CSCI-C311 Programming Languages

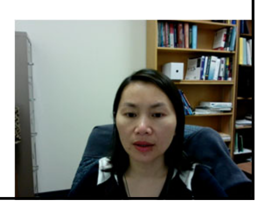
Context-Free Grammars and Parsing

Dr. Hang Dinh

1

Outline and Reading

- After this lecture, you will learn
  - Context-free grammar
  - Parse tree
  - Classes of parsing algorithms
  - Difference between top-down parsing and bottom-up parsing
- Reading
  - Scott 4e - Section 2.1.2, 2.1.3
  - Scott 4e –Section 2.3 (but not 2.3.1 and 2.3.2)



2

Definition of Context-Free Grammars (CFG)

- Recall: CFG is used in syntax analysis, the 2nd phase of compilation
- Formally, a CFG consists of
  - A set *T* of *terminals*
  - A set *N* of *non-terminals* (or *variables*)
  - A *start symbol* *S* (a non-terminal)
  - A set of *productions* (or *rules*), each production is of the form:

$$x \rightarrow w$$

where x is a non-terminal and w is a string of terminals and non-terminals.

- The first rule of a CGF must have the start symbol on the left-hand side

3

CFG Notation

- Several rules of the same left-hand variables can be abbreviated into a single rule using the | meta-symbol :
  - Original rules: $X \rightarrow w_1$

$$X \rightarrow w_2$$

$$\dots$$

$$X \rightarrow w_k$$

The order of these rules doesn't matter.

  - Abbreviated rule: $X \rightarrow w_1 \,|\, w_2 \,|\, \dots \,|w_k$

$\boxed{op \rightarrow + \,|\, * \,|\, := \,|\, -}$ is shorthand for

$$\begin{array}{l} op \rightarrow + \\ op \rightarrow - \\ op \rightarrow * \\ op \rightarrow := \end{array}$$

or

$$\begin{array}{l} op \rightarrow + \\ \quad\ \ \rightarrow - \\ \quad\ \ \rightarrow * \\ \quad\ \ \rightarrow := \end{array}$$

4

Example of CFG

- The expression grammar
  - Terminals: id number + − * / ( )
  - Non-terminals: *expr* *op*
  - Start symbol: *expr*
  - The set of rules:

$$
\begin{aligned}
\mathit{expr} &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ \mathit{expr} \mid \texttt{(}\ \mathit{expr}\ \texttt{)} \\
&\qquad \mid \mathit{expr}\ \mathit{op}\ \mathit{expr} \\
\mathit{op} &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

- Convention: In this lecture, all non-terminals are written *italic*.

5

Backus-Naur-Form (BNF)

- The notation for CFG is sometimes called Backus-Naur Form (BNF)
  - Kleene star and meta-level parentheses of regular languages are not allowed
- Extended BNF allows
  - Kleene star, and meta-level parentheses
  - Kleene plus (+), which indicates one or more instances of symbol or group of symbols in front of it.
- Example:
  - $\mathit{idlist} \rightarrow \mathrm{id}\ (\ ,\ \mathrm{id}\ )^*$ is shorthand for
    $\mathit{idlist} \rightarrow \mathrm{id}$
    $\mathit{idlist} \rightarrow \mathit{idlist}\ ,\mathrm{id}$
  - $\mathit{idlist} \rightarrow \mathrm{id}\ (\ ,\ \mathrm{id}\ )^+$ is shorthand for
    $\mathit{idlist} \rightarrow \mathrm{id}\ ,\mathrm{id}$
    $\mathit{idlist} \rightarrow \mathit{idlist}\ ,\mathrm{id}$

6

Avoid Confusion in Extended BNF

- The ( ) in the following grammar are characters that are part of the language defined by the grammar

$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad\ \mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

- To avoid confusing with the meta-symbols ( ) in extended BNF, some authors use quote marks around non-meta-symbol characters
  - $idlist \rightarrow$ id (‘ , ’ id)*
  - $expr \rightarrow$ ‘ (’ $expr$ ‘) ’

