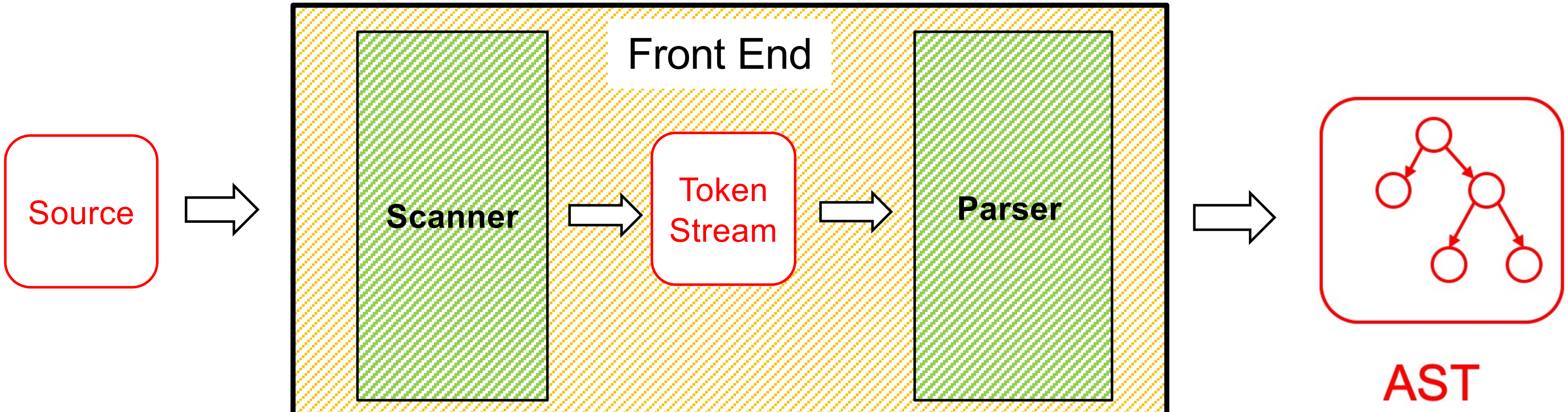
CMSC 330: Organization of Programming Languages

Parsing

Recall: Front End Scanner and Parser



- Scanner / lexer / tokenizer converts program source into tokens (keywords, variable names, operators, numbers, etc.) with regular expressions
- Parser converts tokens into an AST (abstract syntax tree) based on a context free grammar

Scanning (“tokenizing”)

- Converts textual input into a stream of tokens
  - These are the terminals in the parser’s CFG
  - Example tokens are keywords, identifiers, numbers, punctuation, etc.
- Scanner typically ignores/eliminates whitespace

Scanning (“tokenizing”)

```
type token =
   Tok_Num of char
 | Tok_Sum
```

```
tokenize "1+2" =
  [Tok_Num '1'; Tok_Sum; Tok_Num '2']
```

A Scanner in OCaml

```
type token =
   Tok_Num of char
 | Tok_Sum

let tokenize (s:string) = (* returns token list *)
```

```
let re_num = Str.regexp "[0-9]" (* single digit *)
let re_add = Str.regexp "+"
let tokenize str =
 let rec tok pos s =
   if pos >= String.length s then
     []
   else
     if (Str.string_match re_num s pos) then
       let token = Str.matched_string s in
         (Tok_Num token.[0])::(tok (pos+1) s)
     else if (Str.string_match re_add s pos) then
       Tok_Sum::(tok (pos+1) s)
     else
       raise (IllegalExpression "tokenize")
 in
 tok 0 str
```

Uses **`Str`** library module for regexps

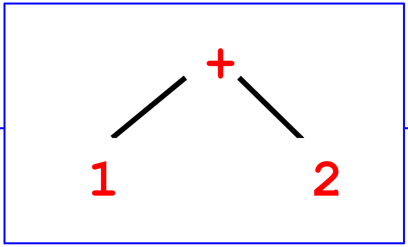
Parsing (to an AST)

```
type token =
   Tok_Num of char
 | Tok_Sum
```

```
type expr =
  Num of int
| Sum of expr * expr
```

```
let tokens = tokenize "1+2" in
(* tokens = [Tok_Num '1’; Tok_Sum; Tok_Num '2'] *)

parse tokens
  = Sum (Num 1, Num 2)
```



