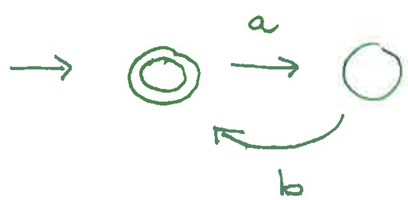
CFGs

exer Write an NFA that is equivalent to the following CFGs

(1) $S \to abS \mid \varepsilon$ (2) $S \to aSb \mid \varepsilon$.

soln (1)



(2) Impossible!

(take 452 for proof!)

$\square$

exer Is the following CFG ambiguous?
If so show two left-most derivations.
Identify the difference between these parse trees.

$$E \to a \mid b \mid E+E \mid E*E$$

soln. There are two sources of ambiguity.

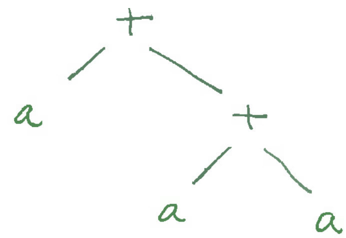
(1) Unspecified associativity.

- $\underline{E} \to \underline{E}+E \to a+\underline{E} \to a+\underline{E}+E$
  $\to a+a+\underline{E} \to a+a+a$

- $\underline{E} \to \underline{E} + E \to \underline{E} + E + E \to a + \underline{E} + E$
  $\to a + a + \underline{E} \to a + a + a$

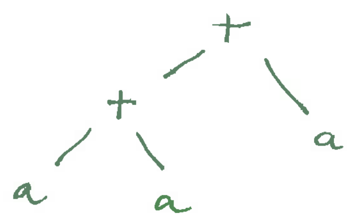
Corresponding to

- [Right Assoc.]



$(a + (a + a))$

- [Left Assoc.]



$((a + a) + a)$

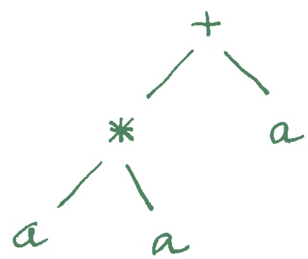
(2) *Unspecified Precedence*

- $\underline{E} \to \underline{E} + E \to \underline{E} * E + E \to a * \underline{E} + E$
  $\to a * a + \underline{E} \to a * a + a$

- $\underline{E} \to \underline{E} * E \to a * \underline{E} \to a * \underline{E} + E$
  $\to a * a + \underline{E} \to a * a + a$

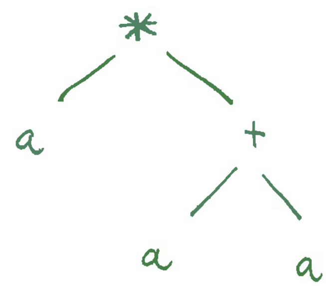
Corresponding to

- [Normal Precedence]



$((a*a)+a)$

- [Wrong Precedence]



$(a*(a+a))$

∎

So, we can modify the grammar to specify these and make it unambiguous.

(1) <u>Specify</u> <u>associativity</u> by forcing recursion on one side.

$$E \to T+E \mid T*E \mid T$$
$$T \to a \mid b$$

<u>exer</u> What associativity does this correspond to?

(2) Specify precedence by giving a hierarchy to the operators.

$$
\begin{aligned}
E &\rightarrow T + E \mid T \\
T &\rightarrow P * T \mid P \\
P &\rightarrow a \mid b
\end{aligned}
$$

↑ low prec.

high prec. ↓

A CFG describes a language in a way that easily let's us generate strings. We usually don't care about that, we already have the string (source code) we care about the parse tree. This process is called parsing.

$$\text{Source} \rightarrow \boxed{\text{Lexer}} \rightarrow \text{tokens} \rightarrow \boxed{\text{Parser}} \rightarrow \text{AST} \rightarrow \boxed{\text{Interp.}} \rightarrow \text{result}$$

Each of these is interesting in its own right.

Lexer — regular expression matching (you know how this works already, P3)

Parser — many techniques, in 330 we'll see recursive descent parsing, but in 430 you'll see LR parsing (and there are many more)

Interp. — we use a definitional interpreter based on operational semantics (future lecture)

Recursive Descent Parsing

goal Is a string in my language? If so, what's its parse tree?

how Recursively replace non-terminals using a lookahead to guide your choice e.g. $E \rightarrow id = n \mid \{ L \}$

