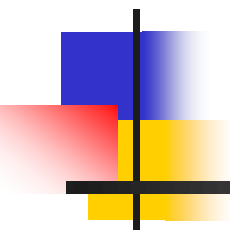


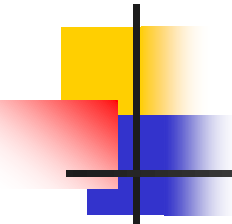
Programming Languages and Compilers (CS 421)



William Mansky

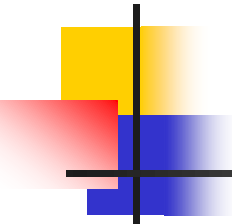
<http://courses.engr.illinois.edu/cs421/>

Based in part on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, Elsa Gunter, and Dennis Griffith



Parsing Programs

- Parsing is the process of tracing or constructing a parse tree for a given input string
- Process usually broken into two phases:
 - Lexer: generating tokens from string or character stream
 - Parser: generating parse tree from token list or stream

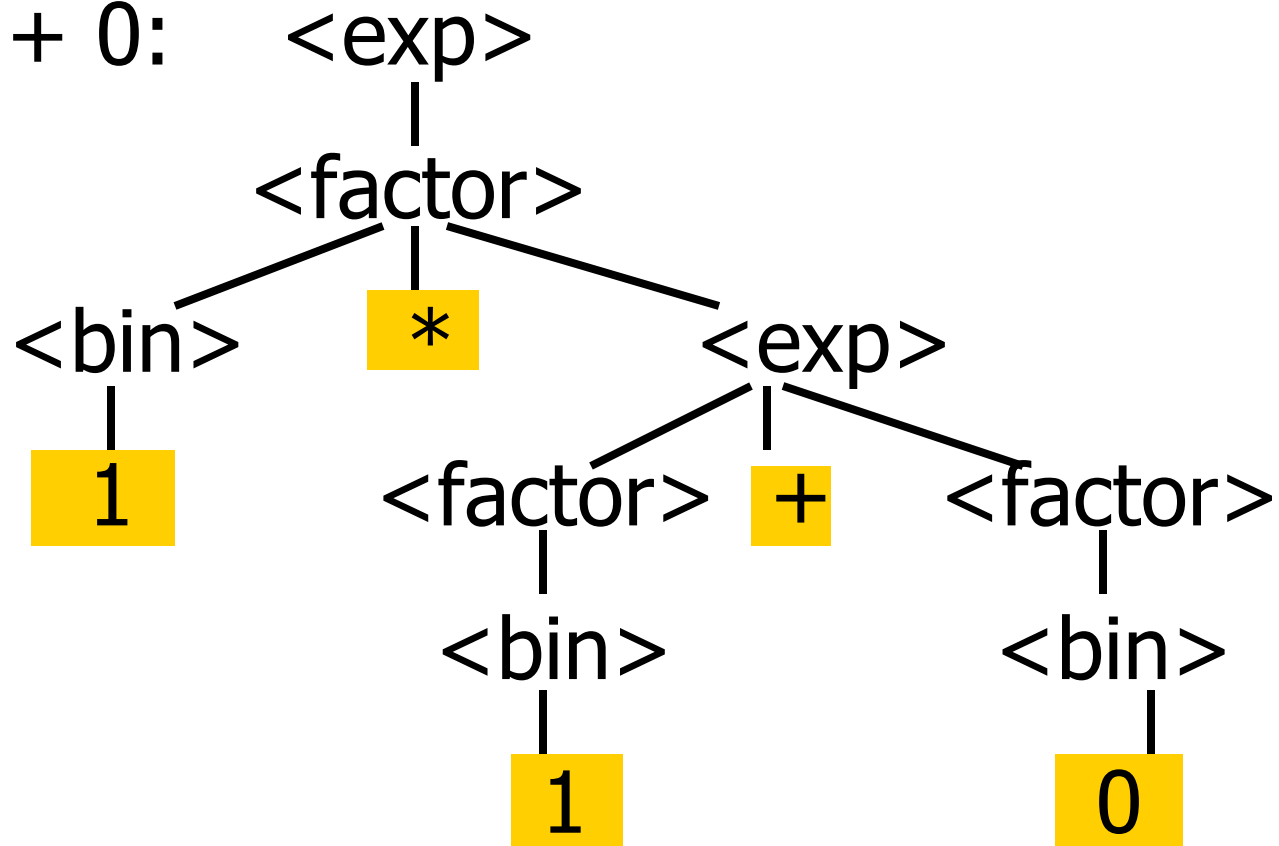


Example BNF Grammar

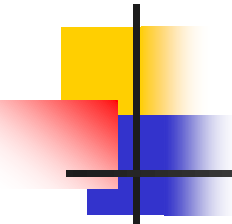
■ $\langle \text{exp} \rangle ::= \langle \text{factor} \rangle$
 $| \langle \text{factor} \rangle + \langle \text{factor} \rangle$
 $\langle \text{factor} \rangle ::= \langle \text{bin} \rangle$
 $| \langle \text{bin} \rangle * \langle \text{exp} \rangle$
 $\langle \text{bin} \rangle ::= 0 \mid 1$

Example Parse Tree

■ 1 * 1 + 0:

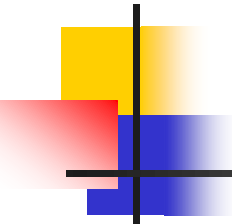


Fringe of tree is string generated by grammar



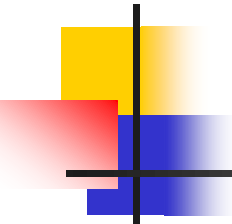
Recursive Descent Parsing

- Algorithm for constructing parse trees given BNF grammar
- Builds parse tree top-down, always builds leftmost derivation (LL – left-to-right scanning, leftmost derivation)
- One subprogram/function to parse each nonterminal
- Each nonterminal in right-hand side of a rule corresponds to a recursive call to the associated subprogram



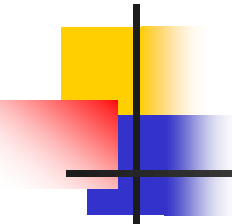
Recursive Descent Parsing

- Each subprogram must be able to decide on a production by looking at the left-most character in the string to be parsed
 - May do so directly, or indirectly by calling another parsing subprogram
 - Cannot be built from left-recursive grammars
 - Sometimes can modify grammar to suit



Sample Grammar

$$\begin{aligned} \langle \text{expr} \rangle ::= & \langle \text{term} \rangle \mid \langle \text{term} \rangle + \langle \text{expr} \rangle \\ & \mid \langle \text{term} \rangle - \langle \text{expr} \rangle \end{aligned}$$
$$\begin{aligned} \langle \text{term} \rangle ::= & \langle \text{factor} \rangle \mid \langle \text{factor} \rangle * \langle \text{term} \rangle \\ & \mid \langle \text{factor} \rangle / \langle \text{term} \rangle \end{aligned}$$
$$\langle \text{factor} \rangle ::= \langle \text{id} \rangle \mid (\langle \text{expr} \rangle)$$



Tokens as OCaml Types

- + - * / () <id>

- Becomes an OCaml datatype

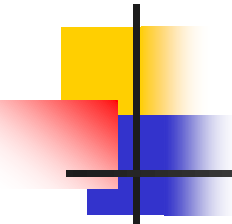
type token =

Id_token of string

| Left_parenthesis | Right_parenthesis

| Times_token | Divide_token

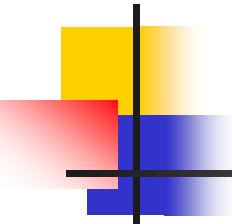
| Plus_token | Minus_token



Parse Trees as Datatypes

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle$
 $| \langle \text{term} \rangle + \langle \text{expr} \rangle$
 $| \langle \text{term} \rangle - \langle \text{expr} \rangle$

type expr =
 Term_as_Expr of term
 | Plus_Expr of (term * expr)
 | Minus_Expr of (term * expr)

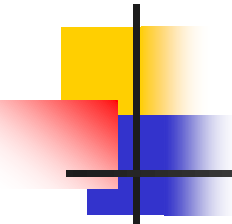


Parse Trees as Datatypes

$\langle \text{term} \rangle ::= \langle \text{factor} \rangle$
| $\langle \text{factor} \rangle * \langle \text{term} \rangle$
| $\langle \text{factor} \rangle / \langle \text{term} \rangle$

and term =

Factor_as_Term of factor
| Mult_Term of (factor * term)
| Div_Term of (factor / term)



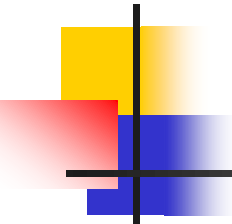
Parse Trees as Datatypes

$\langle \text{factor} \rangle ::= \langle \text{id} \rangle \mid (\langle \text{expr} \rangle)$

and factor =

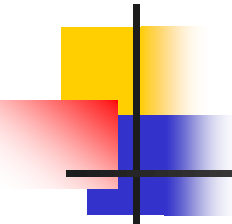
Id_as_Factor of string

| Parenthesized_Expr_as_Factor of expr



Parsing Lists of Tokens

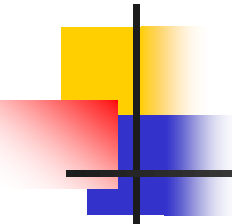
- For recursive descent, three mutually recursive functions:
 - $\text{expr} : \text{token list} \rightarrow (\text{expr} * \text{token list})$
 - $\text{term} : \text{token list} \rightarrow (\text{term} * \text{token list})$
 - $\text{factor} : \text{token list} \rightarrow (\text{factor} * \text{token list})$
- Each parses what it can and gives back parse and remaining tokens



Parse Trees as Datatypes

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle$
| $\langle \text{term} \rangle + \langle \text{expr} \rangle$
| $\langle \text{term} \rangle - \langle \text{expr} \rangle$

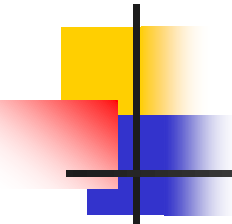
type expr =
 Term_as_Expr of term
 | Plus_Expr of (term * expr)
 | Minus_Expr of (term * expr)



Parsing an Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle [(+ \mid -) \langle \text{expr} \rangle]$

