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
```

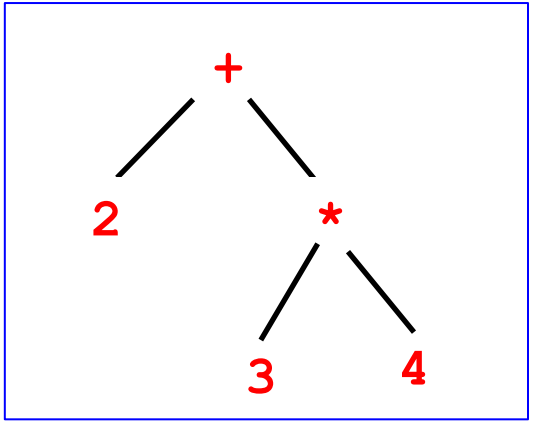
Uses **`Str`** library module for regexps

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)

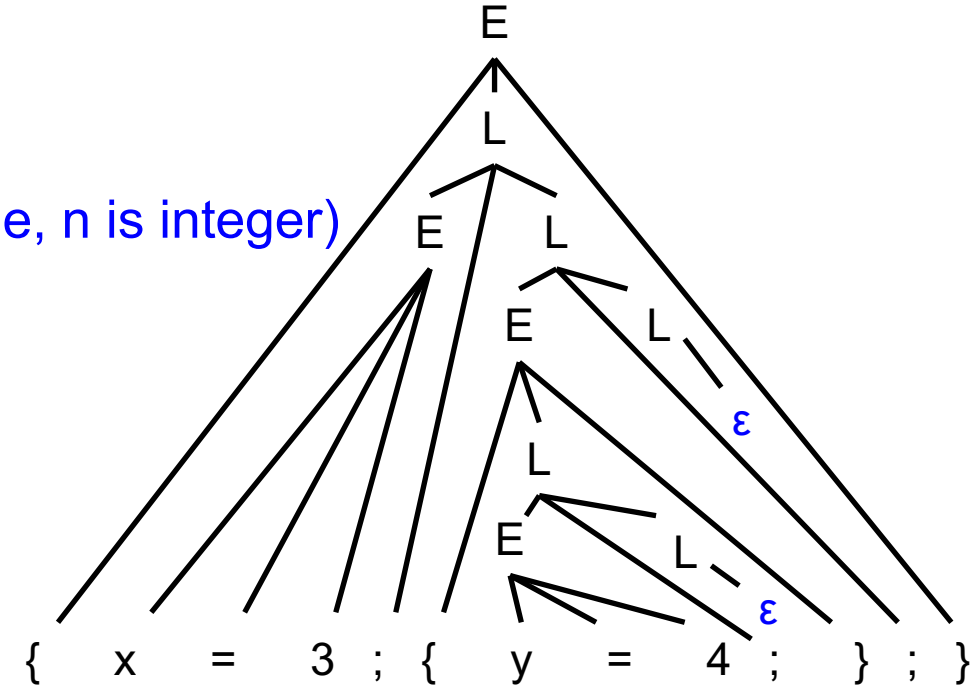
E → id = n | { L }

L → E ; L | ε

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

Show parse tree for

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



Recursive Descent Parsing

- 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 -> A + S | A
A -> B * A | B
B -> 0 | 1 … | 9
```

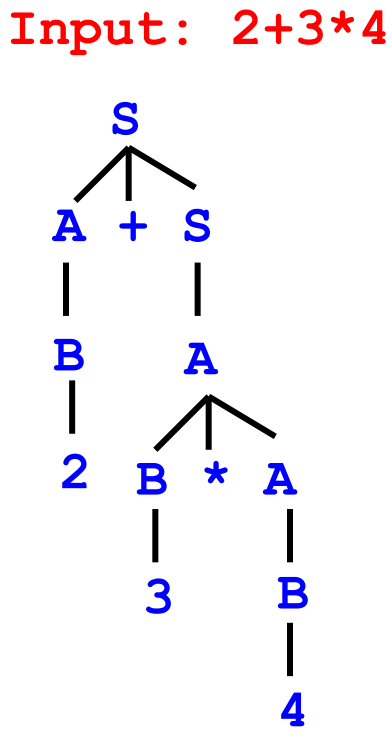
```
Σ={0-9,*,+}

Input: 2 + 3 * 4
```

Recursive Descent Parsing: Example

Grammar:

```
S -> A + S | A
A -> B * A | B
B -> 0 | 1 … | 9
```

Parsing



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 = tok -> 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')
```

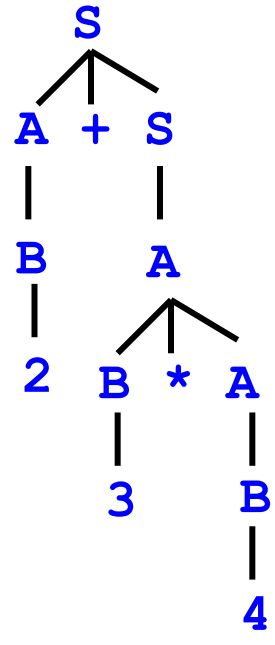
A. 2
B. raise Exception
C. Tok_Num '2'
D. [Tok_Add; Tok_Num '3'; Tok_END]

Example

Grammar:

```
S -> A + S | A
A -> B * A | B
B -> 0 | 1 … | 9
```

Input: 2+3*4



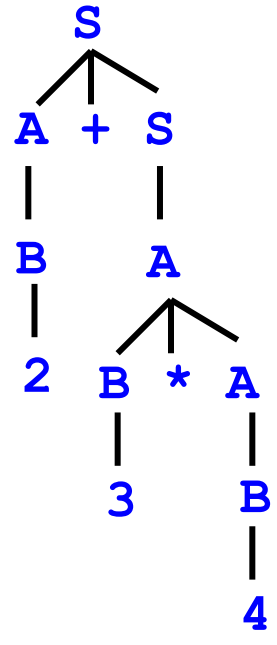

```
let rec parse_S tokens =
  let e1, t1 = parse_A tokens in
    match lookahead t1 with
    | Tok_Add -> (* S -> A Tok_Add E *)
       let t2 = match_token t1 Tok_Add in
       let e2, t3 = parse_S t2 in
       (Add (e1, e2), t3)
    | _ -> (e1, t1) (* S -> A *)
```

Example

Grammar:

```
S -> A + S | A
A -> B * A | B
B -> 0 | 1 … | 9
```

Input: 2+3*4



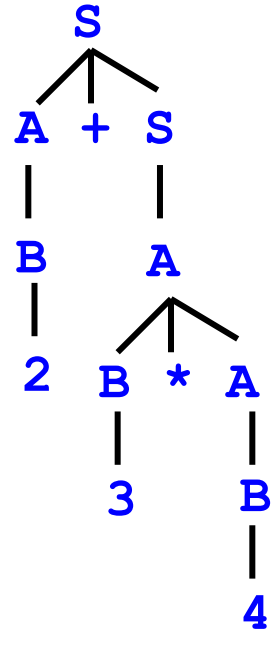

```
let rec parse_A tokens =
  let e1, tokens = parse_B tokens in
   match lookahead tokens with
    | Tok_Mult -> (* A -> B Tok_Mult A *)
      let t2 = match_token tokens Tok_Mult in
      let e2, t3 = parse_A t2 in
      (Mult (e1, e2), t3)
    | _ -> (e1, tokens)
```

Example

Grammar:

```
S -> A + S | A
A -> B * A | B
B -> 0 | 1 … | 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