CSE211: Compiler Design

Oct. 16, 2023

- **Topic**: Parsing with derivatives

- **Questions**:
  - How is Homework 1 going?

- 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!
  - If you don’t have a partner by today it is 20% off and you have to do it by yourself. Please update the google sheet.
  - Use Piazza to ask about any language clarification questions
  - By the end of today you should be able to do the whole homework
  - Office hours on Thursday if you need help (only office hour before homework is due!)

- **Paper review**: paper needs to be approved by me by 1 week (preferably earlier!)

Announcements

- End of Module 1 today, next time starting module 2: analysis and optimization

- I will be gone Monday and Wednesday next week to attend a khronos group meeting.
  - The schedule is still in flux:
    - either I will hold class synchronously on Zoom
    - Or provide asynchronous lectures
    - Maybe a combination, stay tuned

Review

Review

- Scope

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?

- 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

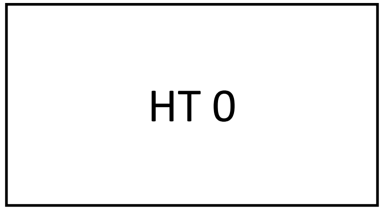
remove the scope S

Think about how to solve with production rules

How to implement a symbol table?

- Example

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



Stack of hash tables

Next

- Parsing with derivatives!

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

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

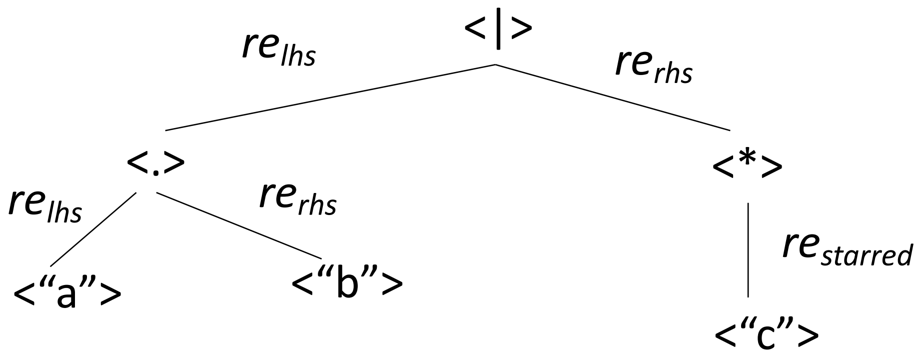
Language Derivatives Examples

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

AST for a regular expression

`input: a.b |c*`

abstract syntax tree

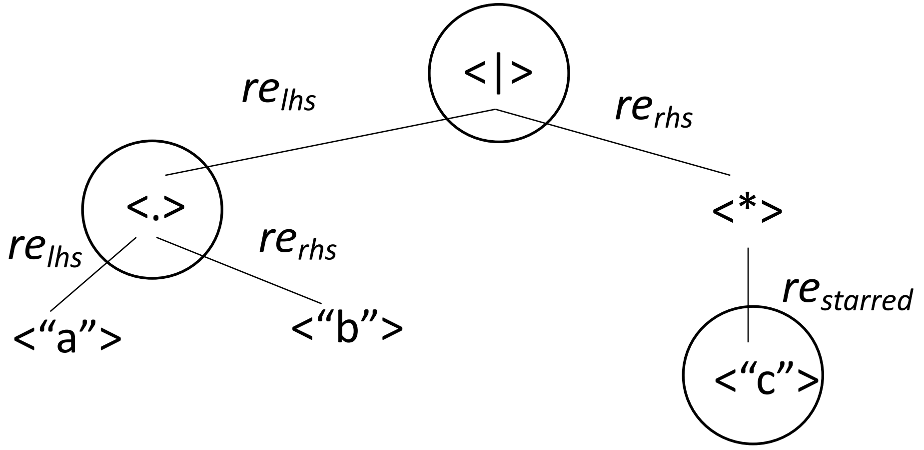


- re =
  - |{}
  - | “”
  - | a (single character)
  - | re_{lhs} | re_{rhs}
  - | re_{lhs} . re_{rhs}
  - | $re_{starred}$ *