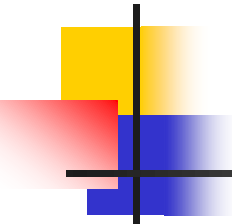
```
let rec expr tokens =  
  match term tokens  
  with (term_parse, more_tokens) ->  
    match more_tokens  
    with (Plus_token  :: rest) -> ...  
        | (Minus_token :: rest) -> ...  
        | _ -> ...
```



Parsing an Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle [(+ \mid -) \langle \text{expr} \rangle]$

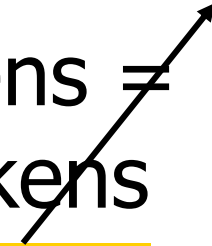
```
let rec expr tokens =  
  match term tokens with  
    with (term_parse, more_tokens) ->  
      match more_tokens  
        with (Plus_token :: rest) -> ...  
             | (Minus_token :: rest) -> ...  
             | _ -> ...
```



Parsing an Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle [(+ \mid -) \langle \text{expr} \rangle]$

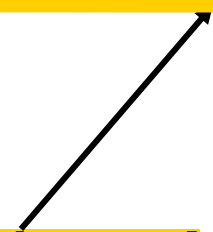
let rec expr tokens =
 match term tokens
 with (term_parse, more_tokens) ->
 match more_tokens
 with (Plus_token :: rest) -> ...
 | (Minus_token :: rest) -> ...
 | _ -> ...



Parsing an Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \underline{[(+ | -) \langle \text{expr} \rangle]}$

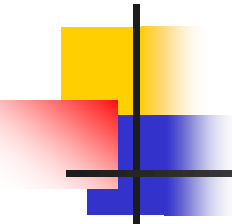
```
let rec expr tokens =  
  match term tokens  
  with (term_parse, more_tokens) ->  
    match more_tokens  
    with (Plus_token  :: rest) -> ...  
        | (Minus_token :: rest) -> ...  
        | _ -> ...
```

A diagram consisting of a black arrow pointing from the underlined part of the grammar rule, $[(+ | -) \langle \text{expr} \rangle]$, down to the `more_tokens` argument in the `with` clause of the `expr` function definition.

Parsing a Plus-Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle + \langle \text{expr} \rangle$

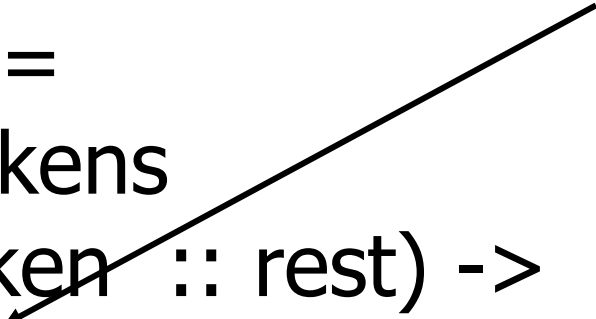
```
let rec expr tokens =  
  ... match more_tokens  
    with (Plus_token :: rest) ->  
      match expr rest  
        with (expr_parse, tokens_after_expr) ->  
          (Plus_Expr (term_parse , expr_parse),  
           tokens_after_expr)  
    | ...
```

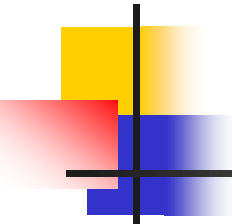


Parsing a Plus-Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle + \langle \text{expr} \rangle$

```
let rec expr tokens =  
  ... match more_tokens  
    with (Plus_token :: rest) ->  
        match expr rest  
          with (expr_parse, tokens_after_expr) ->  
              (Plus_Expr (term_parse , expr_parse),  
               tokens_after_expr))  
  | ...
```

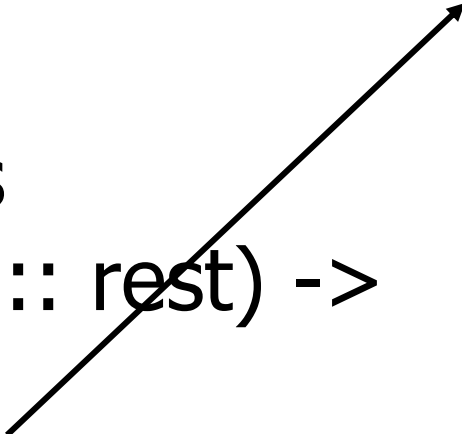




Parsing a Plus-Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle + \langle \text{expr} \rangle$

