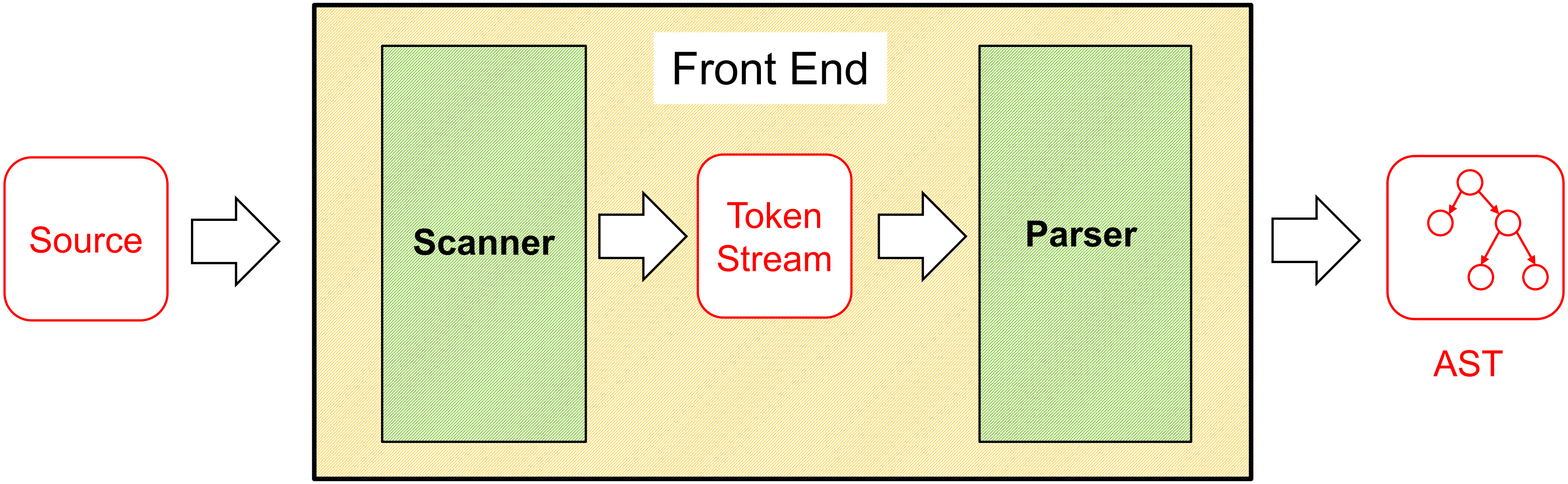
CMSC 330: Organization of Programming Languages

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) using context free grammars

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.
- Tokens determined with regular expressions
  - Identifiers match regexp [a-zA-Z_][a-zA-Z0-9_]*
- Simplest case: a token is just a string
  - **`type token = string`**
  - But representation might be more full featured
- Scanner typically ignores/eliminates whitespace

Simple Scanner in OCaml

```
type token = string

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

```
tokenize “this is a string” =
  [“this”; “is”; “a”; “string”]
```

```
let tokenize s =
  let l = String.length s in
  let rec tok sidx slen =
    if sidx >= l then ("",sidx)
    else if String.get s sidx = ' ' then
      tok (sidx+1) 1
    else if (sidx+slen) >= l then
      (String.sub s sidx slen,l)
    else if String.get s (sidx+slen) = ' ' then
      (String.sub s sidx slen, sidx+slen)
    else
      tok sidx (slen+1) in
  let rec alltoks idx =
    let (t,idx') = tok idx 1 in
    if t = "" then []
    else t::alltoks idx' in
  alltoks 0
```

More Interesting Scanner

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

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

```
tokenize “1+2” =
  [Tok_Num '1';
   Tok_Sum;
   Tok_Num '2';
   Tok_END]
```

```
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
     [Tok_END]
   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

Implementing Parsers

- Many efficient techniques for parsing
  - I.e., for turning strings into parse trees
  - Examples
    - 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

