Programming Languages and Compilers (CS 421)

Sasa Misailovic

4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421A

Based on slides by Elsa Gunter, which were inspired by earlier slides by Mattox Beckman, Vikram Adve, and Gul Agha

LR Parsing

General plan:
- Read tokens left to right (L)
- Create a rightmost derivation (R)

How is this possible?
- Start at the bottom (left) and work your way up
- Last step has only one non-terminal to be replaced so is right-most
- Working backwards, replace mixed strings by non-terminals
- Always proceed so that there are no non-terminals to the right of the string to be replaced

Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>

(0 + 1) + 0

Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>

(0 + 1) + 0

Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>

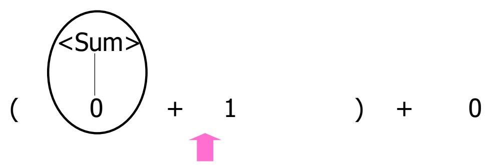
(0 + 1) + 0

Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>

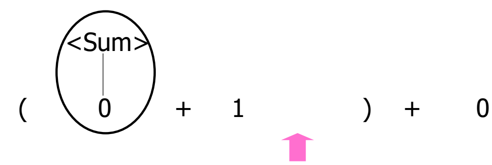
<Sum>

(0 + 1) + 0

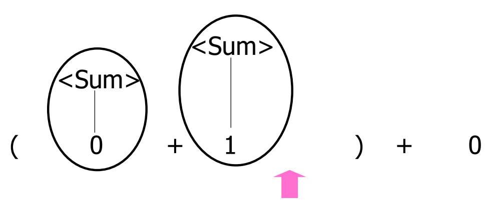
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



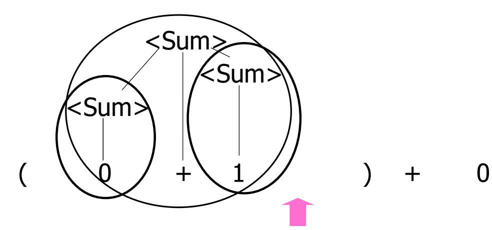
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



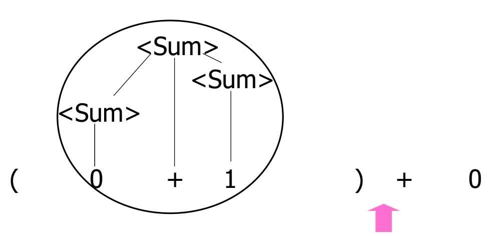
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



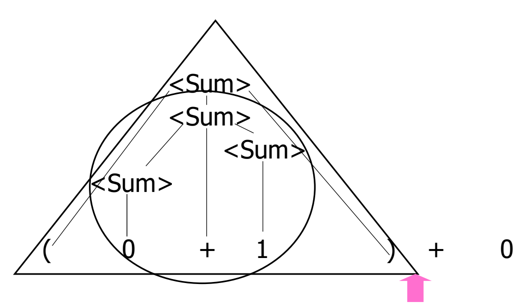
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



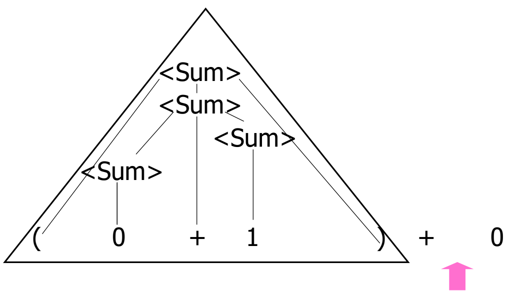
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



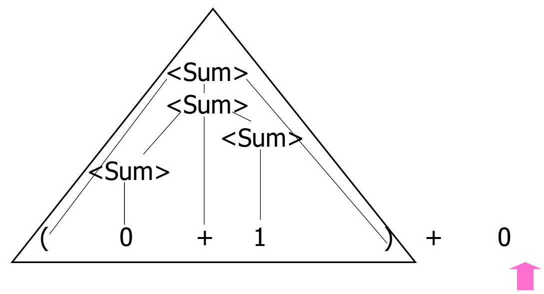
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