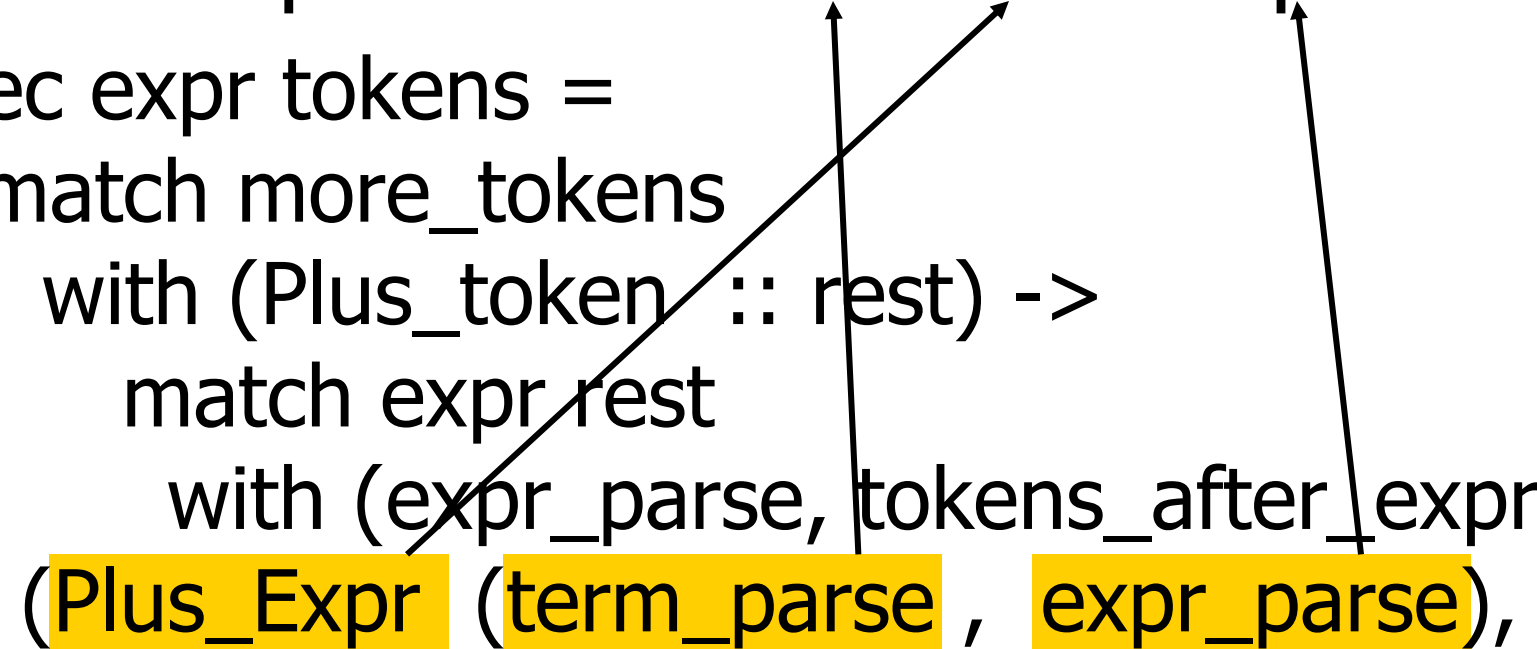
```
let rec expr tokens =  
  ... match more_tokens  
    with (Plus_token :: rest) ->  
      match expr rest  
        with (expr_parse, tokens_after_expr) ->  
          (Plus_Expr (term_parse , expr_parse),  
           tokens_after_expr))  
    | ...
```



Parsing a Plus-Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle + \langle \text{expr} \rangle$

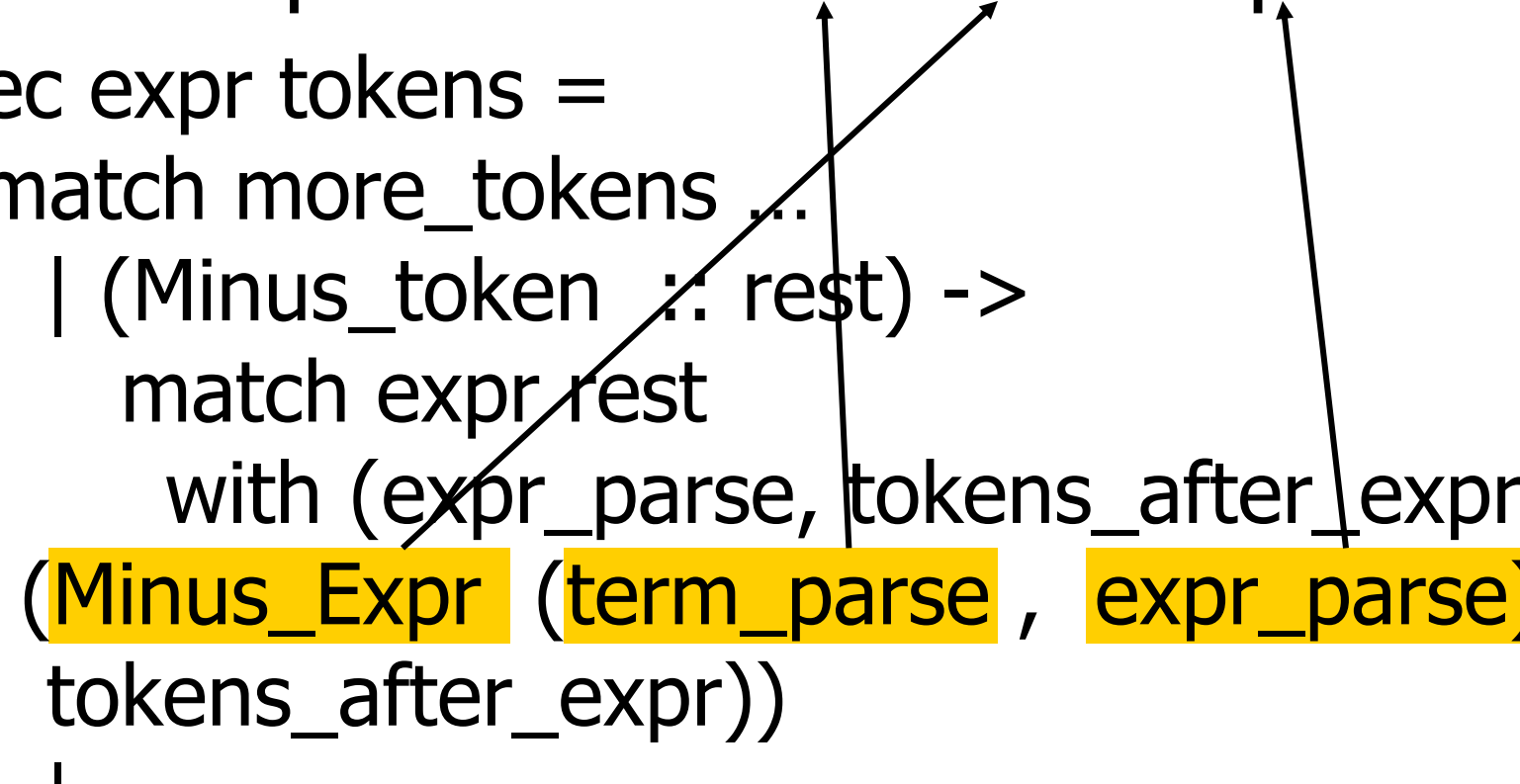
```
let rec expr tokens =  
  ... match more_tokens  
    with (Plus_token :: rest) ->  
        match expr rest  
          with (expr_parse, tokens_after_expr) ->  
              (Plus_Expr (term_parse , expr_parse),  
               tokens_after_expr))  
  | ...
```