$L \rightarrow E \; ; \; L \mid \varepsilon$

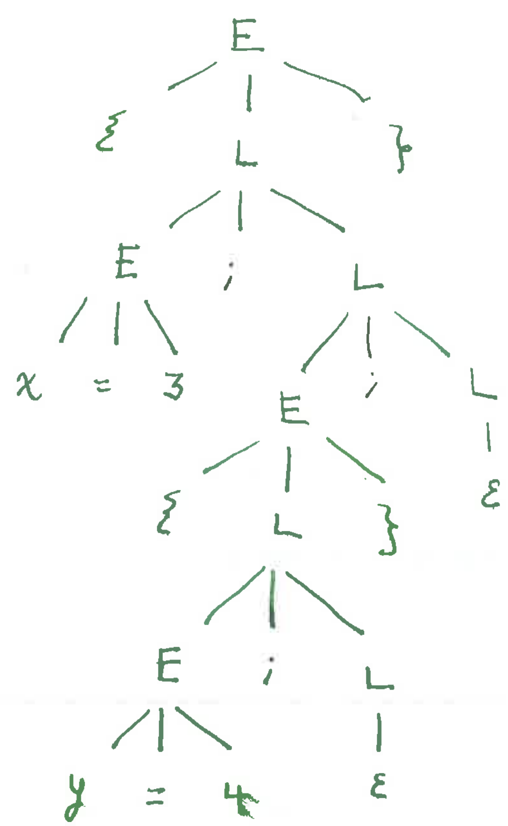
where id is identifier
n is number

e.g. (input)

$\{ \quad x \quad = \quad 3 \quad ; \quad \{ \quad y \quad = \quad 4 \quad ; \quad \} \quad ; \quad \}$

each are tokens

e.g. (parse tree)



First, lexer.ml:

```
module L = List
module S = String
module R = Str

(* Type *)
type token =
| Tok_Var of string
| Tok_Int of int
| Tok_Eq
| Tok_LCurly
| Tok_RCurly
| Tok_Semi
| Tok_EOF

(* Regular expressions and the tokens they generate. *)
let re = [
  (R.regexp "[a-z]+" , fun x -> [Tok_Var x]) ;
  (R.regexp "[0-9]+" , fun x -> [Tok_Int (int_of_string x)]) ;
  (R.regexp "=" , fun _ -> [Tok_Eq]) ;
  (R.regexp "{" , fun _ -> [Tok_LCurly]) ;
  (R.regexp "}" , fun _ -> [Tok_RCurly]) ;
  (R.regexp ";" , fun _ -> [Tok_Semi]) ;
  (R.regexp " " , fun _ -> [])
]

(* Given source code returns a token list. *)
let rec lexer (s : string) : token list =
  lexer' s 0

(* Helper for lexer takes in a position offset. *)
and lexer' (s : string) (pos : int) : token list =
  if pos >= S.length s then [Tok_EOF]
  else
    let (_, f) = L.find (fun (re, _) -> R.string_match re s pos) re in
    let s' = R.matched_string s in
    (f s') @ (lexer' s (pos + (S.length s')))

(* Returns a string representation of a token list. *)
let rec string_of_tokens (ts : token list) : string =
  S.concat "" (L.map string_of_token ts)

(* Returns string representationof a single token. *)
and string_of_token (t : token) : string =
  match t with
```

```
  | Tok_Var x -> x
  | Tok_Int n -> string_of_int n
  | Tok_Eq -> "="
  | Tok_LCurly -> "{"
  | Tok_RCurly -> "}"
  | Tok_Semi -> ";"
  | Tok_EOF -> ""
```

And now, parser.ml:

```
open Lexer

(* Types *)

type var = string
type expr =
| Assign of var * int
| Seq of expr * expr
| Skip

(* Parsing helpers *)

let tok_list = ref []

(* Returns next token in the list. *)
let lookahead () : token =
  match !tok_list with
  | [] -> raise (Failure "no tokens")
  | h :: t -> h

(* Matches the top token in the list. *)
let match_tok (a : token) : unit =
  match !tok_list with
  | h :: t when a = h -> tok_list := t
  | _ -> raise (Failure "bad match")

(* Parses a token list. *)
let rec parser (toks : token list) : expr =
  tok_list := toks;
  let exp = parse_E () in
  if !tok_list <> [Tok_EOF] then
    raise (Failure "did not reach EOF")
  else
    exp

(* Parses the E rule. *)
and parse_E () : expr =
```

```
  match lookahead () with
  | Tok_Var id ->
      (match_tok (Tok_Var id);
       match_tok Tok_Eq;
       match lookahead () with
       | Tok_Int n -> match_tok (Tok_Int n);
                      Assign (id, n)
       | _ -> raise (Failure "parse_E failure"))
  | Tok_LCurly ->
      (match_tok Tok_LCurly;
       let e = parse_L () in
       match_tok Tok_RCurly;
       e)
  | _ -> raise (Failure "parse_E failure")

(* Parses the L rule. *)
and parse_L () : expr =
  match lookahead () with
  | Tok_Var _ | Tok_LCurly ->
      let m = parse_E () in
      match_tok Tok_Semi;
      let n = parse_L () in
      Seq (m, n)
  | _ -> Skip

(* Returns string representation of the AST. *)
let rec string_of_expr (m : expr) : string =
  match m with
  | Assign (id, n) -> id ^ " = " ^ (string_of_int n)
  | Seq (m, n) -> "{" ^ (string_of_expr m) ^ ";" ^ (string_of_expr n) ^ "}"
  | Skip -> ""
```