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
General Input

{ header }
let ident = regexp ...
rule entrypoint [arg1... argn] = parse
  regexp { action }
| ...
| ...
| regexp { action }
and entrypoint [arg1... argn] = parse ...and ...
{ trailer }
Ocamllex Input

- *header* and *trailer* contain arbitrary ocaml code put at top and bottom of `<filename>.ml`

- `let ident = regexp ...` Introduces *ident* for use in later regular expressions
<filename>.ml contains one lexing function per `entrypoint`

- Name of function is name given for `entrypoint`
- Each entry point becomes an Ocaml function that takes \( n+1 \) arguments, the extra implicit last argument being of type `Lexing.lexbuf`

- `arg1... argn` are for use in `action`
Ocamllex Regular Expression

- Single quoted characters for letters: ‘a’
- _: (underscore) matches any letter
- Eof: special “end_of_file” marker
- Concatenation same as usual
- “string”: concatenation of sequence of characters
- $e_1 / e_2$: choice - what was $e_1 \lor e_2$
Ocamlllex Regular Expression

- \([c_1 - c_2]\): choice of any character between first and second inclusive, as determined by character codes
- \([^c_1 - c_2]\): choice of any character NOT in set
- \(e^*\): same as before
- \(e^+\): same as \(e\ e^*\)
- \(e?\): option - was \(e_1 \lor \varepsilon\)
Ocamllex Regular Expression

- $e_1 \# e_2$: the characters in $e_1$ but not in $e_2$; $e_1$ and $e_2$ must describe just sets of characters

- **ident**: abbreviation for earlier reg exp in
  
  ```
  let ident = regexp
  ```

- $e_1$ as **id**: binds the result of $e_1$ to **id** to be used in the associated **action**
More details can be found at

http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html
Example: test.mll

```ml
{ type result = Int of int | Float of float | String of string }

let digit = ['0'-'9']
let digits = digit +
let lower_case = ['a'-'z']
let upper_case = ['A'-'Z']
let letter = upper_case | lower_case
let letters = letter +
```
Example : test.mll

rule main = parse
    (digits)\'.\'digits as f  { Float (float_of_string f) } 
| digits as n             { Int (int_of_string n) } 
| letters as s            { String s} 
| _  { main lexbuf } 
{ let newlexbuf = (Lexing.from_channel stdin) in
print_newline ();
main newlexbuf  }
Example

# use "test.ml";;

...

val main : Lexing.lexbuf -> result = <fun>
val __ocaml_lex_main_rec : Lexing.lexbuf -> int ->
  result = <fun>

hi there 234 5.2
- : result = String "hi"

What happened to the rest?!?
Example

```ocaml
# let b = Lexing.from_channel stdin;;
# main b;;
hi 673 there
- : result = String "hi"
# main b;;
- : result = Int 673
# main b;;
- : result = String "there"
```
Your Turn

- Work on ML5
  - Add a few keywords
  - Implement booleans and unit
  - Implement Ints and Floats
  - Implement identifiers
Problem

- How to get lexer to look at more than the first token at one time?
  - Generally you DON’T want this

- Answer: *action* has to tell it to -- recursive calls
- Side Benefit: can add “state” into lexing
- Note: already used this with the _ case
Example

rule main = parse
  (digits) '.' digits as f { Float (float_of_string f) :: main lexbuf}
| digits as n { Int (int_of_string n) :: main lexbuf }
| letters as s { String s :: main lexbuf }
| eof { [] }
| _ { main lexbuf }
Example Results

hi there 234 5.2

- : result list = [String "hi"; String "there"; Int 234; Float 5.2]

Used Ctrl-d to send the end-of-file signal
Dealing with comments

First Attempt
let open_comment = "(*)"
let close_comment = "*)"
rule main = parse
  (digits) '.' digits as f { Float (float_of_string f) :: main lexbuf}
| digits as n { Int (int_of_string n) :: main lexbuf }
| letters as s { String s :: main lexbuf}
Dealing with comments