7

Derivations in CFG

- A CFG shows how to generate a syntactically valid string of terminals
  - Begin with start symbol S
  - Choose a rule $S \rightarrow w_1$
  - Replace S with w_1
  - Choose a nonterminal A in resulting string
  - Choose a rule $A \rightarrow w_2$
  - Replace A with w_2, so on….

$$
\begin{aligned}
expr &\longrightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\ expr \mid \texttt{(}\ expr\ \texttt{)} \\
&\quad\ \mid expr\ op\ expr \\
op &\longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
expr &\Longrightarrow expr\ op\ \underline{expr} \\
&\Longrightarrow expr\ \underline{op}\ \texttt{id} \\
&\Longrightarrow \underline{expr}\ \texttt{+ id} \\
&\Longrightarrow expr\ op\ \underline{expr}\ \texttt{+ id} \\
&\Longrightarrow expr\ \underline{op}\ \texttt{id + id} \\
&\Longrightarrow \underline{expr}\ \texttt{* id + id} \\
&\Longrightarrow \underset{\texttt{(slope)}}{\texttt{id}}\ \texttt{*}\ \underset{\texttt{(x)}}{\texttt{id}}\ \texttt{+}\ \underset{\texttt{(intercept)}}{\texttt{id}}
\end{aligned}
$$

•Grammar on the left generates the string **"slope * x + intercept"**

8

Derivations in CFG

- The $\Rightarrow$ metasymbol means "derives"
  - Indicates a replacement operation
- *Derivation*
  - is a series of replacement operations that shows how to derive a string of terminals from the start symbol
  - A string along the way is a *sentential form*.
  - Final sentential form is called the *yield*
- The $\Rightarrow^*$ metasymbol means
  - "derives after zero or more replacements"

$$
\begin{aligned}
\mathit{expr} &\Longrightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}} \\
&\Longrightarrow \mathit{expr}\ \underline{\mathit{op}}\ \texttt{id} \\
&\Longrightarrow \underline{\mathit{expr}}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \mathit{expr}\ \underline{\mathit{op}}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \underline{\mathit{expr}}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \underset{(\texttt{slope})}{\texttt{id}}\ \texttt{*}\ \underset{(\texttt{x})}{\texttt{id}}\ \texttt{+}\ \underset{(\texttt{intercept})}{\texttt{id}}
\end{aligned}
$$

$$\mathit{expr} \Rightarrow^* \text{slope} * \text{x} + \text{intercept}$$

9

Leftmost vs. Rightmost Derivations

- *Right-most* derivation
  - At each step, choose to replace the *right-most* nonterminal with the right-hand side of some production
- *Left-most* derivation
  - At each step, choose to replace the *left-most* nonterminal with the right-hand side of some production

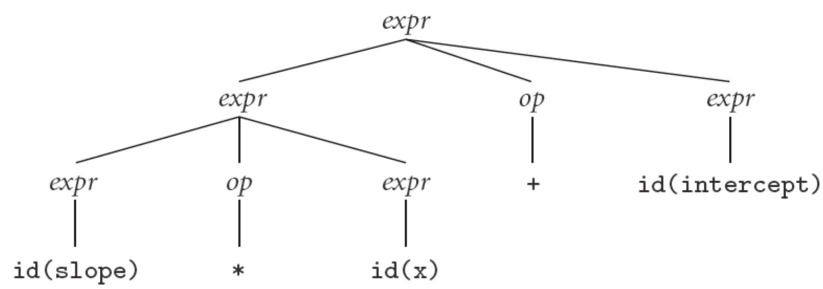
$$
\begin{aligned}
\mathit{expr} &\Longrightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}} \\
&\Longrightarrow \mathit{expr}\ \underline{\mathit{op}}\ \texttt{id} \\
&\Longrightarrow \underline{\mathit{expr}}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \mathit{expr}\ \underline{\mathit{op}}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \underline{\mathit{expr}}\ \texttt{*}\ \texttt{id}\ \texttt{+}\ \texttt{id} \\
&\Longrightarrow \underset{(\texttt{slope})}{\texttt{id}}\ \texttt{*}\ \underset{(\texttt{x})}{\texttt{id}}\ \texttt{+}\ \underset{(\texttt{intercept})}{\texttt{id}}
\end{aligned}
$$