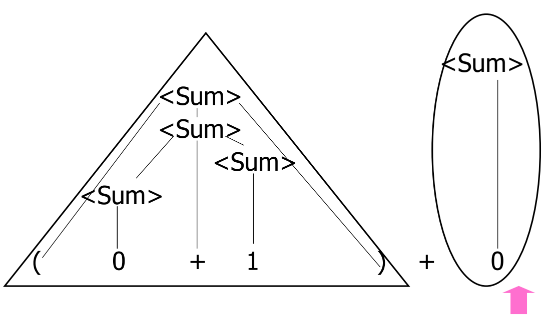
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



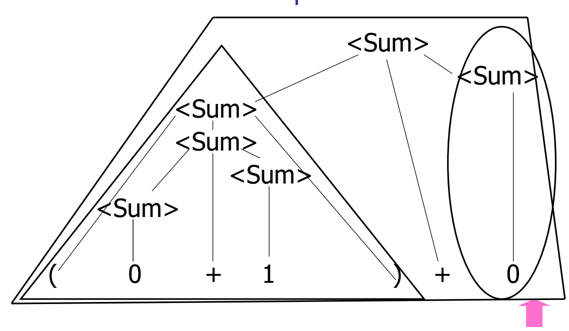
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



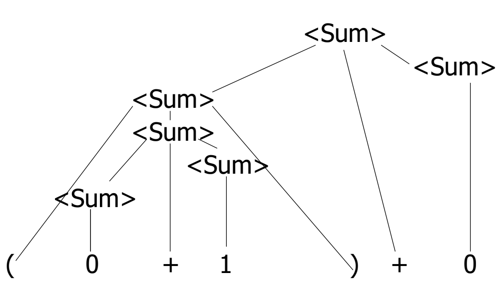
Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



Example: <Sum> ::= 0 | 1 | (<Sum>) | <Sum> + <Sum>



LR Parsing Tables

- Build a pair of tables, Action and Goto, from the grammar
  - This is the hardest part, we omit here
  - Rows labeled by states
  - For Action, columns labeled by terminals and “end-of-tokens” marker
    - (more generally strings of terminals of fixed length)
  - For Goto, columns labeled by non-terminals

Action and Goto Tables

- Given a state and the next input, Action table says either
  - **shift** and go to state *n*, or
  - **reduce** by production *k* (explained in a bit)
  - **accept** or **error**
- Given a state and a non-terminal, Goto table says
  - go to state *m*

LR(i) Parsing Algorithm

- Based on push-down automata
- Uses states and transitions (as recorded in Action and Goto tables)
- Uses a stack containing states, terminals and non-terminals

LR(i) Parsing Algorithm

0. Insure token stream ends in special “end-of-tokens” symbol
1. Start in state 1 with an empty stack
2. Push **state**(1) onto stack
3. → Look at next *i* tokens from token stream (*toks*) (don’t remove yet)
4. If top symbol on stack is **state**(*n*), look up action in Action table at (*n*, *toks*)

LR(i) Parsing Algorithm

5. If action = **shift** *m*,
   a) Remove the top token from token stream and push it onto the stack
   b) Push **state**(*m*) onto stack
   c) Go to step 3

LR(i) Parsing Algorithm

6. If action = **reduce** *k* where production *k* is E ::= u
   a) Remove 2 * length(u) symbols from stack (u and all the interleaved states)
   b) If new top symbol on stack is **state**(*m*), look up new state *p* in Goto(*m*,E)
   c) Push E onto the stack, then push **state**(*p*) onto the stack
   d) Go to step 3

LR(i) Parsing Algorithm

7. If action = **accept**
   - Stop parsing, return success
8. If action = **error**,
   - Stop parsing, return failure

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

0. Insure token stream ends in special "end-of-tokens" symbol
1. Start in state 1 with an empty stack
2. Push **state**(1) onto stack
3. → Look at next *i* tokens from token stream (*toks*) (don' t remove yet)

= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

5. If action = **shift** *m*,
   a) Remove the top token from token stream and push it onto the stack
   b) Push **state**(*m*) onto stack
   c) Go to step 3

= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= (● 0 + 1) + 0 shift
= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

6. If action = **reduce** *k* where production *k* is E ::= u
   a) Remove 2 * length(u) symbols from stack (u and all the interleaved states)
   b) If new top symbol on stack is **state**(*m*), look up new state *p* in Goto(*m*,E)
   c) Push E onto the stack, then push **state**(*p*) onto the stack
   d) Go to step 3