<table>
<thead>
<tr>
<th>open_comment</th>
<th>{ comment lexbuf }</th>
</tr>
</thead>
<tbody>
<tr>
<td>eof</td>
<td>{ [] }</td>
</tr>
<tr>
<td>_</td>
<td>{ main lexbuf }</td>
</tr>
<tr>
<td>_</td>
<td>{ comment lexbuf }</td>
</tr>
</tbody>
</table>

and comment = parse

    close_comment   | { main lexbuf } |
| _               | { main lexbuf }  |
Dealing with nested comments

rule main = parse ...
| open_comment        { comment 1 lexbuf}
| eof                  { [] }
| _ { main lexbuf }
and comment depth = parse
    open_comment        { comment (depth+1) lexbuf }
| close_comment       { if depth = 1
    then main lexbuf
    else comment (depth - 1) lexbuf }
| _                    { comment depth lexbuf }
Dealing with nested comments

rule main = parse
    (digits) '.' digits as f { Float (float_of_string f) :: main lexbuf}
| digits as n    { Int (int_of_string n) :: main lexbuf }
| letters as s   { String s :: main lexbuf }
| open_comment   { (comment 1 lexbuf}
| eof             { [] } 
| _   { main lexbuf }
Dealing with nested comments

and comment depth = parse

open_comment   { comment (depth+1) lexbuf }
| close_comment { if depth = 1
                   then main lexbuf
                   else comment (depth - 1) lexbuf }
| _            { comment depth lexbuf }
Types of Formal Language Descriptions

- Regular expressions, regular grammars
- Context-free grammars, BNF grammars, syntax diagrams
- Finite state automata

- Whole family more of grammars and automata – covered in automata theory
Sample Grammar

- Language: Parenthesized sums of 0’s and 1’s

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

- Start with a set of characters, $a, b, c, \ldots$
  - We call these *terminals*
- Add a set of different characters, $X, Y, Z, \ldots$
  - We call these *nonterminals*
- One special nonterminal $S$ called *start symbol*
BNF Grammars

- BNF rules (aka *productions*) have form
  \[ X ::= y \]
  where \( X \) is any nonterminal and \( y \) is a string of terminals and nonterminals

- BNF *grammar* is a set of BNF rules such that every nonterminal appears on the left of some rule
Sample Grammar

- **Terminals:** 0 1 + ( )
- **Nonterminals:** <Sum>
- **Start symbol:** <Sum>

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

Can be abbreviated as:
- <Sum> ::= 0 | 1 | <Sum> + <Sum> | (<Sum>)
BNF Derivations

- Given rules
  \[ X ::= yZw \quad \text{and} \quad Z ::= v \]
  we may replace \( Z \) by \( v \) to say
  \[ X \Rightarrow yZw \Rightarrow yvw \]

- Sequence of such replacements called \textit{derivation}

- Derivation called \textit{right-most} if always replace the right-most non-terminal
The meaning of a BNF grammar is the set of all strings consisting only of terminals that can be derived from the Start symbol.
BNF Derivations

- Start with the start symbol:

\(<\text{Sum}> \Rightarrow\)
BNF Derivations

- Pick a non-terminal

\(<\text{Sum}>\) =>
BNF Derivations

- Pick a rule and substitute:
  - `<Sum> ::= <Sum> + <Sum>`

  `<Sum>` => `<Sum> + <Sum>`
BNF Derivations

- Pick a non-terminal:

\[ \text{<Sum>} \rightarrow \text{<Sum>} + \text{<Sum>} \]
Pick a rule and substitute:

- `<Sum> ::= ( <Sum> )`
- `<Sum> => <Sum> + <Sum>`
- `=> ( <Sum> ) + <Sum>`
BNF Derivations

- Pick a non-terminal:

\[
<\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}>
\]

\[
\Rightarrow ( <\text{Sum}> ) + <\text{Sum}>
\]
BNF Derivations

- Pick a rule and substitute:
  - \( <\text{Sum}> ::= <\text{Sum}> + <\text{Sum}> \)
  - \( <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \)
    - \( \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \)
    - \( \Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \)
Pick a non-terminal:

\[
\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \\
\Rightarrow ( \langle \text{Sum} \rangle ) + \langle \text{Sum} \rangle \\
\Rightarrow ( \langle \text{Sum} \rangle + \langle \text{Sum} \rangle ) + \langle \text{Sum} \rangle
\]
BNF Derivations

- Pick a rule and substitute:
  - `<Sum> ::= 1`

`<Sum>` => `<Sum>` + `<Sum>`

=> ( `<Sum>` ) + `<Sum>`

=> ( `<Sum>` + `<Sum>` ) + `<Sum>`

=> ( `<Sum>` + 1 ) + `<Sum>`
BNF Derivations

pick a non-terminal:

<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>
Pick a rule and substitute:

- `<Sum> ::= 0`

`<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>
=> ( <Sum> + 1 ) + 0`
Pick a non-terminal:

\[ <\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + 1 ) + <\text{Sum}> \]
\[ \Rightarrow ( <\text{Sum}> + 1 ) + 0 \]
BNF Derivations

- Pick a rule and substitute
  - `<Sum> ::= 0`

```
<Sum>  =>  <Sum> + <Sum>
  => ( <Sum> ) + <Sum>
  => ( <Sum> + <Sum> ) + <Sum>
  => ( <Sum> + 1 ) + <Sum>
  => ( <Sum> + 1 ) 0
  => ( 0 + 1 ) + 0
```
(0 + 1) + 0 is generated by grammar

<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>
=> ( <Sum> + 1 ) + 0
=> (0 + 1) + 0
\[
<\text{Sum}\> ::= 0 \mid 1 \mid <\text{Sum}\> + <\text{Sum}\> \mid (<\text{Sum}\>)
\]

\[
<\text{Sum}\> = >
\]
Regular Grammars

- Subclass of BNF
- Only rules of form
  \(<\text{nonterminal}> ::= <\text{terminal}> <\text{nonterminal}> \) or
  \(<\text{nonterminal}> ::= <\text{terminal}> \) or
  \(<\text{nonterminal}> ::= \varepsilon \)
- Defines same class of languages as regular expressions
- Important for writing lexers (programs that convert strings of characters into strings of tokens)
Example

- Regular grammar:
  
  \[
  \langle \text{Balanced} \rangle ::= \varepsilon \\
  \langle \text{Balanced} \rangle ::= 0\langle \text{OneAndMore} \rangle \\
  \langle \text{Balanced} \rangle ::= 1\langle \text{ZeroAndMore} \rangle \\
  \langle \text{OneAndMore} \rangle ::= 1\langle \text{Balanced} \rangle \\
  \langle \text{ZeroAndMore} \rangle ::= 0\langle \text{Balanced} \rangle 
  \]

- Generates even length strings where every initial substring of even length has same number of 0’s as 1’s
Extended BNF Grammars

- Alternatives: allow rules of from $X ::= y/z$
  - Abbreviates $X ::= y, X ::= z$

- Options: $X ::= y[v]z$
  - Abbreviates $X ::= yvz, X ::= yz$

- Repetition: $X ::= y\{v\}*z$
  - Can be eliminated by adding new nonterminal $V$ and rules $X ::= yz, X ::= yVz, V ::= v, V ::= vW$
Parse Trees

- Graphical representation of derivation
- Each node labeled with either non-terminal or terminal
- If node is labeled with a terminal, then it is a leaf (no sub-trees)
- If node is labeled with a non-terminal, then it has one branch for each character in the right-hand side of rule used to substitute for it
Example

- Consider grammar:
  
  \[
  \begin{align*}
  &\text{<exp>} ::= \text{<factor>} \\
  &\quad \mid \text{<factor>} + \text{<factor>} \\
  &\text{<factor>} ::= \text{<bin>} \\
  &\quad \mid \text{<bin>} \ast \text{<exp>} \\
  &\text{<bin>} ::= 0 \mid 1
  \end{align*}
  \]

- Problem: Build parse tree for 1 * 1 + 0 as an <exp>
Example cont.

- \( 1 \times 1 + 0: \quad \langle \text{exp} \rangle \)

\( \langle \text{exp} \rangle \) is the start symbol for this parse tree
Example cont.

1 * 1 + 0: \[ \langle \text{exp} \rangle \]
\[ \langle \text{factor} \rangle \]

Use rule: \( \langle \text{exp} \rangle ::= \langle \text{factor} \rangle \)
Example cont.

1 * 1 + 0:  \texttt{<exp>}

\texttt{<factor>}

\texttt{<bin> \ast <exp>}

Use rule:  \texttt{<factor> ::= <bin> \ast <exp>}
Example cont.

- $1 \times 1 + 0$: 
  ```
  <exp>
  \text{<factor>}
  \text{<bin> \times <exp>}
  1 \text{ <factor>} + \text{<factor>}
  ```

Use rules:

- $<\text{bin}> ::= 1$ and
- $<\text{exp}> ::= <\text{factor}> + <\text{factor}>$
Example cont.

1 * 1 + 0:  

Use rule: <factor> ::= <bin>
Example cont.

