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
Ambiguous Grammars and Languages

- A BNF grammar is *ambiguous* if its language contains strings for which there is more than one parse tree.
- If all BNF’s for a language are ambiguous then the language is *inherently ambiguous*.
Example: Ambiguous Grammar

\[ 0 + 1 + 0 \]
Example

What is the result for:

\[ 3 + 4 \times 5 + 6 \]
Example

What is the result for:

\[ 3 + 4 \times 5 + 6 \]

Possible answers:

- \[ 41 = ((3 + 4) \times 5) + 6 \]
- \[ 47 = 3 + (4 \times (5 + 6)) \]
- \[ 29 = (3 + (4 \times 5)) + 6 = 3 + ((4 \times 5) + 6) \]
- \[ 77 = (3 + 4) \times (5 + 6) \]
Example

What is the value of:

$$7 - 5 - 2$$
Example

What is the value of:

$7 - 5 - 2$

Possible answers:

- In Pascal, C++, SML assoc. left
  $7 - 5 - 2 = (7 - 5) - 2 = 0$

- In APL, associate to right
  $7 - 5 - 2 = 7 - (5 - 2) = 4$
Two Major Sources of Ambiguity

- Lack of determination of operator precedence
- Lack of determination of operator associativity

- Not the only sources of ambiguity
Disambiguating a Grammar

- Given ambiguous grammar $G$, with start symbol $S$, find a grammar $G'$ with same start symbol, such that
  
  \[
  \text{language of } G = \text{language of } G'
  \]

- Not always possible

- No algorithm in general
Disambiguating a Grammar

- Idea: Each non-terminal represents all strings having some property
- Identify these properties (often in terms of things that can’t happen)
- Use these properties to inductively guarantee every string in language has a unique parse
Steps to Grammar Disambiguation

- Identify the rules and a smallest use that display ambiguity
- Decide which parse to keep; why should others be thrown out?
- What syntactic restrictions on subexpressions are needed to throw out the bad (while keeping the good)?
- Add a new non-terminal and rules to describe this set of restricted subexpressions (called stratifying, or refactoring)
- Replace old rules to use new non-terminals
- Rinse and repeat
Example

- Ambiguous grammar:
  \[<\text{exp}> ::= 0 \mid 1 \mid <\text{exp}> + <\text{exp}> \mid <\text{exp}> * <\text{exp}>\]

- String with more than one parse:
  \[0 + 1 + 0\]
  \[1 * 1 + 1\]

- Source of ambiguity: associativity and precedence
Two Major Sources of Ambiguity

- Lack of determination of operator precedence
- Lack of determination of operator associativity

- Not the only sources of ambiguity
How to Enforce Associativity

- Have at most one recursive call per production

- When two or more recursive calls would be natural, leave right-most one for right associativity, left-most one for left associativity
Example

- `<Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)`

Becomes

- `<Sum> ::= <Num> | <Num> + <Sum>`
- `<Num> ::= 0 | 1 | (<Sum>)`
Operator Precedence

- Operators of highest precedence evaluated first (bind more tightly).

- Precedence for infix binary operators given in following table

- Needs to be reflected in grammar
<table>
<thead>
<tr>
<th></th>
<th>Fortan</th>
<th>Pascal</th>
<th>C/C++</th>
<th>Ada</th>
<th>SML</th>
</tr>
</thead>
<tbody>
<tr>
<td>highest</td>
<td>**</td>
<td>**</td>
<td>++, --</td>
<td>**</td>
<td>div, mod, /, *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*, /, div, mod</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>**</td>
<td>**</td>
<td>++, --</td>
<td>**</td>
<td>div, mod, /, *</td>
</tr>
<tr>
<td></td>
<td>*, /</td>
<td>+, -</td>
<td>*, /, %</td>
<td>*, /, mod</td>
<td>+, -, ^</td>
</tr>
<tr>
<td></td>
<td>+, -</td>
<td>+, -</td>
<td>+, -</td>
<td>+, -</td>
<td>::</td>
</tr>
</tbody>
</table>
First Example Again

