# Recursive Data Structures

FOCS, Fall 2020

# Recursive data structures

A recursive data structure is a data structure made up of pieces that are instances of the data structure

Simplest example: a linked list is one of:

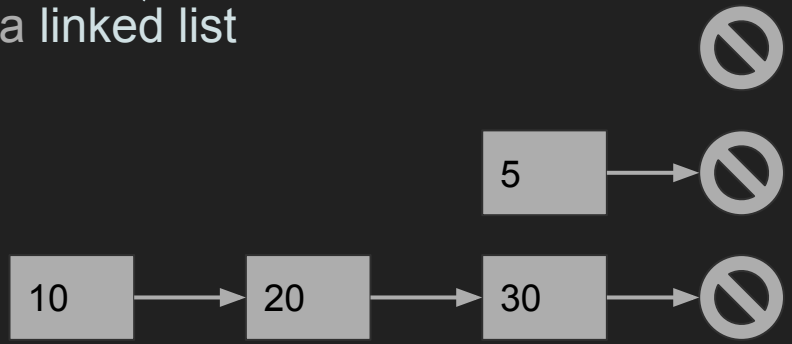
- an empty list, or
- a pair of an item and a linked list

# Recursive data structures

A recursive data structure is a data structure made up of pieces that are instances of the data structure

Simplest example: a linked list is one of:

- an empty list, or
- a pair of an item and a linked list



# Recursive functions over recursive data structures

Most functions that works recursive data structures tend to be recursive functions

- Functions usually follow the shape of the data they work on

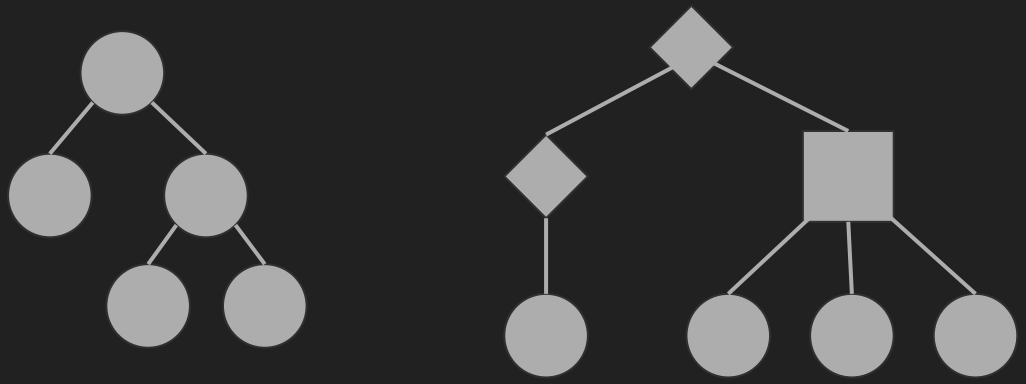
To sum all the elements of a list:

- if the list is the empty list: the sum is 0
- if the list if an integer N followed by a list L:
  recursively sum all the elements in L, and add N to it

We know all of this already for lists

# Tree data structures

The general form of a recursive data structure is a tree:

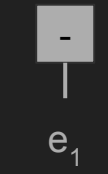


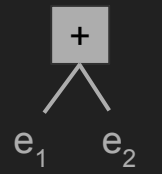
- where do we store values?
- kind of nodes?
- number of children?

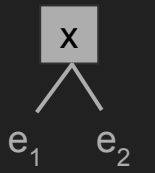
# Abstract syntax trees: arithmetic expressions


An arithmetic expression is one of:

- a number *n*
- - ($e_1$) where $e_1$ is an arithmetic expression
- ($e_1$ + $e_2$) where $e_1$ and $e_2$ are arithmetic expressions
- ($e_1$ x $e_2$) where $e_1$ and $e_2$ are arithmetic expressions



# Abstract syntax trees: arithmetic expressions

An arithmetic expression is one of:

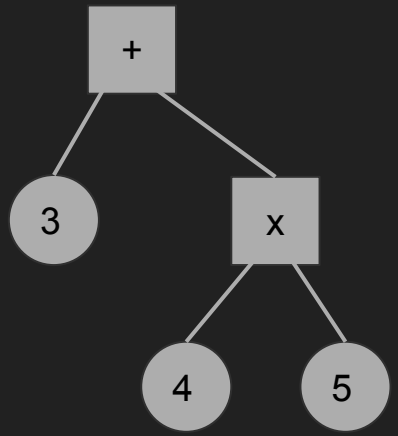
- a number *n*
- - ($e_1$) where $e_1$ is an arit
- ($e_1$ + $e_2$) where $e_1$ and
- ($e_1$ x $e_2$) where $e_1$ and

Often written as a BNF grammar:

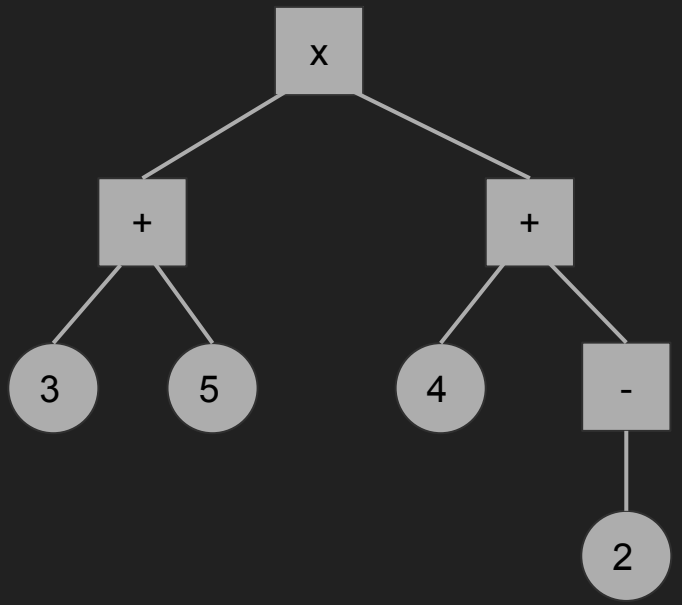
*e* ::= *num*
- ($e_1$)
($e_1$ + $e_2$)
($e_1$ x $e_2$)

# Examples

3 + (4 x 5)




(3 + 5) x (4 + (-2))



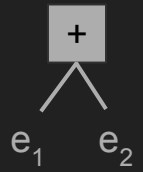
# Evaluation

eval 

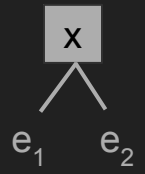
= n

eval 

= - eval $e_1$

eval 

= eval $e_1$ + eval $e_2$

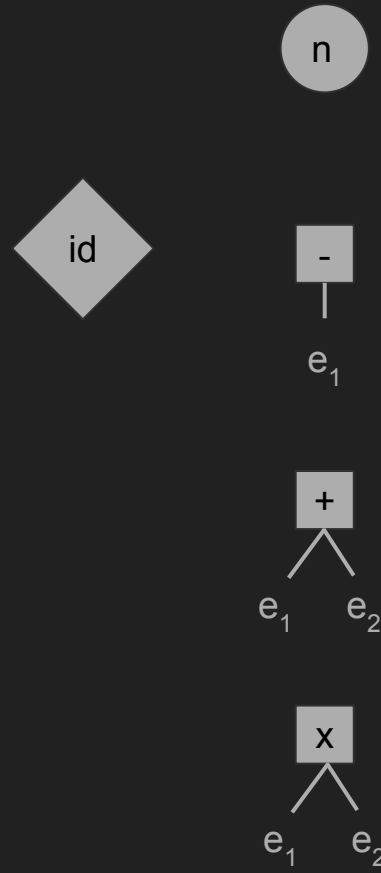
eval 

= eval $e_1$ * eval $e_2$

# Abstract syntax trees: adding identifiers

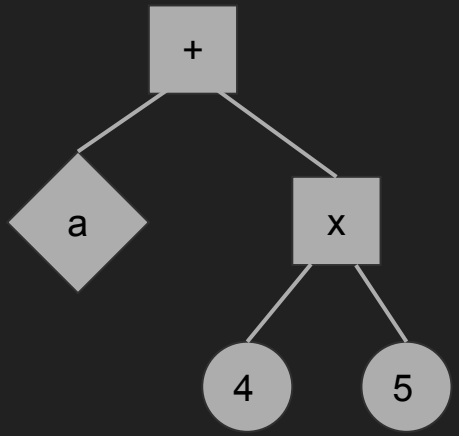
An arithmetic expression is one of:

- a number *n*
- an identifier *id*
- - ($e_1$) where $e_1$ is an arithmetic expression
- ($e_1$ + $e_2$) where $e_1$ and $e_2$ are arithmetic expressions
- ($e_1$ x $e_2$) where $e_1$ and $e_2$ are arithmetic expressions

