

Programming Languages and Compilers (CS 421)



Elsa L Gunter

2112 SC, UIUC

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

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



Recursive Descent Parsing

- Recursive descent parsers are a class of parsers derived fairly directly from BNF grammars
- A recursive descent parser traces out a parse tree in top-down order, corresponding to a left-most derivation (LL - left-to-right scanning, leftmost derivation)



Recursive Descent Parsing

- Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all phrases that the nonterminal can generate
- 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 how to begin parsing by looking at the left-most character in the string to be parsed
 - May do so directly, or indirectly by calling another parsing subprogram
- Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars
 - Sometimes can modify grammar to suit



Sample Grammar

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

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

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



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

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

type expr =

Term_as_Expr of term

| Plus_Expr of (term * expr)

| Minus_Expr of (term * expr)



Parse Trees as Datatypes

$$\begin{aligned} \langle \text{term} \rangle ::= & \langle \text{factor} \rangle \mid \langle \text{factor} \rangle * \\ & \langle \text{term} \rangle \\ & \mid \langle \text{factor} \rangle / \langle \text{term} \rangle \end{aligned}$$

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

- Will create 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



Parsing an Expression

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

let rec expr tokens =

(match term tokens

with (term_parse , tokens_after_term) ->

(match tokens_after_term

with(Plus_token :: tokens_after_plus) ->



Parsing an Expression

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

let rec expr tokens =

(match **term tokens**

with (term_parse , tokens_after_term) ->

(match tokens_after_term

with (Plus_token :: tokens_after_plus) ->



Parsing a Plus Expression

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

let rec expr tokens =

(match term tokens

with (**term_parse** , tokens_after_term) ->

(match tokens_after_term

with (Plus_token :: tokens_after_plus) ->

Parsing a Plus Expression

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

let rec expr tokens =

(match term tokens

with (**term_parse**, tokens_after_term) ->

(match **tokens_after_term**

with (Plus_token :: tokens_after_plus) ->



Parsing a Plus Expression

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

let rec expr tokens =

(match term tokens

with (term_parse , tokens_after_term) ->

(match tokens_after_term

with (**Plus_token** :: tokens_after_plus) ->



Parsing a Plus Expression

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

(match **expr tokens_after_plus**
with (expr_parse , tokens_after_expr) ->
(Plus_Expr (term_parse , expr_parse),
tokens_after_expr))



