Programming Languages and Compilers (CS 421)

Sasa Misailovic
4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421A

Based on slides by Elsa Gunter, which were inspired by earlier slides by Mattox Beckman, Vikram Adve, and Gul Agha
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>` | ( )
BNF Derivations

- Given rules
  \[ X ::= yZw \] and \[ Z ::= v \]

  we may replace \( Z \) by \( v \) to say
  \[ X \rightarrow yZw \rightarrow yvw \]

- Sequence of such replacements called **derivation**

- Derivation called **right-most** if always replace the right-most non-terminal
BNF Semantics

- 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

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

- Start with the start symbol:

\[ \text{<Sum>} => \]
BNF Derivations

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

- Pick a non-terminal

<Sum> =>
BNF Derivations

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

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

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

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

Pick a non-terminal:

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

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

- Pick a rule and substitute:
  - <Sum> ::= ( <Sum> )
  - <Sum> => <Sum> + <Sum>
  - => ( <Sum> ) + <Sum>
  - => ( <Sum> ) + <Sum>
BNF Derivations

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

- Pick a non-terminal:

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

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

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

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

=> ( <Sum> ) + <Sum>

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

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

Pick a non-terminal:

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

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

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

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

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

- Pick a non-terminal:

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

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

- Pick a rule and substitute:
  - \(<\text{Sum}> ::= 0\)

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

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

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

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

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

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

- Pick a non-terminal:

<Sum> => <Sum> + <Sum>
  => ( <Sum> ) + <Sum>
  => ( <Sum> + <Sum> ) + <Sum>
  => ( <Sum> + 1 ) + <Sum>
  => ( <Sum> + 1 ) + 0
BNF Derivations

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

- Pick a rule and substitute
  - \[ <\text{Sum}> ::= 0 \]

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

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

( 0 + 1 ) + 0 is generated by grammar

<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>
=> ( <Sum> + 1 ) + 0
=> ( 0 + 1 ) + 0
Regular Grammars

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

- Alternatives: allow rules of form \( X ::= y \mid z \)
  - 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
    \( X ::= yz, X ::= yVz, \)
    \( V ::= v, V ::= vV \)
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:
  \[
  \begin{align*}
  \langle \text{exp} \rangle & : = \langle \text{factor} \rangle \\
  & \quad \mid \langle \text{factor} \rangle \ + \ \langle \text{factor} \rangle \\
  \langle \text{factor} \rangle & : = \langle \text{bin} \rangle \\
  & \quad \mid \langle \text{bin} \rangle \ * \ \langle \text{exp} \rangle \\
  \langle \text{bin} \rangle & : = \ 0 \ \mid \ 1
  \end{align*}
  \]

- Problem: Build parse tree for \(1 \ast 1 + 0\) as an \langle \text{exp} \rangle.
Example cont.

1 * 1 + 0:  <exp>

<exp> is the start symbol for this parse tree
Example cont.

1 * 1 + 0: \( <\text{exp}> \)

Use rule: \( <\text{exp}> ::= <\text{factor}> \)
Example cont.

$1 \times 1 + 0$:  

```
<exp>
   
<factor>
  
<bin>  *  <exp>
```

Use rule:  

```
<factor> ::= <bin> * <exp>
```
Example cont.

- $1 \times 1 + 0$:  
  
  \[
  \begin{array}{c}
  \text{<exp>} \\
  \text{<factor>} \\
  \text{<bin>} \quad \times \quad \text{<exp>} \\
  \quad \quad \quad \quad \quad \quad \text{1} \quad \text{<factor>} \quad + \quad \text{<factor>}
  \end{array}
  \]

  Use rules:  
  
  - \text{<bin>} ::= 1 
  - \text{<exp>} ::= \text{<factor>} \quad + \quad \text{<factor>
Example cont.

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

  $<\text{bin}>$  $1$  $<\text{factor}>$  $+$  $<\text{factor}>$

  $<\text{bin}>$  $<\text{bin}>$

Use rule: $<\text{factor}> ::= <\text{bin}>$
Example cont.

