# Programming Languages and Compilers (CS 421)
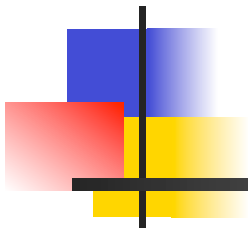
Elsa L Gunter

2112 SC, UIUC

http://www.cs.illinois.edu/class/cs421/

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

# Grammars

- Grammars are formal descriptions of which strings over a given character set are in a particular language

- Language designers write grammar

- Language implementers use grammar to know what programs to accept

- Language users use grammar to know how to write legitimate programs

# 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,...**
  - We call these *terminals*
- Add a set of different characters, **X,Y,Z, ...**
  - 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 Deriviations

- Given rules

$$\mathbf{X} ::= y\mathbf{Z}w \text{ and } \mathbf{Z} ::= v$$

we may replace **Z** by *v* to say

$$\mathbf{X} \Rightarrow y\mathbf{Z}w \Rightarrow yvw$$

- Sequence of such replacements called *derivation*

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

# BNF Derivations

- Start with the start symbol:

<Sum> =>

# BNF Derivations

- Pick a non-terminal

<Sum> =>

# BNF Derivations

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

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

# BNF Derivations

- Pick a non-terminal:

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

# BNF Derivations

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

\<Sum\> => <mark>\<Sum\></mark> + \<Sum \>

=> <mark>( \<Sum\> )</mark> + \<Sum\>

# BNF Derivations

- Pick a non-terminal:

$$\text{<Sum> => <Sum> + <Sum >}$$
$$\text{=> ( <Sum> ) + <Sum>}$$

# BNF Derivations

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

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

=> ( <mark>\<Sum\></mark> ) + \<Sum\>

=> ( <mark>\<Sum\> + \<Sum\></mark> ) + \<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:

&lt;Sum&gt; =&gt; &lt;Sum&gt; + &lt;Sum &gt;

=&gt; ( &lt;Sum&gt; ) + &lt;Sum&gt;

=&gt; ( &lt;Sum&gt; + &lt;Sum&gt; ) + &lt;Sum&gt;

=&gt; ( &lt;Sum&gt; + 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 non-terminal:

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

# BNF Derivations

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

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

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

<Sum> =>
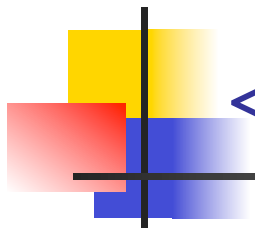
# BNF Semantics

- The meaning of a BNF grammar is the set of all strings consisting only of terminals that can be derived from the Start symbol

# 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::=*vV*

# Regular Grammars

- Subclass of BNF
- Only rules of form
  <nonterminal>::=<terminal><nonterminal> or
  <nonterminal>::=<terminal> or
  <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:

  <Balanced> ::= ε

  <Balanced> ::=  0<OneAndMore>

  <Balanced> ::= 1<ZeroAndMore>

  <OneAndMore> ::= 1<Balanced>

  <ZeroAndMore> ::= 0<Balanced>

- Generates even length strings where every initial substring of even length has same number of 0's as 1's

# 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:     &lt;exp&gt;
                    |
                 &lt;factor&gt;


Use rule: &lt;exp&gt; ::=   &lt;factor&gt;

# Example cont.

- 1 * 1 + 0:     &lt;exp&gt;
                   |
                &lt;factor&gt;
              /      |      \
      &lt;bin&gt;        *          &lt;exp&gt;
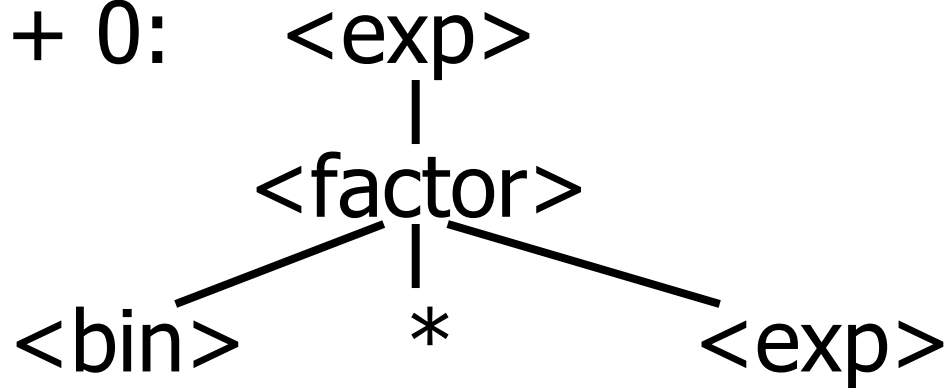
Use rule:  &lt;factor&gt; ::=  &lt;bin&gt; *  &lt;exp&gt;