Parsing a Plus Expression

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

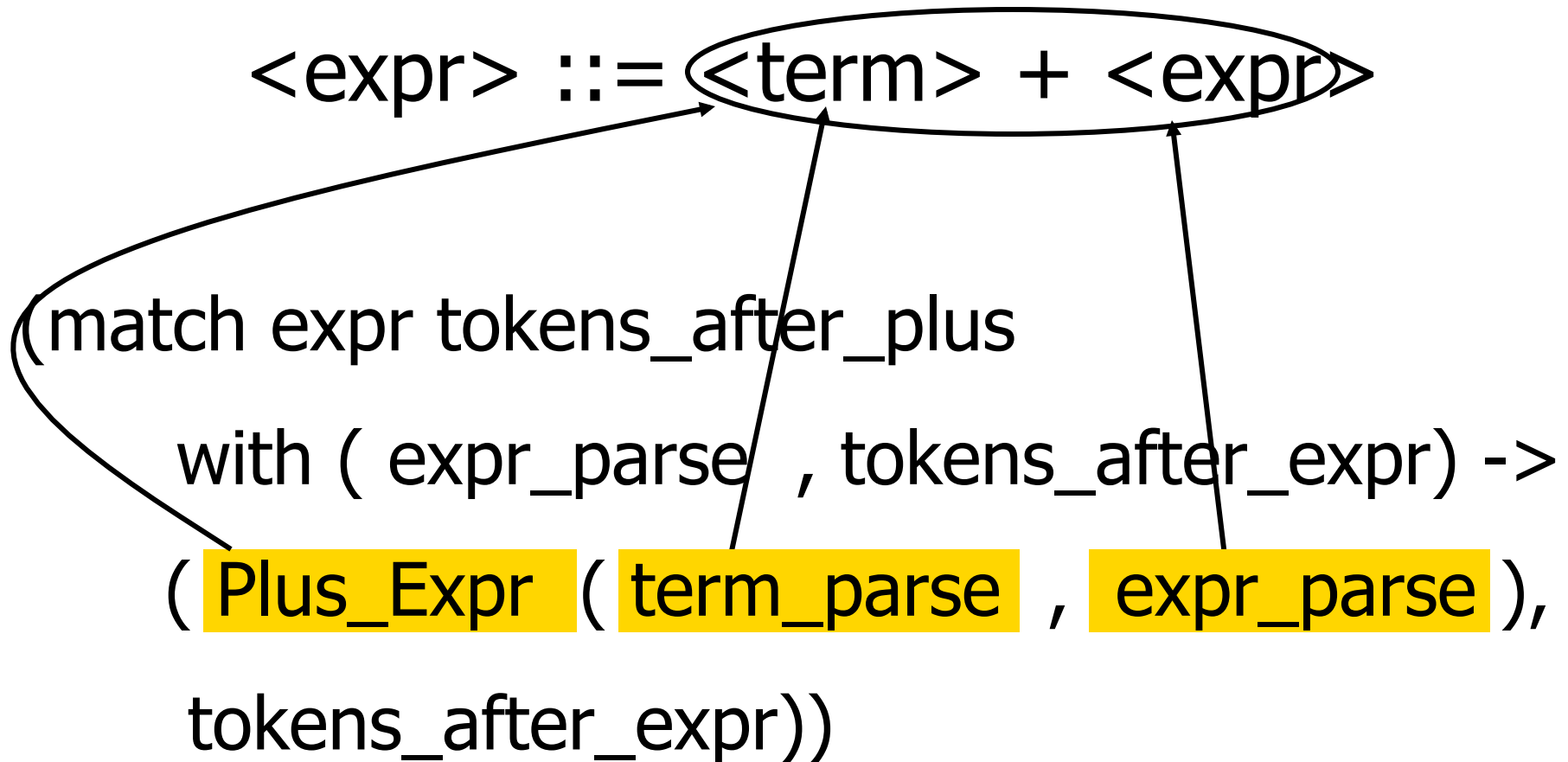
(match expr tokens_after_plus

with (**expr_parse** , tokens_after_expr) ->

(Plus_Expr (term_parse , expr_parse),

tokens_after_expr))

Building Plus Expression Parse Tree





Parsing a Minus Expression

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

```
| ( Minus_token :: tokens_after_minus) ->  
  (match expr tokens_after_minus  
   with ( expr_parse , tokens_after_expr) ->  
        ( Minus_Expr ( term_parse , expr_parse ),  
          tokens_after_expr))
```

Parsing a Minus Expression

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

| (**Minus_token** :: tokens_after_minus) ->
(match expr tokens_after_minus
with (expr_parse , tokens_after_expr) ->
(**Minus_Expr** (**term_parse** , **expr_parse**),
tokens_after_expr))



Parsing an Expression as a Term

`<expr> ::= <term>`

| `_ -> (Term_as_Expr term_parse , tokens_after_term))`

- Code for **term** is same except for replacing addition with multiplication and subtraction with division

Parsing Factor as Id

$\langle \text{factor} \rangle ::= \langle \text{id} \rangle$

and factor tokens =

(match tokens

with (Id_token id_name :: tokens_after_id) =


(**Id_as_Factor** id_name, tokens_after_id)



Parsing Factor as Parenthesized Expression

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

| factor (**Left_parenthesis** :: tokens) =
(match expr tokens
with (**expr_parse** , tokens_after_expr) ->



Parsing Factor as Parenthesized Expression

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

(match tokens_after_expr

with **Right_parenthesis** :: tokens_after_rparen ->

(**Parenthesized_Expr_as_Factor** **expr_parse** ,
tokens_after_rparen)



Error Cases

- What if no matching right parenthesis?

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

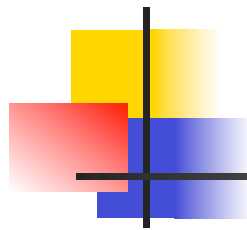
- What if no leading id or left parenthesis?