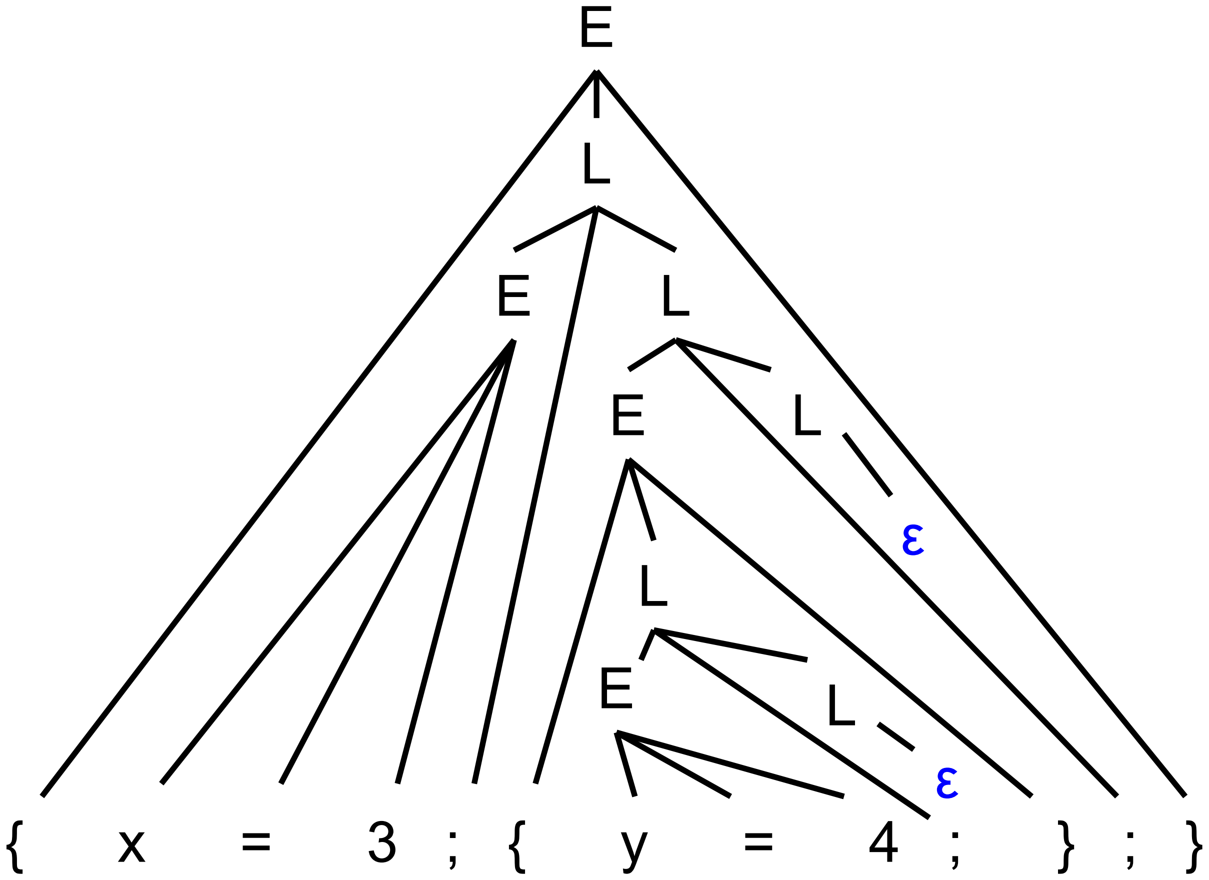
E → id = n | { L }

L → E ; L | ε

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

Show parse tree for

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

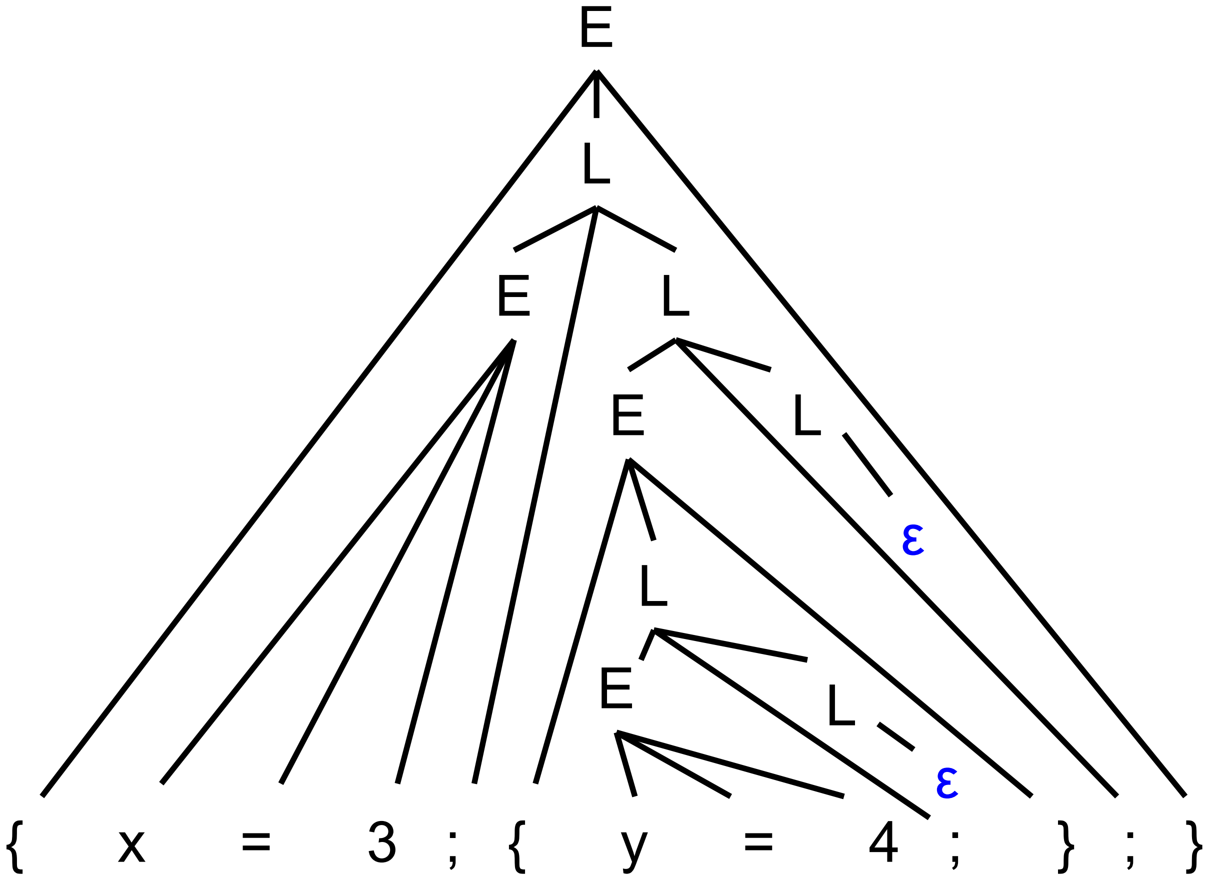


Bottom-up Parsing

E → id = n | { L }
L → E ; L | ε

Show parse tree for
{ x = 3 ; { y = 4 ; } ; }

Note that final trees constructed are same as for top-down; only order in which nodes are added to tree is different



Example: Shift-Reduce Parsing

- Replaces RHS of production with LHS (nonterminal)
- Example grammar
  - $S \rightarrow aA$, $A \rightarrow Bc$, $B \rightarrow b$
- Example parse
  - $abc \Rightarrow aBc \Rightarrow aA \Rightarrow S$
  - Derivation happens in reverse
- Something to look forward to in CMSC 430
- Complicated to use; requires tool support
  - Bison, yacc produce shift-reduce parsers from CFGs

Tradeoffs

- Recursive descent parsers
  - Easy to write
    - The formal definition is a little clunky, but if you follow the code then it's almost what you might have done if you weren't told about grammars formally
  - Fast
    - Can be implemented with a simple table
- Shift-reduce parsers handle more grammars
  - Error messages may be confusing
- Most languages use hacked parsers (!)
  - Strange combination of the two

Recursive Descent Parsing

- Goal
  - Determine if we can produce the string to be parsed from the grammar's start symbol
- Approach
  - Recursively replace nonterminal with RHS of production
- At each step, we'll keep track of two facts
  - What tree node are we trying to match?
  - What is the lookahead (next token of the input string)?
    - Helps guide selection of production used to replace nonterminal

Recursive Descent Parsing (cont.)

- At each step, 3 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

Parsing 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

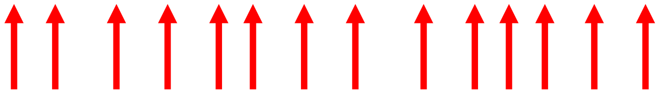
    { x = 3 ; { y = 4 ; } ; }
  - And we want to turn it into a parse tree

Parsing Example (cont.)