AST 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}

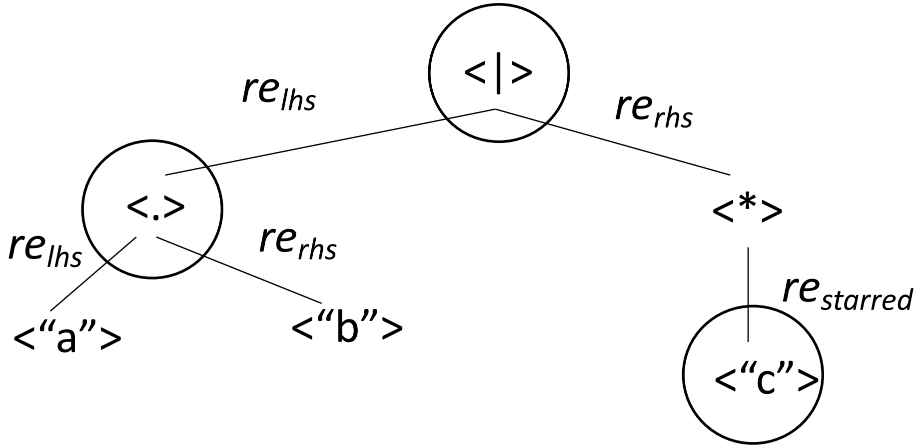
  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

AST 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 | b*
- *L = {"a", "b"}*
- $\delta_a(re)$ = *""*
- $\delta_b(re)$ = *""*

Regular expressions are closed under derivatives

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

Regular expressions are closed under derivatives

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

- *L = {"", "abc", "abcabc", ...}*
- δ_a *(L)= {"bc", "bcabc", "bcabc", ...}*
- δ_a*(re) = 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

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

Derivative Recursive Cases

Consider the recursive cases:

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

    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”, ...}*

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

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

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

    return ??

- re =

  | {}

  | ε

  | a (single character)

  | re_{lhs} | re_{rhs}

  | re_{lhs} . re_{rhs}

  | $re_{starred}$ *

Some properties/optimizations

How do certain regular expressions combine?

- 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*

$\delta_a(re)$ = ?

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:

$re = c^{*}.a$

$\delta_a(re)$ = “”

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) =

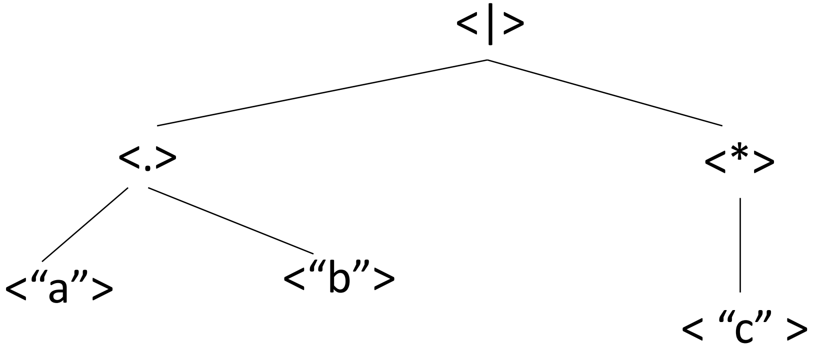
  if “” $\in re$ then: *“”*
  else: {}

Nullable operator

- NULL(re) =

  if “” $\in re$ 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)

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

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

  | $re_{starred} *$

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*?

	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*?

