

CMSC 330: Organization of Programming Languages

Parsing

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

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

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

A Scanner in OCaml

```
type token = Tok_Num of char | Tok_Add | Tok_Mul | Tok_END
```

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

```
let re_num = Str.regexp "[0-9]" (* single digit *)
let re_add = Str.regexp "+"
let re_mul = 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_Add::(tok (pos+1) s)
      else
        raise (IllegalExpression "tokenize")
  in
    tok 0 str
```

Uses **Str**
library module
for regexps

Parsing (to an AST)

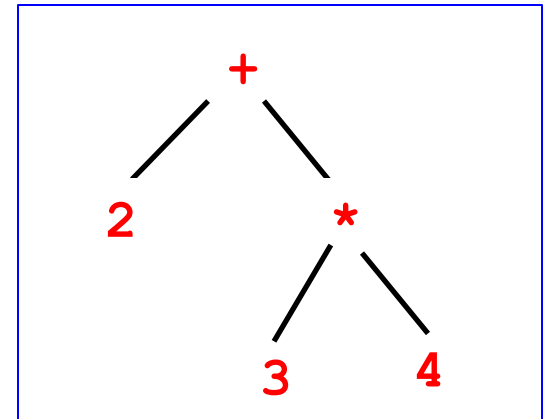
```
type token =  
  Tok_Num of char  
| Tok_Add  
| Tok_Mul  
| Tok_END
```

```
let tokens= tokenize "2+3*4";;
```

```
tokens = [Tok_Num '2'; Tok_Add;  
Tok_Num '3'; Tok_Mul; Tok_Num '4';  
Tok_END]
```

```
type expr =  
  Num of int  
| Add of expr * expr  
| Mult of expr * expr
```

```
parse tokens = Add (Num 2, Mul (Num 3, Num 4))
```

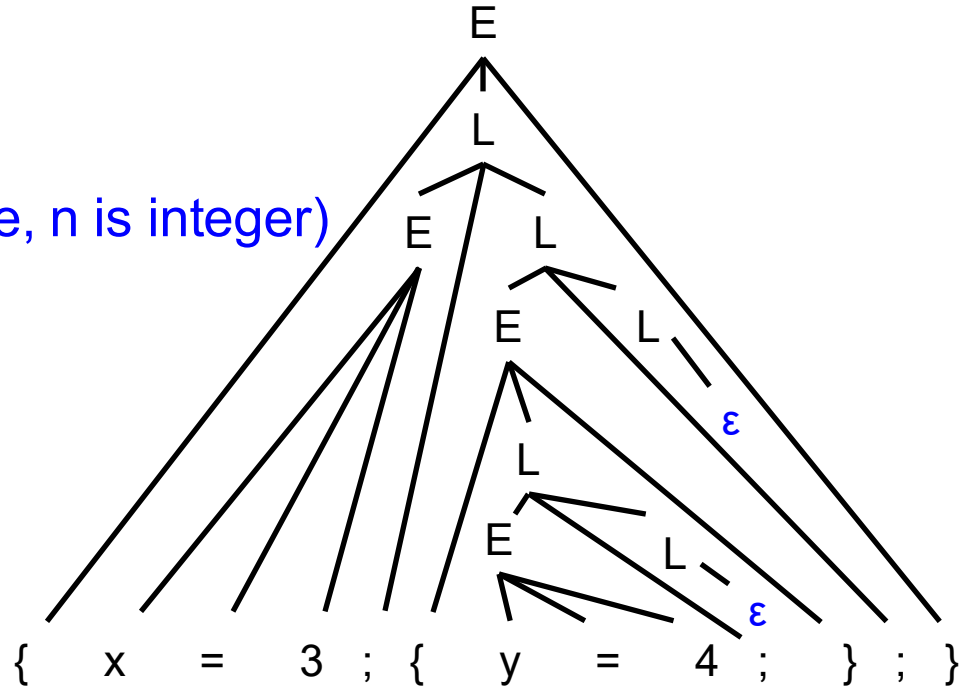


Top-Down Parsing (Intuition)

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

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

Show parse tree for

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


Source code of this example:

https://github.com/anwarmamat/cmssc330spring2024/tree/main/examples/parser_class_example

Recursive Descent Parsing

- 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

Grammar:

$S \rightarrow A + S \mid A$

$A \rightarrow B * A \mid B$

$B \rightarrow 0 \mid 1 \dots \mid 9$

$\Sigma = \{0-9, *, +\}$

Input: $2 + 3 * 4$

Recursive Descent Parsing: Example

Grammar:

$S \rightarrow A + S \mid A$

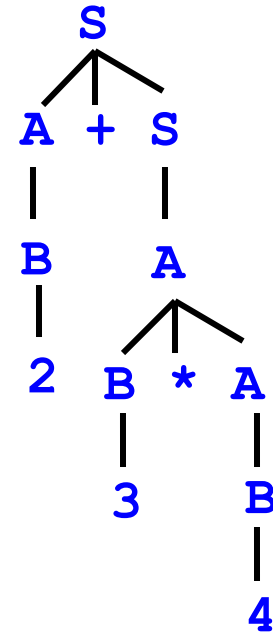
$A \rightarrow B * A \mid B$

$B \rightarrow 0 \mid 1 \mid \dots \mid 9$

Parsing



Input: 2+3*4



Example

```
let lookahead tokens =  
  match tokens with  
  [] -> raise (ParseError "no tokens")  
| (h::t) -> h
```

```
let match_token tokens token =  
  match tokens with  
  | [] -> raise Exception  
  | h :: t when h = token -> t  
  | h :: _ -> raise Exception
```


Quiz 1

- What is the value of x?

```
let tokens =  
    [Tok_Num '2'; Tok_Add; Tok_Num '3'; Tok_END]  
let x = lookahead tokens
```

- A. 2
- B. Tok_Num
- C. Tok_Num '2'
- D. [Tok_Add; Tok_Num '3'; Tok_END]

Quiz 1

- What is the value of x?

```
let tokens =  
    [Tok_Num '2'; Tok_Add; Tok_Num '3'; Tok_END]  
let x = lookahead tokens
```

- A. 2
- B. Tok_Num
- C. Tok_Num '2'
- D. [Tok_Add; Tok_Num '3'; Tok_END]

Quiz 2

- What is the value of x?

```
let tokens =  
    [Tok_Num '2'; Tok_Add; Tok_Num '3'; Tok_END]  
let x = match_token tokens Tok_Add
```

- A. 2
- B. raise Exception
- C. Tok_Num '2'
- D. [Tok_Add; Tok_Num '3'; Tok_END]

Quiz 2

- What is the value of x?

```
let tokens =  
    [Tok_Num '2'; Tok_Add; Tok_Num '3'; Tok_END]  
let x = match_token tokens Tok_Add
```

- A. 2
- B. raise Exception
- C. Tok_Num '2'
- D. [Tok_Add; Tok_Num '3'; Tok_END]

Quiz 3

- What is the value of x?

```
let tokens =  
    [Tok_Num '2'; Tok_Add; Tok_Num '3'; Tok_END]  
let x = match_token tokens (Tok_Num '2')
```

- A. 2
- B. raise Exception
- C. Tok_Num '2'
- D. [Tok_Add; Tok_Num '3'; Tok_END]

Quiz 3

- What is the value of x?

```
let tokens =  
    [Tok_Num '2'; Tok_Add; Tok_Num '3'; Tok_END]  
let x = match_token tokens (Tok_Num '2')
```

- A. 2
- B. raise Exception
- C. Tok_Num '2'
- D. [Tok_Add; Tok_Num '3'; Tok_END]

Example

Grammar:

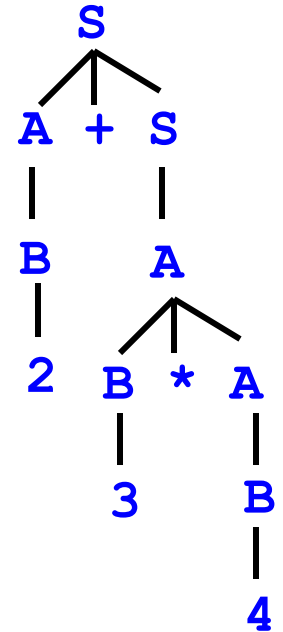
$S \rightarrow A + S \mid A$

$A \rightarrow B * A \mid B$

$B \rightarrow 0 \mid 1 \dots \mid 9$

Input: 2+3*4

```
let rec parse_S tokens =  
  let e1, t1 = parse_A tokens in  
  match lookahead t1 with  
  | Tok_Add -> (* S -> A Tok_Add E *)  
    let t2 = match_token t1 Tok_Add in  
    let e2, t3 = parse_S t2 in  
    (Add (e1, e2), t3)  
  | _ -> (e1, t1) (* S -> A *)
```