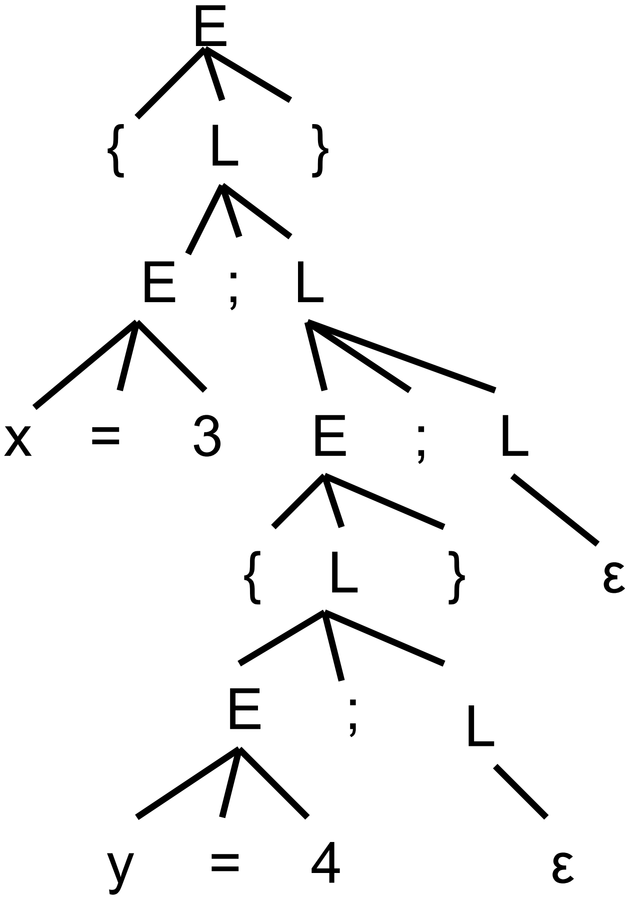
E → id = n | { L }

L → E ; L | ε

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



lookahead



Recursive Descent Parsing (cont.)

- Key step
  - Choosing which production should be selected
- Two approaches
  - Backtracking
    - Choose some production
    - If fails, try different production
    - Parse fails if all choices fail
  - Predictive parsing
    - 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

First Sets

- Motivating example
  - The lookahead is x
  - Given grammar S → xyz | abc
    - Select S → xyz since 1st terminal in RHS matches x
  - Given grammar S → A | B   A → x | y   B → z
    - 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 what production to apply
- Examples
  - Given grammar S → xyz | abc
    - First(xyz) = { x }, First(abc) = { a }
    - First(S) = First(xyz) U First(abc) = { x, a }
  - Given grammar S → A | B   A → x | y   B → z
    - First(x) = { x }, First(y) = { y }, First(A) = { x, y }
    - First(z) = { z }, First(B) = { z }
    - First(S) = { x, y, z }

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
A -> CBC
B -> b
C -> cC | ε
```

What is First(S)?

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

Quiz #1

Given the following grammar:

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

What is First(S)?

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 #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}`

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 in OCaml

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

exception ParseError of string

let match_tok a =
  match !tok_list with
    (* checks lookahead; 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
    [] -> raise (ParseError ”no tokens")
  | (h::t) -> 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 First(β_i) $\cap$ First(β_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 () = "x" then (* S → xyz *)
    (match_tok "x";
     match_tok "y";
     match_tok "z")
   else if lookahead () = "a" then (* S → abc *)
     (match_tok "a";
      match_tok "b";
      match_tok "c")
   else raise (ParseError "parse_S")
```

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 () = "x" ||
     lookahead () = "y" then
    parse_A () (* S → A *)
  else if lookahead () = "z" then
    parse_B () (* S → B *)
  else raise (ParseError "parse_S")

and parse_A () =
  if lookahead () = "x" then
    match_tok "x" (* A → x *)
  else if lookahead () = "y" then
    match_tok "y" (* A → y *)
  else raise (ParseError "parse_A")

and parse_B () = …
```

Example

E → id = n | { L }

L → E ; L | ε

First(E) = { id, "{" }

Parser:

```
let rec parse_E () =
  if lookahead () = "id" then
    (* E → id = n *)
    (match_tok "id";
     match_tok "=";
     match_tok "n")
   else if lookahead () = "{" then
    (* E → { L } *)
    (match_tok "{";
     parse_L ();
     match_tok "}")
  else raise (ParseError "parse_A")
```

```
and parse_L () =
  if lookahead () = "id"
  || lookahead () = "{" then
    (* L → E ; L *)
    (parse_E ();
     match_tok ";";
     parse_L ())
  else
    (* L → ε *)
    ()
```

Things to Notice

- 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
```

Things to Notice (cont.)