= (● 0 + 1) + 0 shift
= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

5. If action = **shift** *m*,
   a) Remove the top token from token stream and push it onto the stack
   b) Push **state**(*m*) onto stack
   c) Go to step 3

= (<Sum> ● + 1) + 0	
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= (<Sum> + ● 1) + 0	shift
= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

6. If action = **reduce** *k* where production *k* is E ::= u
   a) Remove 2 * length(u) symbols from stack (u and all the interleaved states)
   b) If new top symbol on stack is **state**(*m*), look up new state *p* in Goto(*m*,E)
   c) Push E onto the stack, then push **state**(*p*) onto the stack
   d) Go to step 3

=> (<Sum> + 1 ●) + 0	
= (<Sum> + ● 1) + 0	shift
= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=> (<Sum> + 1 ●) + 0	reduce
= (<Sum> + ● 1) + 0	shift
= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

6. If action = **reduce** *k* where production *k* is E ::= u
   a) Remove 2 * length(u) symbols from stack (u and all the interleaved states)
   b) If new top symbol on stack is **state**(*m*), look up new state *p* in Goto(*m*,E)
   c) Push E onto the stack, then push **state**(*p*) onto the stack
   d) Go to step 3

=> (<Sum> + <Sum> ●) + 0	
=> (<Sum> + 1 ●) + 0	reduce
= (<Sum> + ● 1) + 0	shift
= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=> (<Sum> + <Sum> ●) + 0	reduce
=> (<Sum> + 1 ●) + 0	reduce
= (<Sum> + ● 1) + 0	shift
= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= (<Sum> ●) + 0	shift
=> (<Sum> + <Sum> ●) + 0	reduce
=> (<Sum> + 1 ●) + 0	reduce
= (<Sum> + ● 1) + 0	shift
= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

=> (<Sum>) ● + 0	reduce
= (<Sum> ●) + 0	shift
=> (<Sum> + <Sum> ●) + 0	reduce
=> (<Sum> + 1 ●) + 0	reduce
= (<Sum> + ● 1) + 0	shift
= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= <Sum> ● + 0	shift
=> (<Sum>) ● + 0	reduce
= (<Sum> ●) + 0	shift
=> (<Sum> + <Sum> ●) + 0	reduce
=> (<Sum> + 1 ●) + 0	reduce
= (<Sum> + ● 1) + 0	shift
= (<Sum> ● + 1) + 0	shift
=> (0 ● + 1) + 0	reduce
= (● 0 + 1) + 0	shift
= ● (0 + 1) + 0	shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>

= <Sum> + ● 0 shift
= <Sum> ● + 0 shift
=> (<Sum>) ● + 0 reduce
= (<Sum> ●) + 0 shift
=> (<Sum> + <Sum> ●) + 0 reduce
=> (<Sum> + 1 ●) + 0 reduce
= (<Sum> + ● 1) + 0 shift
= (<Sum> ● + 1) + 0 shift
=> (0 ● + 1) + 0 reduce
= (● 0 + 1) + 0 shift
= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> =>
=> <Sum> + 0 ● reduce
= <Sum> + ● 0 shift
= <Sum> ● + 0 shift
=> (<Sum>) ● + 0 reduce
= (<Sum> ●) + 0 shift
=> (<Sum> + <Sum> ●) + 0 reduce
=> (<Sum> + 1 ●) + 0 reduce
= (<Sum> + ● 1) + 0 shift
= (<Sum> ● + 1) + 0 shift
=> (0 ● + 1) + 0 reduce
= (● 0 + 1) + 0 shift
= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> => <Sum> + <Sum > ● reduce
=> <Sum> + 0 ● reduce
= <Sum> + ● 0 shift
= <Sum> ● + 0 shift
=> (<Sum>) ● + 0 reduce
= (<Sum> ●) + 0 shift
=> (<Sum> + <Sum> ●) + 0 reduce
=> (<Sum> + 1 ●) + 0 reduce
= (<Sum> + ● 1) + 0 shift
= (<Sum> ● + 1) + 0 shift
=> (0 ● + 1) + 0 reduce
= (● 0 + 1) + 0 shift
= ● (0 + 1) + 0 shift