10

$$
\begin{array}{lll}
\textit{expr} & \Longrightarrow & \textit{expr op } \underline{\textit{expr}} \\
 & \Longrightarrow & \textit{expr } \underline{\textit{op}} \texttt{ id} \\
 & \Longrightarrow & \underline{\textit{expr}} \texttt{ + id} \\
 & \Longrightarrow & \textit{expr op } \underline{\textit{expr}} \texttt{ + id} \\
 & \Longrightarrow & \textit{expr } \underline{\textit{op}} \texttt{ id + id} \\
 & \Longrightarrow & \underline{\textit{expr}} \texttt{ * id + id} \\
 & \Longrightarrow & \underset{\texttt{(slope)}}{\texttt{id}} \texttt{ * } \underset{\texttt{(x)}}{\texttt{id}} \texttt{ + } \underset{\texttt{(intercept)}}{\texttt{id}}
\end{array}
$$

Parse Trees

- Represent a derivation graphically as a parse tree
  - Parse tree for **`"slope * x + intercept"`**



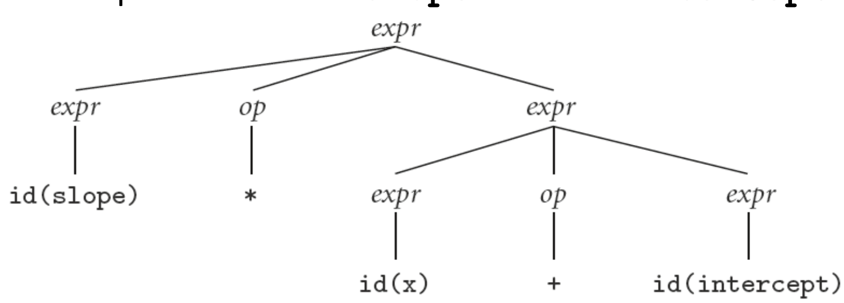
- Many derivations can be represented by the same parse tree.

11

Ambiguous Grammars

$$
\begin{array}{lll}
\textit{expr} & \longrightarrow & \texttt{id} \mid \texttt{number} \mid \texttt{-}\ \textit{expr} \mid \texttt{(}\ \textit{expr}\ \texttt{)} \\
 & & \mid \textit{expr op expr} \\
\textit{op} & \longrightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{array}
$$

- Alternative parse tree for **`"slope * x + intercept"`**



- A grammar that generates different parse trees for the same string of terminals is said to be *ambiguous*.

12

Example of Unambiguous Grammars

- Unambiguous version of expression grammar, which captures
  - *Precedence*: multiplication, division before addition, subtraction
  - *Associativity*: operators group left to right,

    so that **`3+4+5`** means **`(3+4)+5`** rather than **`3+(4+5)`**

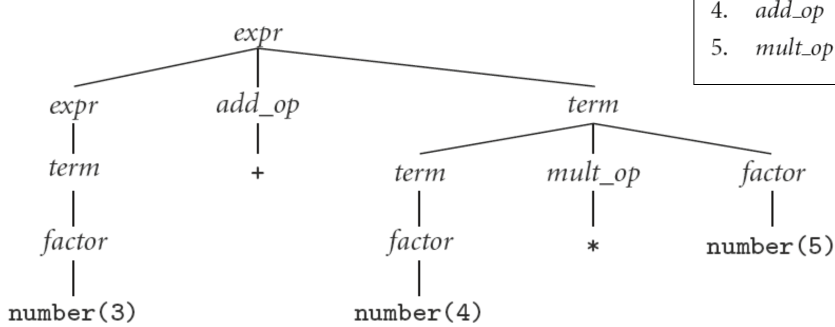
1. *expr* $\longrightarrow$ *term* | *expr add_op term*
2. *term* $\longrightarrow$ *factor* | *term mult_op factor*
3. *factor* $\longrightarrow$ `id` | `number` | `-` *factor* | `(` *expr* `)`
4. *add_op* $\longrightarrow$ `+` | `-`
5. *mult_op* $\longrightarrow$ `*` | `/`

