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

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

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

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

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

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

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

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)
```

Parse Trees as Datatypes

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

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

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

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) ->
```

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) ->
```

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) ->
```

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) ->
```

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) ->
```

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))
```

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))
```

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))

Parsing a Minus Expression

<expr> ::= <term> - <expr>

| (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

<expr> ::= <term> - <expr>

| (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

<factor> ::= <id>

```
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

<factor> ::= (<expr>)

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

Parsing Factor as Parenthesized Expression

<factor> ::= (<expr>)

(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"));;

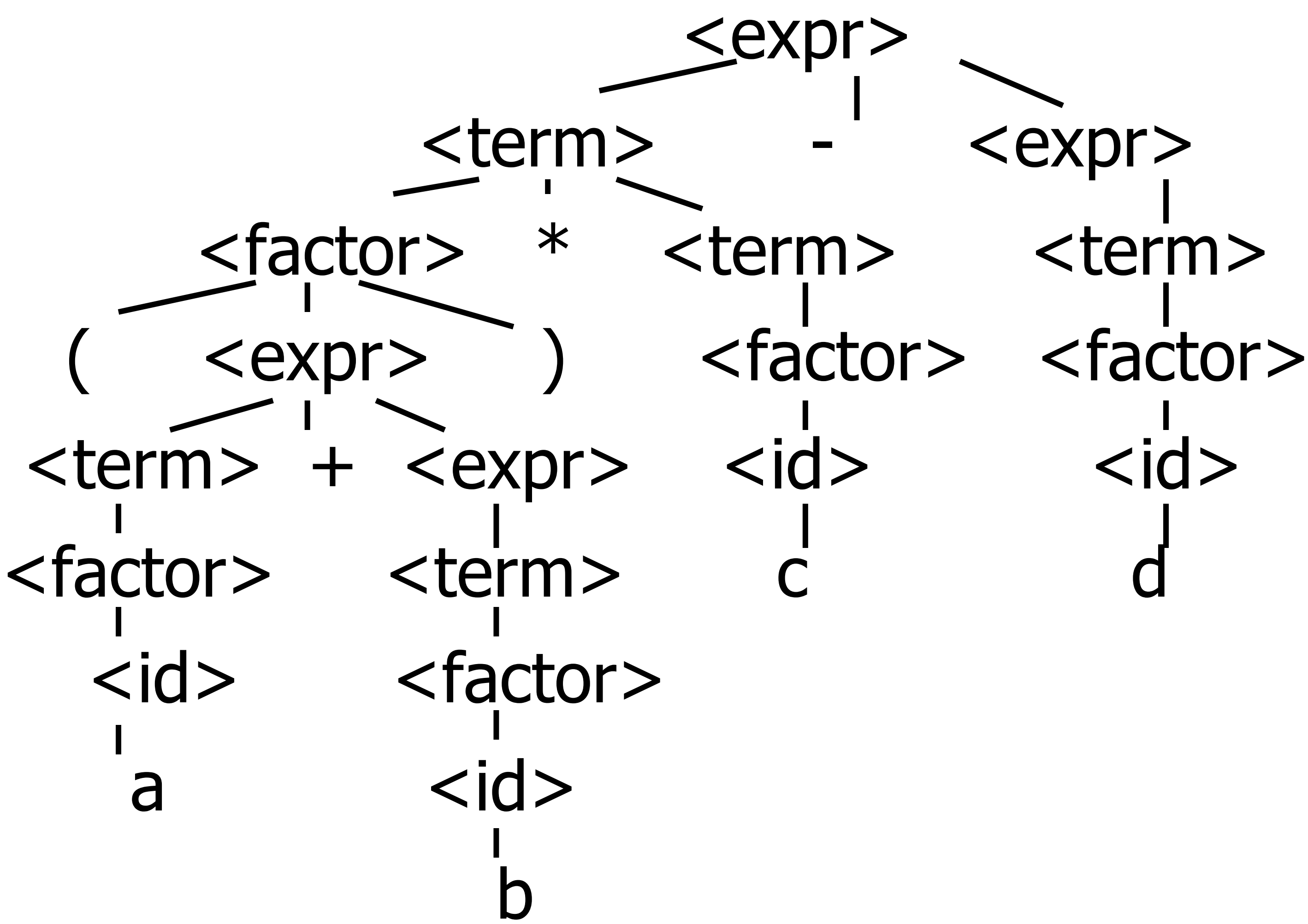
(a + b) * c - d

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"))),
 [])
```

