Compilation Principle
编 译 原 理

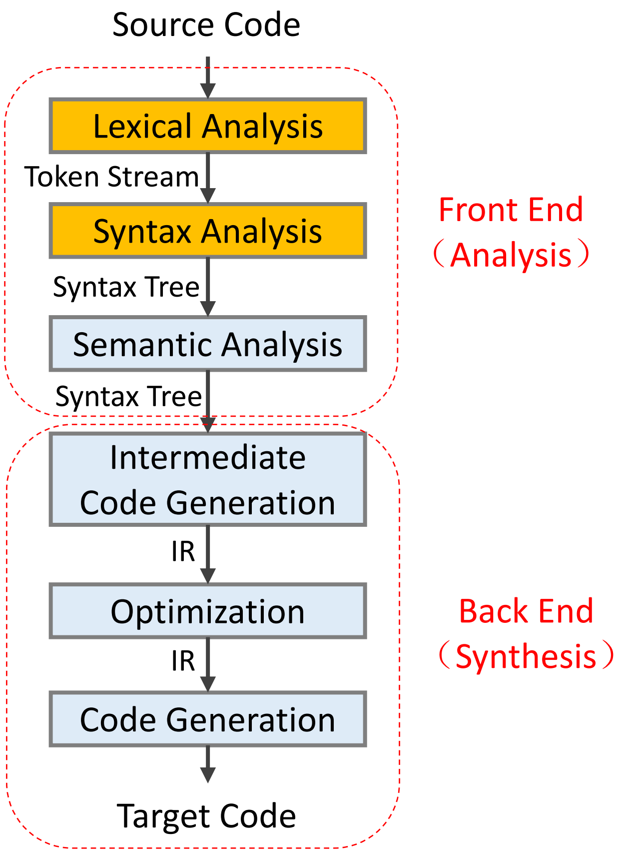
第4讲：语法分析(1)

张献伟

xianweiz.github.io

DCS290, 03/11/2021

Compilation Phases



Syntax Analysis[语法分析]

- Second phase of compilation
  - Also called as **parser**

- Parser obtains a string of tokens from the lexical analyzer
  - **Lexical analyzer** reads the chars of the source program, groups them into lexically meaningful units called **lexemes**
  - and produces as output **tokens** representing these lexemes
    - Token: <token name, attribute value>
  - Token names are used by parser for syntax analysis
    - tokens → parse tree/AST
- Parse tree[分析树]
  - Graphically represent the syntax structure of the token stream

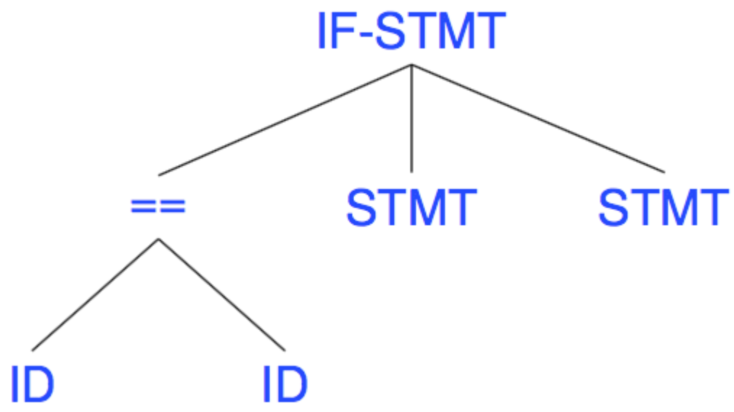
Parsing Example

- Input: if(x==y) … else …

- Parser input (Lexical output):

  KEY(IF) ‘(‘ ID(x) OP(‘==‘) ID(y) ‘)’ … KEY(ELSE) …

- Parser output



Parsing Example (cont.)

- Example: <id, x> <op, *> <op, %>
  - Is it a valid token stream in C language? **YES**
  - Is it a valid statement in C language? **NO**

- Not every string of tokens are valid
  - Parser must distinguish between valid and invalid token strings

- We need a method to describe what is valid string?
  - To specify the syntax of a programming language