13

Example of Unambiguous Grammars

- Parse tree in unambiguous expression grammar for **`3 + 4 * 5`**

1. *expr* $\longrightarrow$ *term* | *expr add_op term*
2. *term* $\longrightarrow$ *factor* | *term mult_op factor*
3. *factor* $\longrightarrow$ `id` | `number` | `-` *factor* | `(` *expr* `)`
4. *add_op* $\longrightarrow$ `+` | `-`
5. *mult_op* $\longrightarrow$ `*` | `/`



14

CFG and Parsing

- CFG is a *generator* for a *context-free language* (CFL)
  - *The language of a CFG* is the set of strings of terminals generated by the CFG.
  - A CFL is simply the language of some context-free grammar.
- There is an infinite number of CFGs for every CFL
  - not all grammars are created equal, however
- A parser is a context-free language *recognizer*
- For any CFG, we can create a parser that runs in $O(n^3)$ time
  - *Early's algorithm*
  - *Cooke-Younger-Kasami (CYK) algorithm*

15

Parsing Algorithms

- $O(n^3)$ time is **too slow** for parsing sizable programs
- There are large classes of grammars for which we can build parsers that run in linear time $O(n)$
  - The two most important classes are called: **LL** and **LR**

Class	Direction of Scanning	Derivation discovered	Parse tree construction	Algorithm used
LL	**L**eft-to-right	**L**eft-most	Top-down	predictive
LR	**L**eft-to-right	**R**ight-most	Bottom-up	shift-reduce

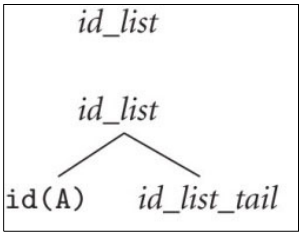
  - **LR** parsers are more common than **LL** parsers

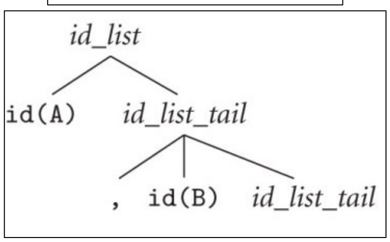
16

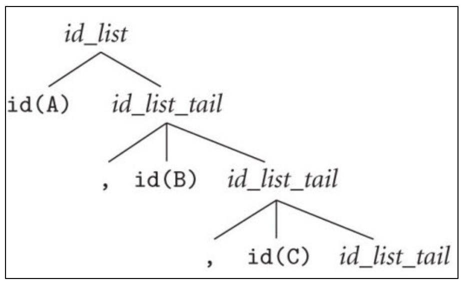
Example of Top-down Parsing

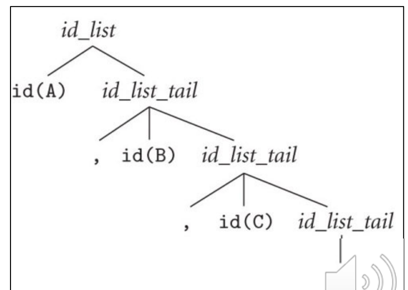
- Top-down parsing for input string
  `A, B, C;`
- Given the CFG for comma-separated list of identifiers ended by a ;

$$
\begin{aligned}
&id\_list \longrightarrow \texttt{id}\ id\_list\_tail\\
&id\_list\_tail \longrightarrow \texttt{,}\ \texttt{id}\ id\_list\_tail\\
&id\_list\_tail \longrightarrow \texttt{;}
\end{aligned}
$$