Implementing Parsers

- Many efficient techniques for parsing
  - LL(k), SLR(k), LR(k), LALR(k)…
  - Take CMSC 430 for more details
- One simple technique: recursive descent parsing
  - This is a top-down parsing algorithm
- Other algorithms are bottom-up

Top-Down Parsing (Intuition)

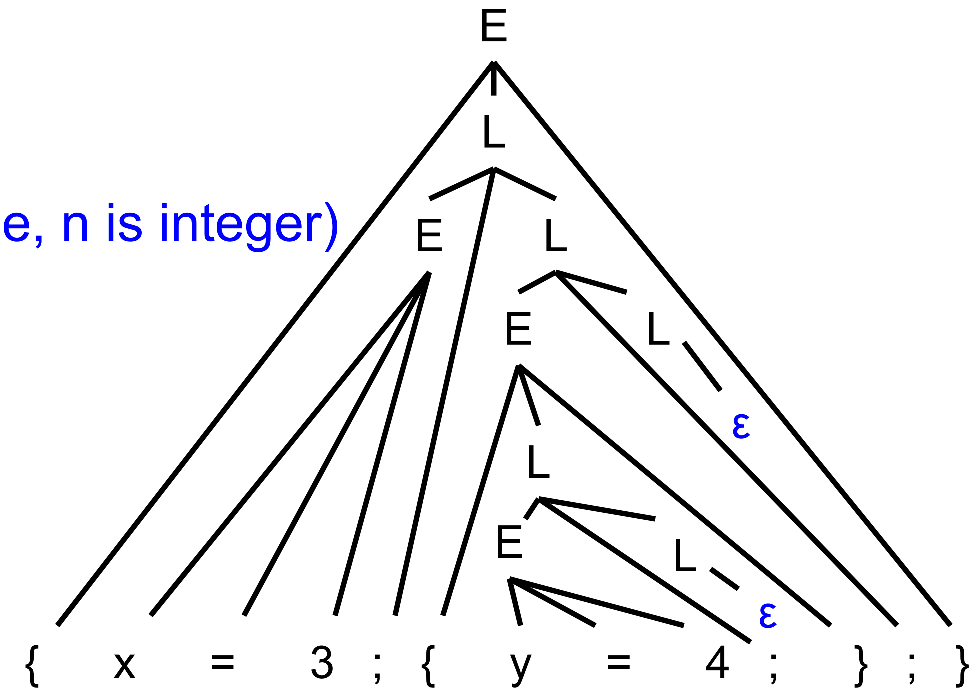
E → id = n | { L }

L → E ; L | ε

(Assume: id is variable name, n is integer)

Show parse tree for

{ x = 3 ; { y = 4 ; } ; }



Recursive Descent Parsing

- Goal
  - Can we "parse" a string – does it match our grammar?
    - We will talk about constructing an AST later

- Approach: Try to produce leftmost derivation

```
Begin with start symbol S, and input tokens t
Repeat:
  Rewrite S and consume tokens in t via a production in the grammar
Until all tokens matched, or failure
```

Recursive Descent Parsing

- At each step, we keep track of two facts
  - What grammar element are we trying to match/expand?
  - What is the lookahead (next token of the input string)?
- At each step, apply one of three possible cases
  - If we're trying to match a terminal
    - If the lookahead is that token, then succeed, advance the lookahead, and continue
  - If we' re trying to match a nonterminal
    - Pick which production to apply based on the lookahead
  - Otherwise fail with a parsing error

Example

E → id = n | { L }

L → E ; L | ε

- Here n is an integer and id is an identifier

- One input might be
  - { x = 3; { y = 4; }; }
  - This would get turned into a list of tokens

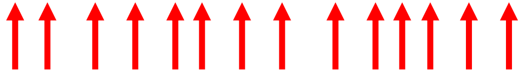
    { x = 3 ; { y = 4 ; } ; }
  - And we want to parse it
    - i.e., just determine if it's in the grammar's language; no AST for now