Example

Grammar:

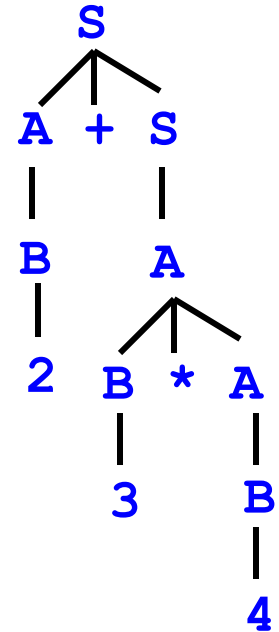
$S \rightarrow A + S \mid A$

$A \rightarrow B * A \mid B$

$B \rightarrow 0 \mid 1 \dots \mid 9$

Input: 2+3*4

```
let rec parse_A tokens =  
  let e1, tokens = parse_B tokens in  
  match lookahead tokens with  
  | Tok_Mult -> (* A -> B Tok_Mult A *)  
    let t2 = match_token tokens Tok_Mult in  
    let e2, t3 = parse_A t2 in  
    (Mult (e1, e2), t3)  
  | _ -> (e1, tokens)
```



Example

Grammar:

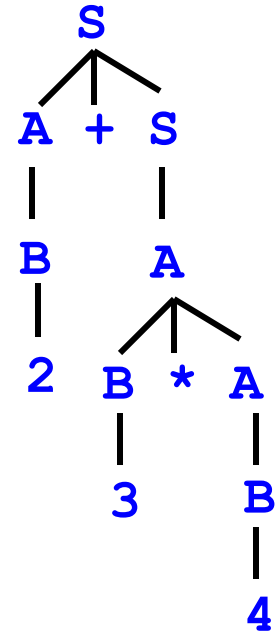
$S \rightarrow A + S \mid A$

$A \rightarrow B * A \mid B$

$B \rightarrow 0 \mid 1 \dots \mid 9$

Input: 2+3*4

```
let rec parse_B tokens =  
  match lookahead tokens with  
  | Tok_Num c -> (* B -> Tok_Num *)  
    let t = match_token tokens (Tok_Num c) in  
    (Num (int_of_string c), t)  
  | _ -> raise (ParseError "parse_B")
```



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)
 - Compare with lookahead to decide which production to select
 - Parse fails if lookahead does not match any production

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

Example Parser

- ▶ Given grammar $S \rightarrow xyz \mid abc$
- ▶ 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 $S \rightarrow A \mid B$ $A \rightarrow x \mid y$ $B \rightarrow z$

```
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 () = ...
```

Execution Trace = Parse Tree

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

- ▶ Examples

- Grammar

$S \rightarrow xyz$

$S \rightarrow abc$

- String “xyz”

parse_S ()

match_tok “x”

match_tok “y”

match_tok “z”

S
/ | \
x y z

- Grammar

$S \rightarrow A \mid B$

$A \rightarrow x \mid y$

$B \rightarrow z$

- String “x”

parse_S ()

parse_A ()

match_tok
“x”

S
|
A
|
x

Left Recursion

- ▶ Consider grammar $S \rightarrow Sa \mid \varepsilon$
 - 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 $S \rightarrow aS \mid \varepsilon$ Again, $\text{First}(aS) = a$

- 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

- $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta$
 - β must exist or no derivation will yield a string

- ▶ Rewrite grammar as (repeat as needed)

- $A \rightarrow \beta L$
- $L \rightarrow \alpha_1 L \mid \alpha_2 L \mid \dots \mid \alpha_n L \mid \epsilon$

- ▶ Replaces left recursion with right recursion

- ▶ Examples

- | | | |
|-------------------------------------|--------------------------------|--|
| • $S \rightarrow Sa \mid \epsilon$ | $\Rightarrow S \rightarrow L$ | $L \rightarrow aL \mid \epsilon$ |
| • $S \rightarrow Sa \mid Sb \mid c$ | $\Rightarrow S \rightarrow cL$ | $L \rightarrow aL \mid bL \mid \epsilon$ |

Quiz 4

- What does the following code parse?

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

- A. $S \rightarrow axyq$
- B. $S \rightarrow a \mid q$
- C. $S \rightarrow aaxy \mid qq$
- D. $S \rightarrow axy \mid q$

Quiz 4

- What does the following code parse?