1 * 1 + 0:  

```
<exp>
  <factor>
    <bin> * <exp>
      1 <factor> + <factor>
        <bin> <bin>
          1 0

Use rules:  <bin> ::= 1 | 0
```
Example cont.

\[ 1 \times 1 + 0: \]

```
<factor>
  <bin>  *  <exp>
  1  <factor>  +  <factor>
        <bin>  <bin>
        1    0
```

Fringe of tree is string generated by grammar
Your Turn: $1 \times 0 + 0 \times 1$

```
<exp>
  /
 <fact>  +  <fact>
  /
 <b>  *  <e> <b>  *  <e>
```


Parse Tree Data Structures

- Parse trees may be represented by OCaml datatypes
- One datatype for each nonterminal
- One constructor for each rule
- Defined as mutually recursive collection of datatype declarations
Example

- Recall grammar:
  \[
  \begin{align*}
  \langle \text{exp} \rangle & \ ::= \langle \text{factor} \rangle \ | \ \langle \text{factor} \rangle \ + \ \langle \text{factor} \rangle \\
  \langle \text{factor} \rangle & \ ::= \langle \text{bin} \rangle \ | \ \langle \text{bin} \rangle \ * \ \langle \text{exp} \rangle \\
  \langle \text{bin} \rangle & \ ::= \ 0 \ | \ 1
  \end{align*}
  \]

- type \( \text{exp} = \text{Factor2Exp of factor} \)
  \[
  \begin{align*}
  & \ | \ \text{Plus of factor} \ * \ \text{factor} \\
  \text{and factor} & = \text{Bin2Factor of bin} \\
  & \ | \ \text{Mult of bin} \ * \ \text{exp} \\
  \text{and bin} & = \text{Zero} \ | \ \text{One}
  \end{align*}
  \]
Example cont.

1 * 1 + 0:  

```
<exp>  
|    
<factor>  
|   *  
<exp>  
|   +  
<factor>  
|        
<bin>  
|    
1  
|    
<bin>  
|    
1  
|    
<bin>  
|    
0  
```

1 * 1 + 0:
Example cont.

- Can be represented as

\[
\text{Factor2Exp} \\
(\text{Mult}(\text{One}, \text{Plus}(\text{Bin2Factor One, Bin2Factor Zero})))
\]
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:
  \[ <\text{exp}> ::= 0 \mid 1 \mid <\text{exp}> + <\text{exp}> \]
  \[ \mid <\text{exp}> \ast <\text{exp}> \]

- 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

- \(<\text{Sum}\> ::= 0 \mid 1 \mid <\text{Sum}\> + <\text{Sum}\>
  \mid (<\text{Sum}\>)\)

- Becomes

- \(<\text{Sum}\> ::= <\text{Num}\> \mid <\text{Num}\> + <\text{Sum}\>
- <\text{Num}\> ::= 0 \mid 1 \mid (<\text{Sum}\>)\)
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><strong>highest</strong></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></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>* , /, mod</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>
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
  \( <\text{exp}> ::= <\text{mult}\_\text{exp}> \mid <\text{exp}> + <\text{mult}\_\text{exp}> \)
  \( <\text{mult}\_\text{exp}> ::= <\text{id}> \mid <\text{mult}\_\text{exp}> \ast <\text{id}> \)
  \( <\text{id}> ::= 0 \mid 1 \)
Parser Code

- `<grammar>.ml` defines one parsing function per entry point
- Parsing function takes a lexing function (lexer buffer to token) and a lexer buffer as arguments
- Returns semantic attribute of corresponding entry point
Ocamlyacc Input

- File format:

```plaintext
%{
  <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>

- \textit{nonterminal} :
  
  \texttt{symbol ... symbol \{ semantic\_action \}}

- [...]
  
  \texttt{symbol ... symbol \{ semantic\_action \}}

;,

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

(* File: expr.ml *)

```ml
let rec parse_expr expr =
    match expr with
    | Expr_of_term term -> Term_of_expr_of_term term
    | Plus_of_expr (term * expr) -> Plus_of_expr (term * expr)
    | Minus_of_expr (term * expr) -> Minus_of_expr (term * expr)

and term =
    match term with
    | Expr_of_term factor -> Term_of_term_of_factor factor
    | Mult_of_term (factor * term) -> Mult_of_term (factor * term)
    | Div_of_term (factor * term) -> Div_of_term (factor * term)

and factor =
    match factor with
    | Id_of_factor string -> Id_of_factor string
    | Parenthesized_of_expr_of_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)

```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 (exprprase.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) }
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 }
Example - Using Parser

# 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