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

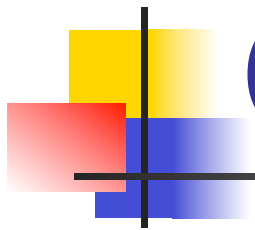


$(a + b) * c - d$

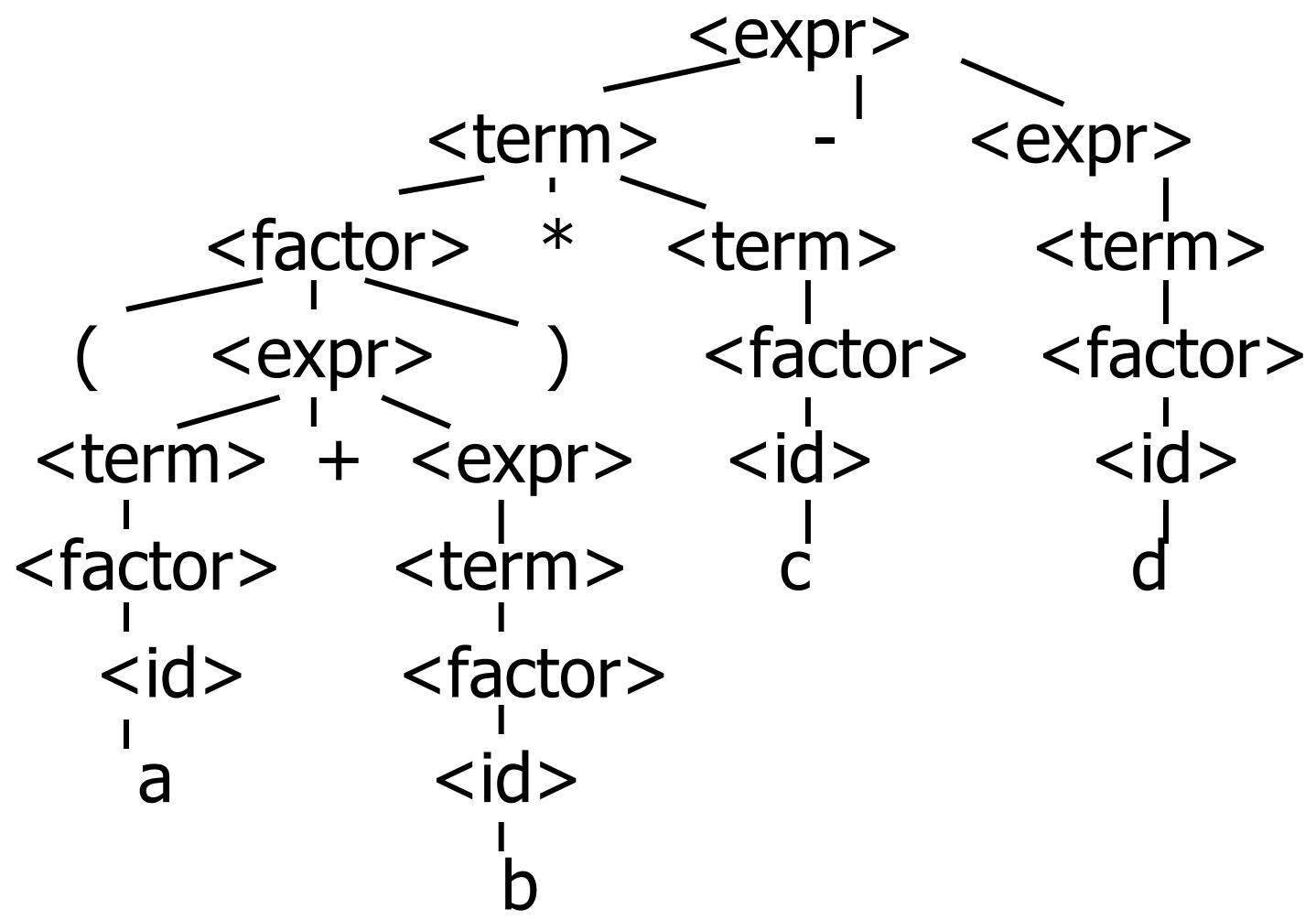
```
expr [Left_parenthesis; Id_token "a";  
      Plus_token; Id_token "b";  
      Right_parenthesis; Times_token;  
      Id_token "c"; Minus_token;  
      Id_token "d"];;
```

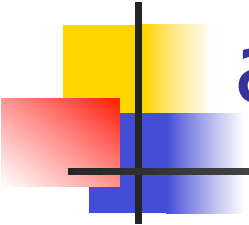

$$(a + b) * c - d$$

```
- : 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
```

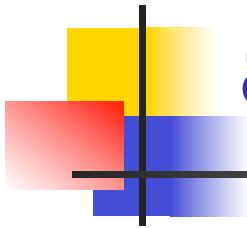
```
  (Factor_as_Term (Id_as_Factor "a"),
```

```
  Minus_Expr
```