Parsing Example Input

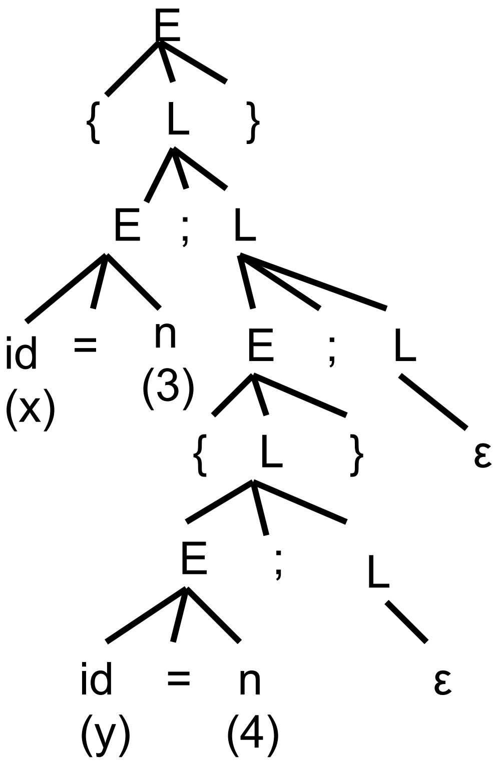
E → id = n | { L }

L → E ; L | ε

{ x = 3 ; { y = 4 ; } ; }



lookahead



Parsing Example: Previewing the Code

E → id = n | { L }
L → E ; L | ε

```
type token = Tok_Num (* of int *)
           | Tok_Id (* of string *)
           | Tok_Eq | Tok_Semi
           | Tok_Lbrace
           | Tok_Rbrace
```

```
let rec parse_E () =
  match lookahead () with
  | Some Tok_Id ->
    (* E → id = n *)
    (match_tok Tok_Id;
     match_tok Tok_Eq;
     match_tok Tok_Num)
  | Some Tok_Lbrace ->
    (* E → { L } *)
    (match_tok Tok_Lbrace;
     parse_L ();
     match_tok Tok_Rbrace)
  | _ -> raise (ParseError "parse_A")
```

```
and parse_L () =
  match lookahead () with
  | Some Tok_Id | Some Tok_Lbrace ->
    (* L → E ; L *)
    (parse_E ();
     match_tok Tok_Semi;
     parse_L ())
  | _ ->
    (* L → ε *)
    ()
```

Parsing Example: Previewing the Code

$E \rightarrow id = n \mid \{ L \}$
$L \rightarrow E \; ; L \mid \varepsilon$

```
type token = Tok_Num (* of int *)
           | Tok_Id (* of string *)
           | Tok_Eq | Tok_Semi
           | Tok_Lbrace
           | Tok_Rbrace
```

```
let rec parse_E () = …
and parse_L () = …
```

```
tok_list := tokenize "{ x = 3 ; { y = 4 ; } ; }";;
  (* tok_list := [ Tok_Lbrace; Tok_Id; Tok_Eq; Tok_Num; Tok_Semi; …] *)
parse_E ();;
  (* returns () -- successfully parses input *)

tok_list := tokenize "{ x = ; }";;
  (* tok_list := [ Tok_Lbrace; Tok_Id; Tok_Eq; Tok_Semi; Tok_Rbrace ] *)
parse_E ();;
  (* raises exception ParseError "bad match" *)
```

Recursive Descent Parsing: Key Step

- Key step: Choosing the right production
- Two approaches
  - Backtracking
    - Choose some production
    - If fails, try different production
    - Parse fails if all choices fail
  - Predictive parsing (what we will do)
    - Analyze grammar to find FIRST sets for productions
    - Compare with lookahead to decide which production to select
    - Parse fails if lookahead does not match FIRST

Selecting a Production

- Motivating example
  - If grammar S → xyz | abc and lookahead is x
    - Select S → xyz since 1st terminal in RHS matches x
  - If grammar S → A | B   A → x | y   B → z
    - If lookahead is x, select S → A, since A can derive string beginning with x
- In general
  - Choose a production that can derive a sentential form beginning with the lookahead
  - Need to know what terminal may be first in any sentential form derived from a nonterminal / production

First Sets

- Definition
  - First(γ), for any terminal or nonterminal γ, is the set of initial terminals of all strings that γ may expand to
  - We'll use this to decide which production to apply
- Example: Given grammar

