Parsing Arithmetic Expressions

https://courses.missouristate.edu/anthonyclark/333/

Outline

Topics and Learning Objectives

- Learn about parsing arithmetic expressions
- Learn how to handle associativity with a grammar
- Learn how to handle precedence with a grammar

Assessments

- ANTLR grammar for math

Parsing Expressions

There are a variety of special purpose algorithms to make this task more efficient:

- The shunting yard algorithm
- Precedence climbing
- Pratt parsing

https://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing

For this class we are just going to use recursive descent

- Simpler
- Same as the rest of our parser

Grammar for Expressions

Needs to account for operator associativity

- Also known as fixity
- Determines how you apply operators of the same precedence
- Operators can be left-associative or right-associative (Binary Operator)

Needs to account for operator precedence

- Precedence is a concept that you know from mathematics
- Think PEMDAS
- Apply higher precedence operators first

Associativity

By convention

`7 + 3 + 1` is equivalent to `(7 + 3) + 1`,

`7 - 3 - 1` is equivalent to `(7 - 3) – 1`, and

`12 / 3 * 4` is equivalent to `(12 / 3) * 4`

- If we treated `7 - 3 - 1` as `7 - (3 - 1)` 2
  the result would be 5 instead of the 3.

- Another way to state this convention is *associativity*

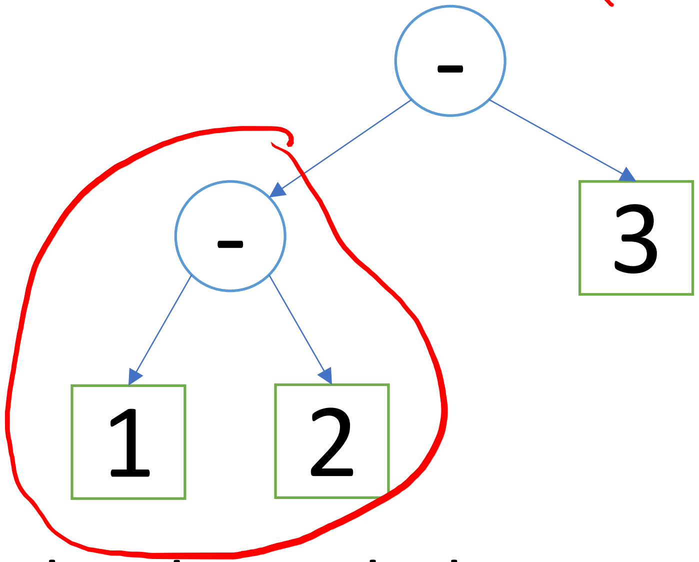
Associativity

Addition, subtraction, multiplication, and division are **left-associative**

AST

What does this mean?

1 - 2 - 3



You have:

- operators (+, -, *, /, etc.) and
- operands (numbers, ids, etc.)
- Left-associativity: if an operand has operators on both sides with the same precedence, it associates with the operator on its left