How to Specify Syntax

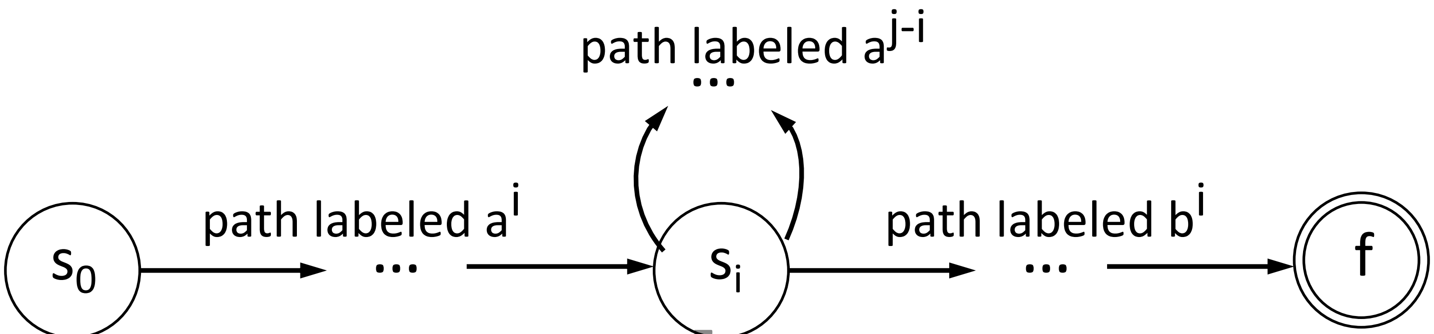
- How can we specify a syntax with nested structures?
  - Is it possible to use RE/FA?
  - L(Regular Expression) ≡ L(Finite Automata)

- RE/FA is not powerful enough

- Example: matching parenthesis: # of '(' == # of ')'
  - (x+y)*z ✓
  - ((x+y)+y)*z ✓
  - (...(((x+y)+y)+y)...) ✓
  - ((x+y)+y)+y)*z ✗

RE/FA is NOT Powerful Enough

- $L = \{a^n b^n \mid n \geq 1\}$ is NOT a Regular Language
  - Suppose ***L*** were the language defined by regular expression
  - Then we could construct a DFD ***D*** with ***k*** states to accept ***L***
  - Since ***D*** has only ***k*** states, for an input beginning with more than ***k a***'s, ***D*** must enter some state twice, say s_i
  - Suppose that the path from s_i back to itself is labeled with a^{j-i}
  - Since $a^i b^i$ is in ***L***, there must be a path labeled b^i from s_i to an accepting state ***f***
  - But, there is also a path from s_0 through s_i to ***f*** labelled $a^j b^i$
  - Thus, ***D*** also accepts $a^j b^i$, which is not in ***L***, contradicting the assumption that ***L*** is the language accepted by ***D***



RE/FA is NOT Powerful Enough(cont.)

- *L* = $\{a^n b^n \mid n \geq 1\}$ is not a Regular Language
  - Proof → Pumping Lemma (泵引理)
  - FA does not have any memory (FA cannot count)
    - The above *L* requires to keep count of a's before seeing b's

- Matching parenthesis is not a RL
- Any language with nested structure is not a RL
  - if … if … else … else
- Regular Languages
  - Weakest formal languages that are widely used

What Language Do We Need?

- C-language syntax: **Context Free Language** (CFL)[上下文无关语言]
  - A broader category of languages that includes languages with nested structures
- Before discussing CFL, we need to learn a more general way of specifying languages than RE, called **Grammars**[文法]
  - Can specify both RL and CFL
  - and more ...
- Everything that can be described by a regular expression can also be described by a grammar
  - Grammars are most useful for describing nested structures

Concepts

- **Language**[语言]
  - Set of strings over alphabet
    - *String*: finite sequence of symbols
    - *Alphabet*: finite set of symbols

- **Grammar**[文法]
  - To systematically describe the syntax of programming language constructs like expressions and statements

- **Syntax**[语法]
  - Describes the proper form of the programs
  - Specified by grammar

Grammar[文法]

- Formal definition[形式化定义]: 4 components **{T, N, s, δ}**
- ***T***: set of terminal symbols[终结符]
  - Basic symbols from which strings are formed
  - Essentially tokens - leaves in the parse tree
- ***N***: set of non-terminal symbols[非终结符]
  - Each represents a set of strings of terminals – internal nodes
  - E.g.: declaration, statement, loop, ...
- **s**: start symbol[开始符号]
  - One of the non-terminals
- ***σ***: set of productions[产生式]
  - Specify the manner in which the terminals and non-terminals can be combined to to form strings
  - “LHS → RHS”: left-hand-side produces right-hand-side

Grammar (cont.)