  S → A | B

  A → x | y

  B → z

  - First(A) = { x, y } since First(x) = { x }, First(y) = { y }
  - First(B) = { z } since First(z) = { z }
- So: If we are parsing S and see x or y, we choose S → A; if we see z we choose S → B

Calculating First(γ)

- For a terminal a
  - First(a) = { a }
- For a nonterminal N
  - If $N \rightarrow \varepsilon$, then add ε to First(N)
  - If $N \rightarrow \alpha_1\ \alpha_2\ \ldots\ \alpha_n$, then (note the α_i are all the symbols on the right side of one single production):
    - Add First($\alpha_1\alpha_2 \ldots \alpha_n$) to First(N), where First($\alpha_1\ \alpha_2 \ldots \alpha_n$) is defined as
      - First(α_1) if $\varepsilon \notin$ First(α_1)
      - Otherwise (First(α_1) – ε) $\cup$ First($\alpha_2 \ldots \alpha_n$)
    - If $\varepsilon \in$ First(α_i) for all i, $1 \leq i \leq k$, then add ε to First(N)

First( ) Examples

E → id = n | { L }
L → E ; L | ε

First(id) = { id }
First("=") = { "=" }
First(n) = { n }
First("{")= { "{" }
First("}")= { "}" }
First(";")= { ";" }
First(E) = { id, "{" }
First(L) = { id, "{", ε }

E → id = n | { L } | ε
L → E ; L

First(id) = { id }
First("=") = { "=" }
First(n) = { n }
First("{")= { "{" }
First("}")= { "}" }
First(";")= { ";" }
First(E) = { id, "{", ε }
First(L) = { id, "{", ";" }

Quiz #1

Given the following grammar:

```
S -> aAB |B
A -> CBC
B -> b
C -> cC | ε
```

What is First(S)?

A. {b,c}

B. {b}

C. {a,b}

D. {c}

Quiz #1

Given the following grammar:

```
S -> aAB |B
A -> CBC
B -> b
C -> cC | ε
```

What is First(S)?

- A. {b,c}
- B. {b}
- **C. {a,b}**
- D. {c}

Quiz #2

Given the following grammar:

```
S  ->  aAB
A  ->  CBC
B  ->  b
C  ->  cC  |  ε
```

What is First(B)?

A. {a}

B. {b,c}

C. {b}

D. {c}

Quiz #2

Given the following grammar:

```
S -> aAB
A -> CBC
B -> b
C -> cC | ε
```

What is First(B)?

A. {a}

B. {b,c}

C. {b}

D. {c}

Quiz #3

Given the following grammar:

```
S -> aAB
A -> CBC
B -> b
C -> cC | ε
```

What is First(A)?

A. {a}

B. {b,c}

C. {b}

D. {c}

Quiz #3

Given the following grammar:

```
S -> aAB
A -> CBC
B -> b
C -> cC | ε
```

What is First(A)?

A. `{a}`

B. `{b,c}`

C. `{b}`

D. `{c}`

Note:

First(B) = **`{b}`**

First(C) = **`{c,ε}`**

Recursive Descent Parser Implementation

- For all terminals, use function match_tok a
  - If lookahead is a it consumes the lookahead by advancing the lookahead to the next token, and returns
  - Fails with a parse error if lookahead is not a
- For each nonterminal N, create a function parse_N
  - Called when we're trying to parse a part of the input which corresponds to (or can be derived from) N
  - parse_S for the start symbol S begins the parse

match_tok, lookahead in OCaml

```
let tok_list = ref [] (* list of parsed tokens *)

exception ParseError of string

let match_tok a =
  match !tok_list with
    (* checks current token; advances on match *)
    | (h::t) when a = h -> tok_list := t
    | _ -> raise (ParseError "bad match")

(* used by parse_X *)
let lookahead () =
  match !tok_list with
    [] -> None
  | (h::t) -> Some h
```