# Example cont.

- 1 * 1 + 0:

```
                  <exp>
                    |
                <factor>
              /     |      \
         <bin>      *        <exp>
           |                /   |    \
           1          <factor>  +  <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>
                   |            |
                   1            0
```
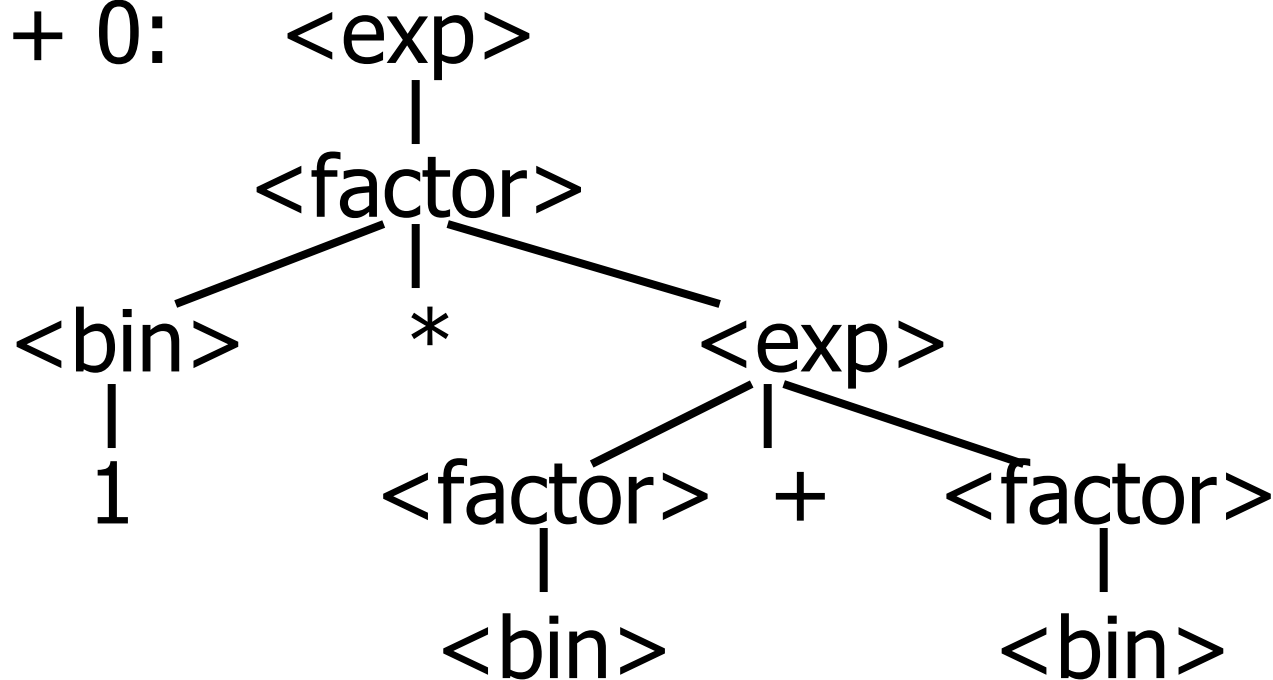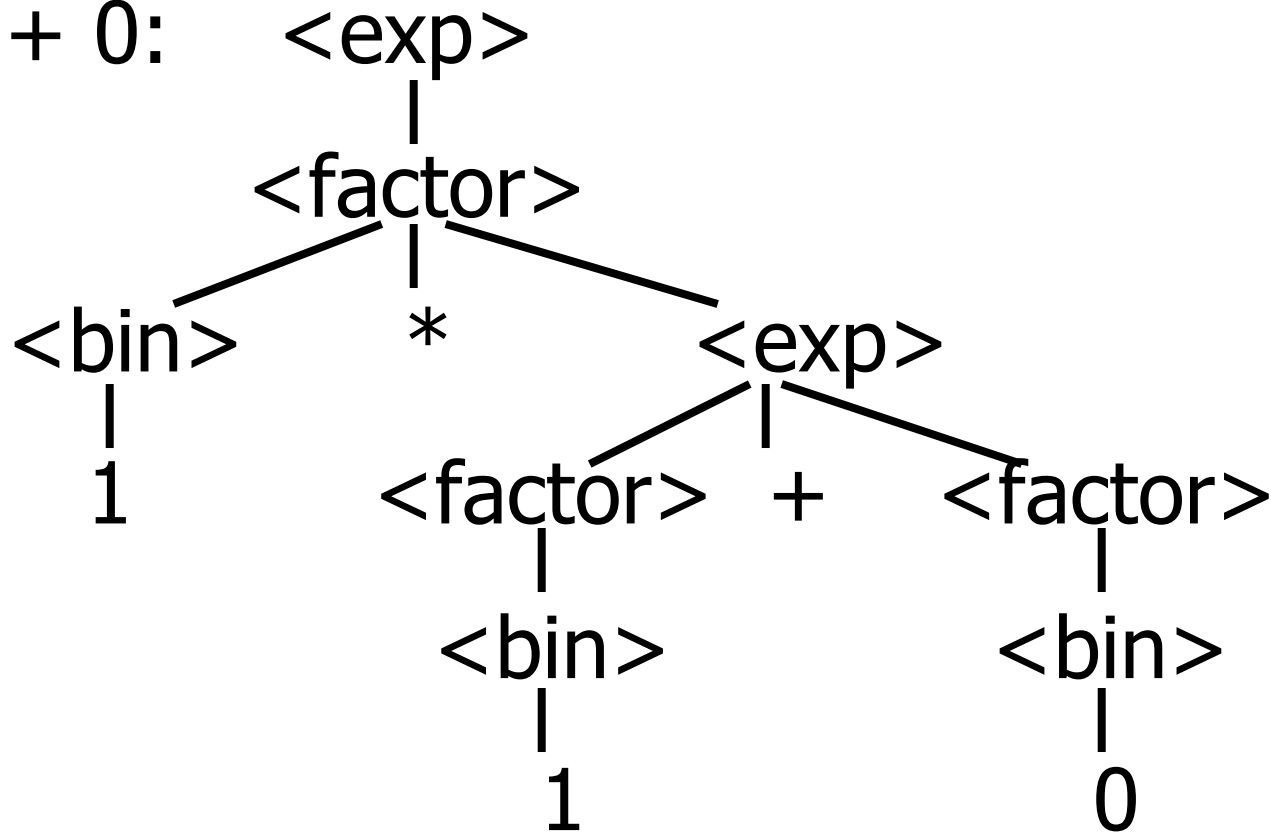
Use rules:  <bin> ::= 1 | 0

