编译原理
Complier Principles

Lecture 3

Syntax Analysis: Intro & Parser & CFG

赵帅

计算机学院

中山大学

Revisit

Lexical Specification → Regular expressions → NFA → DFA → Table-driven Implementation

Transition Flow

1. **Converting REs to NFA**
   - Thompson Algorithm[汤普森算法] (Inductive method)
2. **Converting NFA to DFA**
   - Subset-Construction Algorithm[子集构造法]
3. **Minimizing DFA**
   - Partition Algorithm[分割法]

The Limits of Regular Languages

- L = $\{a^n ba^n \mid n \geq 0\}$ is not a Regular Language
  - FA does not have any memory (FA cannot count)
    - The above L requires to keep count of a's before seeing b's

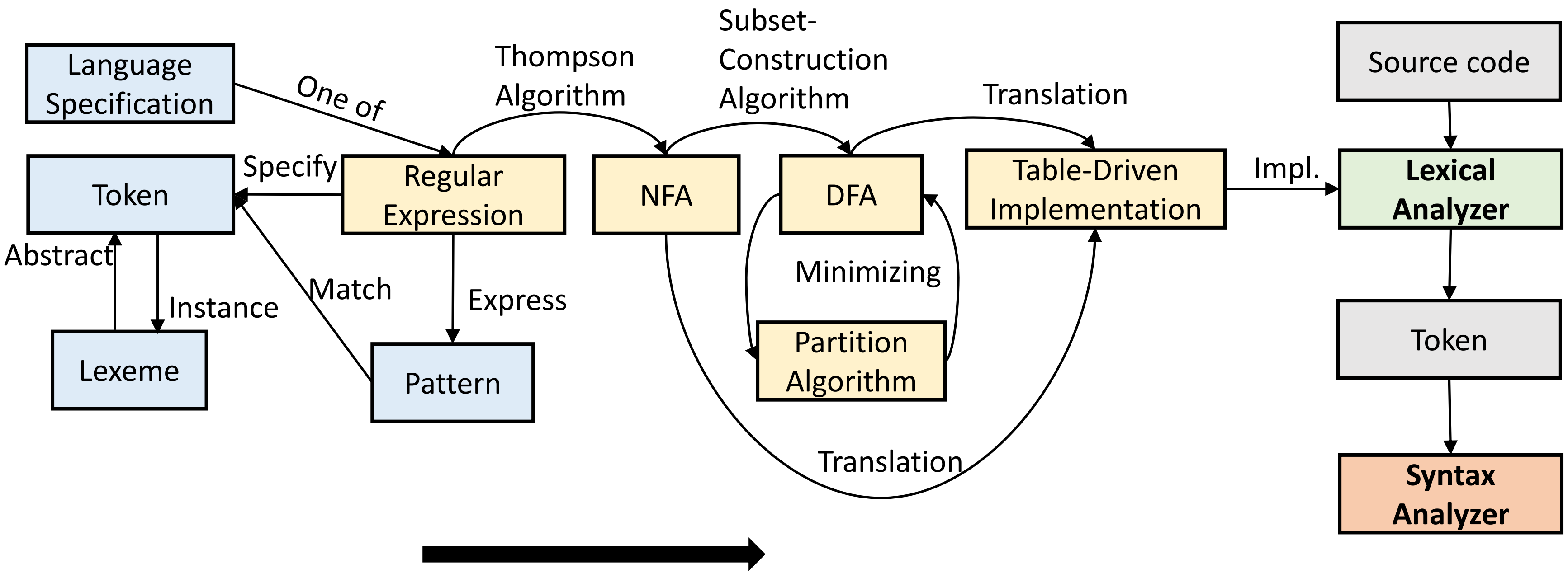
- Matching parenthesis is not a RL[括号匹配不是正则语言]
- Any language with nested structure is not a RL

  if … if … else … else
- Regular Languages
  - Weakest formal languages that are widely used [最弱的形式语言]
- We need a more powerful formalism

