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

- 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
## 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>*, /,</td>
<td>++, --</td>
<td>**</td>
<td>div,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>div,</td>
<td></td>
<td></td>
<td>mod, /</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mod</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*, /</td>
<td></td>
<td>+, -</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*, /,</td>
<td></td>
<td>+, -</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>%</td>
<td></td>
<td>^</td>
</tr>
<tr>
<td>+, -</td>
<td></td>
<td></td>
<td>+, -</td>
<td></td>
<td>+, -</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+, -</td>
<td></td>
<td>::</td>
</tr>
</tbody>
</table>

10/4/07
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:
  \[ \texttt{<exp>} ::= 0 \mid 1 \mid \texttt{<exp>} + \texttt{<exp>} \mid \texttt{<exp>} * \texttt{<exp>} \]
- Becomes
  \[ \texttt{<exp>} ::= \texttt{<mult\_exp>} \mid \texttt{<exp>} + \texttt{<mult\_exp>} \]
  \[ \texttt{<mult\_exp>} ::= \texttt{id} \mid \texttt{<mult\_exp>} * \texttt{id} \]
  \[ \texttt{id} ::= 0 \mid 1 \]
Ocamlyacc Input

- File format:

```%{
    <header>
%
}
    <declarations>

%%%
    <rules>

%%%
    <trailer>```
Ocamlyacc \textit{<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
- \textit{<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 <declarations>

- %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** :
  
  \[
  \text{symbol} \ldots \text{symbol} \{ \text{semantic\_action} \} \\
  \text{...} \\
  \text{symbol} \ldots \text{symbol} \{ \text{semantic\_action} \} \\
  ;
  \]

- 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)

```
{ (*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}  
  | [' ' '	' '\n'] {token lexbuf}  
  | eof {EOL}  
```
Example - Parser (exprparse.mly)

```plaintext
{%
  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) }

10/29/15
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) }
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"))))
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

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

\[ \text{<term>} ::= \text{<factor>} | \text{<factor>} * \text{<term>} \]
\[ \quad | \text{<factor>} / \text{<term>} \]

\[ \text{<factor>} ::= \text{id} | ( \text{<expr>} ) \]
Tokens as OCaml Types

- + - * / ( ) <id>
- Becomes an OCaml datatype

```ocaml
type token =
    | Id_token of string
    | Left_parenthesis | Right_parenthesis
    | Times_token | Divide_token
    | Plus_token | Minus_token
```
Parse Trees as Datatypes

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

type expr = 
    Term_as_Expr of term \\
    | Plus_Expr of (term * expr) \\
    | Minus_Expr of (term * expr)
Parse Trees as Datatypes

\[
<\text{term}> ::= <\text{factor}> \mid <\text{factor}> \ast <\text{term}>
\mid <\text{factor}> \div <\text{term}>
\]

and \( \text{term} = \)

\[
\text{Factor\_as\_Term \ of \ factor} \mid \text{Mult\_Term \ of \ (factor \ast \ term)} \mid \text{Div\_Term \ of \ (factor \ast \ term)}
\]
Parse Trees as Datatypes

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

and \(\text{factor} = \)

\(\text{Id\_as\_Factor} \text{ of string}\)

\(\mid \text{Parenthesized\_Expr\_as\_Factor} \text{ of expr}\)
Will create three mutually recursive functions:

- \( \text{expr} : \text{token list} \rightarrow (\text{expr} \times \text{token list}) \)
- \( \text{term} : \text{token list} \rightarrow (\text{term} \times \text{token list}) \)
- \( \text{factor} : \text{token list} \rightarrow (\text{factor} \times \text{token list}) \)

Each parses what it can and gives back parse and remaining tokens
Parsing an Expression

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

let rec expr tokens =

\[(\text{match term tokens with\}}\]

\[\text{(term\_parse, tokens\_after\_term)} \rightarrow\]\n
\[(\text{match tokens\_after\_term with\}}\]

\[\text{(Plus\_token :: tokens\_after\_plus)} \rightarrow\]\n
Parsing an Expression

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

let rec expr tokens =

\[
(\text{match term tokens with ( term_parse, tokens_after_term) -> (match tokens_after_term with ( Plus_token :: tokens_after_plus) -> })
\]