Associativity in the Grammar

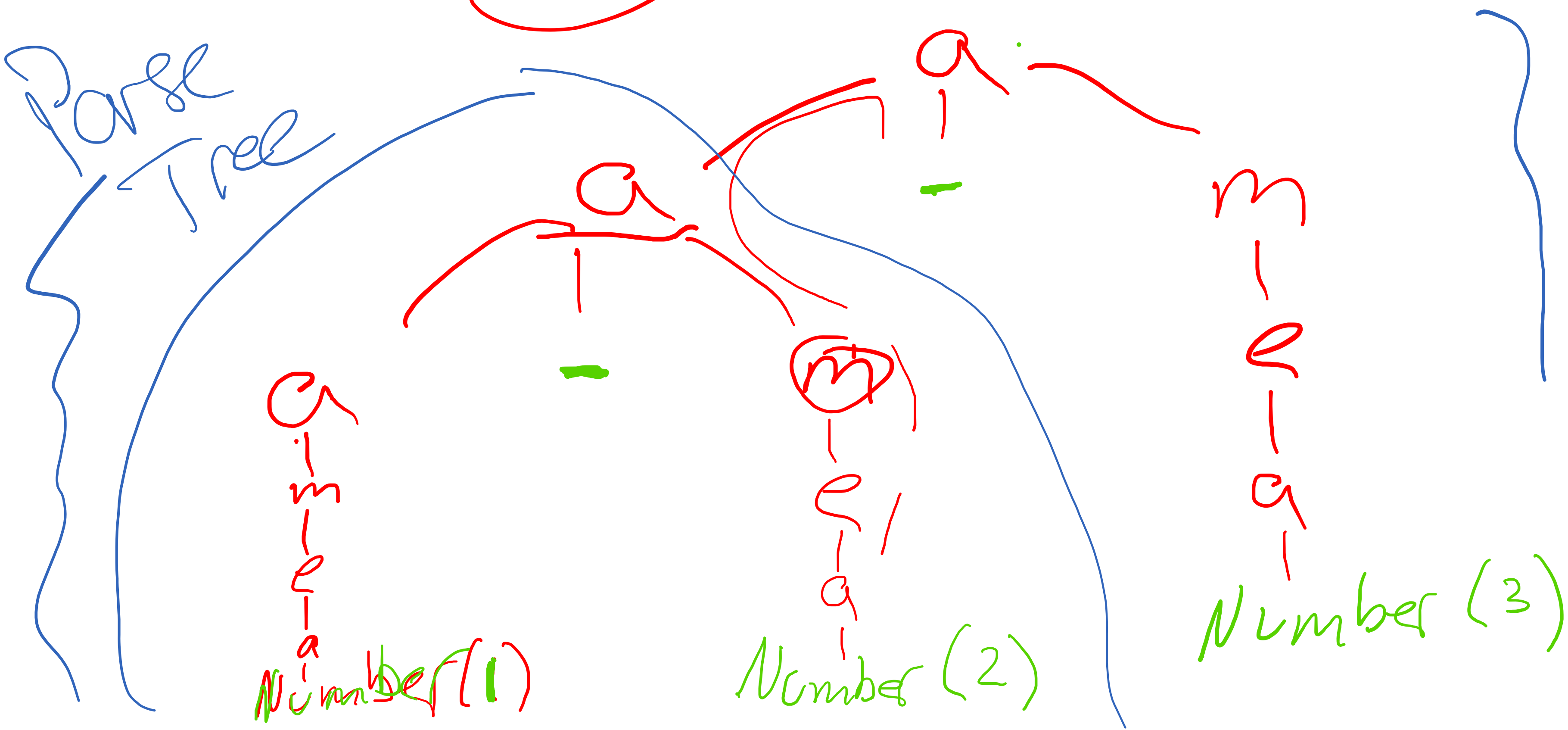
For left-associativity use left-recursion

For right-associativity use right-recursion

```
add_sub      : add_sub ('+' | '-') mul_div | mul_div;
mul_div      : mul_div ('*' | '/') exponentiate | exponentiate;
exponentiate : atom_unit '^' exponentiate | atom_unit;
atom_unit    : Number | '(' add_sub ')';
```