Beyond Regular Language

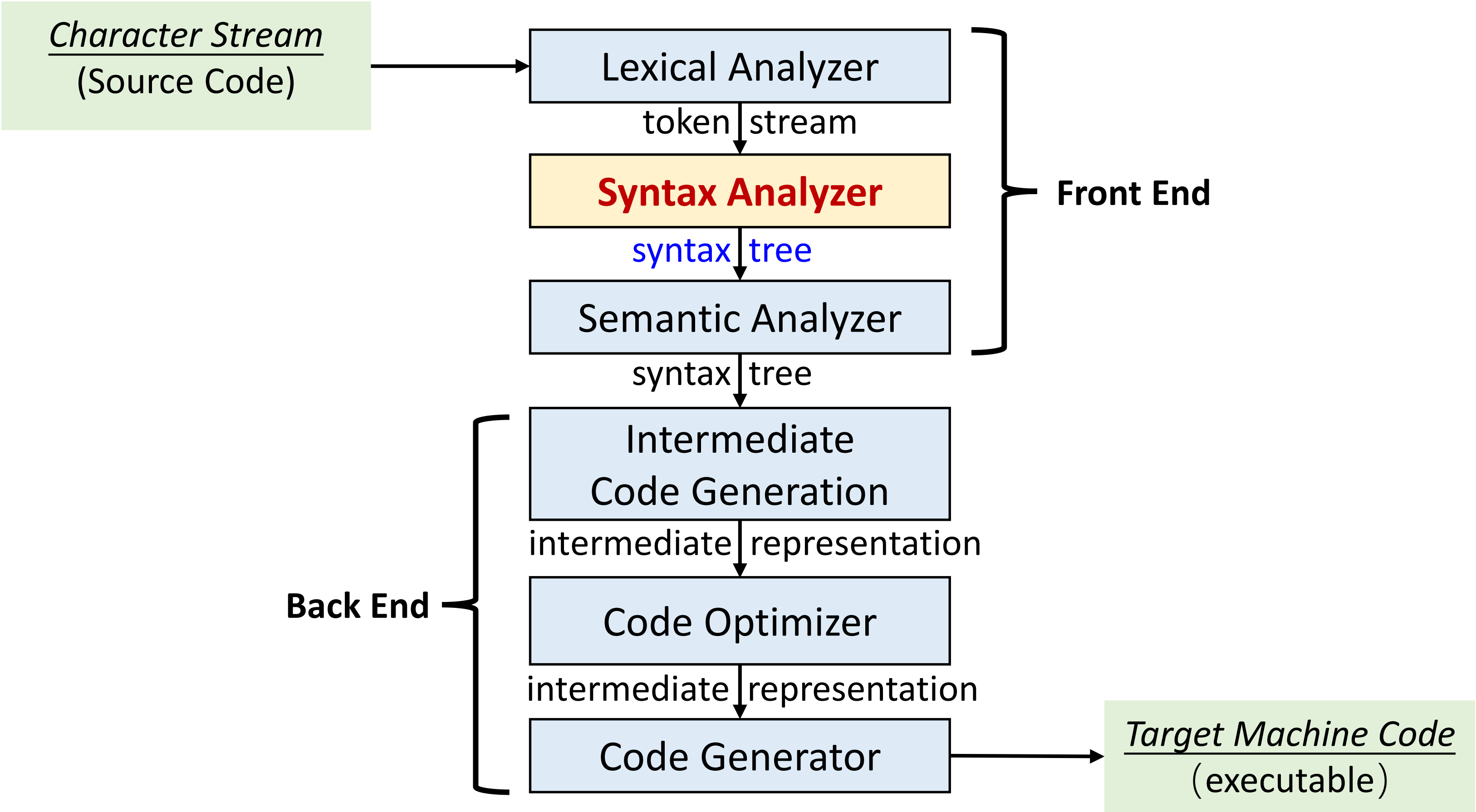
- Regular languages are expressive enough for tokens
  - Can express identifiers, strings, comments, etc.
- However, it is the weakest (least expressive) language
  - Many languages are not regular
  - C programming language is not
    - The language matching braces "{{{...}}}" is also not
  - FA does not have any memory (FA cannot count)
    - L = $\{a^n b^n \mid n \geq 1\}$
    - Crucial for analyzing languages with nested structures[嵌套结构] (e.g. nested for loop in C language)
- We need a more powerful language for parsing
  - Later, we will discuss context-free languages (CFGs)

Revisit



From Specification to Implementation

Compilation Phases[编译阶段]