10/29/15
Parsing a Plus 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> ::= <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) ->

   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 \text{expr tokens_after_plus}

with ( \text{expr_parse} , \text{tokens_after_expr}) ->

( \text{Plus_Expr} ( \text{term_parse} , \text{expr_parse} ),
\text{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 expr tokens_after_plus

with ( expr_parse , tokens_after_expr) ->

( Plus_Expr ( term_parse , expr_parse ),
  tokens_after_expr))
Parsing a Minus Expression

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

\[ | ( \text{Minus\_token} :: \text{tokens\_after\_minus}) \rightarrow \]
\[ (\text{match expr tokens\_after\_minus with ( expr\_parse , tokens\_after\_expr) } \rightarrow \]
\[ (\text{Minus\_Expr ( term\_parse , expr\_parse )}, \]
\[ \text{tokens\_after\_expr})) \]
Parsing a Minus Expression

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

\[
| (\text{Minus_token} :: \text{tokens_after_minus}) -> \\
| (\text{match expr tokens_after_minus with ( expr_parse , tokens_after_expr) -> }
| (\text{Minus.Expr ( term_parse , expr_parse ),
| tokens_after_expr}))
\]
Parsing an Expression as a Term

\[
<expr> ::= <term> \\
| _ -> (Term_as_Expr term_parse, tokens_after_term))
\]

- Code for \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))
Parsing Factor as Parenthesized Expression

\[ \langle \text{factor} \rangle ::= ( \langle \text{expr} \rangle ) \]

\[ | \text{factor} \ ( \text{Left_parenthesis} \ ::= \text{tokens} ) = \]

\[ (\text{match expr tokens with} \ (\text{expr_parse} \ , \ \text{tokens_after_after_expr}) \rightarrow) \]
\[ <\text{factor}> ::= ( <\text{expr}> ) \]

(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 * 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
( 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"])

10/29/15 63
Q: How to guarantee whole string parses?
A: Check returned tokens empty

```ml
let parse tokens =
  match expr tokens
  with (expr_parse, []) -> expr_parse
  | _ -> raise (Failure "No parse");
```

Fixes `<expr>` as start symbol
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 \((\text{token} \times (\text{unit} \to \text{token}))\) or \((\text{token} \times (\text{unit} \to \text{token option}))\) in place of token list.
Problems for Recursive-Descent Parsing

- Left Recursion:
  
  \[ A ::= A \omega \]
  
  translates to a subroutine that loops forever

- Indirect Left Recursion:
  
  \[ A ::= B \omega \]

  \[ B ::= A \omega \]

  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 \[ \text{FIRST}(y) \cap \text{FIRST}(z) = \{ \} \]
Grammar:
\[ <S> ::= <A> \ a \ <B> \ b \]
\[ <A> ::= <A> \ b \mid b \]
\[ <B> ::= a \ <B> \mid a \]

FIRST (\langle A \rangle \ b) = \{b\}
FIRST (b) = \{b\}
Rules for \langle A \rangle \text{ not pairwise disjoint}
Eliminating Left Recursion

- Rewrite grammar to shift left recursion to right recursion
  - Changes associativity

- Given

  \[
  \begin{align*}
  \langle expr \rangle & \ ::= \langle expr \rangle \ + \ \langle term \rangle \quad \text{and} \\
  \langle expr \rangle & \ ::= \ \langle term \rangle \\
  \end{align*}
  \]

- Add new non-terminal \( \langle e \rangle \) and replace above rules with

  \[
  \begin{align*}
  \langle expr \rangle & \ ::= \ \langle term \rangle \langle e \rangle \\
  \langle e \rangle & \ ::= \ + \ \langle term \rangle \langle e \rangle \quad | \quad \varepsilon
  \end{align*}
  \]
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:

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

Transform grammar to:

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

- **%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 *)

```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}
  | "-" {Minus_token}
  | "*" {Times_token}
  | "/" {Divide_token}
  | "(" {Left_parenthesis}
  | ")" {Right_parenthesis}
  | letter (letter|numeric|"_")* as id {Id_token id}
  | [' ' '	' '
'] {token lexbuf}
  | eof {EOL}
```
Example - Parser (exprpparse.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 (exprparse.mly)

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

10/29/15
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")))