- In any above language, $3 + 4 \times 5 + 6 = 29$
- In APL, all infix operators have same precedence
  - Thus we still don’t know what the value is (handled by associativity)
- How do we handle precedence in grammar?
Predence in Grammar

- Higher precedence translates to longer derivation chain
- Example:
  \[
  \text{<exp>} ::= 0 \mid 1 \mid <\text{exp}> + <\text{exp}> \\
  \mid <\text{exp}> \ast <\text{exp}>
  \]
- Becomes
  \[
  \text{<exp>} ::= <\text{mult}\_\text{exp}> \\
  \mid <\text{exp}> + <\text{mult}\_\text{exp}>
  \]
  \[
  <\text{mult}\_\text{exp}> ::= <\text{id}> \mid <\text{mult}\_\text{exp}> \ast <\text{id}>
  \]
  \[
  <\text{id}> ::= 0 \mid 1
  \]
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

\[
\begin{align*}
\text{<expr>} & ::= \text{<term>} \mid \text{<term>} + \text{<expr>} \\
& \quad \mid \text{<term>} - \text{<expr>}
\end{align*}
\]

\[
\begin{align*}
\text{<term>} & ::= \text{<factor>} \mid \text{<factor>} \ast \text{<term>} \\
& \quad \mid \text{<factor>} / \text{<term>}
\end{align*}
\]

\[
\begin{align*}
\text{<factor>} & ::= \text{id} \mid ( \text{<expr>} )
\end{align*}
\]
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

\[<\text{expr}> ::= <\text{term}> \mid <\text{term}> + <\text{expr}> \mid <\text{term}> - <\text{expr}>\]

type expr =
  Term\_as\_Expr of \text{term}
  \mid \text{Plus}\_Expr of (\text{term} \ast \text{expr})
  \mid \text{Minus}\_Expr of (\text{term} \ast \text{expr})
Parse Trees as Datatypes

\[
\langle \text{term} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{factor} \rangle \ast \langle \text{term} \rangle \\
\quad \mid \langle \text{factor} \rangle \div \langle \text{term} \rangle
\]

and term =

Factor\_as\_Term of factor

| Mult\_Term of (factor \ast term)

| Div\_Term of (factor \ast term)
Parse Trees as Datatypes

\[ \text{<factor>} ::= \text{id} \mid ( \text{<expr>} ) \]

and \text{factor} =

\[ \text{Id\_as\_Factor of string} \]
\[ \mid \text{Parenthesized\_Expr\_as\_Factor of expr} \]
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

\[
<\text{expr}> ::= <\text{term}> \ [(\ + \ |\ -\ ) \ <\text{expr}> \ ]
\]

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

\[<\text{expr}> ::= <\text{term}> \ [( + | - ) <\text{expr}> \]\]

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

\[
\text{<expr>} ::= \text{<term>} \ [(\ +\ |\ -\ )\ \text{<expr>}\ ]
\]

let rec expr tokens =

(match term tokens

with (\text{term\_parse},\ tokens\_after\_term) \rightarrow

(match tokens\_after\_term

with (\text{Plus\_token} :: tokens\_after\_plus) \rightarrow

)
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

\[ \text{<expr>} ::= \text{<term>} + \text{<expr>} \]

(match expr tokens_after_plus
with ( expr_parse , tokens_after_expr) ->
( Plus_Expr ( term_parse , expr_parse ),
tokens_after_expr)))
 Parsing a Plus Expression

\[ \text{<expr>} ::= \text{<term>} + \text{<expr>} \]