- 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 $E \rightarrow ab \mid ac$
  - First(ab) = a
  - 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 $\varnothing$
- Solution
  - Rewrite grammar using left factoring

Left Factoring Algorithm

- Given grammar
  - $A \rightarrow x\alpha_1 \mid x\alpha_2 \mid \dots \mid x\alpha_n \mid \beta$
- Rewrite grammar as
  - $A \rightarrow xL \mid \beta$
  - $L \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$
- Repeat as necessary
- 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 () = "+" then (* E → a+b *)
     (match_tok "+";
      match_tok "b")

  else if lookahead () = "*" 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 () = “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 () = “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 derivation will not yield 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
  - $S \rightarrow Sa \mid \varepsilon$ ⇨ $S \rightarrow L$ $L \rightarrow aL \mid \varepsilon$
  - $S \rightarrow Sa \mid Sb \mid c$ ⇨ $S \rightarrow cL$ $L \rightarrow aL \mid bL \mid \varepsilon$

Quiz #4

- What Does the following code parse?

```
let parse_S () =
  if lookahead () = “a” then
    (match_tok "a";
     match_tok "x";
     match_tok "y")
  else if lookahead () = “q” then
    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 () = “a” then
    (match_tok "a";
     match_tok "x";
     match_tok "y")
  else if lookahead () = “q” then
    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 () = “a” then
    (match_tok "a";
     parse_S ())
  else if lookahead () = “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 #5

- What Does the following code parse?

```
let rec parse_S () =
  if lookahead () = “a” then
    (match_tok "a";
     parse_S ())
  else if lookahead () = “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 -> aBa
B -> bC
C -> ε | Cc

A. Yes
B. No

Quiz #6

Can recursive descent parse this grammar?

S -> aBa
B -> bC
C -> ε | Cc

A. Yes
B. No
(due to left recursion)

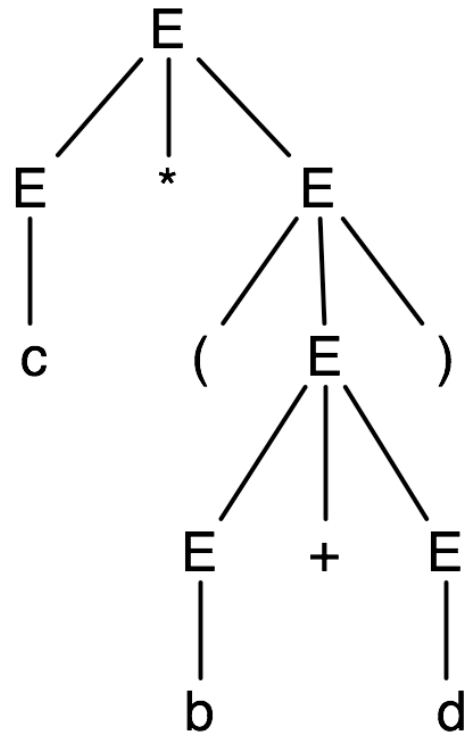
What’s Wrong With Parse Trees?

- Parse trees contain too much information
  - Example
    - Parentheses
    - Extra nonterminals for precedence
  - This extra stuff is needed for parsing

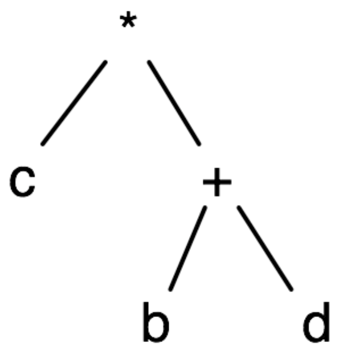
- But when we want to reason about languages
  - Extra information gets in the way (too much detail)

Abstract Syntax Trees (ASTs)

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



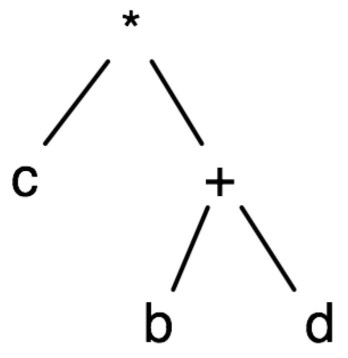
parse tree



AST

Abstract Syntax Trees (cont.)

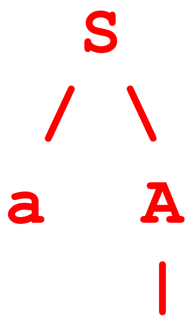
- Intuitively, ASTs correspond to the data structure you'd use to represent strings in the language
  - Note that grammars describe trees
    - So do OCaml datatypes, as we have seen already
  - E → a | b | c | E+E | E-E | E*E | (E)



Producing an AST

- To produce an AST, we can modify the parse() functions to construct the AST along the way
  - match_tok a returns an AST node (leaf) for a
  - parse_A returns an AST node for A
    - AST nodes for RHS of production become children of LHS node
- Example
  - S → aA

```
let rec parse_S () =
      if lookahead () = “a” then
         let n1 = match_tok “a” in
         let n2 = parse_A () in
         Node(n1,n2)
      else raise ParseError “parse_S”
```



The Compilation Process