Parsing a Minus-Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle - \langle \text{expr} \rangle$

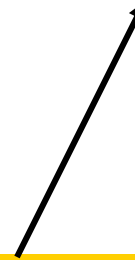
```
let rec expr tokens =  
  ... match more_tokens ...  
  | (Minus_token :: rest) ->  
    match expr rest  
    with (expr_parse, tokens_after_expr) ->  
    (Minus_Expr (term_parse , expr_parse),  
     tokens_after_expr)  
  | ...
```

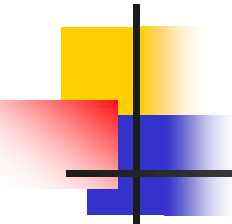


Parsing an Expression as a Term

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle$

```
let rec expr tokens =  
  ... match more_tokens ...  
    | _ -> (Term_as_Expr term_parse,  
            more_tokens)
```

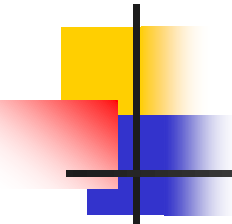




Parsing an Expression

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle [(+ \mid -) \langle \text{expr} \rangle]$

```
let rec expr tokens =  
  match term tokens  
  with (term_parse, more_tokens) ->  
    match more_tokens  
    with (Plus_token  :: rest) -> ...  
        | (Minus_token :: rest) -> ...  
        | _ -> ...  
and term tokens = ...    (* defined similarly *)
```



Parsing Factor as Id

$\langle \text{factor} \rangle ::= \langle \text{id} \rangle \mid (\langle \text{expr} \rangle)$

and factor tokens =
match tokens

with (Id_token id_name::more_tokens) ->
(Id_as_Factor id_name, more_tokens)

| ...

Parsing Factor as Parenthesized Expr

$\langle \text{factor} \rangle ::= \langle \text{id} \rangle \mid (\langle \text{expr} \rangle)$

and factor tokens =

match tokens ...

| (Left_parenthesis :: more_tokens) ->

(match **expr more_tokens**

with (expr_parse, more) ->

match more with Right_parenthesis :: rest ->

(Parenthesized_Expr_as_Factor expr_parse,
rest)

Parsing Factor as Parenthesized Expr

$\langle \text{factor} \rangle ::= \langle \text{id} \rangle \mid (\langle \text{expr} \rangle)$

and factor tokens =

match tokens ...

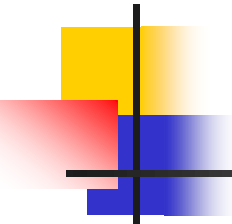
| (Left_parenthesis :: more_tokens) ->

(match expr more_tokens

with (expr_parse, more) ->

match more with Right_parenthesis :: rest ->

(Parenthesized_Expr_as_Factor expr_parse,
rest)



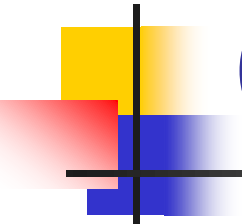
Error Cases

- What if no matching right parenthesis?

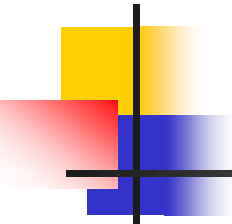
```
| _ -> raise (Failure "No matching  
rparen") ) )
```

- What if no leading id or left parenthesis?

