Parser: due Monday

Interpreter architecture (compiles too)

3 stages

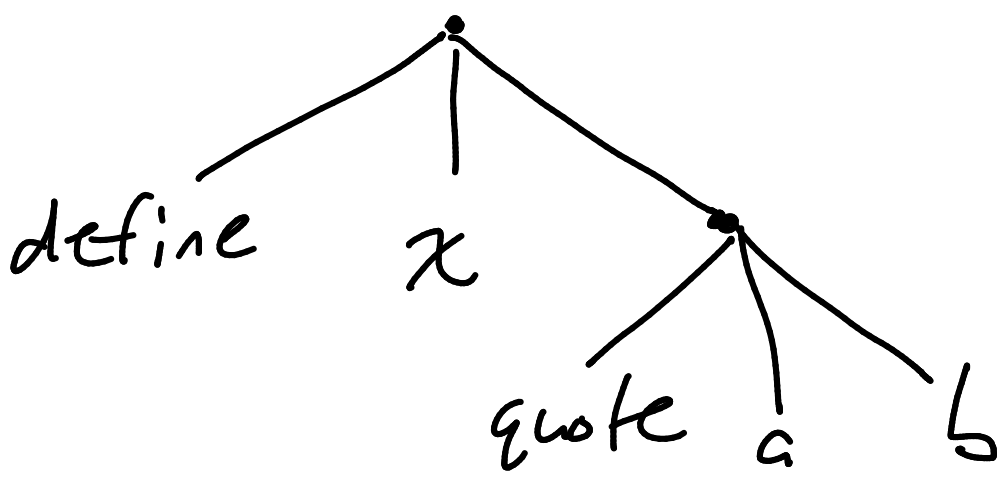
(1) Tokenizing (finding basic units)

(2) Parsing (assembly tokens into a structure, typically a tree or collection of trees)

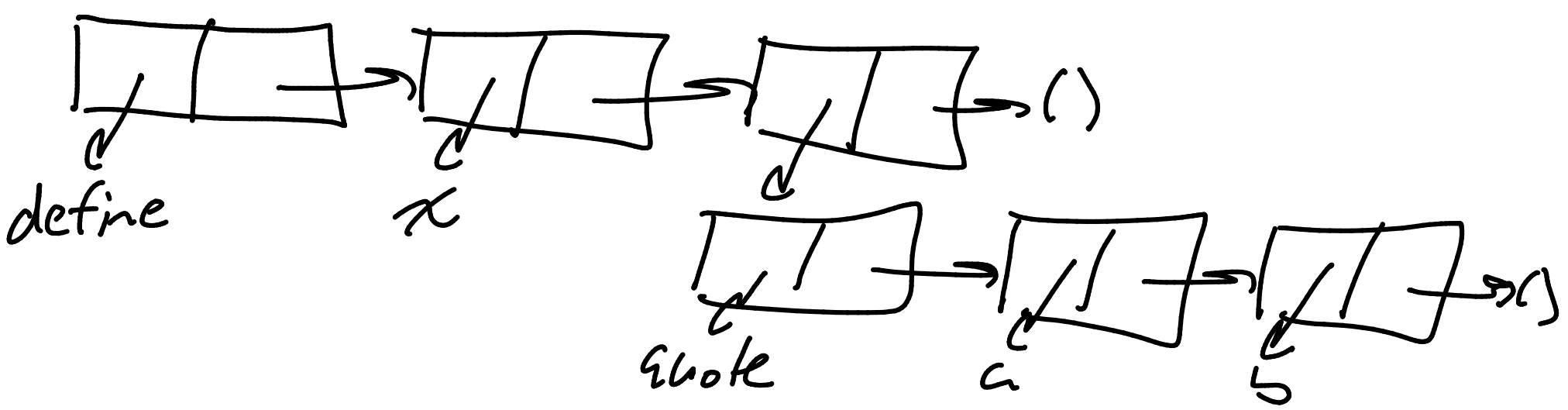
(3) Interpret: code eval
Compile: code translation