Parsing Nonterminals

- The body of parse_N for a nonterminal N does the following
  - Let $N \rightarrow \beta_1 \mid \ldots \mid \beta_k$ be the productions of N
    - Here β_i is the entire right side of a production- a sequence of terminals and nonterminals
  - Pick the production $N \rightarrow \beta_i$ such that the lookahead is in First(β_i)
    - It must be that $\text{First}(\beta_i) \cap \text{First}(\beta_j) = \varnothing$ for $i \neq j$
    - If there is no such production, but $N \rightarrow \varepsilon$ then return
    - Otherwise fail with a parse error
  - Suppose $\beta_i = \alpha_1\ \alpha_2 \ldots \alpha_n$. Then call parse_$\alpha_1$(); ... ; parse_$\alpha_n$() to match the expected right-hand side, and return

Example Parser

- Given grammar S → xyz | abc
  - First(xyz) = { x }, First(abc) = { a }
- Parser

```
let parse_S () =
  if lookahead () = Some "x" then (* S → xyz *)
     (match_tok "x";
      match_tok "y";
      match_tok "z")
   else if lookahead () = Some "a" then (* S → abc *)
      (match_tok "a";
      match_tok "b";
      match_tok "c")
   else raise (ParseError "parse_S")
```

Note: We are not producing an AST here; we are only determining if the string is in the language. We'll produce an AST later.

Another Example Parser

- Given grammar S → A | B   A → x | y   B → z
  - First(A) = { x, y }, First(B) = { z }
- Parser:

```
let rec parse_S () =
  if lookahead () = Some "x" ||
     lookahead () = Some "y" then
    parse_A () (* S → A *)
  else if lookahead () = Some "z" then
    parse_B () (* S → B *)
  else raise (ParseError "parse_S")
and parse_A () =
  if lookahead () = Some "x" then
    match_tok "x" (* A → x *)
  else if lookahead () = Some "y" then
    match_tok "y" (* A → y *)
  else raise (ParseError "parse_A")
and parse_B () = …
```

Syntax for *mutually recursive* functions in OCaml – **`parse_S`** and **`parse_A`** and **`parse_B`** can each call the other

Execution Trace = Parse Tree

- If you draw the execution trace of the parser
  - You get the parse tree
- Examples
  - Grammar

    S → xyz

    S → abc
  - String “xyz”

    parse_S ()
      match_tok “x”
      match_tok “y”
      match_tok “z”

```
  S
 /|\
x y z
```

  - Grammar

    S → A | B

    A → x | y

    B → z
  - String “x”

    parse_S ()
      parse_A ()
        match_tok “x”

```
S
|
A
|
x
```

Predictive Parsing

- This is a predictive parser
  - Because the lookahead determines exactly which production to use
- This parsing strategy may fail on some grammars
  - Production First sets overlap
  - Production First sets contain ε
  - Possible infinite recursion
- Does not mean grammar is not usable
  - Just means this parsing method not powerful enough
  - May be able to change grammar

Conflicting First Sets

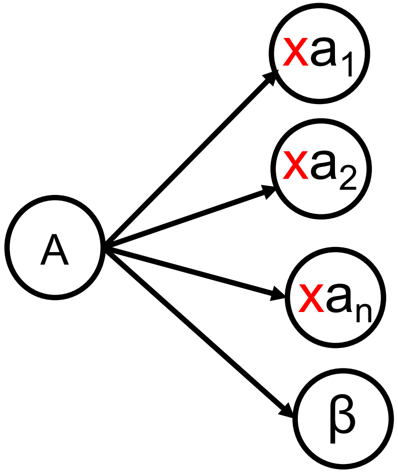
- Consider parsing the grammar $E \rightarrow ab \mid ac$
  - $\text{First}(ab) = a$
  - $\text{First}(ac) = a$

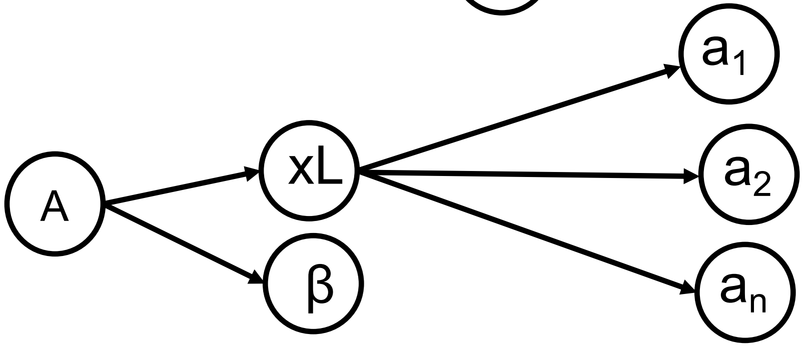
Parser cannot choose between RHS based on lookahead!