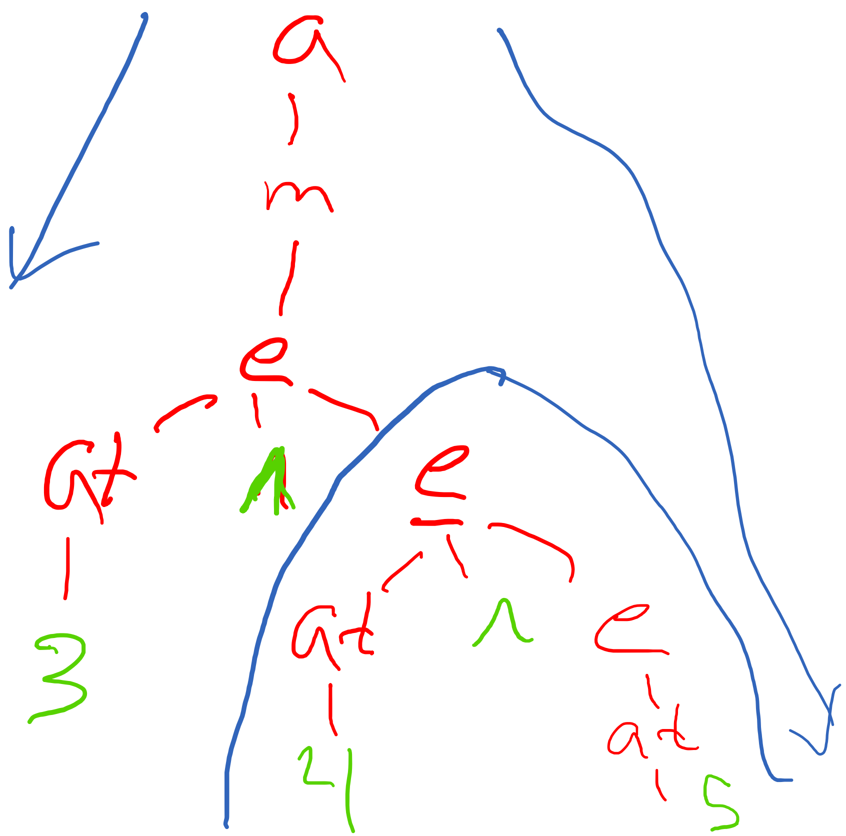
```
add_sub      : add_sub ('+' | '-') mul_div | mul_div;
mul_div      : mul_div ('*' | '/') exponentiate | exponentiate;
exponentiate : atom_unit '^' exponentiate | atom_unit;
atom_unit    : Number | '(' add_sub ')';
```

1 - 2 - 3

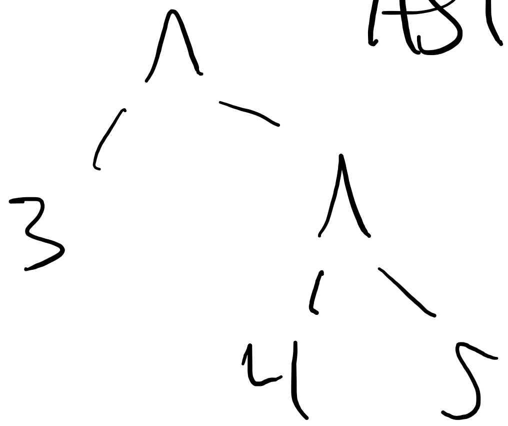


```
add_sub        : add_sub ('+' | '-') mul_div | mul_div;

mul_div        : mul_div ('*' | '/') exponentiate | exponentiate;

exponentiate : atom_unit '^' exponentiate | atom_unit;

atom_unit      : Number | '(' add_sub ')';
```

$\boxed{3 \wedge 4 \wedge 5}$



Parse Tree



AST

Associativity in the Grammar

We cannot implement this grammar.

Recursive Descent Parser

For left-associativity use left-recursion

For right-associativity use right-recursion

Any issues with implementing this grammar?

```
add_sub      : add_sub ('+' | '-') mul_div | mul_div;

mul_div      : mul_div ('*' | '/') exponentiate | exponentiate;

exponentiate : atom_unit '^' exponentiate | atom_unit;

atom_unit    : Number | '(' add_sub ')';
```

Precedence

What about the following expression: `7 + 5 * 2`

```
7 + (5 * 2)
(7 + 5) * 2
```

- Associativity is not helpful here…
- Now you just fall back to PEMDAS

Precedence in the Grammar

P	E	M	D	A	S

Higher-precedence should appear in “lower” (called later) grammar rules

```
add_sub       : add_sub ('+' | '-') mul_div | mul_div;

mul_div       : mul_div ('*' | '/') exponentiate | exponentiate;

exponentiate  : atom_unit '^' exponentiate | atom_unit;

atom_unit     : Number | '(' add_sub ')';
```