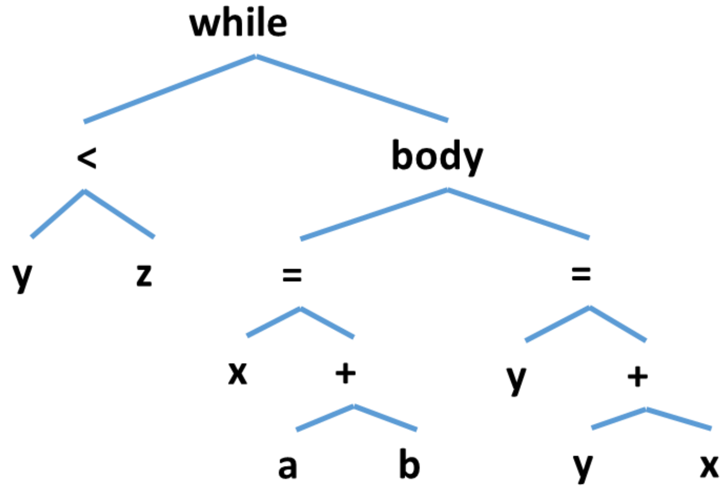
Compilation Phases[编译阶段]

```
while (y<z){
    int x = a + b;
    y += x;
}
```

⇨

(keyword, **while**)
(id, **y**)
(sym, **<**)
(id, **z**)
(id, **x**)
(id, **a**)
(sym, **+**)
(id, **b**)
(sym, **;**)
(id, **y**)
(sym, **+=**)
(id, **x**)
(sym, **;**)

⇨ **?**

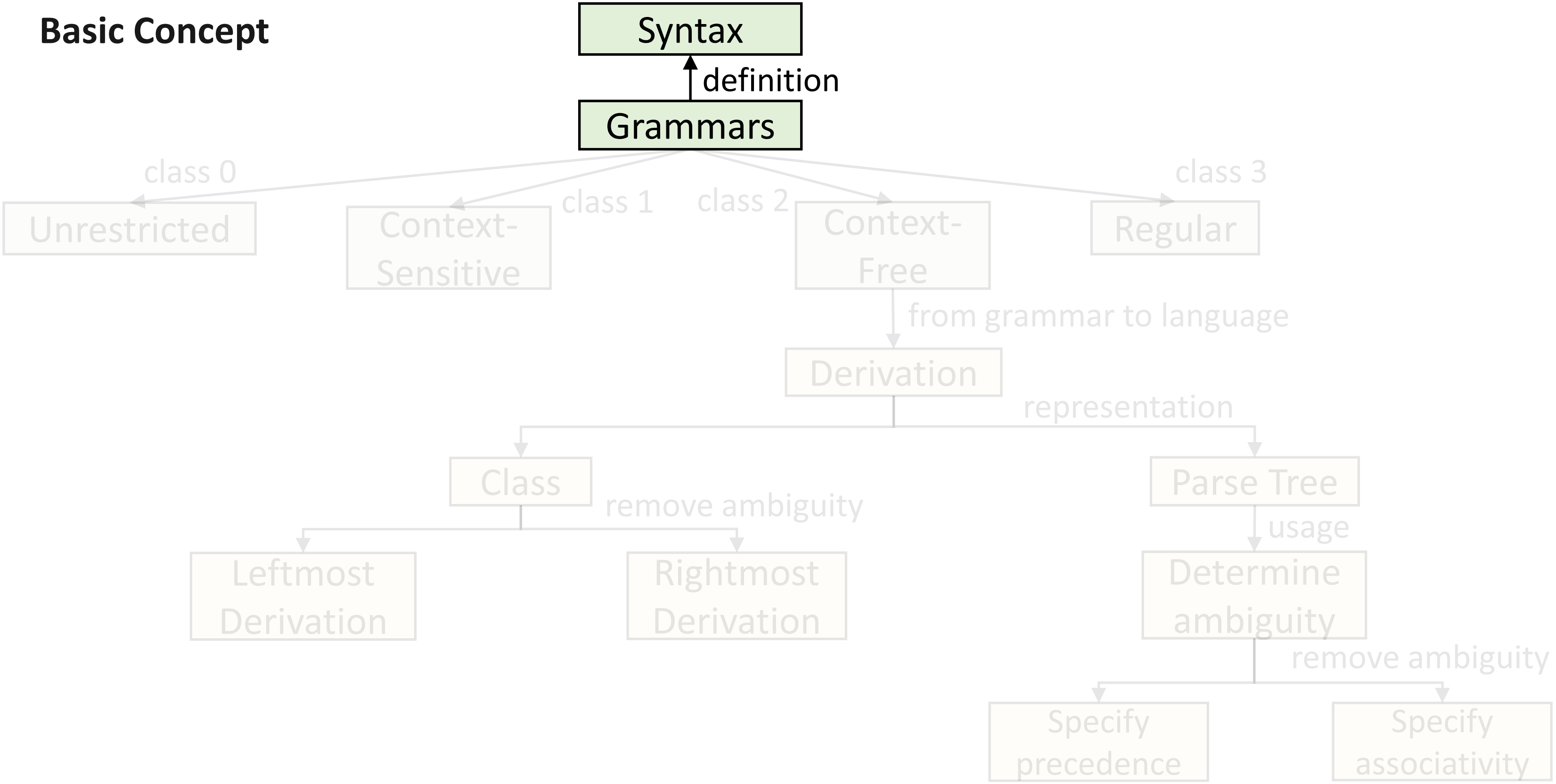


Abstract Syntax Tree(AST)

