BNF Derivations

- Given rules $X ::= yZw$ and $Z ::= v$
- We may replace $Z$ by $v$ to say $X ::= yZw$ and $X ::= yv w$
- 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:
  
  $$<\text{Sum}> ::=$$

BNF Derivations

- Pick a non-terminal
  
  $$<\text{Sum}> ::=$$

BNF Derivations

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

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

Pick a rule and substitute:

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

Pick a non-terminal:

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

Pick a rule and substitute:

\[ <\text{Sum}> ::= 1 \]
\[ <\text{Sum}> => <\text{Sum}> + <\text{Sum}> \]
\[ => ( <\text{Sum}> ) + <\text{Sum}> \]
\[ => ( <\text{Sum}> + <\text{Sum}> ) + <\text{Sum}> \]
\[ => ( <\text{Sum}> + 1 ) + <\text{Sum}> \]
Pick a non-terminal:

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

BNF Derivations

Pick a rule and substitute:

\[ <\text{Sum} > \rightarrow 0 \]
\[ <\text{Sum} > \rightarrow <\text{Sum} > + <\text{Sum} > \]
\[ \rightarrow ( <\text{Sum} > ) + <\text{Sum} > \]
\[ \rightarrow ( <\text{Sum} > + <\text{Sum} > ) + <\text{Sum} > \]
\[ \rightarrow ( <\text{Sum} > + 1 ) + <\text{Sum} > \]
\[ \rightarrow ( <\text{Sum} > + 1 ) + 0 \]

BNF Derivations

(0 + 1) + 0 is generated by grammar

\[ <\text{Sum} > \rightarrow <\text{Sum} > + <\text{Sum} > \]
\[ \rightarrow ( <\text{Sum} > ) + <\text{Sum} > \]
\[ \rightarrow ( <\text{Sum} > + <\text{Sum} > ) + <\text{Sum} > \]
\[ \rightarrow ( <\text{Sum} > + 1 ) + <\text{Sum} > \]
\[ \rightarrow ( <\text{Sum} > + 1 ) + 0 \]
\[ \rightarrow ( 0 + 1 ) + 0 \]

Extended BNF Grammars

Alternatives: allow rules of from \( X ::= y | z \)
Abbreviates \( X ::= y, X ::= z \)
Options: \( X ::= y [v] z \)
Abbreviates \( X ::= y v z, X ::= y z \)
Repetition: \( X ::= y \{v\}^{*} z \)
Can be eliminated by adding new nonterminal \( V \) and rules \( X ::= y z, X ::= y V z, V ::= v, V ::= v V \)
**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>
  - <factor>
    - <bin> * <exp>
      - 1
      - <factor> + <factor>

**Example cont.**
- Use rules: <bin> ::= 1 and <exp> ::= <factor> + <factor>
  - Use rule: <factor> ::= <bin> * <exp>
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>
                1 0

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

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

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

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 cont.

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

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

<Sum> <Sum> <Sum> <Sum> <Sum>

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

<Sum> + <Sum> 0 0 <Sum> + <Sum>

0 1 1 0

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