} (re)*) = {.. s[1:] ..}*	*L*($\delta_{c1,c2}$ (re)*) = {.. s[2:] ..}*	L(δ_s*(re)*) = {.. ε ..}

Implementations

Create a parse tree for regular expressions

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

Operator	Name	Productions
|	union	: union \\\| concat | concat
.	concat	: concat . starred | starred
*	starred	: starred * | unit
()	unit	: (union) | CHAR

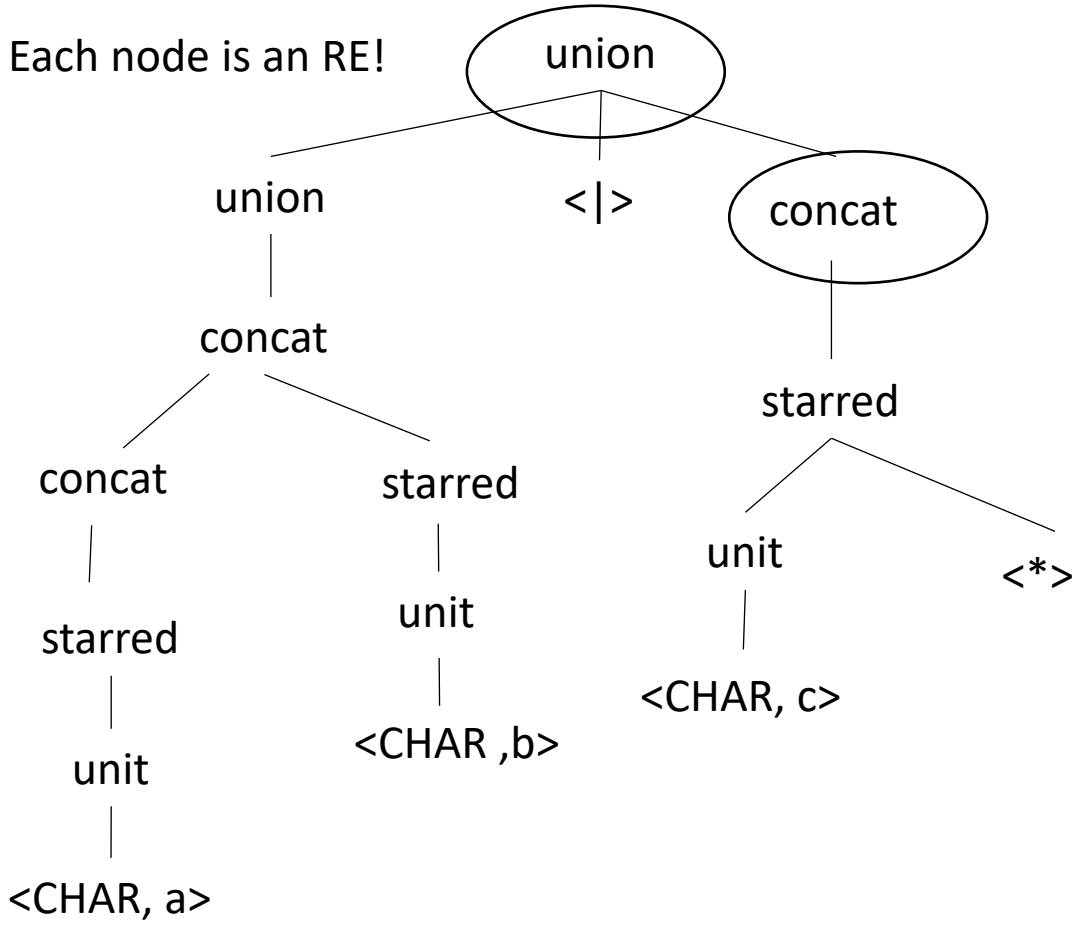


Implementations

Create a parse tree for regular expressions

`input: a.b | c*`

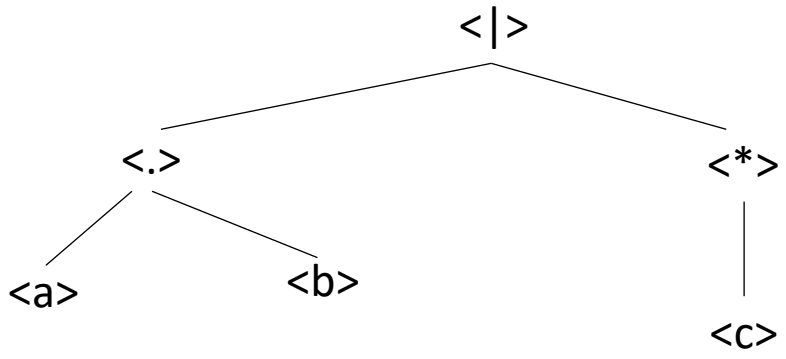
Operator	Name	Productions
|	union	: union \\\| concat | concat
.	concat	: concat . starred | starred
*	starred	: starred * | unit
()	unit	: (union) | CHAR