> traverse tree(s) and evaluate or generate code

(define x (quote a b)) underlines indicate tokens



parsing



Parsing algorithms in general

LL parsing

LR parsing

read your code from left to right

second letter refers to how you navigate through the grammar

from left side to right

<B> → 0 <B> | 1 <B> | 0 | 1

Parse 010

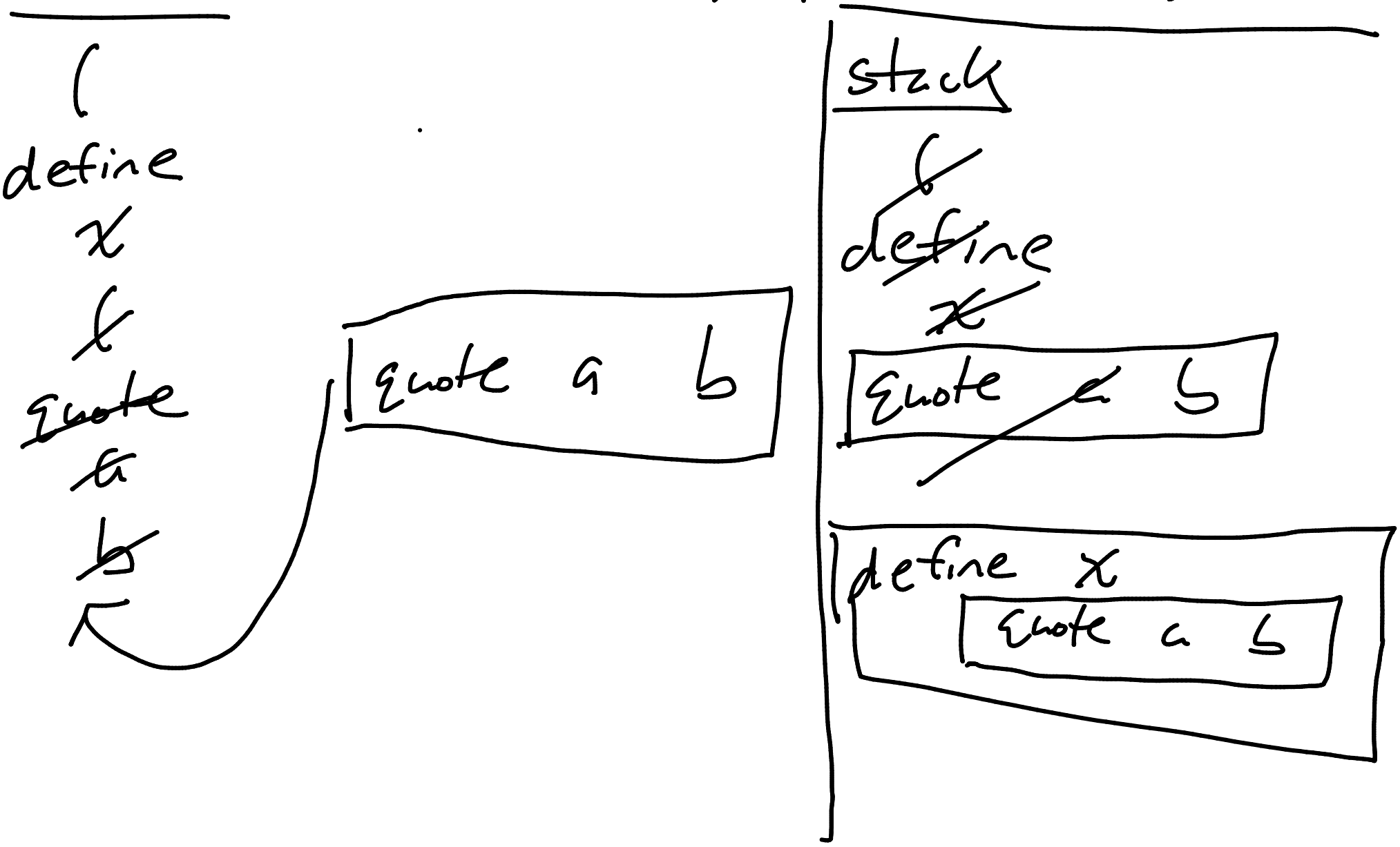
<B>

To parse a Scheme program, we will use an LR algorithm specifically designed for parsing Scheme

- Initialize an empty stack (which we implement as a linked list of SchemeVals).
- While there are more tokens:
  - Get the next token.
  - If the token is anything other than a close paren, push it onto the stack.
  - If the token is a close paren, start popping items from your stack until you pop off an open paren (and form a list of them as you go). Push that list back on the stack.