(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

\[<\text{expr}> ::= <\text{term}> - <\text{expr}>\]

| ( 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

\[ <\text{expr}> ::= <\text{term}> - <\text{expr}> \]

\[
| ( \text{Minus_token} :: tokens\_after\_minus ) -> \\
| ( \text{match expr tokens\_after\_minus with ( expr\_parse , tokens\_after\_expr )} -> \\
| ( \text{Minus\_Expr ( term\_parse , expr\_parse )}, \\
\text{tokens\_after\_expr}) \\
\]
Parsing an Expression as a Term

\[ \langle \text{expr} \rangle ::= \langle \text{term} \rangle \]

\[ \mid \_ \to (\text{Term\_as\_Expr} \text{\_parse} , \text{tokens\_after\_term})) \]

- Code for \textbf{term} is same except for replacing addition with multiplication and subtraction with division
<factor> ::= <id>

and factor tokens =
(match tokens
  with (Id_token id_name :: tokens_after_id) =
    ( Id_as_Factor id_name, tokens_after_id)
)

<factor> ::= ( <expr> )

| factor ( Left_parenthesis :: tokens) =

(match expr tokens

with ( expr_parse, tokens_after_expr) ->
Parsing Factor as Parenthesized Expression

\[
\text{<factor>} ::= ( \text{<expr>} )
\]

(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"))),
   [])

\( (a + b) \times c - d \)
\[ a + b \times c - d \]

```haskell
# 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
\( (a + b) \times c - d \)

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"]

10/30/14 50
Q: How to guarantee whole string parses?
A: Check returned tokens empty

```ocaml
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 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
  \[
  \text{FIRST} (y) = \{a \mid y \Rightarrow^* a w\} \cup \{\varepsilon \mid \text{if } y \Rightarrow^* \varepsilon\}
  \]
- For each pair of rules \( A ::= y \) and \( A ::= z \), require \( \text{FIRST}(y) \cap \text{FIRST}(z) = \{\} \)
Example

Grammar:

\[
\begin{align*}
\langle S \rangle & ::= \langle A \rangle \ a \ \langle B \rangle \ b \\
\langle A \rangle & ::= \langle A \rangle \ b \mid b \\
\langle B \rangle & ::= a \ \langle B \rangle \mid a
\end{align*}
\]

\[
\begin{align*}
\text{FIRST} (\langle A \rangle \ b) & = \{b\} \\
\text{FIRST} (b) & = \{b\}
\end{align*}
\]

Rules for \langle A \rangle 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
  \[ <expr> ::= <term><e> \]
  \[ <e> ::= + <term><e> | \varepsilon \]
Factoring Grammar

- Test too strong: Can’t handle
  \[ <\text{expr}> ::= <\text{term}> \left[ ( + | - ) <\text{expr}> \right] \]
- Answer: Add new non-terminal and replace above rules by
  \[ <\text{expr}> ::= <\text{term}> <\text{e}> \]
  \[ <\text{e}> ::= + <\text{term}> <\text{e}> \]
  \[ <\text{e}> ::= - <\text{term}> <\text{e}> \]
  \[ <\text{e}> ::= \varepsilon \]
- You are delaying the decision point
Example

Both \(<A>\) and \(<B>\) have problems:

\[
\begin{align*}
\langle S \rangle & ::= \langle A \rangle\ a\ \langle B \rangle\ b \\
\langle A \rangle & ::= \langle A \rangle\ b\ |\ b \\
\langle B \rangle & ::= a\ \langle B \rangle\ |\ a
\end{align*}
\]

Transform grammar to:

\[
\begin{align*}
\langle S \rangle & ::= \langle A \rangle\ a\ \langle B \rangle\ b \\
\langle A \rangle & ::= b\langle A1 \rangle \\
\langle B \rangle & ::= a\langle B1 \rangle \\
\langle A1 \rangle & ::= b\langle A1 \rangle\ |\ \varepsilon \\
\langle B1 \rangle & ::= a\langle B1 \rangle\ |\ \varepsilon
\end{align*}
\]
Ocamlyacc Input

