

Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

<http://www.cs.illinois.edu/class/cs421/>

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

10/20/11

1

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
 - Lexer called from parser

10/20/11

2

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)

10/20/11

3

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

10/20/11

4

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

10/20/11

5

Sample Grammar

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

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

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

10/20/11

6

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
```

10/20/11

7

## Parse Trees as Datatypes

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

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

10/20/11

8

## Parse Trees as Datatypes

```
<term> ::= <factor> | <factor> *
 <term>
 | <factor> / <term>
```

```
and term =
 Factor_as_Term of factor
 | Mult_Term of (factor * term)
 | Div_Term of (factor * term)
```

10/20/11

9

## Parse Trees as Datatypes

```
<factor> ::= <id> | (<expr>)
```

```
and factor =
 Id_as_Factor of string
 | Parenthesized_Expr_as_Factor of expr
```

10/20/11

10

## Parsing Lists of Tokens

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

10/20/11

11

## Parsing an Expression

```
<expr> ::= <term> [(+ | -) <expr>]
let rec expr tokens =
 (match term tokens
 with (term_parse , tokens_after_term) ->
 (match tokens_after_term
 with(Plus_token :: tokens_after_plus) ->
```

10/20/11

12

## Parsing an Expression

```
<expr> ::= <term> [(+ | -) <expr>]
let rec expr tokens =
 (match term tokens
 with (term_parse , tokens_after_term) ->
 (match tokens_after_term
 with (Plus_token :: tokens_after_plus) ->
```

10/20/11

13

## Parsing a Plus Expression

```
<expr> ::= <term> [(+ | -) <expr>]
let rec expr tokens =
 (match term tokens
 with (term_parse , tokens_after_term) ->
 (match tokens_after_term
 with (Plus_token :: tokens_after_plus) ->
```

10/20/11

14

## Parsing a Plus Expression

```
<expr> ::= <term> [(+ | -) <expr>]
let rec expr tokens =
 (match term tokens
 with (term_parse , tokens_after_term) ->
 (match tokens_after_term
 with (Plus_token :: tokens_after_plus) ->
```

10/20/11

15

## Parsing a Plus Expression

```
<expr> ::= <term> [(+ | -) <expr>]
let rec expr tokens =
 (match term tokens
 with (term_parse , tokens_after_term) ->
 (match tokens_after_term
 with (Plus_token :: tokens_after_plus) ->
```

10/20/11

16

## Parsing a Plus Expression

```
<expr> ::= <term> + <expr>

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

10/20/11

17

## Parsing a Plus Expression

```
<expr> ::= <term> + <expr>

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

10/20/11

18

## Building Plus Expression Parse Tree

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

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

10/20/11

19

## 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))
```

10/20/11

20

## 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))
```

10/20/11

21

## Parsing an Expression as a Term

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

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

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

10/20/11

22

## 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))
```

10/20/11

23

## 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) ->
```

10/20/11

24

## 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)

10/20/11

## 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" ) );;

10/20/11

26

$( a + b ) * c - d$

expr [Left\_parenthesis; Id\_token "a";  
Plus\_token; Id\_token "b";  
Right\_parenthesis; Times\_token;  
Id\_token "c"; Minus\_token;  
Id\_token "d"];;

10/20/11

27

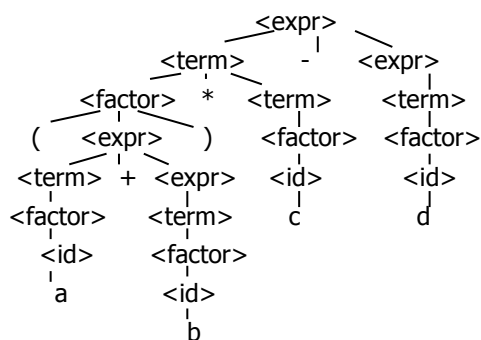
$( a + b ) * c - d$

```
- : 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"))),
 [])
```

10/20/11

28

$( a + b ) * c - d$



10/20/11

29

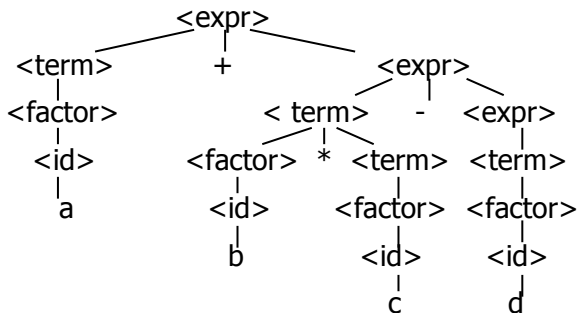
$a + b * c - d$

```
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")))),
 [])
```

10/20/11

30

a + b \* c - d



10/20/11

31

( 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

10/20/11

32

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"])

10/20/11

33

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

10/20/11

34

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
- Will use (token \* (unit -> token)) or (token \* (unit -> token option)) in place of token list

10/20/11

35

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

10/20/11

36

## 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

10/20/11

37

## Pairwise Disjointedness Test

- For each rule  $A ::= y$   
Calculate  $FIRST(y) = \{a \mid y \Rightarrow^* aw\} \cup \{\epsilon \mid y \Rightarrow^* \epsilon\}$
- For each pair of rules  $A ::= y$  and  $A ::= z$ , require  $FIRST(y) \cap FIRST(z) = \{\}$

10/20/11

38

## Example

Grammar:

$\langle S \rangle ::= \langle A \rangle a \langle B \rangle b$

$\langle A \rangle ::= \langle A \rangle b \mid b$

$\langle B \rangle ::= a \langle B \rangle \mid a$

$FIRST(\langle A \rangle b) = \{b\}$

$FIRST(b) = \{b\}$

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

10/20/11

39

## 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  $\langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle e \rangle$   
 $\langle e \rangle ::= + \langle \text{term} \rangle \langle e \rangle \mid \epsilon$

10/20/11

40

## 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  $\langle \text{expr} \rangle ::= \langle \text{term} \rangle \langle e \rangle$   
 $\langle e \rangle ::= + \langle \text{term} \rangle \langle e \rangle$   
 $\langle e \rangle ::= - \langle \text{term} \rangle \langle e \rangle$   
 $\langle e \rangle ::= \epsilon$
- You are delaying the decision point

10/20/11

41

## Example

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

Transform grammar to:

$\langle S \rangle ::= \langle A \rangle a \langle B \rangle b$      $\langle S \rangle ::= \langle A \rangle a \langle B \rangle b$

$\langle A \rangle ::= \langle A \rangle b \mid b$

$\langle A \rangle ::= b \langle A1 \rangle$

$\langle B \rangle ::= a \langle B \rangle \mid a$

$\langle A1 \rangle ::= b \langle A1 \rangle \mid \epsilon$

$\langle B \rangle ::= a \langle B1 \rangle$

$\langle B1 \rangle ::= a \langle B1 \rangle \mid \epsilon$

10/20/11

42