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

<Sum> ● => <Sum> + <Sum > ● reduce
=> <Sum> + 0 ● reduce
= <Sum> + ● 0 shift
= <Sum> ● + 0 shift
=> (<Sum>) ● + 0 reduce

7. If action = **accept**
   - Stop parsing, return success
8. If action = **error**,
   - Stop parsing, return failure

=> (0 ● + 1) + 0 reduce
= (● 0 + 1) + 0 shift
= ● (0 + 1) + 0 shift

Shift-Reduce Conflicts

- **Problem**: can't decide whether the action for a state and input character should be **shift** or **reduce**
- Caused by ambiguity in grammar
- Usually caused by lack of associativity or precedence information in grammar

Example: <Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>

-> <Sum> + <Sum> ● + 0
-> <Sum> + 1 ● + 0 reduce
-> <Sum> + ● 1 + 0 shift
-> <Sum> ● + 1 + 0 shift
-> 0 ● + 1 + 0 reduce
● 0 + 1 + 0 shift

Example - cont

- **Problem:** shift or reduce?
- You can shift-shift-reduce-reduce or reduce-shift-shift-reduce
- Shift first - right associative
- Reduce first- left associative

Reduce - Reduce Conflicts

- **Problem:** can't decide between two different rules to reduce by
- Again caused by ambiguity in grammar
- **Symptom:** RHS of one production suffix of another
- Requires examining grammar and rewriting it
- Harder to solve than shift-reduce errors

Example

- S ::= A | aB A ::= abc B ::= bc

abc ●	
ab ● c	shift
a ● bc	shift
● abc	shift

- Problem: reduce by B ::= bc then by S ::= aB, or by A::= abc then S::=A?

Ocamlyacc Output

LR Parsers:
- Great for arbitrary context-free grammars (and when the strings are in the language)
- But: Debugging and customizing is a pain!

LL 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)

LL Parsing via Recursive Descent Parsers

- 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

LL Parsing via Recursive Descent Parsers

- 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

<expr> ::= <term> | <term> + <expr> | <term> - <expr>

<term> ::= <id> | (<expr>)

```
type token = Id_token of string
           | Left_parenthesis | Right_parenthesis
           | Plus_token | Minus_token

type expr =
            Term_as_Expr of term
          | Plus_Expr of (term * expr)
          | Minus_Expr of (term * expr)
and term =
            Id_as_Term of string
          | Parenthesized_Expr_as_Term of expr
```


Going Back to Sample Grammar

<expr> ::= <term> | <term> + <expr> | <term> - <expr>

<term> ::= <id> | (<expr>)

In extended BNF notation :

<expr> ::= <term> [(+ | -) <expr>]

<term> ::= <id> | (<expr>)

Key observation: Parse tree of each rule has a unique leaf node

- That way the parser knows which rule to immediately apply

Parsing Lists of Tokens

- Create mutually recursive functions:
  - `expr : token list -> (expr * token list)`
  - `term : token list -> (factor * token list)`
- Each parses what it can and gives back the parse and remaining tokens

Parsing Factor

<term> ::= <id> | (<expr>)

```
let rec term tokens =
match tokens with
   (Id_token id_name) :: tokens_after_id ->
      ( Id_as_Factor  id_name, tokens_after_id)
 | Left_parenthesis :: tokens ->
    (match expr tokens
     with (expr_parse, tokens_after_expr) ->
        (match tokens_after_expr with
            Right_parenthesis :: tokens_after_r ->
              (Parenthesized_Expr_as_Term expr_parse,
               tokens_after_r) ));;
```


Parsing Factor as Id

<term> ::= <id> | (<expr>)

```
let rec factor tokens =
match tokens with
   (Id_token id_name :: tokens_after_id) ->
      ( Id_as_Term  id_name, tokens_after_id)
```

Parsing Factor

<term> ::= <id> | (<expr>)

```
let rec factor tokens =
match tokens with
   (Id_token id_name) :: tokens_after_id ->
      ( Id_as_term    id_name, tokens_after_id)
 | Left_parenthesis :: tokens ->
    (match expr tokens
     with (expr_parse, tokens_after_expr) ->
```


