Chapter 2: Context-Free Grammars & Syntax Analysis

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Top-Down Parsing Techniques
- Bottom-Up Parsing Techniques

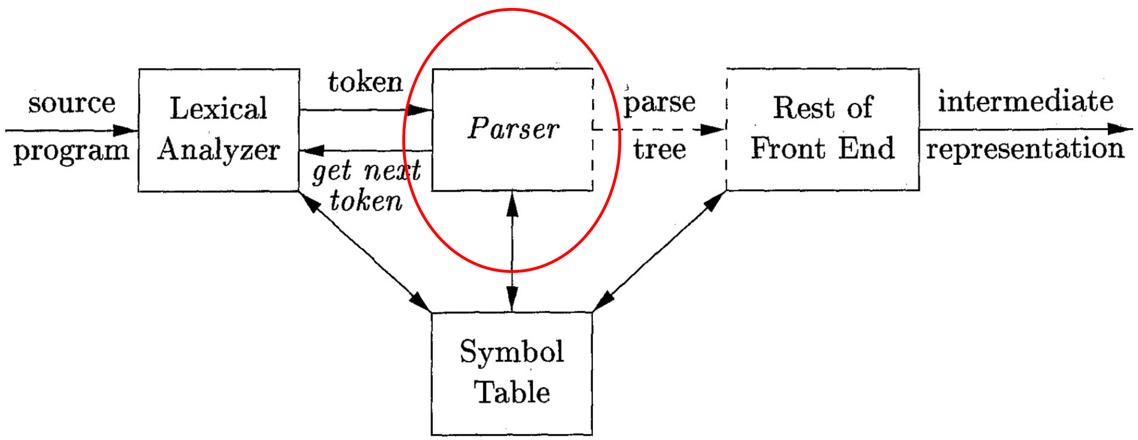
Describing Syntax

- The syntax of programming language constructs can be specified by context-free grammars[1]
  - A grammar gives a precise syntactic specification of a programming language, defining program structures
  - For certain grammars, we can automatically construct an efficient parser for the corresponding language

[1]Can also be specified using BNF (Backus-Naur Form) notation, which basically can be seen as a variant of CFG: http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node23.html

The Role of the Parser

- The parser obtains a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language
- Report syntax errors in an intelligent fashion
- For well-formed programs, the parser constructs a parse tree



Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
  - Formal definition of CFG
  - Derivation and parse tree
  - Ambiguity
  - CFG vs. regexp
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

Context-Free Grammar (上下文无关文法)

- **A context-free grammar (CFG) consists of four parts:**
  - **Terminals (终结符号):** Basic symbols from which strings are formed (token names)
  - **Nonterminals (非终结符号):** Syntactic variables that denote sets of strings
    - Usually correspond to a language construct, such as `stmt` (statements)
  - One nonterminal is distinguished as the **start symbol (开始符号)**
    - The set of strings denoted by the start symbol is the language generated by the CFG
  - **Productions (产生式):** Specify how the terminals and nonterminals can be combined to form strings
    - **Format:** head → body
    - head must be a nonterminal; body consists of zero or more terminals/nonterminals
    - **Example:** *expression* → *expression* + *term*

CFG Example

- The grammar below defines simple arithmetic expressions
  - Terminal symbols: `id`, `+`, `-`, `*`, `/`, `(`, `)`
  - Nonterminals: *expression*, *term* (项), *factor* (因子)
  - Start symbol: *expression*
  - Productions:
    - *expression* → *expression* + *term*
    - *expression* → *expression* – *term*
    - *expression* → *term*
    - *term* → *term* * *factor*
    - *term* → *term* / *factor*
    - *term* → *factor*
    - *factor* → (*expression*)
    - *factor* → **id**

→ can be read as:
can be in the form, can be replaced by, can be re-written as, can produce, can generate, can make…

Notational Simplification

```
expression → expression + term
expression → expression - term
expression → term
term → term * factor
term → term / factor
term → factor
factor → ( expression )
factor → id
```