Precedence in the Grammar

E	M	D	A	S

Higher-precedence should appear in “lower” (called later) grammar rules

```
add_sub      : add_sub ('+' | '-') mul_div | mul_div;
mul_div      : mul_div ('*' | '/') exponentiate | exponentiate;
exponentiate : atom_unit '^' exponentiate | atom_unit;
atom_unit    : Number | '(' add_sub ')'; P
```

Precedence in the Grammar

M	D	A	S

Higher-precedence should appear in “lower” (called later) grammar rules

```
add_sub      : add_sub ('+' | '-') mul_div | mul_div;
mul_div      : mul_div ('*' | '/') exponentiate | exponentiate;
exponentiate : atom_unit '^' E exponentiate | atom_unit;
atom_unit    : Number | '(' add_sub ')'; P
```

Precedence in the Grammar

A S

Higher-precedence should appear in "lower" (called later) grammar rules

```
add_sub       : add_sub ('+' | '-') mul_div | mul_div;
mul_div       : mul_c M ('*' | '/') D ponentiate | exponentiate;
exponentiate  : atom_unit '^' E ponentiate | atom_unit;
atom_unit     : Number | '(' add_sub ')'; P
```

Precedence in the Grammar

Higher-precedence should appear in “lower” (called later) grammar rules

```
add_sub        : add_s A ('+' | '-') S l_div | mul_div;
mul_div        : mul_ M ('*' | '/') D ponentiate | exponentiate;
exponentiate   : atom_unit '^' E ponentiate | atom_unit;
atom_unit      : Number | '(' add_sub ')'; P
```

Table of Operators

Operator	Associativity	Precedence Level
+	left	1
-	left	1
*	left	2
/	left	2
^	right	3

A Grammar from the Table

Operator	Associativity	Level
+	left	1
-	left	1
*	left	2
/	left	2
^	right	3

For each precedence level in the table, define a non-terminal

- The RHS of the rule should match operators for that level
- The RHS should include non-terminals for the next higher precedence
- For left-associativity use left-recursion
- For right-associativity use right-recursion

If you have n precedence levels, you will need n + 1 rules

( ) Numbers, Ids, etc.

A Grammar from the Table

Operator	Associativity	Level
+	left	1
-	left	1
*	left	2
/	left	2
^	right	3

Pause

For each precedence level in the table, define a non-terminal

- The RHS of the rule should match operators for that level
- The RHS should include non-terminals for the next higher precedence
- For left-associativity use left-recursion
- For right-associativity use right-recursion

Activity

If you have n precedence levels, you will need n + 1 rules

```
hashat : hashat '#' dollar
       | dollar '@' hashat
       | dollar ;

dollar : dollar '$' atom
       | atom ;

atom : Numbers | Identifiers | '(' hashat ')' ;
```

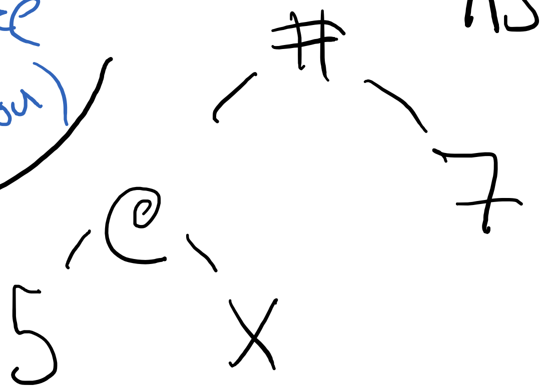
Operator	Associativity	Level
#	Left	1
@	Right	1
$	Left	2

The basic units are Numbers, Identifiers, and parenthetical expressions.

(5 @ x) # 7

Parse Tree (for you)

AST



```
hashat : hashat '#' dollar | dollar '@' hashat | dollar ;
dollar : dollar '$' unit | unit ;
unit   : Number | Identifier | '(' hashat ')' ;
```

Left-Recursion Transformation

Input Pattern:

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

Output Pattern:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

hashat : dollar @ hashat hashat' | dollar

hashat' : # dollar hashat' | ε ;

dollar : unit dollar' ;

dollar' : $ unit dollar' | ε ;

```
expr : ID '+' expr
     | ID '*' expr
     | '(' expr ')'
     | ID
     | NUMBER
```

Try to parse the following:

```
x + y + z
(x + y) * z
z * (x + y)
x + y * z
x * y + z
```

Is the grammar expressive?