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

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 $s = c_1 . c_2 . c_3 ...$ (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)) == “”*
	L(δ_{c1} (re)) = {.. s[1:] ..}	*L($\delta_{c1,c2}$ (re)) = {.. s[2:] ..}*	L(δ_s*(re)*) = {.. “” ..}	

Homework discussion

- Part 2:
  - Create RE AST node classes
  - Base class: RE_AST_node
  - Derive leaf node classes from base class:
    - Character node
    - Empty string node
    - Empty set node

  - Derive RE operator nodes:
    - Unary operators; has one child
      - Star
      - Optional
    - Binary operators:
      - Union
      - Concat

Homework discussion

- Part 2:
  - Create RE AST when parsing:

Example using arithmetic

`input: 1+5*6`

Example: executing a mathematical expression during parsing

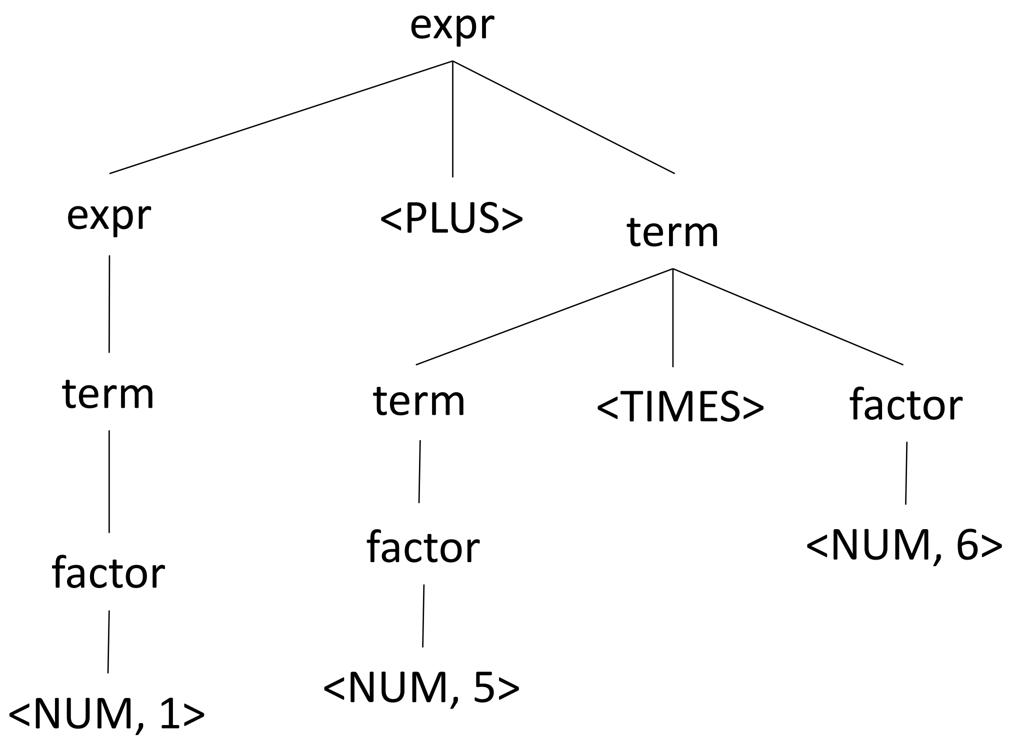
Children values are passed in as an array `C`, indexed from left to right

Operator	Name	Productions	Actions

+,-	expr	: expr PLUS term
		| expr MINUS term
		| term
			`{ret C[0] + C[2]}`
			`{ret C[0] - C[2]}`
			`{ret C[0]}`

*,/	term	: term TIMES factor
		: term DIV factor
		| factor
			`{ret C[0] * C[2]}`
			`{ret C[0] / C[2]}`
			`{ret C[0]}`

()	factor	: LPAR expr RPAR | NUM	`{ret C[1]}` `{ret int(C[0])}`



We have just implemented a simple arithmetic interpreter!

Homework discussion

- Implement the derivative and NULLABLE functions

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:

- 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}
  - | $re_{starred}$ *

Homework discussion

- To match a string:
  - Take the derivative with the first character, then the second, then the third…
  - At the end of the string, check if the resulting RE is nullable

- Consider some tricks to help improve efficiency of your matcher:

How do certain regular expressions combine?

- a | {} = a

- a . “” = a
- a . {} = {}

- “” * = “”
- {} * = {}

Part 1

- Difference between Statement and Expression?
  - Expression returns a value
  - Statement modifies the state of the program
  - Statement production rules are at the top
  - Should the parser accept an empty program?