⇒

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | id
```

- | is a meta symbol to specify alternatives
- (and) are not meta symbols, they are terminal symbols

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
    - Formal definition of CFG
    - Derivation and parse tree
    - Ambiguity
    - CFG vs. regexp
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

Derivations

- **Derivation (推导):** Starting with the start symbol, nonterminals are rewritten using productions until only terminals remain

- Example:
  - **CFG:** $E \rightarrow -\ E \mid E + E \mid E * E \mid (\ E\ ) \mid \mathbf{id}$
  - A derivation (a sequence of rewrites) of $-(\mathbf{id})$ from E
    - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$

Notations

- $\Rightarrow$ means "derives in one step"
- $\overset{*}{\Rightarrow}$ means "derives in zero or more steps"
  - $\alpha \overset{*}{\Rightarrow} \alpha$ holds for any string α
  - If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$
  - Example: $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$ can be written as $E \overset{*}{\Rightarrow} -(\mathbf{id})$
- $\overset{+}{\Rightarrow}$ means "derives in one or more steps"

Terminologies

- If $S \overset{*}{\Rightarrow} \alpha$, where S is the start symbol of a grammar G, we say that α is a ***sentential form*** of G (文法的句型)
  - May contain both terminals and nonterminals, and may be empty
  - **Example:** $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$, here all strings of grammar symbols are sentential forms

- A ***sentence*** **(句子)** of G is a sentential form without nonterminals
  - In the above example, only the last string $-(\mathbf{id}+\mathbf{id})$ is a sentence

- The *language generated* by a grammar is its set of sentences

Leftmost/Rightmost Derivations

- At each step of a derivation, we need to choose which nonterminal to replace

- In **leftmost derivations (最左推导)**, the leftmost nonterminal in each sentential form is always chosen to be replaced
  - $E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$

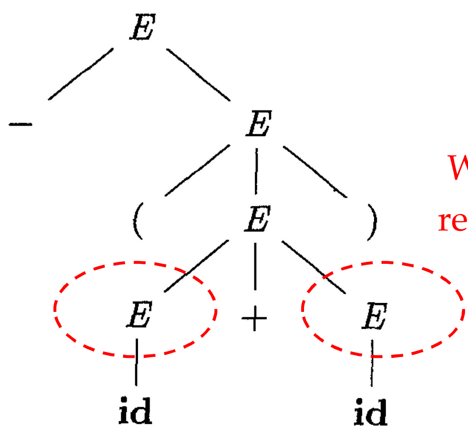
- In **rightmost derivations (最右推导)**, the rightmost nonterminal is always chosen to be replaced
  - $E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+\mathbf{id}) \underset{rm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$

Parse Trees (语法分析树)

- A ***parse tree*** is a graphical representation of a derivation that filters out the order in which productions are applied
  - The root node (根结点) is the start symbol of the grammar
  - Each leaf node (叶子结点) is labeled by a terminal symbol*
  - Each interior node (内部结点) is labeled with a nonterminal symbol and represents the application of a production
    - The interior node is labeled with the nonterminal in the head of the production; the children nodes are labeled, from left to right, by the symbols in the body of the production

CFG: $E \rightarrow - E \mid E + E \mid E * E \mid (E) \mid \textbf{id}$

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\textbf{id}+E) \underset{lm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$



Which one is replaced first?

* Here, we assume that a derivation always produces a string with only terminals, so leaf nodes cannot be non-terminals.

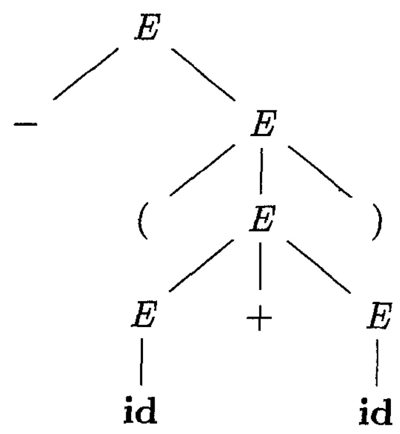
Parse Trees (语法分析树) Cont.

- The leaves, from left to right, constitute a **sentential form** of the grammar, which is called the *yield* or *frontier* of the tree
- There is a **many-to-one** relationship between derivations and parse trees
  - However, there is a **one-to-one** relationship between leftmost/rightmost derivations and parse trees

CFG: $E \rightarrow -\ E \mid E + E \mid E * E \mid (E) \mid \mathbf{id}$

