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
  $X => yZw => yvw$
- 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

- Start with the start symbol:
  \[<Sum> \Rightarrow \]

BNF Derivations

- Pick a non-terminal
  \[<Sum> \Rightarrow \]

BNF Derivations

- Pick a rule and substitute:
  \[<Sum> ::= <Sum> + <Sum>\]
  \[<Sum> \Rightarrow <Sum> + <Sum>\]

BNF Derivations

- Pick a non-terminal:
  \[<Sum> \Rightarrow <Sum> + <Sum>\]
  \[<Sum> \Rightarrow ( <Sum> ) + <Sum>\]
BNF Derivations

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

BNF Derivations

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

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>`

BNF Derivations

- Pick a rule and substitute:
  - `<Sum> ::= 0`
  - `<Sum> => <Sum> + <Sum>`
    - `=> ( <Sum> ) + <Sum>`
    - `=> ( <Sum> + <Sum> ) + <Sum>`
    - `=> ( <Sum> + 1 ) + <Sum>`
    - `=> ( <Sum> + 1 ) + 0`
BNF Derivations

Pick a rule and substitute

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

BNF Derivations

- \( (0 + 1 ) + 0 \) is generated by grammar

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

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

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:
  - \(<\text{Balanced}> ::= \varepsilon \)
  - \(<\text{Balanced}> ::= 0<\text{OneAndMore}>\)
  - \(<\text{Balanced}> ::= 1<\text{ZeroAndMore}>\)
  - \(<\text{OneAndMore}> ::= 1<\text{Balanced}>\)
  - \(<\text{ZeroAndMore}> ::= 0<\text{Balanced}>\)
- 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: \(X::=y[\text{v}]z\)
  - Abbreviates \(X::=yvz, X::=yz\)
- Repetition: \(X::=y(\text{v})^*z\)
  - Can be eliminated by adding new nonterminal \(V\) and rules \(X::=yz, X::=yVz, V::=v, V::=w\)
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:
  
  ```
  <exp>  ::= <factor> 
         |  <factor> +  <factor> 
  <factor>  ::=  <bin> 
           |  <bin>  *  <exp> 
  <bin> ::=  0  | 1 
  ```

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

Example cont.

- 1 * 1 + 0:  <exp>  

  <exp> is the start symbol for this parse tree

Example cont.

- 1 * 1 + 0:  <exp>  

  Use rule: <exp> ::= <factor>

Example cont.

- 1 * 1 + 0:  <exp>  

  Use rule: <factor> ::= <bin> * <exp>

Example cont.

- 1 * 1 + 0:  <exp>  

  Use rules: <bin> ::= 1  and  
              <exp> ::= <factor>  + 

  Use rules: <bin> ::= 1  and  
                  <exp> ::= <factor>  + 
                  <factor>
Example cont.

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

Use rule: <factor> ::= <bin>

Example cont.

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

Use rules: <bin> ::= 1 | 0

Example cont.

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

Fringe of tree is string generated by grammar

Your Turn: 1 * 0 + 0 * 1

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:

<exp> ::= <factor> | <factor> + <factor>  
<factor> ::= <bin> | <bin> * <exp>  
<bin> ::= 0 | 1

type exp = Factor2Exp of factor  
| Plus of factor * factor  
and factor = Bin2Factor of bin  
| Mult of bin * exp  
and bin = Zero | One
Example cont.

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

Example cont.
- Can be represented as

\[
\text{Factor2Exp} \\
\text{(Mult(One,} \\
\text{Plus(Bin2Factor One,} \\
\text{Bin2Factor Zero)))}
\]

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:

\[ 7 - 5 - 2 \]

Possible answers:

- In Pascal, C++, SML associativity left:
  \[ 7 - 5 - 2 = (7 - 5) - 2 = 0 \]
- In APL, associate to right:
  \[ 7 - 5 - 2 = 7 - (5 - 2) = 4 \]

Two Major Sources of Ambiguity

- Lack of determination of operator precedence
- Lack of determination of operator associativity
- Not the only sources of ambiguity

Disambiguating a Grammar

Given ambiguous grammar \( G \), with start symbol \( S \), find a grammar \( G' \) with same start symbol, such that:

- Language of \( G \) = language of \( G' \)
- Not always possible
- No algorithm in general

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 then one parse:
  \[ 0 + 1 + 0 \]
  \[ 1 \ast 1 + 1 \]
- 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>Fortran</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>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>mod</td>
<td></td>
<td>+, ,</td>
</tr>
<tr>
<td></td>
<td>+, -</td>
<td>+, -</td>
<td>+, -</td>
<td>::</td>
<td></td>
</tr>
</tbody>
</table>
First Example Again

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

Predence in Grammar

- Higher precedence translates to longer derivation chain
- Example:

  `<exp> ::= 0 | 1 | <exp> + <exp>
  | <exp> * <exp>

  Becomes

  `<exp> ::= <mult_exp>
  | <exp> + <mult_exp>
  <mult_exp> ::= <id> | <mult_exp> * <id>
  <id> ::= 0 | 1`

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:

  ```
  %{ <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)

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*) }

```ocaml
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)

```ocaml
%
%

%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)

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

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

Example - Parser (exprparse.mly)

```ml
factor:
  Id_token
    { Id_as_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;;
```

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

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

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

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

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

10/31/17 73

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

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

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

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

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

10/31/17 74

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

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

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

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

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

10/31/17 75

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

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

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

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

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

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

10/31/17 76

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

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

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

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

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

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

10/31/17 77

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

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

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

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

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

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

10/31/17 78
Example: \( \text{<Sum> = 0 | 1 | (<Sum>)} \)

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

\( (0 + 1) + 0 \)

Example

(0 + 1) + 0

Example

(0 + 1) + 0

Example

(0 + 1) + 0

Example

(0 + 1) + 0

Example
Example

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

10/31/17 91

Example

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

10/31/17 92

Example

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

10/31/17 93

Example

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

10/31/17 94

Example

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

10/31/17 95

Example

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

10/31/17 96
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

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 \( (n, toks) \)

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

6. If action = reduce \( k \) where production \( k \) is \( E ::= u \)
   a) Remove 2 * 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

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: 

\[ \text{S ::= A | aB} \quad \text{A ::= abc} \quad \text{B ::= bc} \]

- \( \text{abc} \rightarrow \text{shift} \)
- \( \text{a abc} \rightarrow \text{shift} \)
- \( \text{ab c} \rightarrow \text{shift} \)
- \( \text{abc} \)

- **Problem:** reduce by \( B ::= bc \) then by \( S ::= aB \), or by \( A ::= abc \) then \( S ::= A \)?