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
  \(<Sum> ::= 0 \mid 1 \mid <Sum> + <Sum> \mid ( ) \)

BNF Deriviations

- 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.

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 \(<\text{X}> ::= y|z\)
- Abbreviates \(<\text{X}> ::= y, X ::= z\)
- Options: \(<\text{X}> ::= [v]z\)
- Abbreviates \(<\text{X}> ::= yvz, X ::= yz\)
- Repetition: \(<\text{X}> ::= y\{v\}*z\)
- Can be eliminated by adding new nonterminal \(V\) and rules \(X ::= yz, X ::= yWz, V ::= v, V ::= vV\)

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:
  \(<\text{exp}> ::= <\text{factor}>\)
  \(<\text{factor}> ::= <\text{bin}>\)
  \(<\text{bin}> ::= 0 | 1\)
- Problem: Build parse tree for \(1 * 1 + 0\) as an \(<\text{exp}>\)
Example cont.

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

$\text{<exp>}$ is the start symbol for this parse tree

Example cont.

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

Use rule: $\text{<exp>} ::= \text{<factor>}$

Example cont.

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

  $\text{<factor>}$

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

Use rule: $\text{<factor>} ::= \text{<bin> * <exp>}$

Example cont.

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

  $\text{<factor>}$

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

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

Use rules: $\text{<bin>} ::= 1$ and $\text{<exp>} ::= \text{<factor> + <factor>}$

Example cont.

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

  $\text{<factor>}$

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

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

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

Use rules: $\text{<bin>} ::= 1$ | $0$
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.

- Can be represented as
  Factor2Exp
  (Mult(One,
       Plus(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

\[
\begin{array}{c}
\text{<Sum>} \\
\text{<Sum>} + \text{<Sum>} \\
\text{<Sum>} + \text{<Sum>} \\
0 \\
1 \\
0
\end{array}
\]

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

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

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:
  $\text{<exp>} ::= 0 \mid 1 \mid \text{<exp>} + \text{<exp>} \mid \text{<exp>} * \text{<exp>}$
- 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

- `<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, mod, /, *</td>
</tr>
<tr>
<td></td>
<td>*, /,</td>
<td>+, -</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>div,</td>
<td>%</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mod</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>

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?

Precedence in Grammar

- Higher precedence translates to longer derivation chain
- Example:
  - `<exp> ::= 0 | 1 | <exp> + <exp> | <exp> * <exp>`
- Becomes
  - `<exp> ::= <mult_exp> | <exp> + <mult_exp>`
  - `<mult_exp> ::= <id> | <mult_exp> * <id>`
  - `<id> ::= 0 | 1`
Ocamlyacc Input

- File format:
  ```%
  <header>
  %
  <declarations>
  %%
  <rules>
  %%
  <trailer>
  ```

Ocamlyacc `<header>`

- Contains arbitrary Ocaml code
- Typically used to give types and functions needed for the semantic actions of rules and to give specialized error recovery
- May be omitted
- `<footer>` similar. Possibly used to call parser

Ocamlyacc `<declarations>`

- `%token symbol ... symbol`
  Declare given symbols as tokens
- `%token <type> symbol ... symbol`
  Declare given symbols as token constructors, taking an argument of type `<type>`
- `%start symbol ... symbol`
  Declare given symbols as entry points; functions of same names in `<grammar>.ml`

Ocamlyacc `<rules>`

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

Example - Base types

```(* File: expr.ml *)
type expr =
  Term_as_Expr of term |
  Plus_Expr of (term * expr) |
  Minus_Expr of (term * expr) |
and term =
  Factor_as_Term of factor |
  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 (exprparse.mly)

```ml
%{ open Expr
%
%token <string> Id_token
%token Left_parenthesis Right_parenthesis
%token Times_token Divide_token
%token Plus_token Minus_token
%token EOL
%start main
%type <expr> main
%%
```

Example - Parser (exprparse.mly)

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

Example - Parser (exprparse.mly)

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

Example - Parser (exprparse.mly)

```ml
factor:
  Id_token       
  { Id_as_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")))