Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/sp2023

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

\[ X ::= yZw \text{ and } Z ::= v \]

we may replace \( Z \) by \( v \) to say

\[ X \Rightarrow yZw \Rightarrow yv w \]

Sequence of such replacements called *derivation*

Derivation called *right-most* if always replace the right-most non-terminal
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> =>
BNF Derivations

- Pick a non-terminal

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

Pick a rule and substitute:

- `<Sum> ::= <Sum> + <Sum>`

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

- Pick a non-terminal:

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

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

  $\Rightarrow ( <Sum> ) + <Sum>$
BNF Derivations

- Pick a non-terminal:

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

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

Pick a rule and substitute:

- `<Sum> ::= <Sum> + <Sum>`
- `<Sum> => <Sum> + <Sum>`
- `=> ( <Sum> ) + <Sum>`
- `=> ( <Sum> + <Sum> ) + <Sum>`
BNF Derivations

- Pick a non-terminal:

\[
\langle \text{Sum} \rangle \Rightarrow \langle \text{Sum} \rangle + \langle \text{Sum} \rangle \\
\Rightarrow ( \langle \text{Sum} \rangle ) + \langle \text{Sum} \rangle \\
\Rightarrow ( \langle \text{Sum} \rangle + \langle \text{Sum} \rangle ) + \langle \text{Sum} \rangle
\]
BNF Derivations

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

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

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

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

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

```
<Sum> => <Sum> + <Sum>
=> ( <Sum> ) + <Sum>
=> ( <Sum> + <Sum> ) + <Sum>
=> ( <Sum> + 1 ) + <Sum>
=> ( <Sum> + 1 ) + 0
```
Pick a non-terminal:

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

\[<Sum> \Rightarrow <Sum> + <Sum>\]
\[\Rightarrow ( <Sum> ) + <Sum>\]
\[\Rightarrow ( <Sum> + <Sum> ) + <Sum>\]
\[\Rightarrow ( <Sum> + 1 ) + <Sum>\]
\[\Rightarrow ( <Sum> + 1 ) + 0\]
\[\Rightarrow ( 0 + 1 ) + 0\]
Extended BNF Grammars

- Alternatives: allow rules of from $X ::= y \mid z$
  - Abbreviates $X ::= y$, $X ::= z$
- Options: $X ::= y [\nu] z$
  - Abbreviates $X ::= y \nu z$, $X ::= y z$
- Repetition: $X ::= y \{\nu\}^* z$
  - Can be eliminated by adding new nonterminal $V$ and rules $X ::= y z$, $X ::= y \nu V z$, $V ::= \nu$, $V ::= \nu V$
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:

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

Problem: Build parse tree for \( 1 * 1 + 0 \) as an \( \texttt{<exp>} \)
Example cont.

1 * 1 + 0:  <exp>

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

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

Use rule: \[ \text{<exp>} ::= \text{<factor>} \]
Example cont.

1 * 1 + 0: \(<exp>\)

\(<\text{factor}>\)

\(<\text{bin}>\) * \(<exp>\)

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 

$<\text{exp}> ::= <\text{factor}> + <\text{factor}>$
Example cont.

1 * 1 + 0:  <exp>

Use rule:  <factor> ::= <bin>
Example cont.

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

\(<\text{factor}>\)

\(<\text{bin}>\) * \(<\text{exp}>\)

\(<\text{bin}>\) 1 \(<\text{factor}>\) + \(<\text{factor}>\)

\(<\text{bin}>\) 1 \(<\text{bin}>\) 0

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

```
1 * 1 + 0:  <exp>
    /     \
   /       /
 <factor> <exp>
     /     /  \
 <bin> *   <exp>
    /     /  /  \
  1     *  +   <factor>
     /     /  /
 <bin> 1   <bin> 0
```

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

```
<exp>
  /
 /  \
<fact>  +  <fact>
  /
 /  \
 <b>  *  <e> <b>  *  <e>
```
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:

- $\text{exp} ::= \text{factor} \mid \text{factor} + \text{factor}$
- $\text{factor} ::= \text{bin} \mid \text{bin} * \text{exp}$
- $\text{bin} ::= 0 \mid 1$

Type definitions:

- $\text{exp} = \text{Factor2Exp of factor}$
  - $\mid \text{Plus of factor} * \text{factor}$
- $\text{factor} = \text{Bin2Factor of bin}$
  - $\mid \text{Mult of bin} * \text{exp}$
- $\text{bin} = \text{Zero} \mid \text{One}$
Example cont.

1 * 1 + 0:

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

- Can be represented as

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

0 + 1 + 0

<Sum> + <Sum> 0

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

0 1 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:

$$7 - 5 - 2$$
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