Parsing Factor

<term> ::= <id> | (<expr>)

```
let rec factor tokens =
match tokens with
   (Id_token id_name) :: tokens_after_id ->
      ( Id_as_Term    id_name, tokens_after_id)
 | Left_parenthesis :: tokens ->
    (match expr tokens
     with (expr_parse, tokens_after_expr) ->
         (match tokens_after_expr with
             Right_parenthesis :: tokens_after_r ->
               (Parenthesized_Expr_as_Term expr_parse,
                tokens_after_r) ))
```


Error Cases

- What if no matching right parenthesis?

```
(match tokens_after_expr with
     Right_parenthesis :: tokens_after_r ->
             (*...*)
   | _ -> raise (Failure "No matching rparen" )
```

- What if no leading id or left parenthesis?

```
match tokens with
    (Id_token id_name) :: tokens_after_id ->
           (*...*)
  | _ -> raise (Failure "No id or lparen" ));;
```


Parsing an Expression

<expr> ::= <term> [(+ | -) <expr>]

```
and expr tokens =
 (match (term tokens) with (term_parse,tokens_after) ->
    (match tokens_after with
        Plus_token  :: tokens_after_plus -> (*plus case*)
      | Minus_token :: tokens_after_minus -> (*minus case*)
      | _ -> (* this was either single term or error *)
```


Parsing an Expression

<expr> ::= <term> [(+ | -) <expr>]

```
and expr tokens =
 (match (term tokens) with (term_parse,tokens_after) ->
    (match tokens_after with
        Plus_token  :: tokens_after_plus -> (*plus case*)
         (match expr tokens_after_plus with
            (expr_parse, tokens_after_expr) ->
               (Plus_Expr (term_parse, expr_parse ),
                tokens_after_expr))
(* other cases ...*)
```


Parsing an Expression

<expr> ::= <term> [(+ | -) <expr>]

```
and expr tokens =
 (match (term tokens) with (term_parse,tokens_after) ->
    (match tokens_after with
        Plus_token  :: tokens_after_plus -> (*plus case*)
      | Minus_token :: tokens_after_minus -> (*minus case*)
          (match expr tokens_after_minus
           with ( expr_parse  , tokens_after_expr) ->
            (Minus_Expr(term_parse, expr_parse),
             tokens_after_expr))
```

Parsing an Expression

<expr> ::= <term> [(+ | -) <expr>]

```
and expr tokens =
 (match (term tokens) with (term_parse,tokens_after) ->
    (match tokens_after with
        Plus_token  :: tokens_after_plus -> (*plus case*)
      | Minus_token :: tokens_after_minus -> (*minus case*)
      | _ ->
         (Term_as_Expr term_parse, tokens_after_term))) ;;
```


(a + b) + c - d

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


(a + b + c - d

```
# expr [Left_parenthesis; Id_token
"a"; Plus_token; Id_token "b";
Plus_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 ((Id_as_Term "a"),
            Term_as_Expr ((Id_as_Term "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

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
- For each rule A ::= γ
  Calculate FIRST (γ) = $\{a \mid \gamma \Rightarrow^* a w\} \cup \{\varepsilon \mid \text{if } \gamma \Rightarrow^* \varepsilon\}$
- For each pair of rules A ::= γ and A ::= z, require FIRST(γ) $\cap$ FIRST(z) = { }

Example

Grammar:
<S> ::= <A> a <B> b
<A> ::= <A> b | b
<B> ::= a <B> | a

FIRST (<A> b) = {b}
FIRST (b) = {b}
Rules for <A> not pairwise disjoint

Eliminating Left Recursion

- Rewrite grammar to shift left recursion to right recursion
  - **Changes associativity**
- Given

<expr> ::= <expr> + <term> and
<expr> ::= <term>

- Add new non-terminal <e> and replace above rules with

<expr> ::= <term><e>
<e> ::= + <term><e> | ε

Factoring Grammar

- Test too strong: Can't handle
  <expr> ::= <term> [(+ | -) <expr>]
- Answer: Add new non-terminal and replace above rules by
  <expr> ::= <term><e>
  <e> ::= + <term><e>
  <e> ::= - <term><e>
  <e> ::= ε
- You are delaying the decision point

Example

Both <A> and <B> have problems:	Transform grammar to:
<S> ::= <A> a <B> b	<S> ::= <A> a <B> b
<A> ::= <A> b | b	<A> ::-= b<A1>
<B> ::= a <B> | a	<A1> :: b<A1> | ε
	<B> ::= a<B1>
	<B1> ::= a<B1> | ε