- Parser fails whenever $A \rightarrow \alpha_1 \mid \alpha_2$ and
  - $\text{First}(\alpha_1) \cap \text{First}(\alpha_2) \mathrel{!=} \varepsilon$ or $\emptyset$
- Solution
  - Rewrite grammar using left factoring

Left Factoring Algorithm

- Given grammar
  - $A \rightarrow x\alpha_1 \mid x\alpha_2 \mid \ldots \mid x\alpha_n \mid \beta$
- Rewrite grammar as
  - $A \rightarrow xL \mid \beta$
  - $L \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
- Repeat as necessary





Left Factoring Algorithm

- Given grammar
  - $A \rightarrow x\alpha_1 \mid x\alpha_2 \mid \ldots \mid x\alpha_n \mid \beta$
- Rewrite grammar as
  - $A \rightarrow xL \mid \beta$
  - $L \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
- Examples
  - $S \rightarrow ab \mid ac$ ⇨ $S \rightarrow aL$ $L \rightarrow b \mid c$
  - $S \rightarrow abcA \mid abB \mid a$ ⇨ $S \rightarrow aL$ $L \rightarrow bcA \mid bB \mid \varepsilon$
  - $L \rightarrow bcA \mid bB \mid \varepsilon$ ⇨ $L \rightarrow bL' \mid \varepsilon$ $L' \rightarrow cA \mid B$

Alternative Approach

- Change structure of parser
  - First match common prefix of productions
  - Then use lookahead to chose between productions
- Example
  - Consider parsing the grammar E → a+b | a*b | a

```
let parse_E () =
  match_tok "a"; (* common prefix *)

  if lookahead () = Some "+" then (* E → a+b *)
     (match_tok "+";
      match_tok "b")

  else if lookahead () = Some "*" then (* E → a*b *)
     (match_tok "*";
      match_tok "b")

  else () (* E → a *)
```

Left Recursion

- Consider grammar S → Sa | ε
  - Try writing parser

```
let rec parse_S () =
    if lookahead () = Some “a” then
      (parse_S ();
     match_tok “a”) (* S → Sa *)
    else ()
```

  - Body of parse_S () has an infinite loop!
    - Infinite loop occurs in grammar with left recursion

Right Recursion

- Consider grammar S → aS | ε   Again, First(aS) = a
  - Try writing parser

```
let rec parse_S () =
    if lookahead () = Some “a” then
      (match_tok “a”;
      parse_S ())  (* S → aS *)
    else ()
```

  - Will parse_S( ) infinite loop?
    - Invoking match_tok will advance lookahead, eventually stop
  - Top-down parsers handles grammar w/ right recursion

Algorithm To Eliminate Left Recursion

- Given grammar
  - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta$
    - β must exist or no derivation will yield a string
- Rewrite grammar as (repeat as needed)
  - $A \rightarrow \beta L$
  - $L \rightarrow \alpha_1 L \mid \alpha_2 L \mid \ldots \mid \alpha_n L \mid \varepsilon$
- Replaces left recursion with right recursion
- Examples
  - $S \rightarrow Sa \mid \varepsilon$ ⇨ $S \rightarrow L$ $\quad L \rightarrow aL \mid \varepsilon$
  - $S \rightarrow Sa \mid Sb \mid c$ ⇨ $S \rightarrow cL$ $\quad L \rightarrow aL \mid bL \mid \varepsilon$

Quiz #4

- What does the following code parse?

```
let parse_S () =
  if lookahead () = Some “a” then
    (match_tok "a";
     match_tok "x";
     match_tok "y”;
     match_tok "q”)
  else
    raise (ParseError "parse_S")
```

A. S → axyq
B. S → a | q
C. S → aaxy | qq
D. S → axy | q

Quiz #4

- What does the following code parse?

```
let parse_S () =
  if lookahead () = Some “a” then
    (match_tok "a";
     match_tok "x";
     match_tok "y”;
     match_tok "q”)
  else
    raise (ParseError "parse_S")
```