# 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

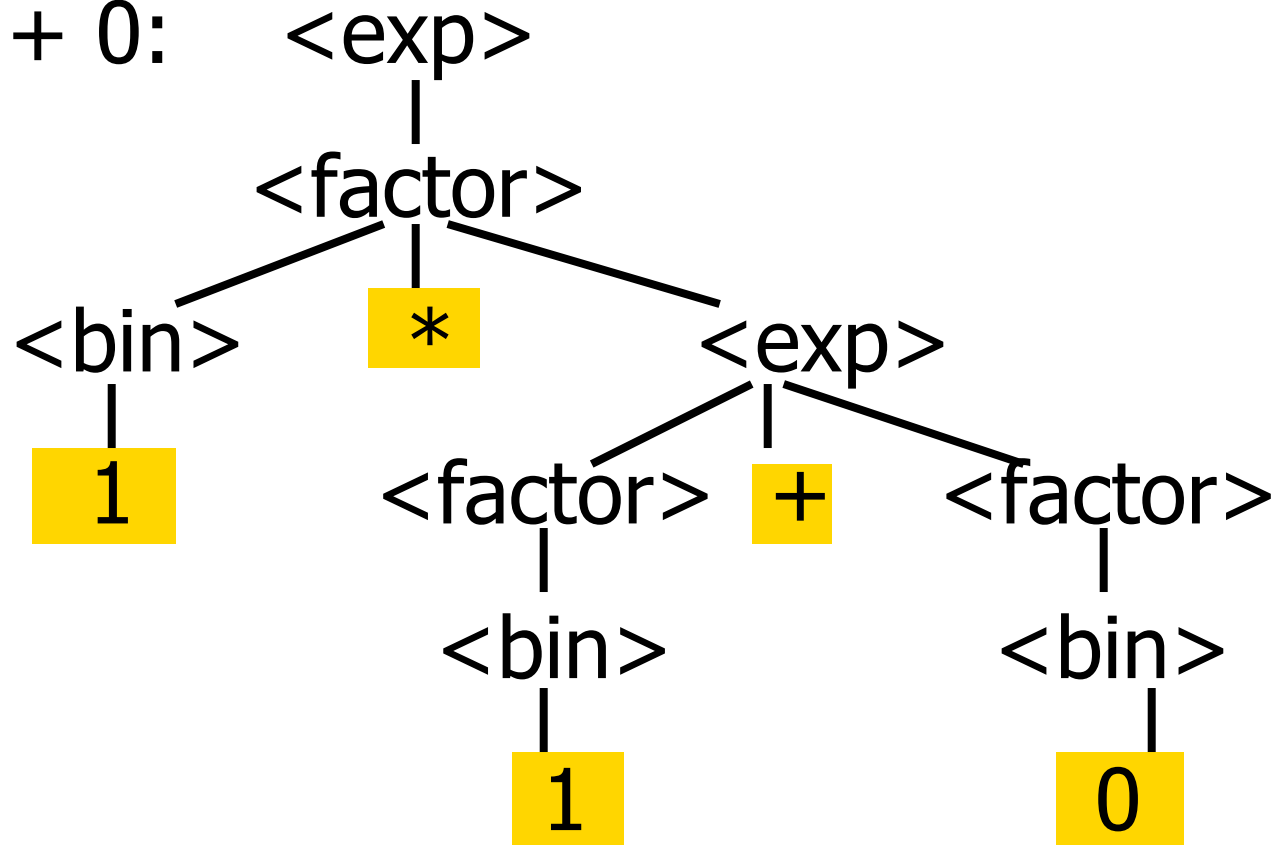# Parse Tree Data Structures