```plaintext
# test "a + b";;
- : expr =

Plus_Expr

(Factor_as_Term (Id_as_Factor "a"),
 Term_as_Expr (Factor_as_Term (Id_as_Factor "b")))
```
LR Parsing

- Read tokens left to right (L)
- Create a rightmost derivation (R)
- How is this possible?
- Start at the bottom (left) and work your way up
- Last step has only one non-terminal to be replaced so is right-most
- Working backwards, replace mixed strings by non-terminals
- Always proceed so that there are no non-terminals to the right of the string to be replaced
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (\langle\text{Sum}\rangle)\)
\mid \langle\text{Sum}\rangle + \langle\text{Sum}\rangle

\langle\text{Sum}\rangle \Rightarrow

= \circ (0 + 1) + 0 \quad \text{shift}
Example: $<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)$

$\mid <\text{Sum}> + <\text{Sum}>$

$<\text{Sum}> \Rightarrow$

$= (0 + 1) + 0$

shift

$= (0 + 1) + 0$

shift
Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)
\mid <\text{Sum}> + <\text{Sum}>\)

\(<\text{Sum}> \Rightarrow \)

\begin{align*}
\Rightarrow & (0 + 1) + 0 & \text{reduce} \\
= & (0 + 1) + 0 & \text{shift} \\
= & (0 + 1) + 0 & \text{shift}
\end{align*}
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (\langle\text{Sum}\rangle) \mid \langle\text{Sum}\rangle + \langle\text{Sum}\rangle\) \\
\langle\text{Sum}\rangle \Rightarrow \\
\begin{align*}
  &= (\langle\text{Sum}\rangle \bullet + 1) + 0 \quad \text{shift} \\
  &\Rightarrow (0 \bullet + 1) + 0 \quad \text{reduce} \\
  &= (\bullet 0 + 1) + 0 \quad \text{shift} \\
  &= \bullet (0 + 1) + 0 \quad \text{shift}
\end{align*}
Example: \(<\text{Sum}> = 0 | 1 | (<\text{Sum}>))
| <\text{Sum}> + <\text{Sum}>

\(<\text{Sum}> \Rightarrow \)

\[
\begin{align*}
= & ( <\text{Sum}> + 1 ) + 0 & \text{shift} \\
= & ( <\text{Sum}> \circ + 1 ) + 0 & \text{shift} \\
=> & ( 0 \circ + 1 ) + 0 & \text{reduce} \\
= & ( 0 + 1 ) + 0 & \text{shift} \\
= & 0 ( 0 + 1 ) + 0 & \text{shift}
\end{align*}
\]
Example: $<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)$

\[
<\text{Sum}> \quad => \\
\]

\[
=> ( <\text{Sum}> + 1 ) + 0 \quad \text{reduce} \\
= ( <\text{Sum}> + 1 ) + 0 \quad \text{shift} \\
= ( <\text{Sum}> + 1 ) + 0 \quad \text{shift} \\
=> ( 0 + 1 ) + 0 \quad \text{reduce} \\
= ( 0 + 1 ) + 0 \quad \text{shift} \\
= ( 0 + 1 ) + 0 \quad \text{shift}
\]
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (\langle\text{Sum}\rangle) \mid \langle\text{Sum}\rangle + \langle\text{Sum}\rangle\)

\[\langle\text{Sum}\rangle \Rightarrow\]

\[\Rightarrow (\langle\text{Sum}\rangle + \langle\text{Sum}\rangle \bullet ) + 0 \quad \text{(reduce)}\]
\[\Rightarrow (\langle\text{Sum}\rangle + 1 \bullet ) + 0 \quad \text{(reduce)}\]
\[= (\langle\text{Sum}\rangle + \bullet 1 ) + 0 \quad \text{(shift)}\]
\[= (\langle\text{Sum}\rangle \bullet + 1 ) + 0 \quad \text{(shift)}\]
\[\Rightarrow (0 \bullet + 1 ) + 0 \quad \text{(reduce)}\]
\[= (\bullet 0 + 1 ) + 0 \quad \text{(shift)}\]
\[= \bullet (0 + 1 ) + 0 \quad \text{(shift)}\]
Example: \(<\text{Sum}> = 0 \mid 1 \mid (\langle\text{Sum}\rangle) \mid \langle\text{Sum}\rangle + \langle\text{Sum}\rangle\rangle

\langle\text{Sum}\rangle \Rightarrow

= (\langle\text{Sum}\rangle \bullet) + 0 \quad \text{shift}
=> (\langle\text{Sum}\rangle + \langle\text{Sum}\rangle \bullet) + 0 \quad \text{reduce}
=> (\langle\text{Sum}\rangle + 1 \bullet) + 0 \quad \text{reduce}
= (\langle\text{Sum}\rangle + \bullet 1) + 0 \quad \text{shift}
= (\langle\text{Sum}\rangle \bullet + 1) + 0 \quad \text{shift}
=> (0 \bullet + 1) + 0 \quad \text{reduce}
= (\bullet 0 + 1) + 0 \quad \text{shift}