(define x (quote a b))

stack (list of SchemeVals, top is at head)



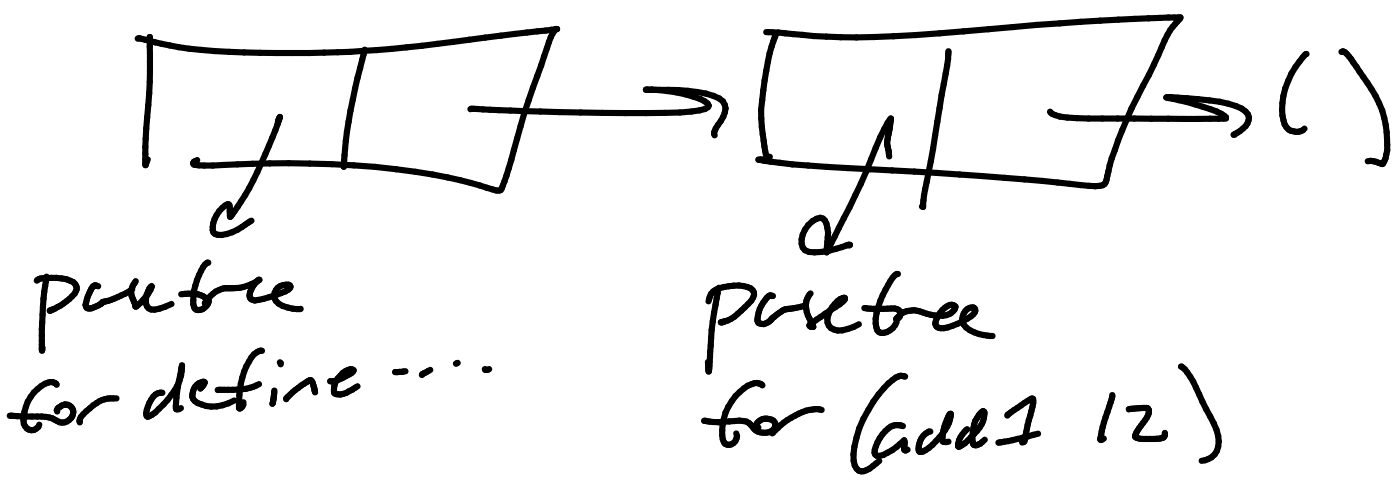
A few details:

One place Scheme does not enclosing parens is at the top level of a program.

```
(define add1 (lambda (x) (+ x 1)))
(add1 12)
```

A Scheme program is really a collection of Scheme expressions.

Your parsing code actually returns a list of parse trees.



One of the detail: the annoying single quote.

(define x '(a b))

↑

abomination

Concept: the single quote wraps everything that follows in (quote .....)

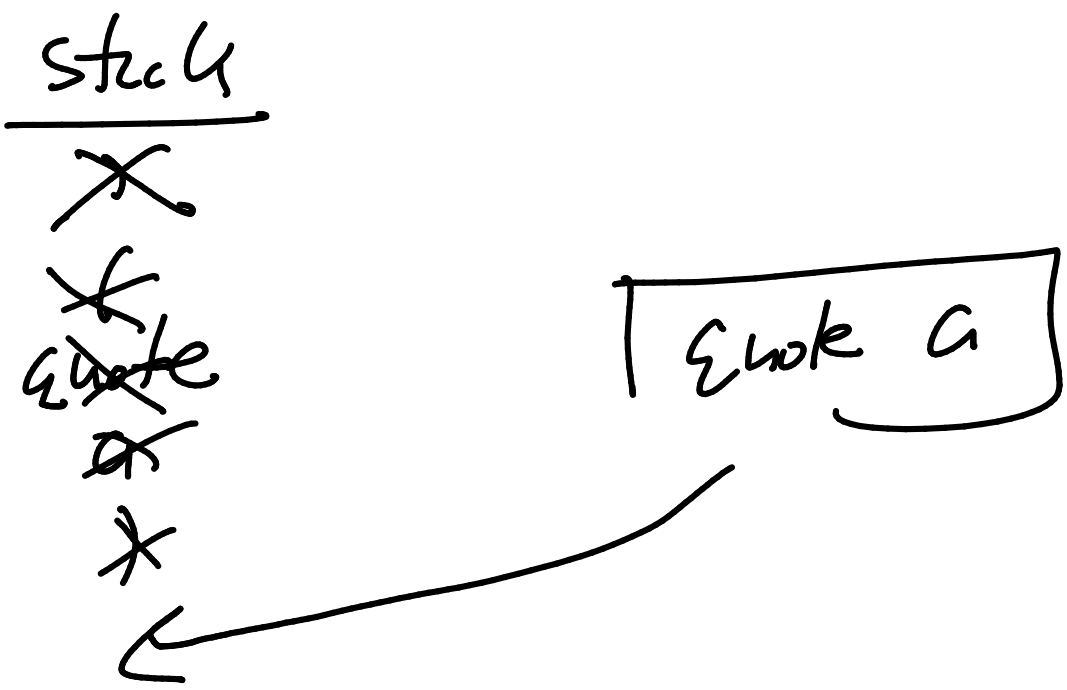
(quote (a b))