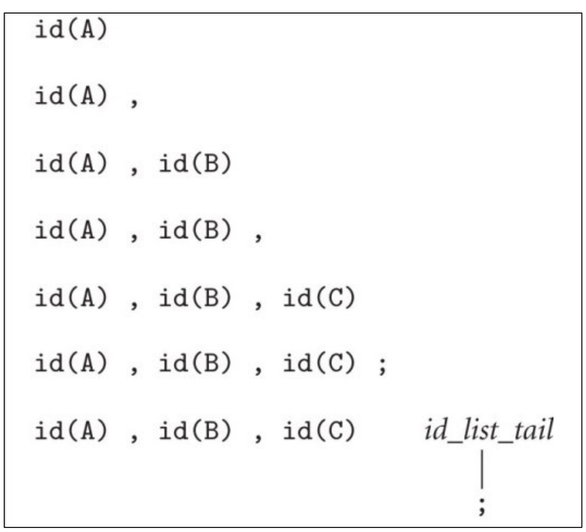


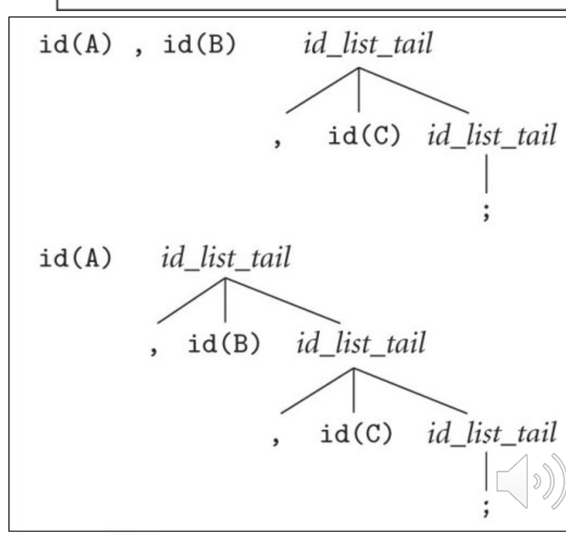
17

Example of Bottom-Up Parsing

$$
\begin{aligned}
&id\_list \longrightarrow \texttt{id}\ id\_list\_tail\\
&id\_list\_tail \longrightarrow \texttt{,}\ \texttt{id}\ id\_list\_tail\\
&id\_list\_tail \longrightarrow \texttt{;}
\end{aligned}
$$

- Top-down parsing for input string
  `A, B, C;`





18

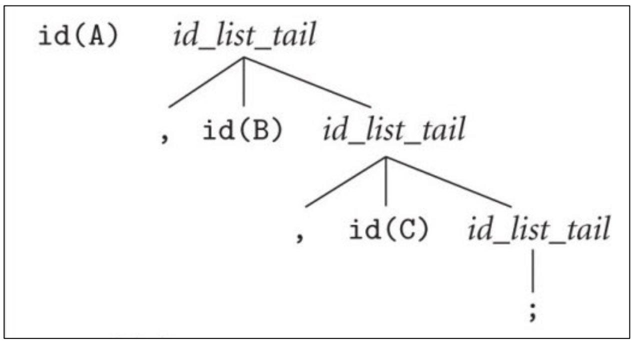
Example of Bottom-Up Parsing

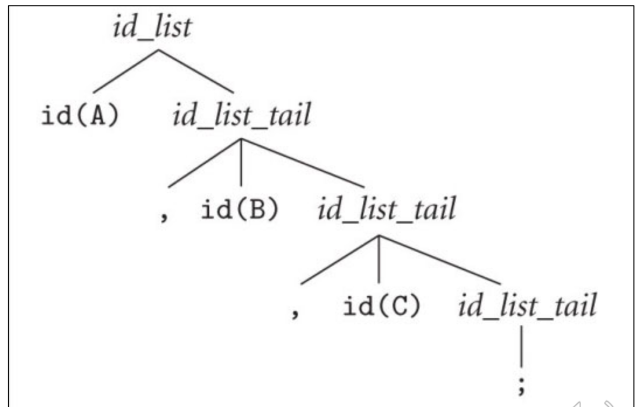
- Top-down parsing for input string
  `A, B, C;`

$id\_list \longrightarrow$ `id` id_list_tail

$id\_list\_tail \longrightarrow$ `, id` id_list_tail

$id\_list\_tail \longrightarrow$ `;`





19

Subclasses of Linear Time Parsers

- There are several important sub-classes of **LR** parsers
  - **SLR** (Simple LR)
  - **LALR** (Look-ahead LR)

  Important for their ease of implementation
  - "full **LR**": Important for its generality
- Parser classes written with a number in ( ) after main class name
  - Examples: **LL(2)** or **LALR(1)**
  - This number indicates how many tokens of look-ahead are required in order to parse.
  - Almost all real compilers use one token of look-ahead
  - We will cover **LL(1)** and **SLR(1)** in the last two lectures.

20