= \bullet (0 + 1) + 0 \quad \text{shift}
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (\langle\text{Sum}\rangle) \mid \text{<Sum>} + \text{<Sum}>\)

\(<\text{Sum}\> \implies \)

\[
\begin{align*}
\implies & (\langle\text{Sum}\rangle) \bullet + 0 & \text{reduce} \\
= & (\langle\text{Sum}\rangle \bullet) + 0 & \text{shift} \\
\implies & (\langle\text{Sum}\rangle + \langle\text{Sum}\rangle \bullet) + 0 & \text{reduce} \\
\implies & (\langle\text{Sum}\rangle + 1 \bullet) + 0 & \text{reduce} \\
= & (\langle\text{Sum}\rangle + \bullet 1) + 0 & \text{shift} \\
= & (\langle\text{Sum}\rangle \bullet + 1) + 0 & \text{shift} \\
\implies & (0 \bullet + 1) + 0 & \text{reduce} \\
= & (\bullet 0 + 1) + 0 & \text{shift} \\
= & \bullet (0 + 1) + 0 & \text{shift}
\end{align*}
\]
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>) \mid <\text{Sum}\> + <\text{Sum}\>)

\(<\text{Sum}\> \Rightarrow \)

\[
= <\text{Sum}> \circ + 0 \quad \text{shift}
\]
\[
=> ( <\text{Sum}> ) \circ + 0 \quad \text{reduce}
\]
\[
= ( <\text{Sum}> \circ ) + 0 \quad \text{shift}
\]
\[
=> ( <\text{Sum}> + <\text{Sum}> \circ ) + 0 \quad \text{reduce}
\]
\[
=> ( <\text{Sum}> + 1 \circ ) + 0 \quad \text{reduce}
\]
\[
= ( <\text{Sum}> + \circ 1 ) + 0 \quad \text{shift}
\]
\[
= ( <\text{Sum}> \circ + 1 ) + 0 \quad \text{shift}
\]
\[
=> ( 0 \circ + 1 ) + 0 \quad \text{reduce}
\]
\[
= ( \circ 0 + 1 ) + 0 \quad \text{shift}
\]
\[
= \circ ( 0 + 1 ) + 0 \quad \text{shift}
\]
Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)
\mid <\text{Sum}> + <\text{Sum}>\)

\(<\text{Sum}> \Rightarrow $

\[
= \text{<Sum>} + \bullet 0 \text{ shift}
\]

\[
= \text{<Sum>} \bullet + 0 \text{ shift}
\]

\[
\Rightarrow (\text{<Sum>}) \bullet + 0 \text{ reduce}
\]

\[
= (\text{<Sum>} \bullet) + 0 \text{ shift}
\]

\[
\Rightarrow (\text{<Sum>} + \text{<Sum>} \bullet) + 0 \text{ reduce}
\]

\[
= (\text{<Sum>} + 1 \bullet) + 0 \text{ reduce}
\]

\[
= (\text{<Sum>} + \bullet 1) + 0 \text{ shift}
\]

\[
= (\text{<Sum>} \bullet + 1) + 0 \text{ shift}
\]

\[
\Rightarrow (0 \bullet + 1) + 0 \text{ reduce}
\]

\[
= (\bullet 0 + 1) + 0 \text{ shift}
\]

\[
= \bullet (0 + 1) + 0 \text{ shift}
\]
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>)\) \\
\mid <\text{Sum}\> + <\text{Sum}\>

\[
\begin{align*}
<\text{Sum}\> & \Rightarrow \\
& \Rightarrow <\text{Sum}\> + 0 \quad \text{reduce} \\
& = <\text{Sum}\> + 0 \\
& = <\text{Sum}\> + 0 \\
& \Rightarrow ( <\text{Sum}\> ) + 0 \\
& = ( <\text{Sum}\> ) + 0 \\
& \Rightarrow ( <\text{Sum}\> + <\text{Sum}\> ) + 0 \quad \text{reduce} \\
& = ( <\text{Sum}\> + <\text{Sum}\> ) + 0 \\
& = ( <\text{Sum}\> + 1 ) + 0 \\
& = ( <\text{Sum}\> + 1 ) + 0 \\
& \Rightarrow ( 0 + 1 ) + 0 \quad \text{reduce} \\
& = ( 0 + 1 ) + 0 \\
& = ( 0 + 1 ) + 0 \\
& \Rightarrow ( 0 + 1 ) + 0 \\
& = ( 0 + 1 ) + 0 \\
& = ( 0 + 1 ) + 0
\end{align*}
\]
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>) \mid <\text{Sum}\> + <\text{Sum}\>)

\[
\begin{align*}
<\text{Sum}\> & \implies <\text{Sum}\> + <\text{Sum}\> \text{ red} \\
& \implies <\text{Sum}\> + 0 \text{ red} \\
& = <\text{Sum}\> + 0 \text{ sh} \\
& = <\text{Sum}\> + 0 \text{ sh} \\
& \implies ( <\text{Sum}\> ) + 0 \text{ red} \\
& = ( <\text{Sum}\> ) + 0 \text{ sh} \\
& \implies ( <\text{Sum}\> + <\text{Sum}\> ) + 0 \text{ red} \\
& \implies ( <\text{Sum}\> + 1 ) + 0 \text{ red} \\
& = ( <\text{Sum}\> + 1 ) + 0 \text{ sh} \\
& = ( <\text{Sum}\> + 1 ) + 0 \text{ sh} \\
& \implies ( 0 + 1 ) + 0 \text{ red} \\
& = ( 0 + 1 ) + 0 \text{ sh} \\
& = ( 0 + 1 ) + 0 \text{ sh}
\end{align*}
\]
Example: `<Sum> = 0 | 1 | (<Sum>) | <Sum> + <Sum>`