- \(1 \times 1 + 0:\)

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

Use rules: \(<\text{bin}> ::= 1 | 0\)
Example cont.

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

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

- Represent as Abstract Data Types:

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

- $1 \times 1 + 0$: 

```
<exp>
  
  <factor>
    
    <bin> * <exp>
    
    1
  
  <factor> + <factor>
    
    <bin> <bin>
    
    1 0
```
Example cont.

- Can be represented as

$$\text{Factor2Exp} \left( \text{Mult}(\text{One,} \right.$$

$$\text{Plus}(\text{Bin2Factor One,}$$

$$\text{Bin2Factor Zero})) \right)$$
Example cont.

- 1 * 1 + 0: Factor2Exp

```
+ 0:
  \( <\text{exp}> \)
  \( <\text{factor}> \)
  \( <\text{bin}> * <\text{exp}> \)
  \( 1 <\text{factor}> + <\text{factor}> \)
  \( <\text{bin}> <\text{bin}> \)
  \( 1 0 \)
```

```
\[ \text{Mult} \]
\[ \text{One} \quad <\text{Plus}> \]
\[ <\text{Bin2Factor}> <\text{Bin2Factor}> \]
\[ <\text{One}> <\text{Zero}> \]
```

- type \( \text{exp} = \text{Factor2Exp} \) of factor
  - Plus of factor * factor
    - Mult of bin * exp
      - bin = Zero | One
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 BNFs 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 * 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
  
  \[
  \text{language of } G = \text{language of } G'
  \]

- 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:
  \[ \langle \text{exp} \rangle ::= 0 \mid 1 \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle \ast \langle \text{exp} \rangle \]

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

- Source of ambiguity: associativity and precedence
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

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

- **Becomes**
  - $<\text{Sum}> ::= <\text{Num}> | <\text{Num}> + <\text{Sum}>$
  - $<\text{Num}> ::= 0 | 1 | (<\text{Sum}>)$
Operator Precedence

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

For instance multiplication (*) has higher precedence than addition (+)

- Needs to be reflected in grammar
Predence in Grammar

- Higher precedence translates to longer derivation chain

- Example:

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

  Becomes

  \[\text{<exp> ::= <mult\_exp> | <exp> + <mult\_exp>}\]
  \[\text{<mult\_exp> ::= <id> | <mult\_exp> * <id>}\]
  \[\text{<id> ::= 0 | 1}\]
Disambiguating a Grammar

- \( \text{exp} ::= 0|1| b\text{exp} | \text{exp}a \\
  | \text{exp}m\text{exp} \)

- Want \textit{a} to have \underline{higher precedence} than \textit{b}, which in turn has \underline{higher precedence} than \textit{m}, and such that \textit{m} associates to the left.
Disambiguating a Grammar

- `<exp>::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>`

- Want `a` to have higher precedence than `b`, which in turn has higher precedence than `m`, and such that `m` associates to the left.

- `<exp> ::= <exp> m <not_m> | <not_m>`
- `<not_m> ::= b <not_m> | <not_b_m>`
- `<not_b_m> ::= <not_b_m>a | 0 | 1`
Disambiguating a Grammar – Take 2

- `<exp>::= 0|1| b<exp> | <exp>a 
  | <exp>m<exp>`
- Want **b** to have higher precedence than **m**, which in turn has higher precedence than **a**, and such that **m** associates to the right.
Disambiguating a Grammar – Take 2

- \(<\text{exp}> :\!::= 0|1| b<\text{exp}> | <\text{exp}>a
  
  | <\text{exp}>m<\text{exp}>

- Want \(b\) has higher precedence than \(m\), which in turn has higher precedence than \(a\), and such that \(m\) associates to the right.