- Usually, we can only write the σ

- Merge rules sharing the same LHS
  - $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \ldots, \alpha \rightarrow \beta_n$
  - $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

$G = (\{id, +, *, (,)\}, \{E\}, E, P)$
$P = \{ E \rightarrow E + E,$
$E \rightarrow E * E,$
$E \rightarrow (E),$
$E \rightarrow id \}$

G: $E \rightarrow E + E,$
$E \rightarrow E * E,$
$E \rightarrow (E),$
$E \rightarrow id \}$

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

Production Rule and Derivation[推导]

- **Production rule**: LHS → RHS
  - Aliases: LHS ≡ head, RHS ≡ body
  - Meaning: LHS can be constructed (or replaced) with RHS
- **Derivation**: a series of applications of production rules
  - Corresponds to the construction of a parse tree
- $\beta \Rightarrow \alpha$
  - Meaning: string α is derived from β
  - $\beta \Rightarrow \alpha$: derives one step
  - $\beta \Rightarrow^* \alpha$: derives in zero or more steps
  - $\beta \Rightarrow^+ \alpha$: derives in one or more steps
- Example: $A \Rightarrow 0A \Rightarrow 00B \Rightarrow 000$
  - $A \Rightarrow^* 000$
  - $A \Rightarrow^+ 000$

Derivation

- If $S \Rightarrow^* \alpha$, where S is the start symbol of grammar G
- α: **sentential form** of G [句型]
  - A sentential form may contain both terminals and non-terminals (and can be empty)
- α: **sentence** of G [句子]
  - A sentential form with no non-terminals
- **Language**[语言] generated by a grammar
  - $L(G) = \{w: S \Rightarrow {}^*w,\ w \in V_T{}^* \}$
  - A string of terminal *w* is in L(G), **iff** *w* is a sentence of G (or $S \Rightarrow^* w$)

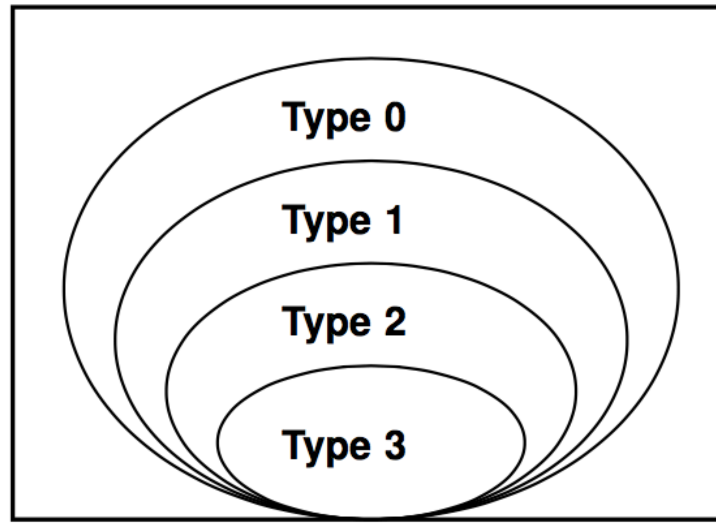
Example

- Grammar $G = \{T, N, s, \delta\}$
  - $T = \{0, 1\}$
  - $N = \{A, B\}$
  - $s = A$
  - $\delta = \{ A \rightarrow 0A \mid 1A \mid 0B,\ B \rightarrow 0 \}$

- Derivation: from grammar to language
  - $A \Rightarrow 0A \Rightarrow 00B \Rightarrow 000$
  - $A \Rightarrow 1A \Rightarrow 10B \Rightarrow 100$
  - $A \Rightarrow 0A \Rightarrow 00A \Rightarrow 000B \Rightarrow 0000$
  - $A \Rightarrow 0A \Rightarrow 01A \Rightarrow \ldots$
  - … …

Language Classification: Chomsky

- **Language classification** based on form of grammar rules
- Four types of grammars:
  - Type 0 — unrestricted grammar
    - 0型文法 – 无限制文法
  - Type 1 — context sensitive grammar(CSG)
    - 1型文法 – 上下文有关文法
  - Type 2 — context free grammar (CFG
    - 2型文法 – 上下午无关文法
  - Type 3 — regular grammar
    - 3型文法 – 正则文法



- Regular Grammar ⊆ CFG ⊆ CSG ⊆ Unrestricted Grammar

Type 0: Unrestricted Grammar

- Form of rules $\alpha \rightarrow \beta$
  - where $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$
- Implied restrictions:
  - LHS: no ε allowed
- Example:
  - aB → aCD: LHS is shorter than RHS
  - aAB → aB : LHS is longer than RHS
  - A → ε: ε-productions are allowed
- Computational complexity: unbounded
  - Derivation strings may contract and expand repeatedly (Since LHS may be longer or shorter than RHS)
  - Unbounded number of productions before target string