```
<Sum> ● => <Sum> + <Sum> ● reduce
=> <Sum> + 0 ● reduce
= <Sum> + ● 0 shift
= <Sum> ● + 0 shift

=> ( <Sum> ) ● + 0 reduce
= ( <Sum> ●) + 0 shift

=> ( <Sum> + <Sum> ● ) + 0 reduce
=> ( <Sum> + 1 ● ) + 0 reduce
= ( <Sum> + ● 1 ) + 0 shift
= ( <Sum> ● + 1 ) + 0 shift

=> ( 0 ● + 1 ) + 0 reduce
= (● 0 + 1 ) + 0 shift
= ● ( 0 + 1 ) + 0 shift
```
Example

\[(0 + 1) + 0\]
Example

\[(0 + 1) + 0\]
Example

\[(0 \quad + \quad 1) \quad + \quad 0\]
Example

\[
\langle \text{Sum} \rangle 
\]

\[
(0 + 1) + 0
\]
Example

\[
\langle \text{Sum} \rangle = (0 + 1) + 0
\]
Example

\[
\langle Sum \rangle (0 + 1) + 0
\]
Example

\[
\langle \text{Sum} \rangle 
\begin{array}{c}
0 \\
\end{array} 
+ 
\langle \text{Sum} \rangle 
\begin{array}{c}
1 \\
\end{array} 
\] 
\) 
+ 
0
Example

\[ \langle \text{Sum} \rangle \langle \text{Sum} \rangle \langle \text{Sum} \rangle \]

\[ (0 + 1) + 0 \]
Example

\[
(0 + 1) + 0
\]
Example

\[(0 + 1) + 0\]
Example

\[(0 + 1) + 0\]
Example

\[
\begin{align*}
\langle \text{Sum} \rangle & \Rightarrow (0 + 1) + 0 \\
\langle \text{Sum} \rangle & \Rightarrow ( \langle \text{Sum} \rangle + \langle \text{Sum} \rangle ) + 0 \\
\langle \text{Sum} \rangle & \Rightarrow (0 + 1) + 0 \\
\end{align*}
\]
Example

\[
\langle \text{Sum} \rangle = \langle \text{Sum} \rangle + \langle \text{Sum} \rangle + \langle \text{Sum} \rangle + 0 + 1 \]

\[
\Rightarrow \langle \text{Sum} \rangle = \langle \text{Sum} \rangle + 0
\]
Example

\( \langle \text{Sum} \rangle (0 + 1) + 0 \)
LR Parsing Tables

- Build a pair of tables, Action and Goto, from the grammar
  - This is the hardest part, we omit here
  - Rows labeled by states
  - For Action, columns labeled by terminals and “end-of-tokens” marker
    - (more generally strings of terminals of fixed length)
  - For Goto, columns labeled by non-terminals
Action and Goto Tables

- Given a state and the next input, Action table says either
  - **shift** and go to state $n$, or
  - **reduce** by production $k$ (explained in a bit)
  - **accept** or **error**

- Given a state and a non-terminal, Goto table says
  - go to state $m$
LR(i) Parsing Algorithm

- Based on push-down automata
- Uses states and transitions (as recorded in Action and Goto tables)
- Uses a stack containing states, terminals and non-terminals
LR(i) Parsing Algorithm

0. Insure token stream ends in special “end-of-tokens” symbol

1. Start in state 1 with an empty stack

2. Push \texttt{state}(1) onto stack

3. Look at next \textit{i} tokens from token stream (\textit{toks}) (don’t remove yet)

4. If top symbol on stack is \texttt{state}(n), look up action in Action table at (\textit{n, toks})
LR(i) Parsing Algorithm