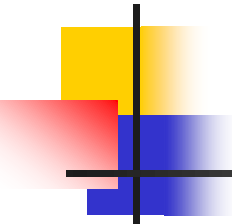
```
| _ -> raise (Failure "No id or lparen" ) );;
```


$$(a + b) * c - d$$

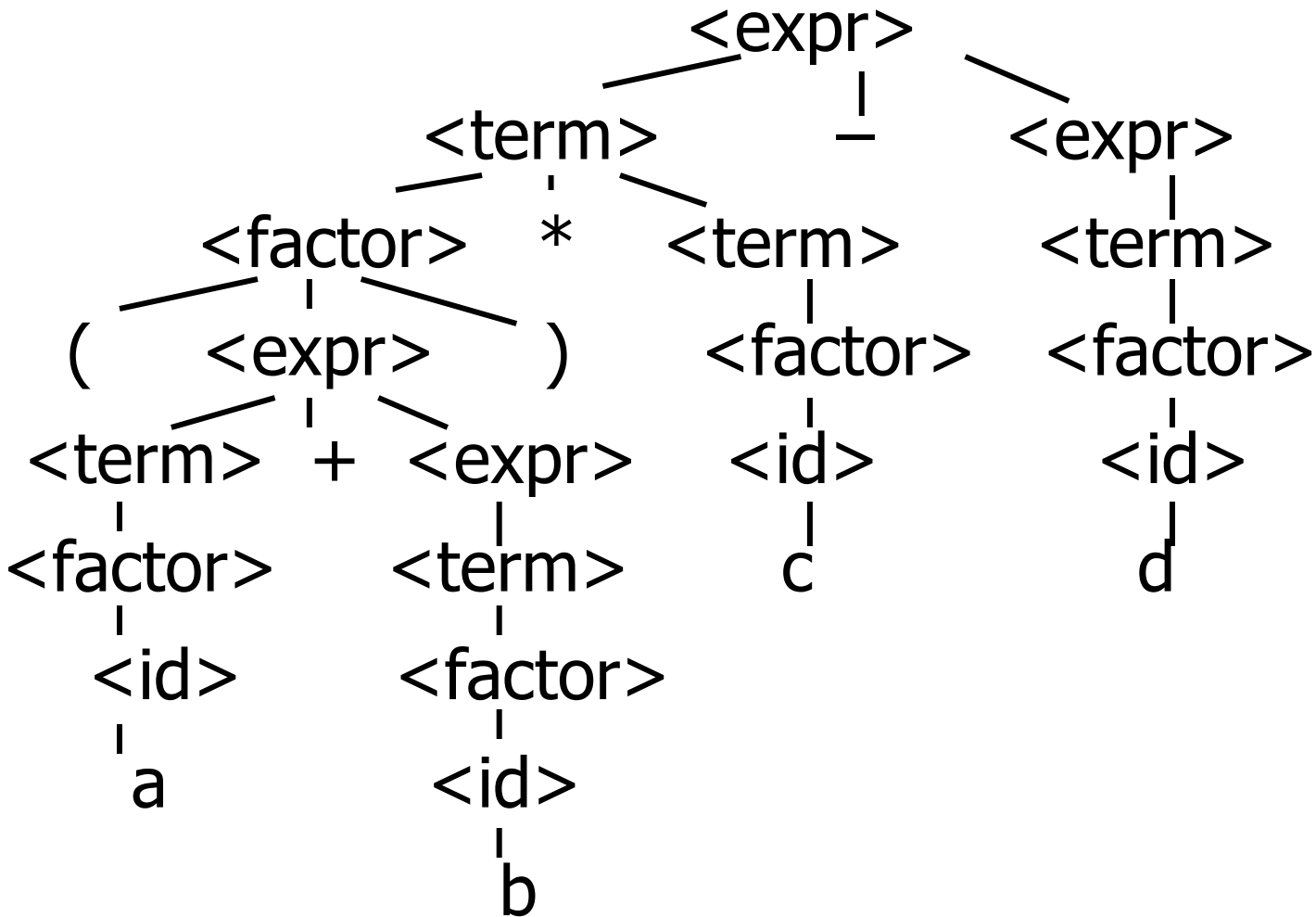
```
expr [Left_parenthesis; Id_token "a";  
      Plus_token; Id_token "b";  
      Right_parenthesis; Times_token;  
      Id_token "c"; Minus_token;  
      Id_token "d"];;
```

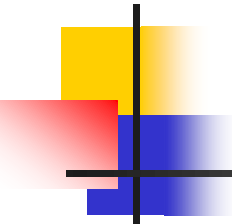

$$(a + b) * c - d$$

```
- : expr * token list =  
(Minus_Expr  
  (Mult_Term  
    (Parenthesized_Expr_as_Factor  
      (Plus_Expr  
        (Factor_as_Term (Id_as_Factor "a"),  
          Term_as_Expr (Factor_as_Term  
            (Id_as_Factor "b")))),  
        Factor_as_Term (Id_as_Factor "c")),  
        Term_as_Expr (Factor_as_Term (Id_as_Factor  
          "d")))),  
  [])
```



$(a + b) * c - d$





$a + b * c - d$

```
# expr [Id_token "a"; Plus_token; Id_token "b";  
      Times_token; Id_token "c"; Minus_token;  
      Id_token "d"];;
```

```
- : expr * token list =
```

```
(Plus_Expr
```

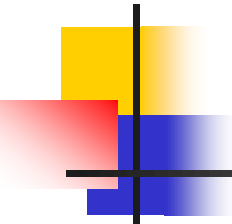
```
  (Factor_as_Term (Id_as_Factor "a"),
```

```
  Minus_Expr
```