- \(<\text{exp}> :\!::=
  
  <\text{no}_a\_m> | <\text{no}_m> m <\text{no}_a>| <\text{exp}> a

- \(<\text{no}_a> :\!::= <\text{no}_a\_m> | <\text{no}_a\_m> m <\text{no}_a>

- \(<\text{no}_m> :\!::= <\text{no}_a\_m> | <\exp> a

- \(<\text{no}_a\_m> :\!::= b <\text{no}_a\_m> | 0 | 1
Disambiguating a Grammar – Take 3

- `<exp>` ::= 0|1| b<exp> | <exp>a
  | <exp>m<exp>

- Want **a** has higher precedence than **m**, which in turn has higher precedence than **b**, and such that **m** associates to the right.

- For you...
How do we disambiguate in this case?

- Our old friend:

  ```
  <exp> ::= <factor>
  | <factor> + <factor>

  <factor> ::= <bin>
  | <bin> * <exp>

  <bin> ::= 0 | 1
  ```

- How do we make multiplication have higher precedence than addition?
Moving On With Richer Expressions

- How do we extend the grammar to support nested additions, e.g., \(1 \times (0 + 1)\)

\[
\begin{align*}
<\text{exp}> & ::= <\text{factor}> \\
& \quad | <\text{factor}> + <\text{exp}>
\\
<\text{factor}> & ::= <\text{bin}> \\
& \quad | <\text{bin}> \times <\text{factor}>
\\
<\text{bin}> & ::= 0 \mid 1
\end{align*}
\]
Moving On With Richer Expressions

- How do we extend the grammar to support nested additions, e.g., $1 \times (0 + 1)$

\[
\begin{align*}
\langle \text{exp} \rangle & \ ::= \ \langle \text{factor} \rangle \\
& \quad | \ \langle \text{factor} \rangle + \ \langle \text{exp} \rangle \\
\langle \text{factor} \rangle & \ ::= \ \langle \text{bin} \rangle \\
& \quad | \ \langle \text{bin} \rangle \times \ \langle \text{factor} \rangle \\
\langle \text{bin} \rangle & \ ::= \ 0 \ | \ 1 \ | \ ( \ \langle \text{exp} \rangle \ )
\end{align*}
\]
Moving On With Richer Expressions

- How do we extend the grammar to support other operations, subtraction and division?

\[
\begin{align*}
\text{<exp>} & ::= \text{<factor>} \\
& \quad | \text{<factor>} + \text{<exp>} | \text{<factor>} - \text{<exp>}
\end{align*}
\]

\[
\begin{align*}
\text{<factor>} & ::= \text{<bin>} \\
& \quad | \text{<bin>} \times \text{<exp>} | \text{<bin>} / \text{<factor>}
\end{align*}
\]

\[
\begin{align*}
\text{<bin>} & ::= 0 | 1 | ( \text{<exp>} )
\end{align*}
\]
Disambiguating Grammars – Dangling Else

- \texttt{stmt ::= ...}
  - | \texttt{if ( expr ) stmt}
  - | \texttt{if ( expr ) stmt else stmt}

- How can we parse
  \[
  \text{if (e1) if (e2) s1 else s2}
  \]
Try: let us try to differentiate if we have `if` inside the `then` branch or not....

- `stmt = open_stmt | closed_stmt`
- `open_stmt ::= if(expr)stmt`
  - `| if(expr)closed_stmt else open_stmt`
- `closed_stmt ::= non_if_statement`
  - `| if(expr)closed_stmt else closed_stmt`

How can we parse `if(e1) if(e2)s1 else s2` now?
Disambiguating Grammars – Overlapping

- seq = ɛ | may_word | word seq
- may_word = ɛ | “word”

- How do you parse “word”? And ɛ?

- How do you fix it?
How do you know you have ambiguity?