5. If action = \textbf{shift} \( m \),
   
   a) Remove the top token from token stream and push it onto the stack
   
   b) Push \textbf{state}(m) onto stack
   
   c) Go to step 3
LR(i) Parsing Algorithm

6. If action = reduce \( k \) where production \( k \) is \( E ::= u \)
   
   a) Remove \( 2 \ast \text{length}(u) \) symbols from stack (\( u \) and all the interleaved states)
   
   b) If new top symbol on stack is \( \text{state}(m) \), look up new state \( p \) in \( \text{Goto}(m,E) \)
   
   c) Push \( E \) onto the stack, then push \( \text{state}(p) \) onto the stack
   
   d) Go to step 3
LR(i) Parsing Algorithm

7. If action = **accept**
   - Stop parsing, return success

8. If action = **error**,
   - Stop parsing, return failure
Adding Synthesized Attributes

- Add to each **reduce** a rule for calculating the new synthesized attribute from the component attributes
- Add to each non-terminal pushed onto the stack, the attribute calculated for it
- When performing a **reduce**,  
  - gather the recorded attributes from each non-terminal popped from stack  
  - Compute new attribute for non-terminal pushed onto stack
Shift-Reduce Conflicts

- **Problem**: can’t decide whether the action for a state and input character should be **shift** or **reduce**
- Caused by ambiguity in grammar
- Usually caused by lack of associativity or precedence information in grammar
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (\langle\text{Sum}\rangle) \mid \langle\text{Sum}\rangle + \langle\text{Sum}\rangle\)

\[
\begin{align*}
0 + 1 + 0 & \quad \text{shift} \\
\rightarrow 0 + 1 + 0 & \quad \text{reduce} \\
\rightarrow \langle\text{Sum}\rangle + 1 + 0 & \quad \text{shift} \\
\rightarrow \langle\text{Sum}\rangle + 1 + 0 & \quad \text{shift} \\
\rightarrow \langle\text{Sum}\rangle + 1 + 0 & \quad \text{reduce} \\
\rightarrow \langle\text{Sum}\rangle + \langle\text{Sum}\rangle + 0 & \quad \text{reduce}
\end{align*}
\]
Example - cont

- **Problem:** shift or reduce?

- You can shift-shift-reduce-reduce or reduce-shift-shift-reduce

- Shift first - right associative
- Reduce first- left associative
Reduce - Reduce Conflicts

- **Problem**: can’t decide between two different rules to reduce by
- Again caused by ambiguity in grammar
- **Symptom**: RHS of one production suffix of another
- Requires examining grammar and rewriting it
- Harder to solve than shift-reduce errors
Example

- $S ::= A \mid aB$
- $A ::= \text{abc}$
- $B ::= \text{bc}$

- $\text{abc}$ shift
- $a \text{bc}$ shift
- $\text{ab} c$ shift
- $\text{abc}$

Problem: reduce by $B ::= \text{bc}$ then by $S ::= aB$, or by $A ::= \text{abc}$ then $S ::= A$?