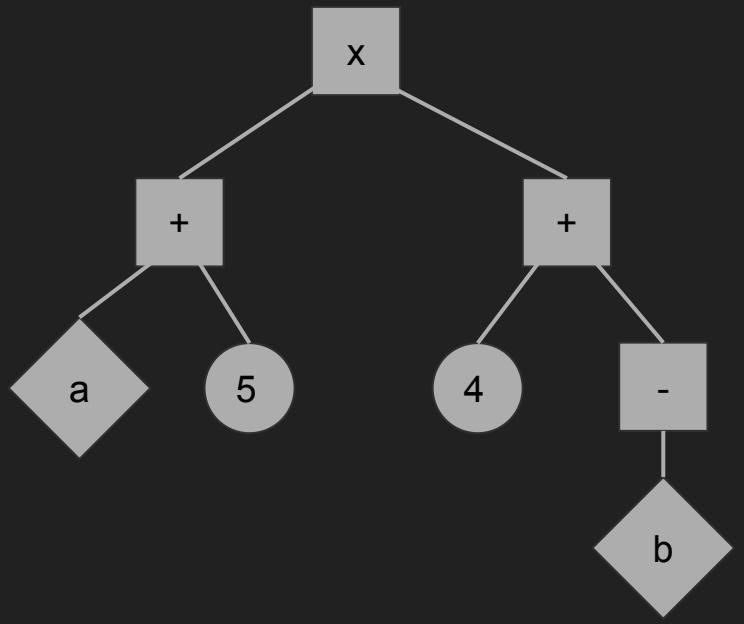


# Examples

a + (4 x 5)




(a + 5) x (4 + (-b))



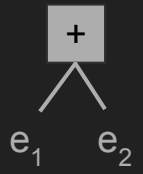
# Evaluation with environments

eval E 

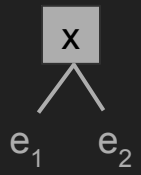
= n

eval E 

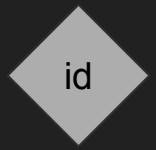
= - eval E $e_1$

eval E 

= eval E $e_1$ + eval E $e_2$

eval E 

= eval E $e_1$ * eval E $e_2$

eval E 

= E(id)

# Implementation using algebraic datatypes

```
type aexp =
   | Num of int
   | Neg of aexp
   | Plus of aexp * aexp
   | Times of aexp * aexp
```

```
let rec eval aexp =
   match aexp with
    | Num n -> n
    | Neg e1 -> - (eval e1)
    | Plus (e1, e2) -> (eval e1) + (eval e2)
    | Times (e1, e2) -> (eval e1) * (eval e2)
```

Example: `Plus (Num 3, Times (Num 4, Num 5))`

# Implementation using algebraic datatypes

```
type aexp =
   | Num of int
   | Neg of aexp
   | Plus of aexp * aexp
   | Times of aexp * aexp
   | Ident of string
```

```
let rec eval env aexp =
   match aexp with
    | Num n -> n
    | Neg e1 -> - (eval env e1)
    | Plus (e1, e2) -> (eval env e1) + (eval env e2)
    | Times (e1, e2) -> (eval env e1) * (eval env e2)
    | Ident s -> lookup env s
```

Example: `Plus (Ident "a", Times (Num 4, Num 5))`

# Implementation using classes

```
abstract class AExp {
  abstract int eval ();
}
```

```
class Num extends AExp {
  int n;

  Num (int n) {
    this.n = n;
  }

  int eval () {
    return this.n;
  }
}
```

# Implementation using classes

```
abstract class AExp {
  abstract int eval ();
}
```

```
class Neg extends AExp {
  AExp e1;

  Neg (AExp e1) {
    this.e1 = e1;
  }

  int eval () {
    return -(this.e1.eval());
  }
}
```

# Implementation using classes

```
abstract class AExp {
  abstract int eval ();
}
```

```
class Plus extends AExp {
  AExp e1, e2;

  Plus (AExp e1, AExp e2) {
    this.e1 = e1;
    this.e2 = e2;
  }

  int eval () {
    return this.e1.eval() + this.e2.eval();
  }
}
```

# Implementation using classes

```
abstract class AExp {
  abstract int eval ();
}
```

```
class Times extends AExp {
  AExp e1, e2;

  Times (AExp e1, AExp e2) {
    this.e1 = e1;
    this.e2 = e2;
  }

  int eval () {
    return this.e1.eval() * this.e2.eval();
  }
}
```

# Implementation using classes

```
abstract class AExp {
  abstract int eval ();
}
```

Example:

```
AExp e = new Plus(new Num(3),
                  new Times(new Num(4),
                            new Num(5)));
```

# JSON in OCaml

A JSON item is one of:

- an integer
- a string
- a list of JSON items
- an object, mapping string keys to JSON items

Standard notation:

```
{
   "a" : [1, 2, 3],
   "b" : { "c" : 1, "d" : "two" }
}
```

# JSON in OCaml

```
type json =
  | JInteger of int
  | JString of string
  | JList of json list
  | JObject of (string * json) list
```

Recursive functions to

- dump a JSON object to a file or a string
- transform (map) a JSON object
- traverse (fold) a JSON object