Dealing with the special case of the single quote (i.e. `'a`, or `'(a b c)`) means that we need to redefine "push onto the stack," which appears a few times in the parsing algorithm. Here is the new definition of "push onto the stack", which you only need to pass all of the E tests:

- Proceed as usual so long if the stack doesn't have a single quote `'` on top.
- Also proceed as usual if pushing on a left paren, no matter what the stack looks like.
- In all other cases...
  - Pop the single quote off the stack. Then proceed as if the token sequence had `(quote ....)` wrapped around the value you're pushing. In other words, create a new subtree consisting of `quote` and the new token, and then push that subtree onto the stack instead of the original value you were going to push.

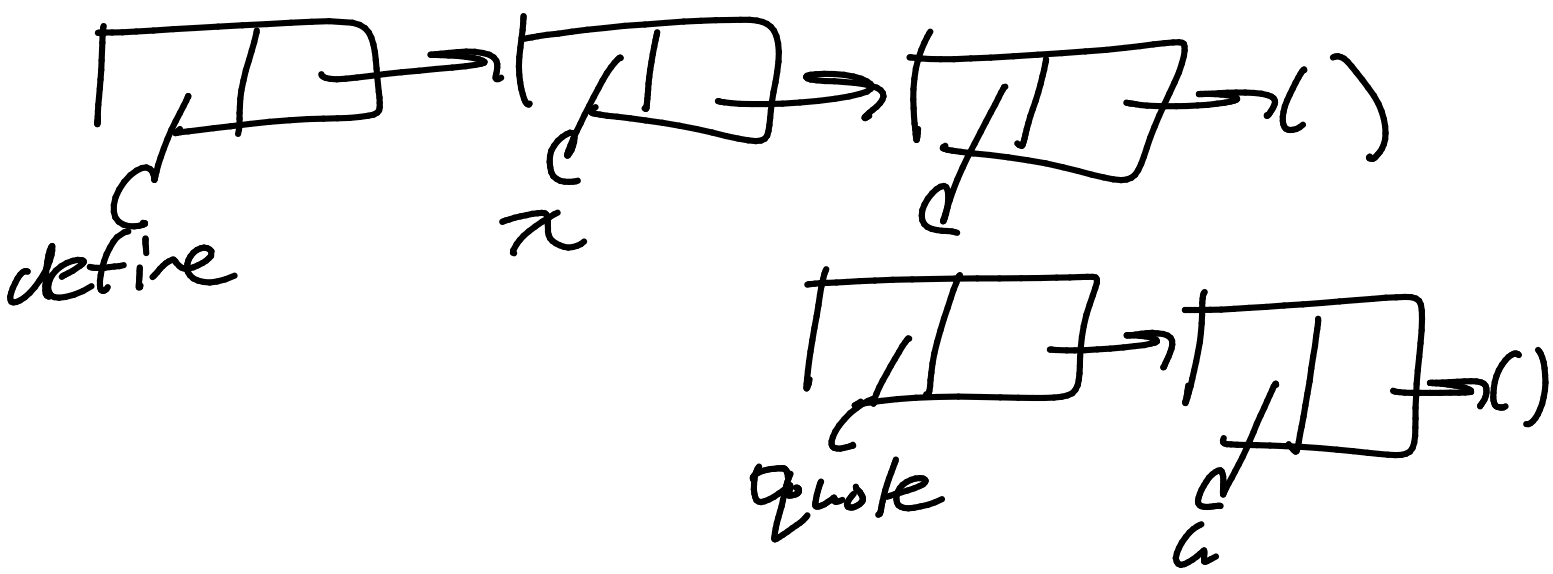
'a

stack



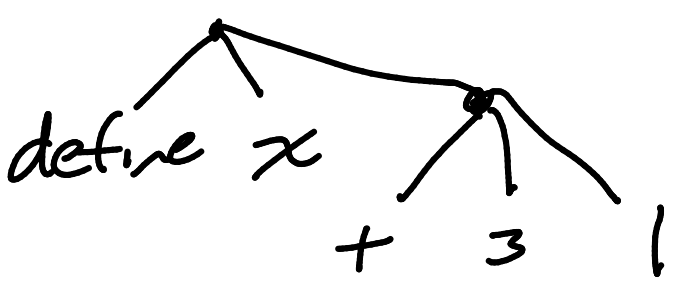
Why don't we actually store the parentheses themselves in the tree?