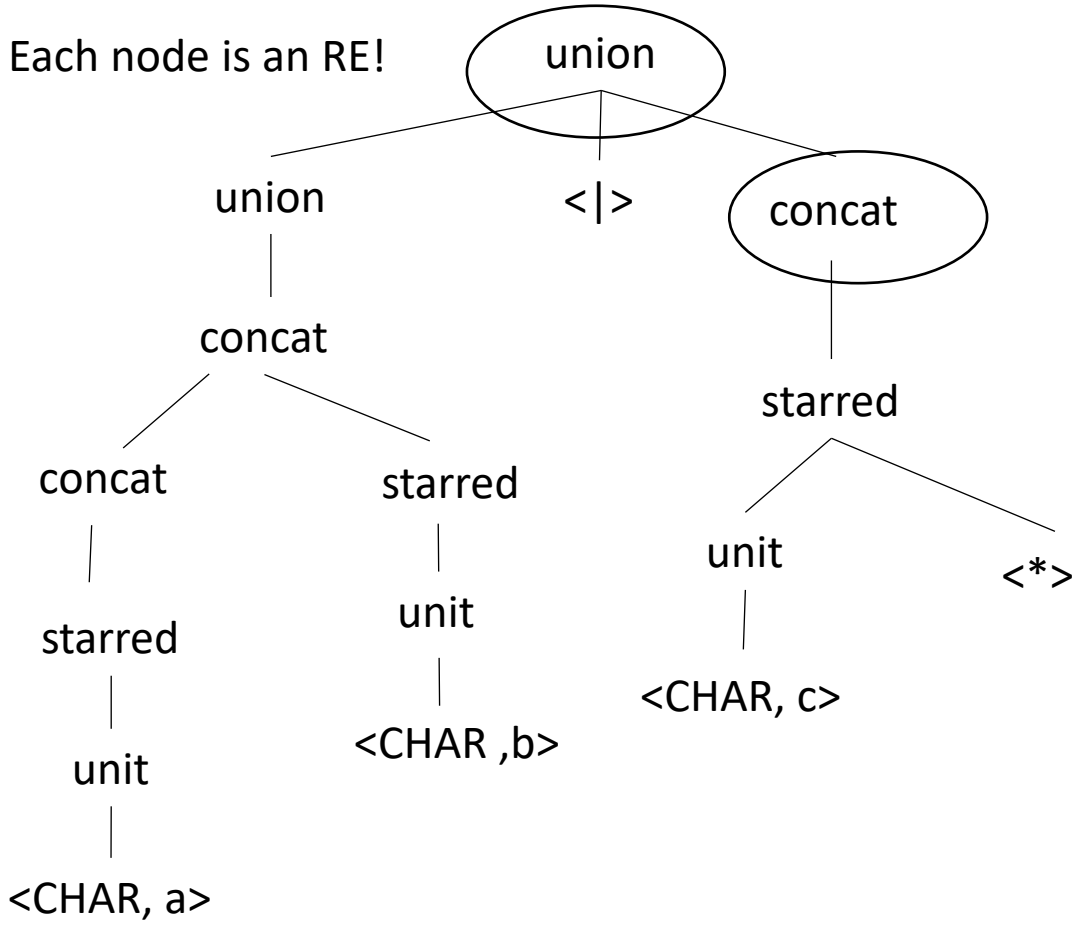


Implementations

Create a parse tree for regular expressions

`input: a.b | c*`

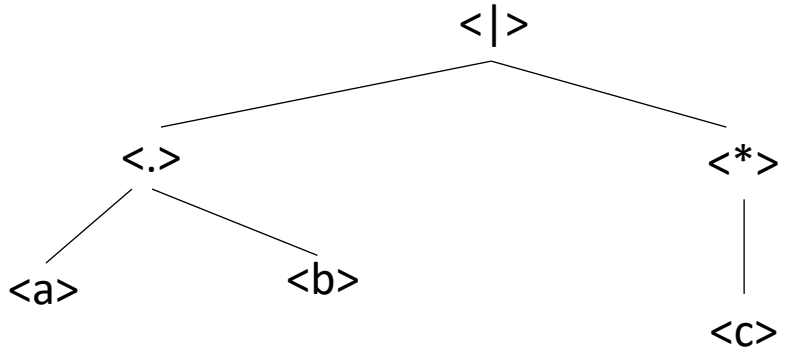




Implementations

Create a parse tree for regular expressions

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

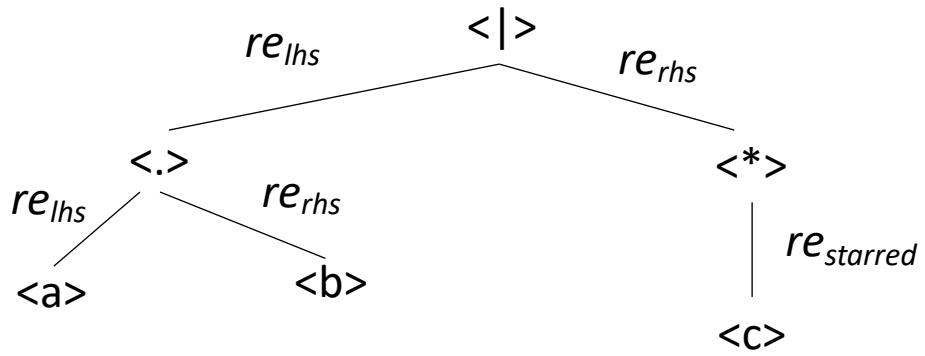


- *regular expression =*
  - *| {}*
  - *| ε*
  - *| a (single character)*
  - *| re_{lhs} | re_{rhs}*
  - *| re_{lhs} . re_{rhs}*
  - *| $re_{starred}$ **

Implementations

Create a parse tree for regular expressions

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



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

Implementations

Homework overview

Next Lecture

- Starting on module 2: Optimizations and Flow analysis

- Start on homework!
  - Visit office hours tomorrow!
  - Post questions/comments on Canvas!