A. S → axyq
B. S → a | q
C. S → aaxy | qq
D. S → axy | q

Quiz #5

- What does the following code parse?

```
let rec parse_S () =
  if lookahead () = Some “a” then
    (match_tok "a";
     parse_S ())
  else if lookahead () = Some “q” then
    (match_tok "q”;
     match_tok ”p”)
  else
    raise (ParseError "parse_S")
```

A. $S \rightarrow aS \mid qp$
B. $S \rightarrow a \mid S \mid qp$
C. $S \rightarrow aqSp$
D. $S \rightarrow a \mid q$

Quiz #5

- What does the following code parse?

```
let rec parse_S () =
  if lookahead () = Some “a” then
    (match_tok "a";
     parse_S ())
  else if lookahead () = Some “q” then
    (match_tok "q”;
     match_tok ”p”)
  else
    raise (ParseError "parse_S")
```

A. S → aS | qp

B. S → a | S | qp

C. S → aqSp

D. S → a | q

Quiz #6

Can recursive descent parse this grammar?

$S \rightarrow aBa$
$B \rightarrow bC$
$C \rightarrow \varepsilon \mid Cc$

A. Yes
B. No

Quiz #6

Can recursive descent parse this grammar?

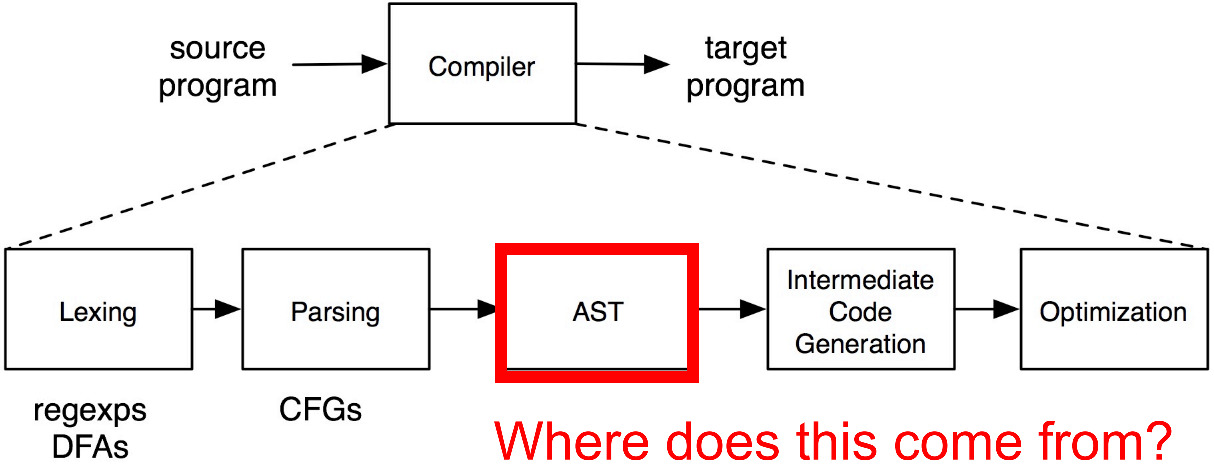
$S \rightarrow aBa$
$B \rightarrow bC$
$C \rightarrow \varepsilon \mid Cc$

A. Yes

B. No

(due to left recursion)

Recall: The Compilation Process


source program

Compiler

target program

Lexing

Parsing

AST

Intermediate Code Generation

Optimization

regexps
DFAs

CFGs

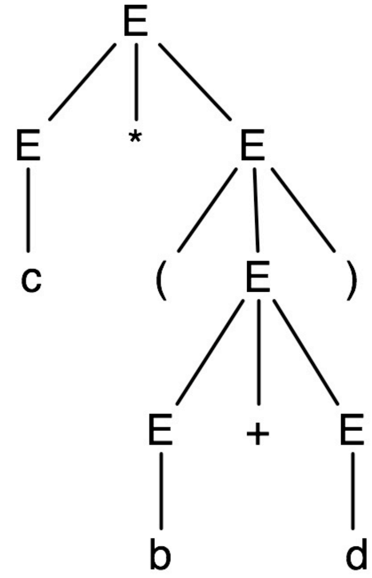
Where does this come from?