- The Ocaml parser generator (ocamlyacc) will report ambiguity in the grammar as “conflicts”:
  - **Shift/reduce**: Usually caused by lack of associativity or precedence information in grammar
  - **Reduce/reduce**: can’t decide between two different rules to reduce by; Not always clear what the problem is, but often right-hand side of one production is the suffix of another

- We will explain what these conflicts mean next time!
**Parser Code**

- **Ocamlyacc** is a parser generator for Ocaml
  - Similar generators exist for other languages
  - Search under: Yacc, Bison, Menhir...
  - Another family: Antlr

- **Input**: high level specification (**<grammar>**.mly file)

- **Output**: tokens (**<grammar>**.mli) and generated parser (**<grammar>**.ml)
  - **<grammar>**.ml defines a 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

- `<grammar>.mly` File format:

```ocaml
%
  <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
- `<trailer>` similar. Possibly used to call parser
Ocamlyacc Input

- `<grammar>.mly` File format:

```ocaml
%%
<%header%>
%
```

```ocaml
<declarations>
%%
```

```ocaml
<rules>
%%
```

```ocaml
<%trailer%>
```
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 Input

- `<grammar>.mly` File format:

```%
%
    <header>
%
%
    <declarations>
%
%
    <rules>
%
%
    <trailer>
```
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 - Grammar

A slight variation of what we’ve seen earlier:

Expr ::= Term | Term + Expr | Term – Expr
Term ::= Factor | Factor * Term | Factor / Term
Factor ::= Id | ( Expr )
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

Expr ::= Term | Term + Expr | Term – Expr
Term ::= Factor | Factor * Term | Factor / Term
Factor ::= Id | ( Expr )
Example - Lexer

{ 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}

Expr ::= Term | Term + Expr | Term - Expr
Term ::= Factor | Factor * Term | Factor / Term
Factor ::= Id | ( Expr )
Example - Parser (exprparse.mly)

```perl
{%
    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

\{
  \text{Term\_as\_Expr} \ $1 \}

\mid \text{term\ Plus\_token\ expr}

\{
  \text{Plus\_Expr} \ ($1, \ $3) \}

\mid \text{term\ Minus\_token\ expr}

\{
  \text{Minus\_Expr} \ ($1, \ $3) \}

Example - Base types

(* File: expr.ml *)
type expr =
  Term\_as\_Expr\ of\ term
| Plus\_Expr\ of\ (term\ *\ expr)
| Minus\_Expr\ of\ (term\ *\ expr)
| ...
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 - 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)  
| expr
**Example - Parser (exprparse.mly)**

factor:

\[
\text{Id}_\text{token} \\
\quad \{ \text{Id}_\text{as}\_\text{Factor} \ \$1 \ \} \\
| \ \text{Left}_\text{parenthesis} \ \text{expr} \ \text{Right}_\text{parenthesis} \\
\quad \{ \text{Parenthesized}_\text{Expr}_\text{as}\_\text{Factor} \ \$2 \ \} \\
\]

main:

\[
| \ \text{expr} \ \text{EOL} \\
\quad \{ \ \$1 \ \} \\
\]

Recall, we previously defined:

\[
\text{%}\text{start main} \\
\text{%}\text{type <expr> main} \\
\]

**Example - Base types**

\[
(* \text{File: expr.ml} *) \\
\text{type expr} = \\
\quad \text{Term}_\text{as}\_\text{Expr} \ \text{of}\ \text{term} \\
\quad | \ \text{Plus}_\text{Expr} \ \text{of}\ \text{(term} * \ \text{expr}) \\
\quad | \ \text{Minus}_\text{Expr} \ \text{of}\ \text{(term} * \ \text{expr}) \\
\quad \text{and} \ \text{term} = \\
\quad | \ \text{Factor}_\text{as}\_\text{Term} \ \text{of}\ \text{factor} \\
\quad | \ \text{Mult}_\text{Term} \ \text{of}\ \text{(factor} * \ \text{term}) \\
\quad | \ \text{Div}_\text{Term} \ \text{of}\ \text{(factor} * \ \text{term}) \\
\quad \text{and} \ \text{factor} = \\
\quad | \ \text{Id}_\text{as}\_\text{Factor} \ \text{of}\ \text{string} \\
\quad | \ \text{Parenthesized}_\text{Expr}_\text{as}\_\text{Factor} \ \text{of}\ \text{expr} \\
\]

Call:
- $ ocamlyacc options exprparse.mly

Get:
- Tokens: exprparse.mli (can be used in lexer)
- Parser: exprparse.ml (included in the rest of code)
Example - Using Parser