- 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:
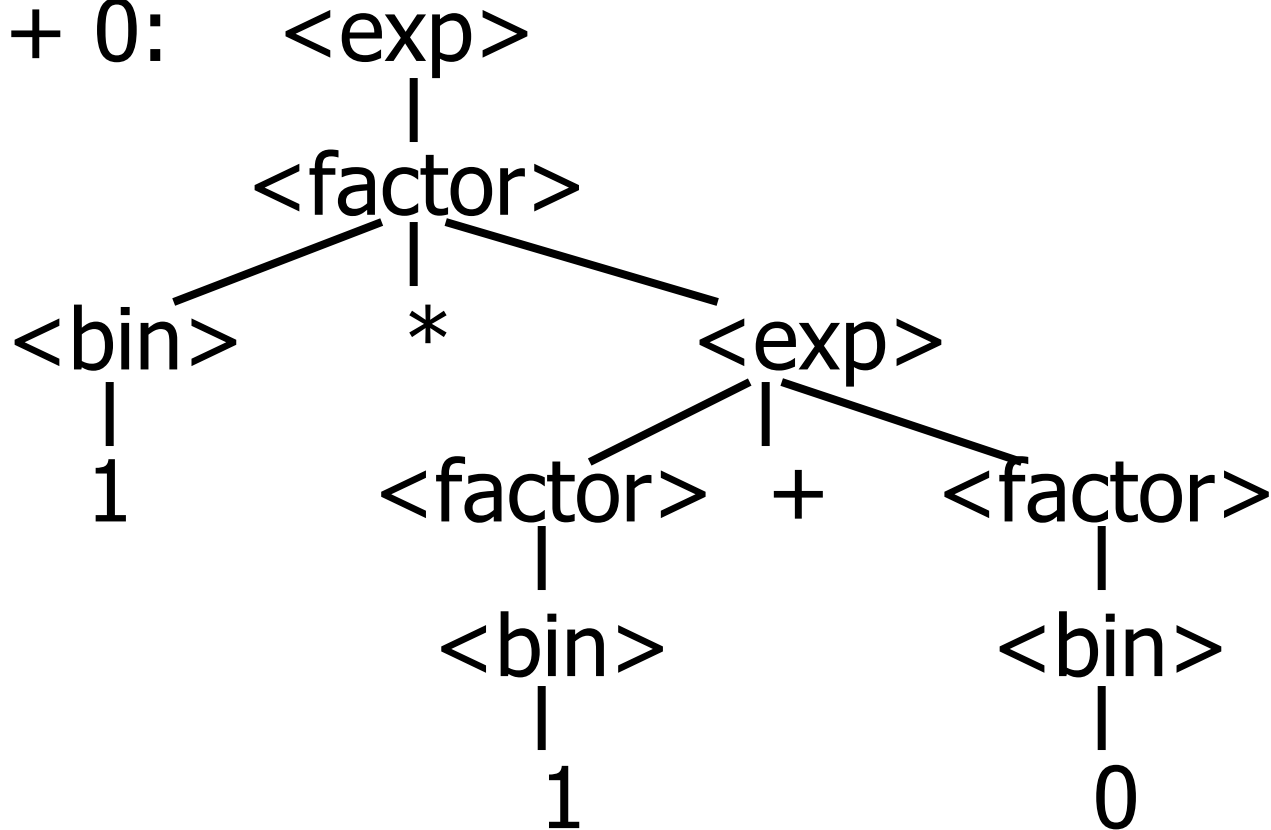
# Example cont.