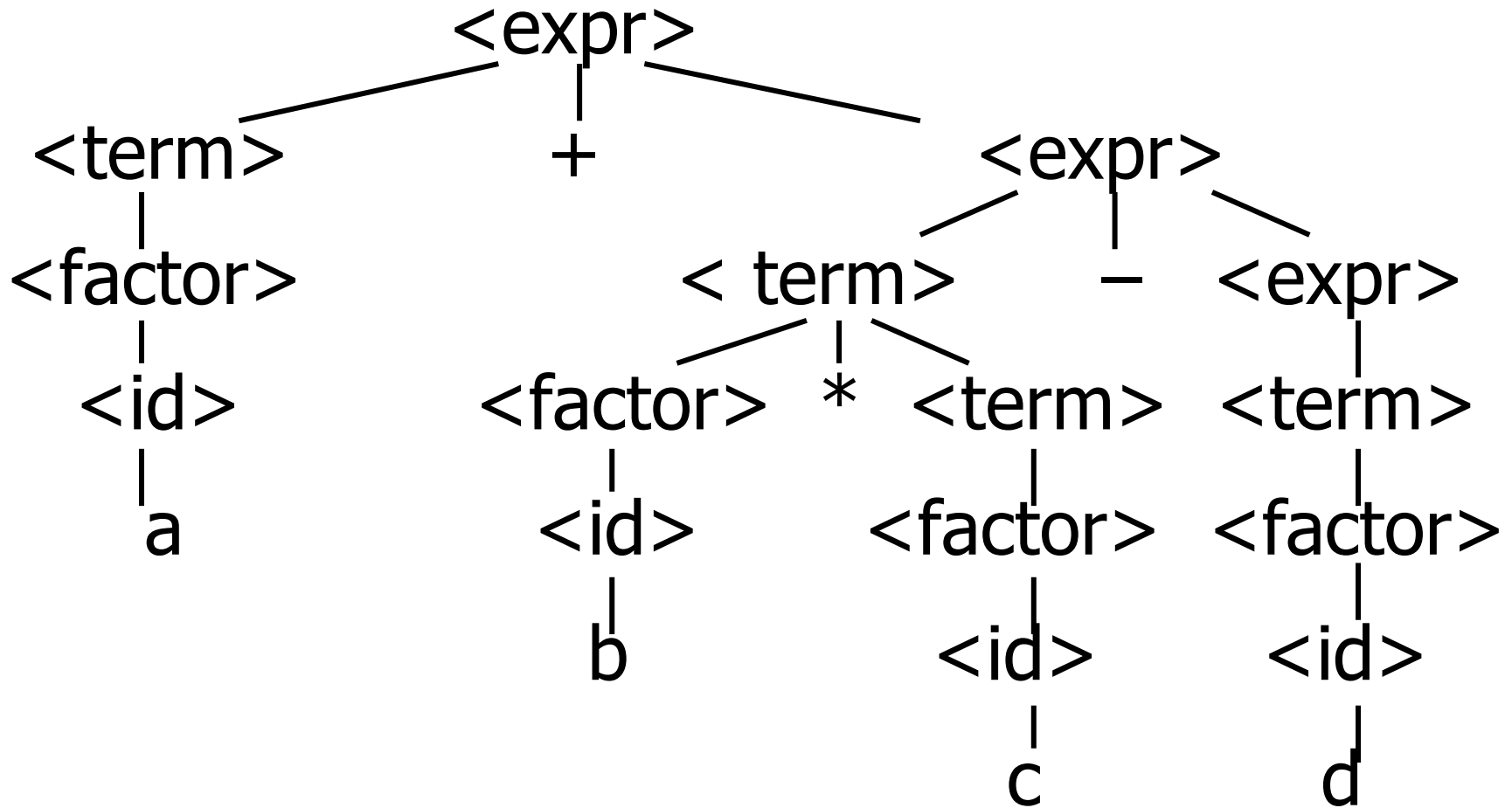
```
    (Mult_Term (Id_as_Factor "b", Factor_as_Term  
      (Id_as_Factor "c")),
```

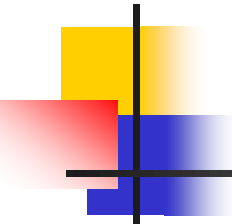
```
    Term_as_Expr (Factor_as_Term (Id_as_Factor  
      "d")))),
```

```
[])
```



$a + b * c - d$



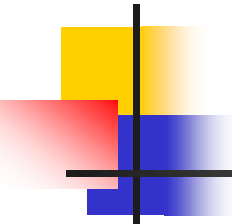


$(a + b * c - d$

```
# expr [Left_parenthesis; Id_token "a";  
Plus_token; Id_token "b"; Times_token;  
Id_token "c"; Minus_token; Id_token "d"];;
```

Exception: Failure "No matching rparen".