# #use "expr.ml";;
...
# #use "exprpparse.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")))

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
LR Parsing

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

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

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

\[
(0 + 1) + 0
\]
Example: \[<\text{Sum}> ::= 0 \mid 1 \mid (<\text{Sum}>)
| <\text{Sum}> + <\text{Sum}>\]
Example: \( <\text{Sum}> ::= 0 | 1 | (<\text{Sum}>) | <\text{Sum}> + <\text{Sum}> \)
Example: \( <\text{Sum}> ::= 0 | 1 | (<\text{Sum}>) \]
\[| \text{<Sum>} + \text{<Sum>}\]
Example: \(<\text{Sum}\> ::= 0 | 1 | (\langle\text{Sum}\rangle) | \langle\text{Sum}\rangle + \langle\text{Sum}\rangle\)
Example: `<Sum> ::= 0 | 1 | ( <Sum> ) | <Sum> + <Sum>`
Example: \(<\text{Sum}> ::= 0 \mid 1 \mid ( <\text{Sum}> ) \mid <\text{Sum}> + <\text{Sum}>\)
Example: \(<\text{Sum}\> \ ::= \ 0 \ | \ 1 \ | \ (\ <\text{Sum}\> \ ) \ |
\<\text{Sum}\> \ + \ <\text{Sum}\>\)
Example: \(<\text{Sum}\> ::= 0 \mid 1 \mid (\langle \text{Sum} \rangle)
\mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle\)
Example: \(<\text{Sum}\> ::= 0 \mid 1 \mid (\text{<Sum>}) \mid \text{<Sum>} + \text{<Sum>}\)
Example: \(<\text{Sum}\> ::= 0 \mid 1 \mid (\langle\text{Sum}\rangle) \]
\[
\mid \langle\text{Sum}\rangle + \langle\text{Sum}\rangle
\]
Example: \(<\text{Sum}\> ::= 0 \mid 1 \mid (\langle \text{Sum} \rangle) \mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle\)
Example: \(<\text{Sum}> ::= 0 \mid 1 \mid (\langle\text{Sum}\rangle) \mid \langle\text{Sum}\rangle + \langle\text{Sum}\rangle\)
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 state(1) onto stack

3. Look at next $i$ tokens from token stream ($toks$) (don’t remove yet)

4. If top symbol on stack is state($n$), look up action in Action table at $(n, toks)$
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 = \textbf{reduce} \ k \text{ where production } \ k \text{ is } E ::= u
   
   a) Remove $2 \times \text{length}(u)$ symbols from stack (u and all the interleaved states)
   
   b) If new top symbol on stack is state($m$), look up new state $p$ in Goto($m,E$)
   
   c) Push E onto the stack, then push state($\rho$) 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
Example: \(<\text{Sum}> = 0 \mid 1 \mid (\langle\text{Sum}\rangle)\)  
\mid \langle\text{Sum}\rangle + \langle\text{Sum}\rangle\)

\(<\text{Sum}> \quad \Rightarrow \quad \bullet (0 + 1) + 0\)  
\text{shift}
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 state(1) onto stack

3. Look at next $i$ tokens from token stream ($toks$) (don’t remove yet)

4. If top symbol on stack is state($n$), look up action in Action table at ($n$, $toks$)
Example: $<\text{Sum}> = 0 \mid 1 \mid (\langle<\text{Sum}\rangle\rangle) \mid <\text{Sum}> + <\text{Sum}>$

$<\text{Sum}> \implies$

$= \bullet (0 + 1) + 0 \quad \text{shift}$
LR(i) Parsing Algorithm

5. If action = **shift** \( m \),
   a) Remove the top token from token stream and push it onto the stack
   b) Push **state**\((m)\) onto stack
   c) Go to step 3
Example: \(<\text{Sum}\> = 0 | 1 | (<\text{Sum}\>) | <\text{Sum}\> + <\text{Sum}\>)