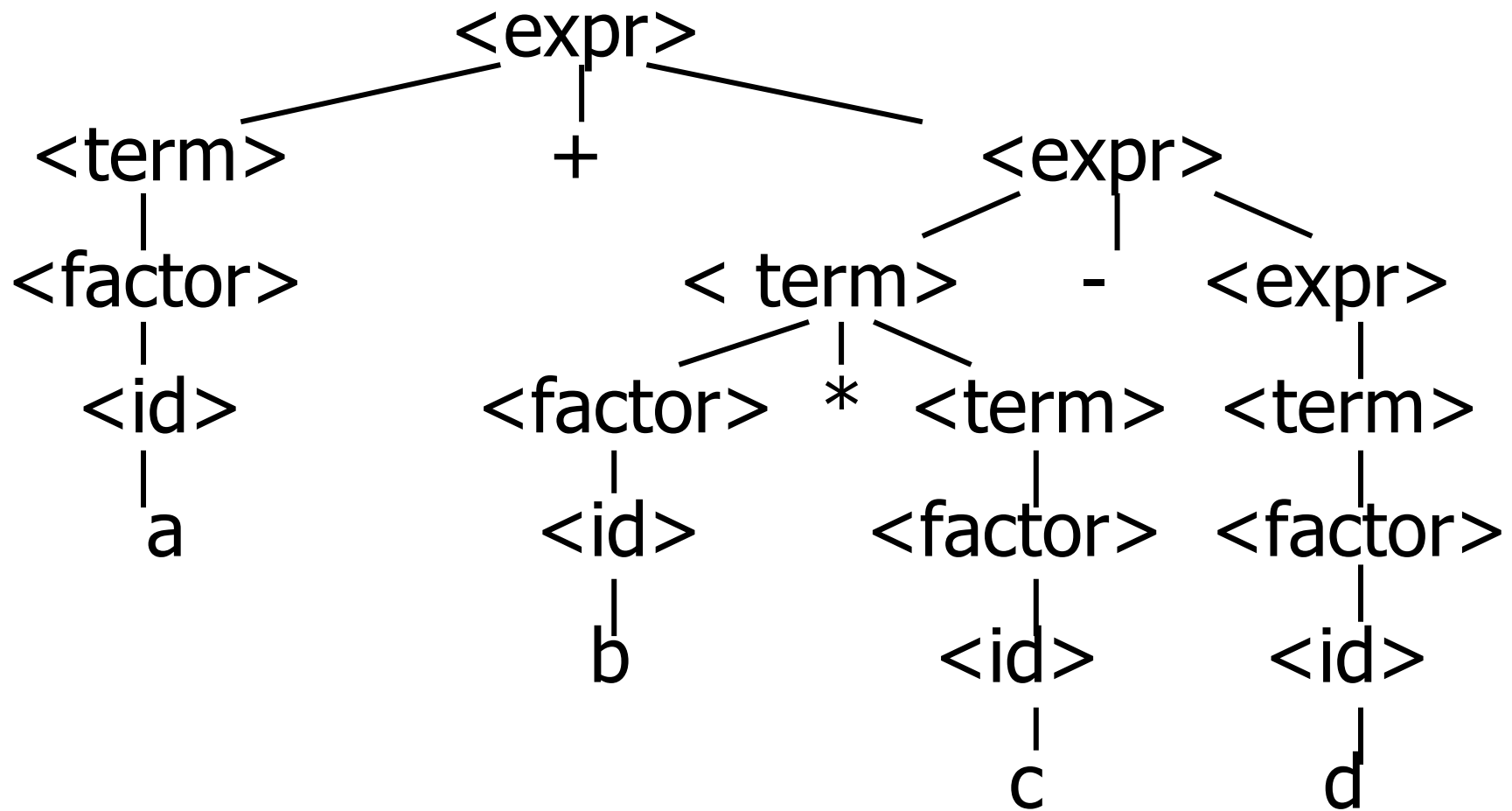
```
    (Mult_Term (Id_as_Factor "b", Factor_as_Term  
              (Id_as_Factor "c")),
```

```
    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 it was expecting a right parenthesis but it got to the end without finding one



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"; Left_parenthesis];;
```

- : 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"; Left_parenthesis])



Parsing Whole String

- Q: How to guarantee whole string parses?
- A: Check returned tokens 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, we don't want to create the entire list of tokens before we can start parsing
- We want to generate one token at a time and use it to make one step in parsing
- Can 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:

$A ::= Aw$

translates to a subroutine that loops forever

- Indirect Left Recursion:

$A ::= Bw$

$B ::= Av$

causes the same problem



Problems for Recursive-Descent Parsing

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



Pairwise Disjointedness Test

- For each rule

$$A ::= y$$

Calculate

$$\text{FIRST}(y) =$$

$$\{a \mid y \Rightarrow^* aw\} \cup \{\varepsilon \mid \text{if } y \Rightarrow^* \varepsilon\}$$

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



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

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

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

- Answer: Add new non-terminal and replace above rules by

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

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

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

$\langle e \rangle ::= \varepsilon$

- You are delaying the decision point



Example

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

Transform grammar
to:

$\langle S \rangle ::= \langle A \rangle a \langle B \rangle b$	$\langle S \rangle ::= \langle A \rangle a \langle B \rangle b$
$\langle A \rangle ::= \langle A \rangle b \mid b$	$\langle A \rangle ::= b \langle A1 \rangle$
$\langle B \rangle ::= a \langle B \rangle \mid a$	$\langle A1 \rangle ::= b \langle A1 \rangle \mid \varepsilon$
	$\langle B \rangle ::= a \langle B1 \rangle$
	$\langle B1 \rangle ::= a \langle B1 \rangle \mid \varepsilon$