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+\mathbf{id}) \underset{rm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$



Both derivations correspond to the parse tree.

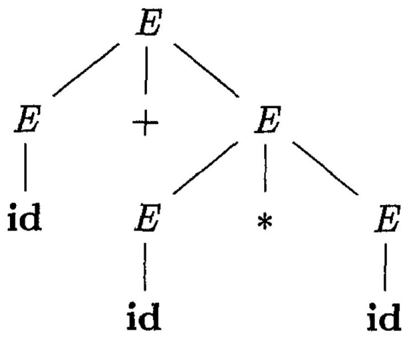
Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
  - Formal definition of CFG
  - Derivation and parse tree
  - Ambiguity
  - CFG vs. regexp
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

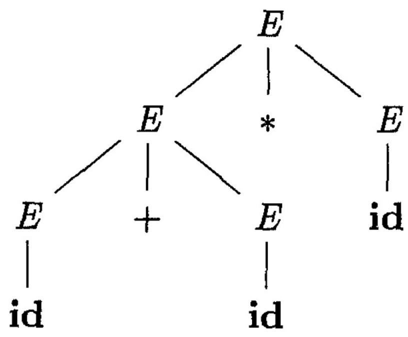
Ambiguity (二义性)

- Given a grammar, if there are more than one parse tree for some sentence, it is ambiguous.
- Example CFG: $E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$



$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$



Both are leftmost derivations

The left tree corresponds to the commonly assumed precedence.

Ambiguity (二义性) Cont.

- The grammar of a programming language typically needs to be unambiguous
  - Otherwise, there will be multiple ways to interpret a program
  - Given $E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$, how to interpret $a + b * c$?

- In some cases, it is convenient to use carefully chosen ambiguous grammars, together with disambiguating rules to discard undesirable parse trees
  - For example: multiplication before addition

Outline

- Introduction: Syntax and Parsers
- Context-Free Grammars
- Overview of Parsing Techniques
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators (Lab)

- Formal definition of CFG
- Derivation and parse tree
- Ambiguity
- CFG vs. regexp

CFG vs. Regular Expressions

- **CFGs are more expressive than regular expressions**

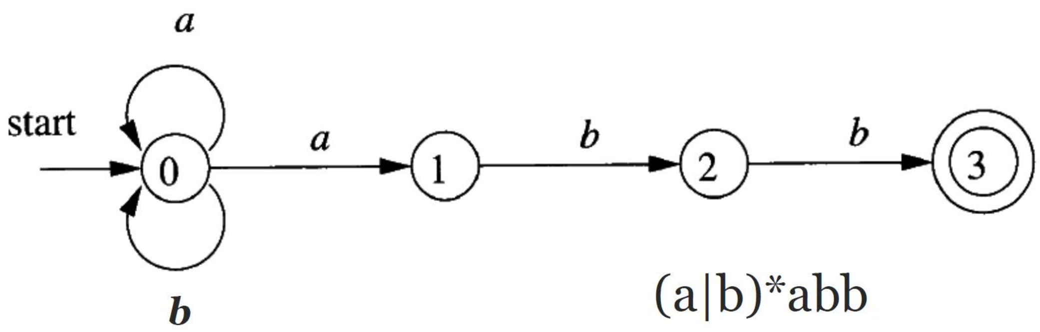
  1. Every language that can be described by a regular expression can also be described by a grammar (i.e., every regular language is also a context-free language)
  2. Some context-free languages cannot be described using regular expressions

Any Regular Language Can be Described by a CFG

- **(Proof by Construction)** Each regular language can be accepted by an NFA. We can construct a CFG to describe the language:
  - For each state i of the NFA, create a nonterminal symbol A_i
  - If state i has a transition to state j on input a, add the production $A_i \rightarrow aA_j$
  - If state i goes to state j on input ϵ, add the production $A_i \rightarrow A_j$
  - If i is an accepting state, add $A_i \rightarrow \epsilon$
  - If i is the start state, make A_i be the start symbol of the grammar

Example: NFA to CFG

- $A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$
- $A_1 \rightarrow bA_2$
- $A_2 \rightarrow bA_3$
- $A_3 \rightarrow \epsilon$



(a|b)*abb

Consider the string *baabb*: The process of the NFA accepting the sentence corresponds exactly to the derivation of the sentence from the grammar

Some Context-Free Languages Cannot be Described Using Regular Expressions

- Example: $L = \{a^n b^n \mid n > 0\}$
  - The language L can be described by CFG $S \rightarrow aSb \mid ab$
  - L cannot be described by regular expressions. In other words, we cannot construct a DFA to accept L

Proof by Contradiction

- Suppose there is a DFA D that accepts L and D has k states
- When processing a^{k+1} ..., D must enter a state s more than once (D enters one state after processing a symbol)[1]
- Assume that D enters the state s after reading the ith and jth a ($i \neq j, i \leq k+1, j \leq k+1$)
- Since D accepts L, $a^j b^j$ must reach an accepting state. There must exist a path labeled b^j from s to an accepting state
- Since a^i reaches the state s and there is a path labeled b^j from s to an accepting state, D will accept $a^i b^j$. Contradiction!!!

[1] $a^{k+1}b^{k+1}$ is a string in L so D must accept it