

## 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

11/6/12

1

## 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)

11/6/12

2

## 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

11/6/12

3

## 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

11/6/12

4

## Sample Grammar

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

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

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

11/6/12

5

## Tokens as OCaml Types

- + - \* / ( )  $\langle \text{id} \rangle$
  - Becomes an OCaml datatype
- type token =
- Id\_token of string
  - | Left\_parenthesis | Right\_parenthesis
  - | Times\_token | Divide\_token
  - | Plus\_token | Minus\_token

11/6/12

6

## 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)

11/6/12

7

## Parse Trees as Datatypes

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

and term =  
 Factor\_as\_Term of factor  
 | Mult\_Term of (factor \* term)  
 | Div\_Term of (factor \* term)

11/6/12

8

## 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

11/6/12

9

## 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

11/6/12

10

## 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 , tokens\_after\_term ) ->  
 (match tokens\_after\_term  
 with ( Plus\_token :: tokens\_after\_plus ) ->

11/6/12

11

## 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 , tokens\_after\_term ) ->  
 (match tokens\_after\_term  
 with ( Plus\_token :: tokens\_after\_plus ) ->

11/6/12

12

## 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 ) ->

```

11/6/12

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 ) ->

```

11/6/12

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 ) ->

```

11/6/12

15

## 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 ))

```

11/6/12

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 ))

```

11/6/12

17

## Building Plus Expression Parse Tree

```

<expr> ::= <term> + <expr>
(match expr tokens_after_plus
 with ( expr_parse , tokens_after_expr ) ->
  ( Plus_Expr ( term_parse , expr_parse ),
    tokens_after_expr ))

```

11/6/12

18

## 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))
```

11/6/12

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))
```

11/6/12

20

## 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

11/6/12

21

## 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)
```

11/6/12

22

## 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) ->
```

11/6/12

23

## 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)
```

11/6/12

## 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" );;
```

11/6/12

25

$(a + b) * c - d$

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

11/6/12

26

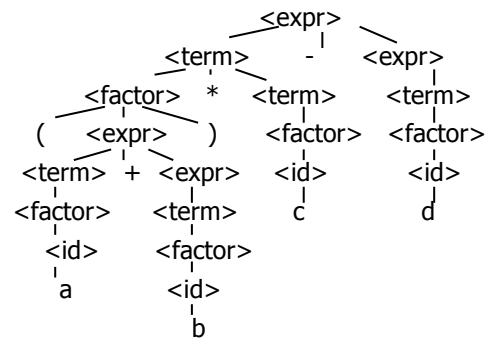
$(a + b) * c - d$

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

11/6/12

27

$(a + b) * c - d$



11/6/12

28

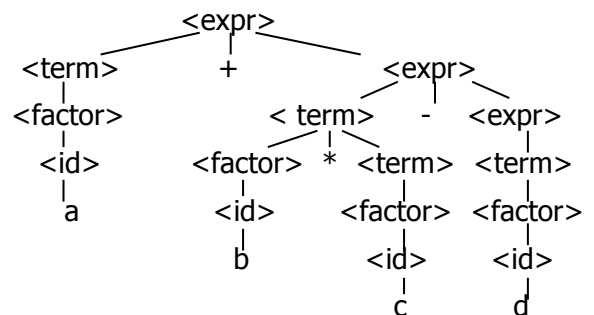
$a + b * c - d$

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

11/6/12


29

$a + b * c - d$



11/6/12

30

 ( 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

11/6/12

31

 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"])

11/6/12

32

## 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

11/6/12

33

## 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

11/6/12

34

## 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

11/6/12

35

## 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

11/6/12

36

## Pairwise Disjointness Test

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

11/6/12

37

## 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

11/6/12

38

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

11/6/12

39

## 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

11/6/12

40

## Example

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

Transform grammar  
to:

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

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

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

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

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

11/6/12

41

## Semantics

- Expresses the meaning of syntax
- Static semantics
  - Meaning based only on the form of the expression without executing it
  - Usually restricted to type checking / type inference

11/6/12

42

## Dynamic semantics

- Method of describing meaning of executing a program
- Several different types:
  - Operational Semantics
  - Axiomatic Semantics
  - Denotational Semantics

11/6/12

43

## Dynamic Semantics

- Different languages better suited to different types of semantics
- Different types of semantics serve different purposes

11/6/12

44

## Operational Semantics

- Start with a simple notion of machine
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (ie, following the *structure* of the program)
- Meaning of program is how its execution changes the state of the machine
- Useful as basis for implementations

11/6/12

45

## Axiomatic Semantics

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from *axioms* and *inference rules*
- Mainly suited to simple imperative programming languages

11/6/12

46

## Axiomatic Semantics

- Used to formally prove a property (*post-condition*) of the *state* (the values of the program variables) after the execution of program, assuming another property (*pre-condition*) of the state before execution
- Written :  
    {Precondition} Program {Postcondition}
- Source of idea of *loop invariant*

11/6/12

47

## Denotational Semantics

- Construct a function  $\mathcal{M}$  assigning a mathematical meaning to each program construct
- Lambda calculus often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs

11/6/12

48





## Natural Semantics

- Aka Structural Operational Semantics, aka “Big Step Semantics”
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like

$(C, m) \Downarrow m'$

or

$(E, m) \Downarrow v$