Can't parse because we ran out of tokens
before seeing a ')'



$(a + b) * c - d *$

```
expr [Id_token "a"; Plus_token; Id_token "b";  
      Right_parenthesis; Times_token; Id_token "c";  
      Minus_token; Id_token "d"];;
```

- : expr * token list =

(Plus_Expr

(Factor_as_Term (Id_as_Factor "a"),

Term_as_Expr (Factor_as_Term (Id_as_Factor
"b"))),

[Right_parenthesis; Times_token; Id_token "c";
Minus_token; Id_token "d"])

Parsing Whole String

- Q: Does the whole string parse?
- A: Check whether returned tokens are empty

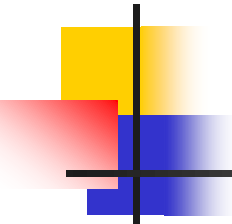
let parse tokens =

match **expr** tokens

with (expr_parse, []) -> expr_parse

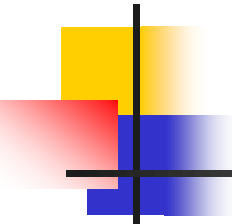
| _ -> raise (Failure "No parse");;

- Fixes <expr> as start symbol



Streams in Place of Lists

- More realistically, might not want to create entire list of tokens before parsing
- We want to generate one token at a time and use it to make one step in parsing
- Use $(\text{token} * (\text{unit} \rightarrow \text{token}))$ or $(\text{token} * (\text{unit} \rightarrow \text{token option}))$ in place of token list



Problems for Recursive-Descent Parsing

- Left Recursion:

$$\mathbf{A} ::= \mathbf{A}w$$

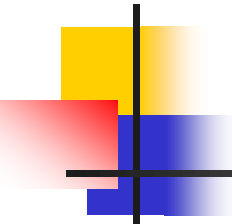
translates to a subroutine that loops forever

- Indirect Left Recursion:

$$\mathbf{A} ::= \mathbf{B}w$$

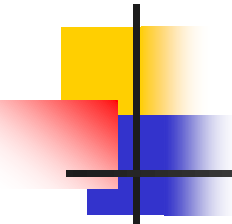
$$\mathbf{B} ::= \mathbf{A}v$$

causes the same problem



Problems for Recursive-Descent Parsing

- Parser must always be able to choose next action based only the very next token
- Pairwise Disjointness Test: Can we always determine which rule (in the non-extended BNF) to choose based on just the first token?



Pairwise Disjointness Test

- For each rule

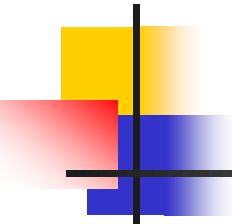
$$\mathbf{A} ::= \gamma$$

Calculate ($\Rightarrow^*$ denotes derivations)

FIRST (γ) =

$$\{a \mid \gamma \Rightarrow^* a w\} \cup \{\varepsilon \mid \text{if } \gamma \Rightarrow^* \varepsilon\}$$

- For each pair $\mathbf{A} ::= \gamma$ and $\mathbf{A} ::= z$,
require $\text{FIRST}(\gamma) \cap \text{FIRST}(z) = \{ \}$

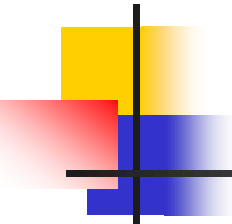


Example

Grammar:

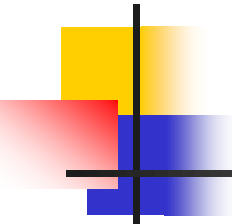
$$\langle S \rangle ::= \langle A \rangle a \langle B \rangle b$$
$$\langle A \rangle ::= \langle A \rangle b \mid b$$
$$\langle B \rangle ::= a \langle B \rangle \mid a$$
$$\text{FIRST}(\langle A \rangle b) = \{b\}$$
$$\text{FIRST}(b) = \{b\}$$

Rules for $\langle A \rangle$ not pairwise disjoint



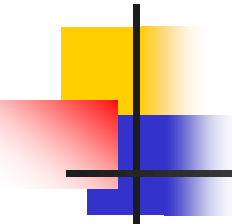
Eliminating Left Recursion

- Rewrite grammar to shift left recursion to right recursion
 - Changes associativity
- Given
$$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle \text{ and }$$
$$\langle \text{expr} \rangle ::= \langle \text{term} \rangle$$
- Add new non-terminal $\langle e \rangle$ and replace above rules with
$$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle e \rangle$$
$$\langle e \rangle ::= + \langle \text{term} \rangle \langle e \rangle \mid \varepsilon$$



Factoring Grammar

- Test too strong: Can't handle
$$\langle \text{expr} \rangle ::= \langle \text{term} \rangle [(+ \mid -) \langle \text{expr} \rangle]$$
- Answer: Add new non-terminal and replace above rules by
$$\begin{aligned}\langle \text{expr} \rangle &::= \langle \text{term} \rangle \langle e \rangle \\ \langle e \rangle &::= + \langle \text{term} \rangle \langle e \rangle \\ \langle e \rangle &::= - \langle \text{term} \rangle \langle e \rangle \\ \langle e \rangle &::= \varepsilon\end{aligned}$$
- Delay decision point



Example

Both $\langle A \rangle$ and $\langle B \rangle$
have problems:

$$\begin{aligned}\langle S \rangle &::= \langle A \rangle a \langle B \rangle b \\ \langle A \rangle &::= \langle A \rangle b \mid b \\ \langle B \rangle &::= a \langle B \rangle \mid a\end{aligned}$$

Transform grammar
to:

$$\begin{aligned}\langle S \rangle &::= \langle A \rangle a \langle B \rangle b \\ \langle A \rangle &::= b \langle A1 \rangle \\ \langle A1 \rangle &::= b \langle A1 \rangle \mid \varepsilon \\ \langle B \rangle &::= a \langle B1 \rangle \\ \langle B1 \rangle &::= a \langle B1 \rangle \mid \varepsilon\end{aligned}$$