(define x (quote a))



They end up being redundant, because tree structure already captures that information.

(define x (+ 3 1))



How can you visualize your work in progress?

- print things out
- gdb

Parsing in general
(not about the project in particular)

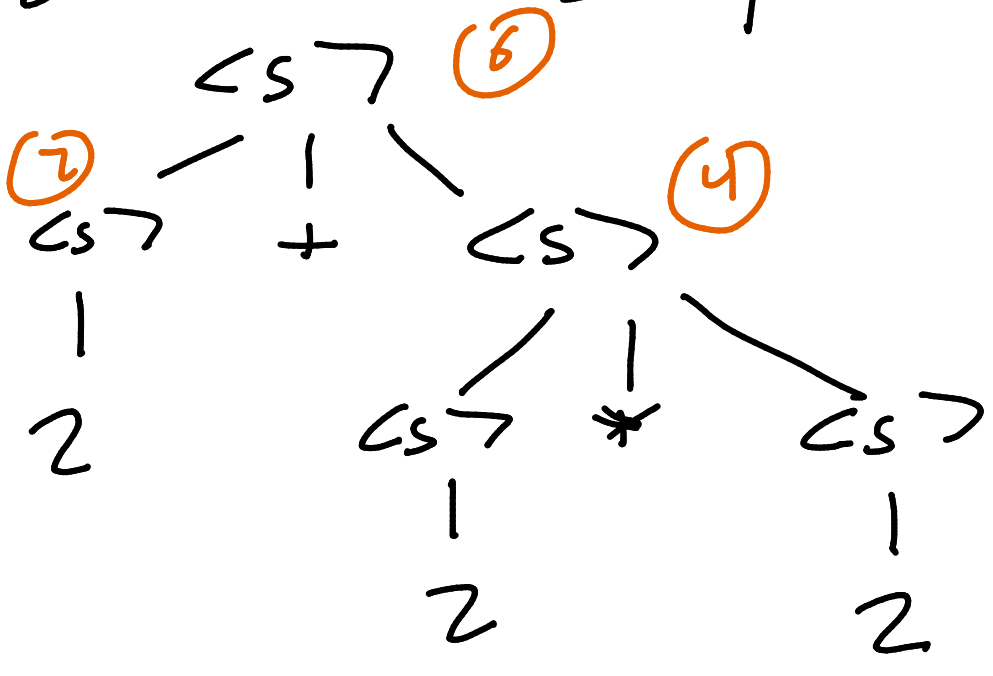
It's often the case that you have a BNF grammar that you're using to define language

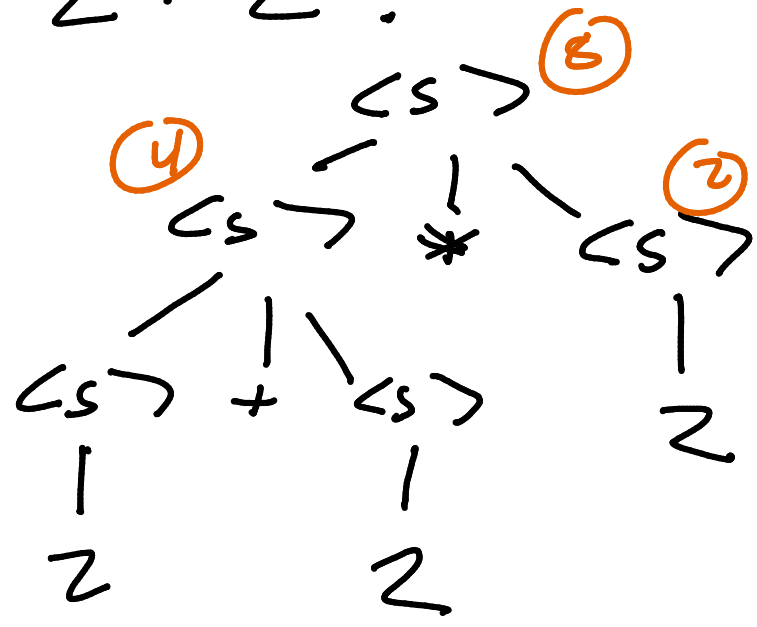
---

Small calculator language

<S> ::= <S> + <S>
| <S> * <S>
| 2

How could it parse 2 + 2*2?