Parse Trees to ASTs

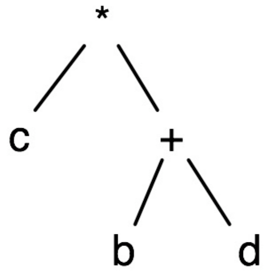
- Parse trees are a representation of a parse, with all of the syntactic elements present
  - Parentheses
  - Extra nonterminals for precedence
- This extra stuff is needed for parsing
- Lots of that stuff is not needed to actually implement a compiler or interpreter
  - So in the abstract syntax tree we get rid of it

Abstract Syntax Trees (ASTs)

- An abstract syntax tree is a more compact, abstract representation of a parse tree, with only the essential parts



parse tree



AST

Example: Simple Assignment

E → *id* = *n* | { L }
L → E ; L | ε

```
type token = Tok_Num (* of string *)
           | Tok_Id (* of string *)
           | Tok_Eq | Tok_Semi
           | Tok_Lbrace
           | Tok_Rbrace
```

- Here, *id* stands for a general identifier (variable), like **`a`**, **`bob`**, **`chandra`**, **`toy`**, etc.
  - The scanner will match this via a regular expression, and can track of what the actual string was; we'll ignore here
- Similar situation for *n*, which represents an integer

Matching Strings; no AST

E → *id* = *n* | { L }
L → E ; L | ε

```
type token = Tok_Num (* of string *)
           | Tok_Id (* of string *)
           | Tok_Eq | Tok_Semi
           | Tok_Lbrace
           | Tok_Rbrace
```

```
let rec parse_E () = (* First(E) = { id, "{"  } *)
  match lookahead () with
   | Some Tok_Id ->
    (* E → id = n *)
    (match_tok Tok_Id;
     match_tok Tok_Eq;
     match_tok Tok_Num)
   | Some Tok_Lbrace ->
    (* E → { L } *)
    (match_tok Tok_Lbrace;
     parse_L ();
     match_tok Tok_Rbrace)
   | _ -> raise (ParseError "parse_A")
```

```
and parse_L () =
  match lookahead () with
   | Some Tok_Id
   | Some Tok_Lbrace ->
    (* L → E ; L *)
    (parse_E ();
     match_tok Tok_Semi;
     parse_L ())
   | _ ->
    (* L → ε *)
    ()
```

Defining the AST

E → *id* = *n* | { L }
L → E ; L | ε

```
let match_tok a : string option =
 match !tok_list,a with
   | (Tok_Id s)::t,(Tok_Id _) ->
     tok_list := t; (Some s)
   | (Tok_Num s)::t,(Tok_Num _) ->
     tok_list := t; (Some s)
   | h::t, _ ->
     if h = a then
       (tok_list := t; None)
     else
       raise (ParseError "bad match")
   | _ -> raise (ParseError "no tokens")
```

```
type token = Tok_Num of string
           | Tok_Id of string
           | Tok_Eq | Tok_Semi
           | Tok_Lbrace
           | Tok_Rbrace
```

```
type stmt =
   Assign of string * int
 | Block of stmt list
```

- The AST is just a sequence of assignment statements
- Match_tok now returns the string that was matched for Tok_Num and Tok_Id

Parsing, producing AST

E → *id* = *n* | { L }
L → E ; L | ε

```
let rec parse_E () : stmt =

  match lookahead () with
    Some (Tok_Id _) ->
      (let Some v = match_tok (Tok_Id "") in
      match_tok Tok_Eq;
      let Some n = match_tok (Tok_Num "") in
      Assign (v, int_of_string n))

  | Some Tok_Lbrace ->
      (match_tok Tok_Lbrace;
      let stms = parse_L () in
      match_tok Tok_Rbrace;
      Block stms)

  | _ -> raise (ParseError "parse_A")
```

```
type token = Tok_Num of string
           | Tok_Id of string
           | Tok_Eq | Tok_Semi
           | Tok_Lbrace
           | Tok_Rbrace

type stmt =
  Assign of string * int
| Block of stmt list


and parse_L () : stmt list =

  match lookahead () with
  | Some (Tok_Id _)
  | Some Tok_Lbrace ->
      (let stm = parse_E () in
      match_tok Tok_Semi;
      let stms = parse_L () in
      stm :: stms)

  | _ -> []
```