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

- A. $S \rightarrow axyq$
- B. $S \rightarrow a \mid q$
- C. $S \rightarrow aaxy \mid qq$
- D. $S \rightarrow axy \mid q$

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. $S \rightarrow aS \mid qp$
- 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 () = "a" then  
    (match_tok "a";  
     parse_S ())  
  else if lookahead () = "q" then  
    (match_tok "q";  
     match_tok "p")  
  else  
    raise (ParseError "parse_S")
```

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

Quiz 6

Can recursive descent parse this grammar?

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

- A. Yes
- B. No

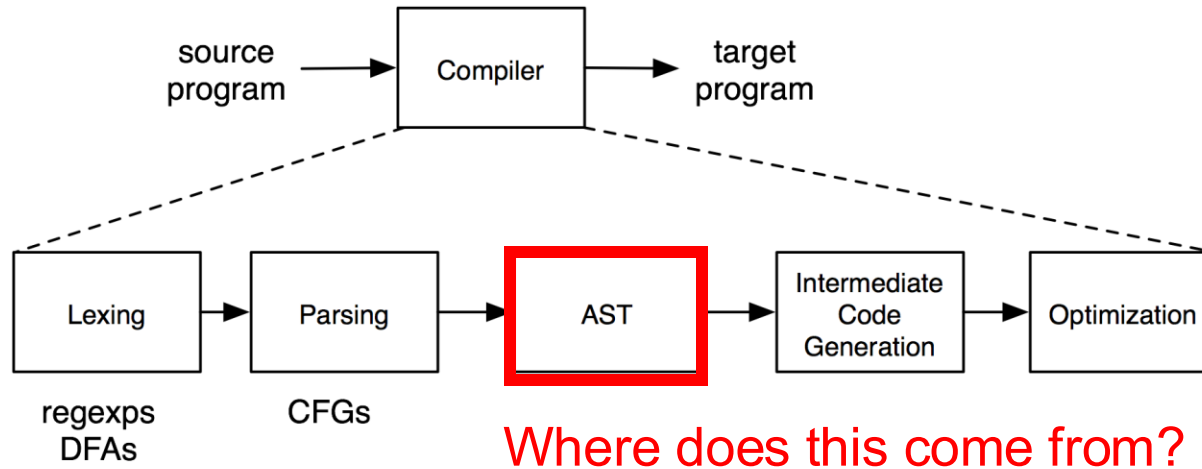
Quiz 6

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

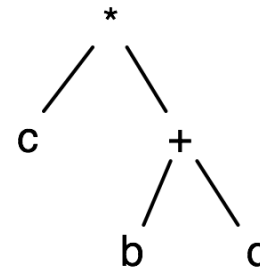
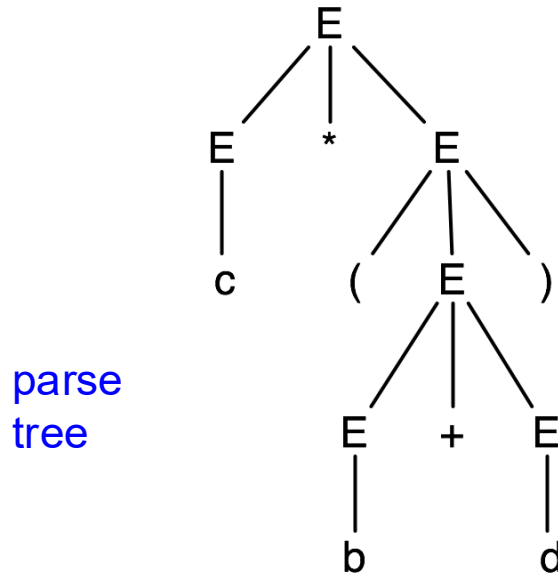


Parse Trees to ASTs

- ▶ 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



AST

Parser Examples:

- ▶ Top-Down Parsing example on slide 5
 - https://github.com/anwarmamat/cmsc330spring2024/tree/main/examples/parser_class_example
- ▶ Expression (addition and multiplication) parser Dune project
 - <https://github.com/anwarmamat/cmsc330spring2024/tree/main/examples/parser>
- ▶ Expression (addition and multiplication) parser
 - https://github.com/anwarmamat/cmsc330spring2024/blob/main/examples/parse_add_mult.ml
- ▶ Postfix expression parser:
 - https://github.com/anwarmamat/cmsc330spring2024/blob/main/examples/parser_postfix_exp.ml