No

Is precedence correct?

No

Is associativity correct?

No

Can we implement the grammar as is?

Yes

```
expr  :  ID '+' expr
      |  ID '*' expr
      |  '(' expr ')'
      |  ID
      |  NUMBER
```

Try to parse the following:

```
x + y + z
(x + y) * z
z * (x + y)
x + y * z
x * y + z
```

Is the grammar expressive?

No

Is precedence correct?

No

Is associativity correct?

No

Can we implement the grammar as is?

Yes

```
expr    :   expr '+' expr
        |   expr '*' expr
        |   '(' expr ')'
        |   ID
        |   NUMBER
```

Try to parse the following:

```
(x + y) * z
x + y + z
```

Is the grammar expressive?

Yes

Is precedence correct?

No

Is associativity correct?

No (ambiguous)

Can we implement the grammar as is?

No (left recursions)

Pause

```
expr   :   term '+' expr
       |   term '*' expr
       |   term
term   :   '(' expr ')'
       |   ID
       |   NUMBER
```

?

Try to parse the following:

```
(x + y) * z
x + y * z
x * y + z
```

Is the grammar expressive?

No

Is precedence correct?

No

Is associativity correct?

No

Can we implement the grammar as is?

Yes

Pause

```
expr   :   expr '+' term
       |   expr '*' term
       |   term
term   :   '(' expr ')'
       |   ID
       |   NUMBER
```

Try to parse the following:

```
(x + y) * z
x + y * z
x * y + z
x + y + z
```

Is the grammar expressive?

Yes

Is precedence correct?

No

Is associativity correct?

Yes

Can we implement the grammar as is?

No (left recursions)

Pause

```
expr   :   expr '+' expr
       |   term
term   :   term '*' term
       |   factor
factor :   '(' expr ')'
       |   ID
       |   NUMBER
```

Try to parse the following:

```
(x + y) * z
x + y * z
x * y + z
x + y + z
```

Is the grammar expressive?

Yes

Is precedence correct?

Yes

Is associativity correct?

No (ambiguous)

Can we implement the grammar as is?

No (left recursions)

```
expr   :  expr '+' term
       |  term
term   :  term '*' factor
       |  factor
factor :  '(' expr ')'
       |  ID
       |  NUMBER
```

Try to parse the following:

```
(x + y) * z
x + y * z
x * y + z
x + y + z
```

Is the grammar expressive?
Yes

Is precedence correct?
Yes

Is associativity correct?
Yes

Can we implement the grammar as is?
No (left recursions)

Pause

Do it

Do it now

Expression Grammar

Start with this:

```
expr   -> expr ('+' | '-') term | term
term   -> term ('*' | '/') factor | factor
factor -> NUMBER | ID | '(' expr ')'
```

Equivalent

Turn it in to this (after removing left recursion):

```
expr   -> term expr'
expr'  -> ('+' | '-') term expr' | ep
term   -> factor term'
term'  -> ('*' | '/') factor term' | ep
factor -> NUMBER | ID | '(' expr ')'
```

Associativity ✓
Precedence ✓
Expressive ✓
Not left recursion ✓

Left recursive grammar (cannot be implemented using recursive descent)

```
expr   -> expr ('+' | '-') term | term
term   -> term ('*' | '/') factor | factor
factor -> NUMBER | ID | '(' expr ')'
```

Expression Grammar

BNF (Backus–Naur form):

```
expr    -> term expr'
expr'   -> ('+' | '-') term expr' | ep
term    -> factor term'
term'   -> ('*' | '/') factor term' | ep
factor -> NUMBER | ID | '(' expr ')'
```

Becomes while-loop

EBNF (Extended Backus–Naur form):

```
expr    -> term [ ('+' | '-') term ]*
term    -> factor [ ('*' | '/') factor ]*
factor -> '(' expr ')' | ID | NUMBER
```

Much easier to implement!