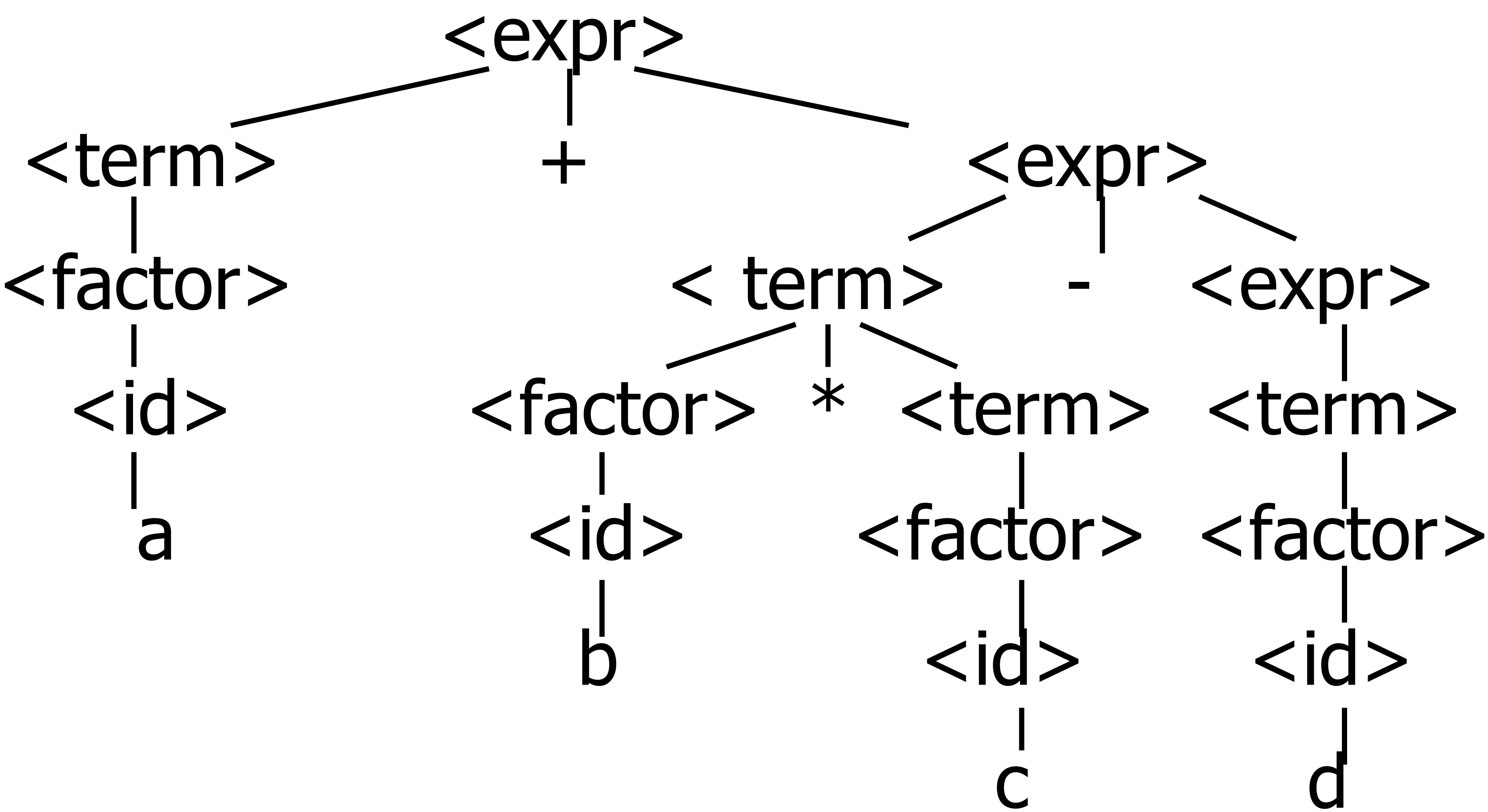
(a + b) * c – d


<expr>
<term>
-
<expr>
<factor>
*
<term>
<term>
(
<expr>
)
<factor>
<factor>
<term>
+
<expr>
<id>
<id>
<factor>
<term>
c
d
<id>
<factor>
a
<id>
b

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


<expr>
<term>
+
<expr>
<factor>
< term>
-
<expr>
<id>
<factor>
*
<term>
<term>
a
<id>
<factor>
<factor>
b
<id>
<id>
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"];;

- : 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: 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
- Will use (token * (unit -> token)) or (token * (unit -> 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 only 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

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 FIRST(*y*) ∩ 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:

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

<A> ::= <A> b | b

<B> ::= a <B> | a

Transform grammar to:

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

<A> ::-= b<A1>

<A1> :: b<A1> | ε

<B> ::= a<B1>

<B1> ::= a<B1> | ε

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

Dynamic semantics

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

Dynamic Semantics

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

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

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

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*

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

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

$$(E, m) \Downarrow v$$