- File format:

```%

    %{
        <header>
    }

    <declarations>

    %
    %

    <rules>

    %
    %

    <trailer>
```
Ocamlyacc <header>

- Contains arbitrary Ocaml code
- Typically used to give types and functions needed for the semantic actions of rules and to give specialized error recovery
- May be omitted

<footer> similar. Possibly used to call parser
Ocamlyacc <declarations>

- %token symbol ... symbol
  - Declare given symbols as tokens
- %token <type> symbol ... symbol
  - Declare given symbols as token constructors, taking an argument of type <type>
- %start symbol ... symbol
  - Declare given symbols as entry points; functions of same names in <grammar>.ml
Ocamlyacc <declareations>

- **%type** `<type> symbol ... symbol`
  Specify type of attributes for given symbols. Mandatory for start symbols

- **%left** `symbol ... symbol`

- **%right** `symbol ... symbol`

- **%nonassoc** `symbol ... symbol`
  Associate precedence and associativity to given symbols. Same line, same precedence; earlier line, lower precedence (broadest scope)
Ocamlyacc <rules>

- `nonterminal : symbol ... symbol { semantic_action }`  
  `...`  
  `symbol ... symbol { semantic_action }`  

;

- Semantic actions are arbitrary Ocaml expressions
- Must be of same type as declared (or inferred) for `nonterminal`
- Access semantic attributes (values) of symbols by position: $1$ for first symbol, $2$ to second ...
Example - Base types

(* File: expr.ml *)

```ocaml
type expr =
  Term_as.Expr of term
 | Plus.Expr of (term * expr)
 | Minus.Expr of (term * expr)
and term =
  Factor_as.Term of factor
 | Mult.Term of (factor * term)
 | Div.Term of (factor * term)
and factor =
  Id_as.Factor of string
 | Parenthesized.Expr_as.Factor of expr
```
Example - Lexer (exprlex.mll)

```ml
{ (*open Exprparse*) }
let numeric = ['0' - '9']
let letter = ['a' - 'z' 'A' - 'Z']
rule token = parse
  | "+" {Plus_token}
  | "-" {Minus_token}
  | "*" {Times_token}
  | "/" {Divide_token}
  | "(" {Left_parenthesis}
  | ")" {Right_parenthesis}
  | letter (letter|numeric|"_")* as id {Id_token id}
  | [' ' '	' '
'] {token lexbuf}
  | eof {EOL}
```
Example - Parser (exprparse.mly)

{% open Expr
%
%
%token <string> Id_token
%token Left_parenthesis Right_parenthesis
%token Times_token Divide_token
%token Plus_token Minus_token
%token EOL
%start main
%type <expr> main
%%
Example - Parser (exprparse.mly)

expr:
  
  term
    { Term_as_Expr $1 }
  | term Plus_token expr
    { Plus_Expr ($1, $3) }
  | term Minus_token expr
    { Minus_Expr ($1, $3) }
Example - Parser (exprpparse.mly)

term: 
  factor 
    { Factor_as_Term $1 } 
  | factor Times_token term 
    { Mult_Term ($1, $3) } 
  | factor Divide_token term 
    { Div_Term ($1, $3) }
factor:
  Id_token
  { Id_as_Factor $1 }
  | Left_parenthesis expr Right_parenthesis
  {Parenthesized_Expr_as_Factor $2 }
main:
  | expr EOL
  { $1 }
Example - Using Parser

```ml
# use "expr.ml";;
...
# use "exprparse.ml";;
...
# use "exprlex.ml";;
...
# let test s =
  let lexbuf = Lexing.from_string (s^"\n") in
  main token lexbuf;;
```
Example - Using Parser

# test "a + b";;

- : expr =

Plus.Expr

(Factor_as_Term (Id_as_Factor "a"),
Term_as.Expr (Factor_as_Term (Id_as_Factor "b")))