\(<\text{Sum}\> \quad \Rightarrow \quad (0 + 1) + 0 \quad \text{shift}

= \quad (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\)

\[
\langle\text{Sum}\rangle \implies (0 \circ 1) + 0
\]
reduce

\[
= (0 + 1) + 0
\]
shift

\[
= 0 \circ (0 + 1) + 0
\]
shift
6. If action = **reduce** $k$ where production $k$ is $E ::= u$
   a) Remove $2 \times \text{length}(u)$ symbols from stack (u and all the interleaved states)
   b) If new top symbol on stack is **state**($m$), look up new state $p$ in Goto($m, E$)
   c) Push $E$ onto the stack, then push **state**($p$) onto the stack
   d) Go to step 3
Example: $<\text{Sum}> = 0 \mid 1 \mid ( <\text{Sum}> ) \mid <\text{Sum}> + <\text{Sum}>$

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

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

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

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

$\langle \text{Sum}\rangle \Rightarrow$

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

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

\[\Rightarrow (\langle\text{Sum}\rangle + \langle\text{Sum}\rangle \cdot 1) + 0 \quad \text{reduce}\]
\[\Rightarrow (\langle\text{Sum}\rangle + 1 \cdot 1) + 0 \quad \text{reduce}\]
\[= (\langle\text{Sum}\rangle + 1 \cdot 1) + 0 \quad \text{shift}\]
\[= (\langle\text{Sum}\rangle \cdot 1 + 1) + 0 \quad \text{shift}\]
\[\Rightarrow (0 \cdot 1 + 1) + 0 \quad \text{reduce}\]
\[= (0 + 1) + 0 \quad \text{shift}\]
\[= 0 (0 + 1) + 0 \quad \text{shift}\]
LR(i) Parsing Algorithm

6. If action = reduce $k$ where production $k$ is $E ::= u$
   
   a) Remove $2 \times \text{length}(u)$ symbols from stack ($u$ and all the interleaved states)
   
   b) If new top symbol on stack is state$(m)$, look up new state $p$ in Goto$(m,E)$
   
   c) Push $E$ onto the stack, then push state$(p)$ onto the stack
   
   d) Go to step 3
Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>))\)

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

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

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

\[
<\text{Sum}> \implies
\]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

\[
<\text{Sum}> \Rightarrow <\text{Sum}> + <\text{Sum}> \Rightarrow <\text{Sum}> + 0 \Rightarrow <\text{Sum}> + 0 \Rightarrow ( <\text{Sum} > ) + 0 \Rightarrow ( <\text{Sum} > + <\text{Sum} > ) + 0 \Rightarrow ( <\text{Sum} > + 1 ) + 0 \Rightarrow ( 0 + 1 ) + 0
\]
LR(i) Parsing Algorithm

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

8. If action = error,
   - Stop parsing, return failure
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 state(1) onto stack
3. Look at next \( i \) tokens from token stream (\( \text{toks} \)) (don’t remove yet)
4. If top symbol on stack is state(\( n \)), look up action in Action table at (\( n, \text{toks} \))
LR(i) Parsing Algorithm

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

6. If action = reduce $k$ where production $k$ is $E ::= u$
   a) Remove $2 \times \text{length}(u)$ symbols from stack ($u$ and all the interleaved states)
   b) If new top symbol on stack is \textit{state($m$)}, look up new state $p$ in 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: $\langle \text{Sum}\rangle = 0 \mid 1 \mid (\langle \text{Sum}\rangle)$
$\mid \langle \text{Sum}\rangle + \langle \text{Sum}\rangle$

- $0 + 1 + 0$ shift
- $0 \bullet + 1 + 0$ reduce
- $\langle \text{Sum}\rangle \bullet + 1 + 0$ shift
- $\langle \text{Sum}\rangle + \bullet 1 + 0$ shift
- $\langle \text{Sum}\rangle + 1 \bullet + 0$ reduce
- $\langle \text{Sum}\rangle + \langle \text{Sum}\rangle \bullet + 0$
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

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

- abc  
  shift

- a bc  
  shift

- ab c  
  shift

- abc

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