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

Possible answers:

41 = ((3 + 4) * 5) + 6
47 = 3 + (4 * (5 + 6))
29 = (3 + (4 * 5)) + 6 = 3 + ((4 * 5) + 6)
77 = (3 + 4) * (5 + 6)

Example

What is the result for:

3 + 4 * 5 + 6

Possible answers:

41 = ((3 + 4) * 5) + 6
47 = 3 + (4 * (5 + 6))
29 = (3 + (4 * 5)) + 6 = 3 + ((4 * 5) + 6)
77 = (3 + 4) * (5 + 6)

Example

What is the result for:

3 + 4 * 5 + 6

Possible answers:

41 = ((3 + 4) * 5) + 6
47 = 3 + (4 * (5 + 6))
29 = (3 + (4 * 5)) + 6 = 3 + ((4 * 5) + 6)
77 = (3 + 4) * (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

language of G = language of G’

Not always possible
No algorithm in general

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:

\[
<exp> ::= 0 | 1 | <exp> + <exp> \\
\quad | <exp> * <exp>
\]

String with more then 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

Precedence Table - Sample

<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>*, /, div, mod</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 * 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:
  \[<exp> ::= 0 | 1 | <exp> + <exp> | <exp> * <exp>\]
  
  Becomes
  \[<exp> ::= <mult_exp> | <exp> + <mult_exp>\]
  \[<mult_exp> ::= <id> | <mult_exp> * <id>\]
  \[<id> ::= 0 | 1\]

**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 <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 *)
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)

{ (*open Exprparse*) }
let numeric = ['0' - '9']
let letter = ['a' - 'z' 'A' - 'Z']
rule token = parse
| "+" {Plus_token}
| ";" {Minus_token}
| "/" {Divide_token}
| "/" {Right_parenthesis}
| letter (letter|numeric|"_")* as id  {Id_token id}
| 

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)

term:
  factor        
   { Factor_as_Term $1 } 
 | factor Times_token term    
   { Mult_Term ($1, $3) } 
 | factor Divide_token term    
   { Div_Term ($1, $3) }

Example - Parser (exprparse.mly)

factor:
  Id_token
   { Id_as_Factor $1 } 
 | Left_parenthesis expr Right_parenthesis
   {Parenthesized_Expr_as_Factor $2 }
main:
  | expr EOL
   { $1 }
```ocaml
# use "expr.ml";;
...
# use "exprparse.ml";;
...
# use "exprlex.ml";;
...
# let test s =
  let lexbuf = Lexing.from_string (s^"\n") in
       main token lexbuf;;
```

```
test "a + b";;
- : expr =
  Plus_Expr
    (Factor_as_Term (Id_as_Factor "a"),
     Term_as.Expr (Factor_as_Term
      (Id_as_Factor "b")))
```
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)

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
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>

(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

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

match \(\text{expr tokens}_{\text{after plus}}\)
with (\(\text{expr}_{\text{parse}}, \text{tokens}_{\text{after expr}}\)) ->
(\(\text{Plus}_\text{Expr} (\text{term}_{\text{parse}}, \text{expr}_{\text{parse}}), \text{tokens}_{\text{after expr}})\)

### Parsing a Minus Expression

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

| (\(\text{Minus}_\text{token} :: \text{tokens}_{\text{after minus}}\)) ->
  (match \(\text{expr tokens}_{\text{after minus}}\)
with (\(\text{expr}_{\text{parse}}, \text{tokens}_{\text{after expr}}\)) ->
(\(\text{Minus}_\text{Expr} (\text{term}_{\text{parse}}, \text{expr}_{\text{parse}}), \text{tokens}_{\text{after expr}})\)

### Parsing an Expression as a Term

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

| _ -> (\(\text{Term}_\text{as}_{\text{Expr}} \text{term}_{\text{parse}}, \text{tokens}_{\text{after term}})\)

\(n\) Code for \text{term} is same except for replacing addition with multiplication and subtraction with division

### Parsing Factor as Id

\[
\text{<factor>} ::= \text{id}
\]

and \(\text{factor tokens} =\)
(match \text{tokens}
with (\(\text{Id}_\text{token} \text{id}_{\text{name}} :: \text{tokens}_{\text{after id}}\) =
(\(\text{Id}_\text{as}_{\text{Factor}} \text{id}_{\text{name}}, \text{tokens}_{\text{after id}})\)

### Parsing Factor as Parenthesized Expression

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

| \(\text{factor} (\text{Left}_\text{parenthesis} :: \text{tokens}) =\)
  (match \text{expr tokens}
with (\(\text{expr}_{\text{parse}}, \text{tokens}_{\text{after expr}}\)) ->
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 * 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> + (factor)
 a <id> + <id> <id>
 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 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
  \[ \text{FIRST}(y) = \{a \mid y \Rightarrow^* aw \} \cup \{\varepsilon \mid y \Rightarrow^* \varepsilon\} \]
- For each pair of rules \( A ::= y \) and \( A ::= z \), require \( \text{FIRST}(y) \cap \text{FIRST}(z) = \{\} \)

Example

Grammar:
- \( <S> ::= <A> \ a \ <B> \ b \)
- \( <A> ::= <A> \ b \mid b \)
- \( <B> ::= a \ <B> \mid a \)

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
    - \( <expr> ::= <expr> + <term> \)
    - \( <expr> ::= <term> \)
  - Add new non-terminal \( <e> \) and replace above rules with
    - \( <expr> ::= <term><e> \)
    - \( <e> ::= + <term><e> \mid \varepsilon \)

Factoring Grammar

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

Example

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

Transform grammar to:

- \( <S> ::= <A> \ a \ <B> \ b \) to \( <S> ::= <A> \ a <B> \ b \)
- \( <A> ::= <A> \ b \mid b \) to \( <A> ::= b <A1> \)
- \( <B> ::= a <B> \mid a \) to \( <B> ::= a <B1> \mid \varepsilon \)
- \( <A1> ::= b <A1> \mid \varepsilon \) to \( <B1> ::= a <B1> \mid \varepsilon \)
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 <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 *)
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}
| "/" {Divide_token}
| "(" {Left_parenthesis}
| ")" {Right_parenthesis}
| letter (letter|numeric|"_")* as id  {Id_token id}
| [' ' '	' '
'] {token lexbuf}
| eof {EOL}
```

Example - Parser (exprparse.mly)

```ml
%{ 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)

```ml
expr: 
  term        
  { Term_as_Expr $1 } 
  | term Plus_token expr     
  { Plus_Expr ($1, $3) } 
  | term Minus_token expr    
  { Minus_Expr ($1, $3) }
```

Example - Parser (exprparse.mly)

```ml
term: 
  factor        
  { Factor_as_Term $1 } 
  | factor Times_token term    
  { Mult_Term ($1, $3) } 
  | factor Divide_token term    
  { Div_Term ($1, $3) }
```

Example - Parser (exprparse.mly)

```ml
factor: 
  Id_token       
  { Id_as_Token $1 } 
  | Left_parenthesis expr Right_parenthesis 
  {Parenthesized_EXPR_as_Factor $2 }
main: 
  expr EOL       
  { $1 }
```

Example - Using Parser

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