Type 1: Context Sensitive Grammar

- Form of rules: $\alpha A\beta \rightarrow \alpha\gamma\beta$
  - where $A \in N$, $\alpha, \beta \in (N \cup T)^*$, $\gamma \in (N \cup T)^+$
- Replace A by γ only if found in the context of α and β
- Implied restrictions:
  - LHS: shorter or equal to RHS
  - RHS: no ε allowed
- Example:
  - aAB→aCB: replace A with C when in between a and B
  - A → C: replace A with C regardless of context
- Computational complexity: likely NP-Complete
  - Derivation strings may only expand
  - Bounded number of derivations before target string

Type 2: Context Free Grammar

- Form of rules: $A \rightarrow \gamma$
  - where $A \in N$, $\gamma \in (N \cup T)^+$
- Replace A by γ (no context can be specified)
- Implied restrictions:
  - LHS: a single non-terminal
  - RHS: no ε allowed
    - Sometimes relaxed to simplify grammar but rules can always be rewritten to exclude ε-productions
- Example:
  - A → aBc: replace A with aBc regardless of context
- Computational complexity:
  - Polynomial $O(n^{2.3728639})$, but most real world CFGs are $O(n)$

Type 3: Regular Grammar

- Form of rules $A \to \alpha$,or $A \to \alpha B$
  - where $A,B \in N$, $\alpha \in T$
- In terms of FA: Move from state A to state B on input α
- Implied restrictions:
  - LHS: a single non-terminal
  - RHS: a terminal or a terminal followed by a non-terminal
- Example: $A \to 1A \mid 0$
- Computational complexity:
  - Linear O(n)
  - Derivation string length increases by 1 at each step

中山大學 SUN YAT-SEN UNIVERSITY

In Practice

- Every regular language is a context-free language
- If PLs are context-sensitive, why use CFGs for parsing?
  - CSG parsers are provably inefficient
  - Most PL constructs are context-free:
    - if-stmt, declarations
  - The remaining context-sensitive constructs can be analyzed at the semantic analysis stage
    - e.g. def-before-use, matching formal/actual parameters
- In PLs
  - Regular language for lexical analysis
  - Context-free language for syntax analysis

Grammar and Derivation

- **Grammar** is used to derive string or construct parser

- A **derivation** is a sequence of applications of rules
  - Starting from the start symbol
  - S⇒... ⇒... ⇒... ⇒ (sentence)

- **Leftmost** and **Rightmost** derivations
  - At each derivation step, leftmost derivation always replaces the leftmost non-terminal symbol
  - Rightmost derivation always replaces the rightmost one

Example

- Two derivations of string "id * id + id * id" using grammar:
  $E \rightarrow E*E \mid E+E \mid (E) \mid id$

- Leftmost derivation
  - $E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id * E + E \Rightarrow id * id + E \Rightarrow \ldots \Rightarrow id * id + id * id$

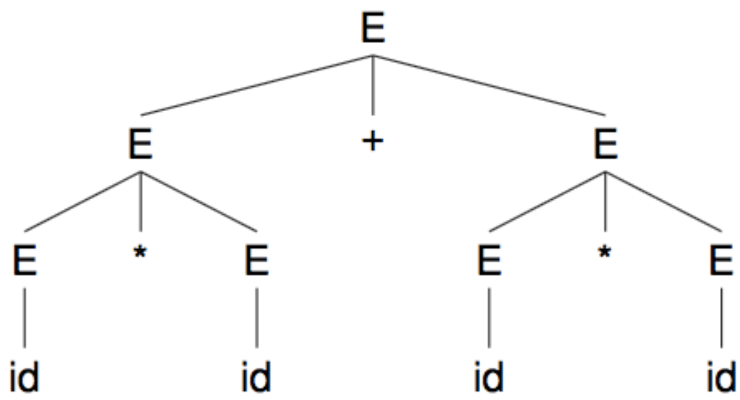
- Rightmost derivation $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * id \Rightarrow E + id * id \Rightarrow \ldots \Rightarrow id * id + id * id$

- Derivations can be summarized as a parse tree

SUN YAT-SEN UNIVERSITY

Parse Trees

- Both previous derivations result in the same parse tree:



- A **parse tree** is a graphical representation of a derivation
  - But filters out the order in which productions are applied to replace non-terminals
  - Each interior node represents the application of a production
    - Labeled with the non-terminal in the LHS of production
  - Leaves are labeled by terminals or non-terminals
    - Constitutes a sentential form (read from left to right)
    - Called the ***yield[产出]*** or ***frontier[边缘]*** of the tree

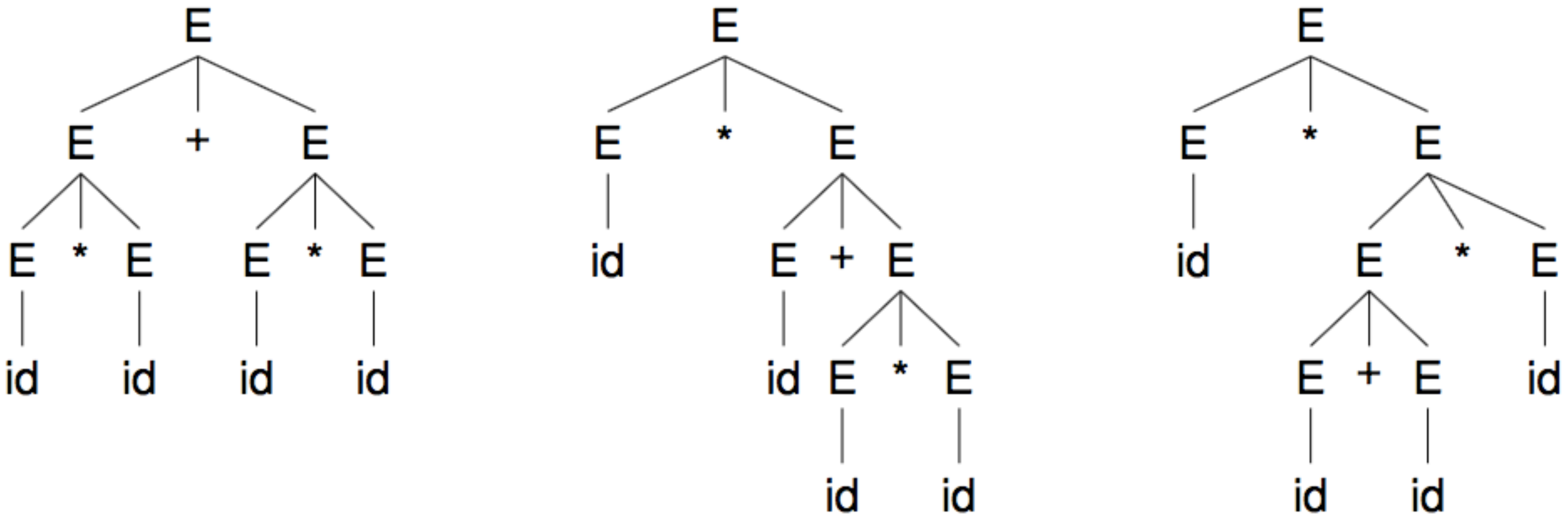
Parse Trees (cont.)

- Derivations and parse trees: many-to-one relationship
  - Leftmost derivation order: builds tree left to right
  - Rightmost derivation order: builds tree right to left
  - Different parser implementations choose different orders
  - One-to-one relationships between parse trees and either leftmost or rightmost derivations

- Program structure does not depend on order of rule application, instead it depends on what production rules are applied
  - Grammar must define unambiguously set of rules applied

Different Parse Trees

- Grammar E→E*E | E+E | (E) | id is ambiguous
  - String id * id + id * id can result in 3 parse trees (and more)



- Grammar can apply different rules to derive same string
  - Meaning of parse tree 1: (id * id) + (id * id)
  - Meaning of parse tree 2: id * (id + (id * id))
  - Meaning of parse tree 3: id * ((id + id) * id)

Ambiguity[二义性]

- grammar G is ambiguous if
  - It produces more than one parse tree some sentence
  - i.e., there exist a string *str* ∈ L(G) such that
  - more than one parse tree derives *str*

    ≡ more than one leftmost derivation derives *str*

    ≡ more than one rightmost derivation derives *str*
- Unambiguous grammars are preferred for most parsers
  - If not, we cannot uniquely determine which parse tree to select for a sentence
  - In minor cases, it is convenient to use carefully chosen ambiguous grammars, together with disambiguating rules that "throw away" undesirable parse trees, leaving only one tree for each sentence