- Can be represented as

Factor2Exp
(Mult(One,
         Plus(Bin2Factor One,
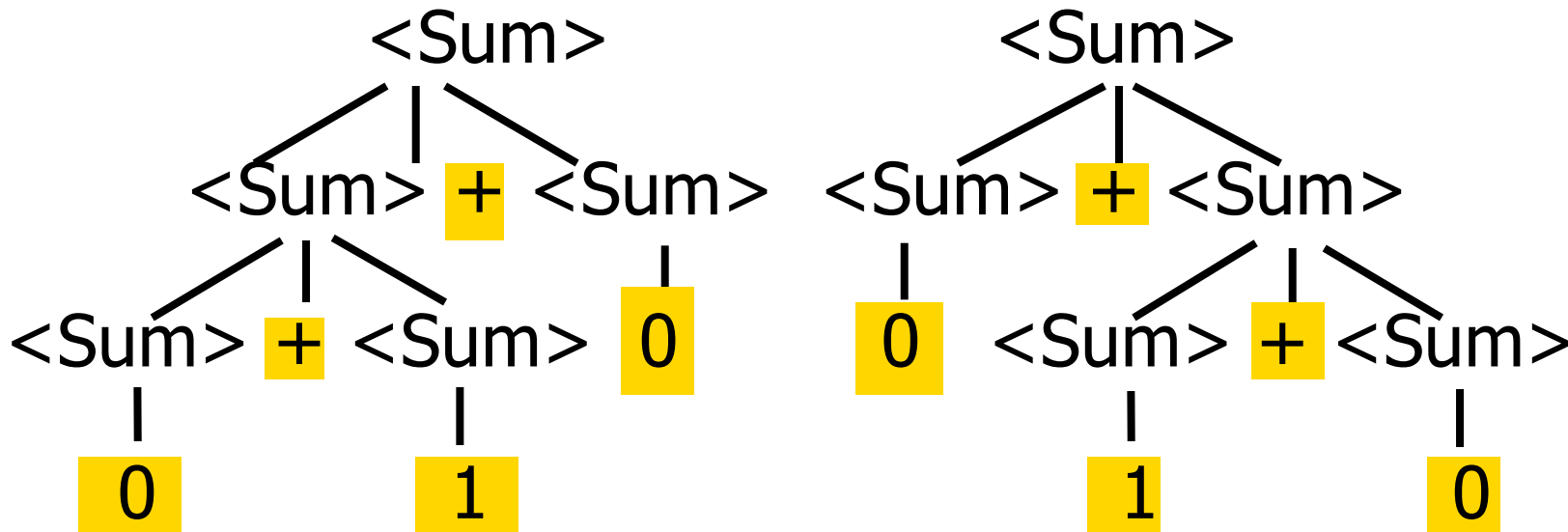                 Bin2Factor Zero)))

# 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

```
            <Sum>                              <Sum>
           /  |  \                            /  |  \
      <Sum>  +  <Sum>                     <Sum>  +  <Sum>
      / |  \       |                        |      / |  \
 <Sum> + <Sum>     0                        0  <Sum> + <Sum>
   |       |                                      |       |
   0       1                                      1       0
```

# Example

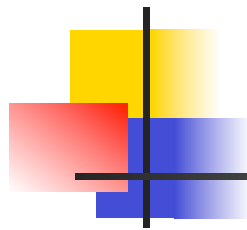- What is the result for:

$$3 + 4 * 5 + 6$$

# 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 assoicativity

- 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 assoicativity, left-most one for left assoiciativity

# 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

| | Fortan | Pascal | C/C++ | Ada | SML |
|---|---|---|---|---|---|
| highest | ** | *, /, div, mod | ++, -- | ** | div, mod, /, * |
| | *, / | +, - | *, /, % | *, /, mod | +, -, ^ |
| | +, - | | +, - | +, - | :: |

# 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> ::= <id> | <exp> + <exp>
        | <exp> * <exp>

- Becomes

 <exp> ::= <mult_exp>
         | <exp> + <mult_exp>
 <mult_exp> ::= <id> | <mult_exp> * <id>