A gramma like this one that results in multiple possible parse trees for the same expression is called an ambiguous gramma (generally bad)

- different parse trees when evaluated typically result in different results

---

one way to fix this gramma looks like:

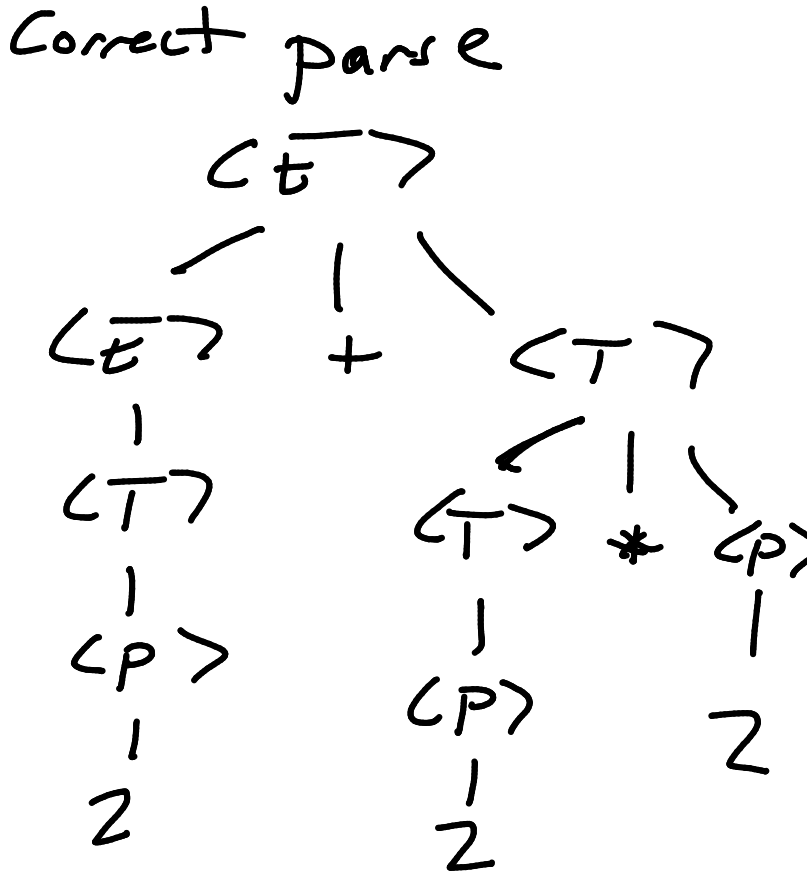
start

<E> ::= <E>+<T> | <T>

<T> ::= <T>*<P> | <P>

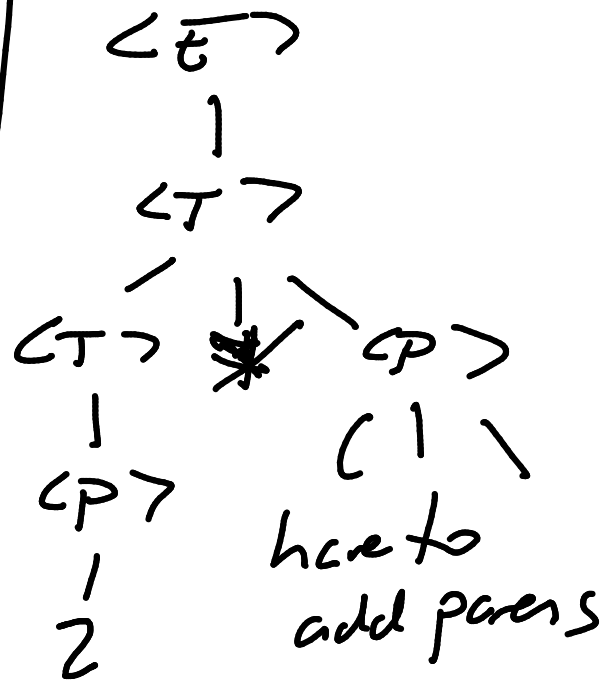
<P> ::= 2 | (<E>)

Correct parse



2 + 2 * 2

Try to mess it up



Deadlines (course schedule)