CS 320: Concepts of Programming Languages

Lecture 6: ADTs Everywhere!

Ankush Das

Nathan Mull

Sep 19, 2024

Course Feedback

- Thank you everyone for taking the time to provide feedback!!
- Highlights:
  - *All typing and semantics rules will come with English explanations*
  - *No more criticisms of programming languages*
  - *Spend more time on explanations of typing and semantics derivations*
  - *More on this today and in future lectures*
  - *A lot more live coding in the lectures, taking students' suggestions*

This Week's Abstraction

- **Today, we will continue studying data structures and Algebraic Data Types (ADTs), also called Abstract Data Types, User-Defined Types**
- **As usual, we will study syntax, type system, and semantics of lists**
- **We will do more examples today!**
- *Homework: Get a lot of practice with ADTs, they are the most important concept in OCaml and this course*

Binary Trees


```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

- Very much like list, except has two recursive calls instead of one
- There is no element at the leaf (just my choice); we can also change it to have elements at leaf nodes

```
type 'a tree' =
  | Leaf' of 'a
  | Node' of 'a * 'a tree' * 'a tree'
```

- They are equivalent! Why?

Proof of Equivalence


```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

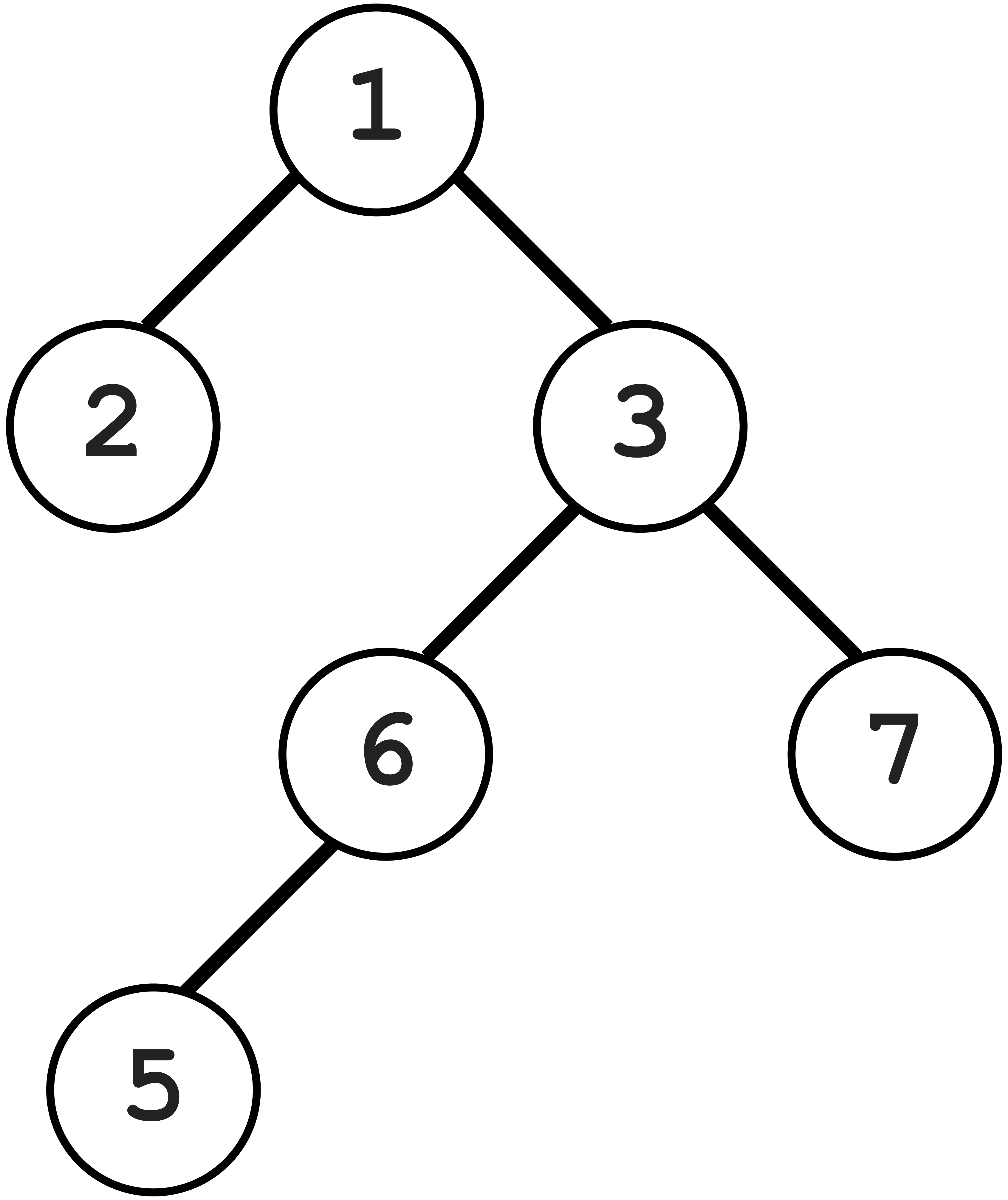
```
type 'a tree' =
  | Leaf' of 'a
  | Node' of 'a * 'a tree' * 'a tree'
```

'a tree' -> 'a tree

```
let rec convert tr' =
  match tr' with
  | Leaf'(x) -> Node(x, Leaf, Leaf)
  | Node'(x, l, r) -> Node(x, convert l, convert r)
```

- *Homework: Write the function for the opposite direction:* `convert' : 'a tree -> 'a tree'`
- We will use `'a tree`

Constructing Trees




```
int tree
let n5 = Node (5, Leaf, Leaf)

int tree
let n6 = Node (6, n5, Leaf)

int tree
let n7 = Node (7, Leaf, Leaf)

int tree
let n3 = Node (3, n6, n7)

int tree
let n2 = Node (2, Leaf, Leaf)

int tree
let n1 = Node (1, n2, n3)
```

Size and Depth of Tree


```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

'a tree -> int

```
let rec size tr =
  match tr with
  | Leaf -> 0
  | Node(_, l, r) -> 1 + (size l) + (size r)
```

'a tree -> int

```
let rec depth tr =
  match tr with
  | Leaf -> 0
  | Node(_, l, r) -> 1 + max (depth l) (depth r)
```

Pre/In/Post Orders of Tree


```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

- `@` operator appends two lists

```
@ : 'a list ->
    'a list ->
    'a list
```

```
'a tree -> 'a list
let rec pre_order tr =
  match tr with
  | Leaf -> []
  | Node(x, l, r) -> [x] @ pre_order l @ pre_order r

'a tree -> 'a list
let rec in_order tr =
  match tr with
  | Leaf -> []
  | Node(x, l, r) -> in_order l @ [x] @ in_order r

'a tree -> 'a list
let rec post_order tr =
  match tr with
  | Leaf -> []
  | Node(x, l, r) -> post_order l @ post_order r @ [x]
```

Homework

- Read OCaml book: 3.8, 3.9, 3.11, 3.12
- Take functions from your programming assignments
  - *Write typing derivations for them*
  - *Test them on some small simple input by writing semantic derivations for them*