Mind Map[思维导图]

SUN YAT-SEN UNIVERSITY

Basic Concept



Syntax Analysis [语法分析]

SUN YAT-SEN UNIVERSITY

- Second phase of compilation, also called **parser**.

- The parser obtains ***a string of tokens***[词法单元组成的串] from the **lexical analyzer**, and verifies that the string of token names [*Token: <token name, attribute value>*] can be generated by **the grammar** [文法] for the source language.

- 语法分析验证tokens是否满足源语言的语法规则

Syntax Analysis [语法分析]

- The parser will construct ***a parse tree*** [语法分析树] and passes it to the rest of the compiler for further processing.
  - Parse tree: Graphically represent the syntax structure of the token stream.

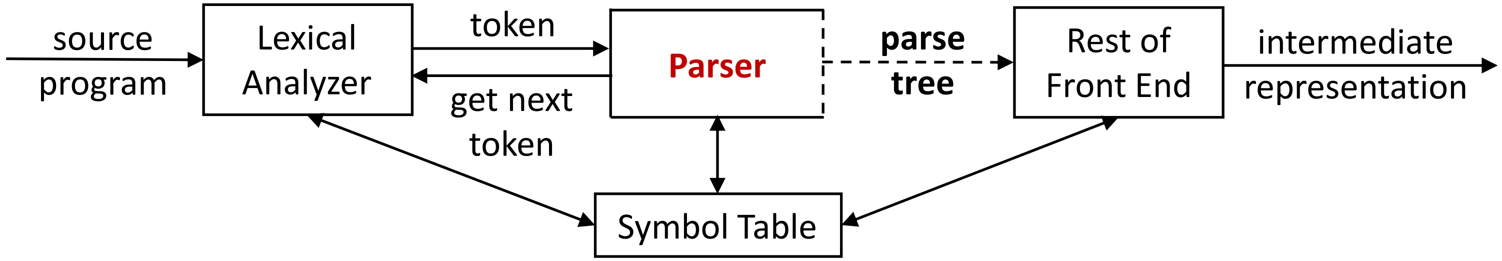
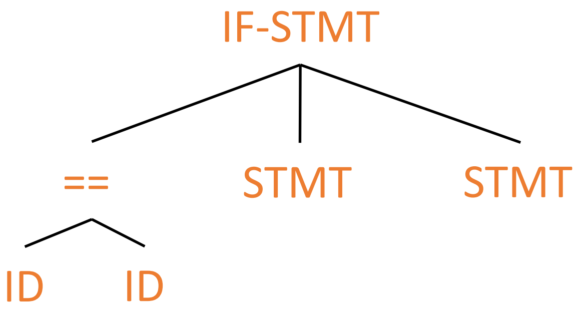


Fig. Position of parser in compiler model

Parsing Example[语法分析举例]

- **Example1:** Input: if (x == y) stmt1 else stmt2 [源程序输入]
  - Parser input (Lexical output) [语法分析输入]

    KEY(IF) ‘(’ ID(x) OP(‘==’) ID(y) ‘)’ … KEY(ELSE) …
  - Parser output [语法分析输出]



- **Example2:** <id, x> <op, *> <op, %>
  - Is it a valid token stream in C language? **YES**
  - Is it a valid statement in C language (x *%)? **NO**
  - So not every string of tokens is valid, **Parser** must distinguish between valid and invalid token strings. [通过语法分析来辨别有效token流]

How to Specify Syntax? [如何定义语法]

- Natural Language[自然语言]: The language spoken by human beings and different countries have different languages.
- Formal language[形式语言]: The language defined by precise mathematics or machine-processable formulas which have strict syntax rules. [严格的语法规则]
- **Programming Language is also a formal language**[编程语言也是一种形式语言], which is used to define computer programs.
- A formal language can define itself **in many ways**:(1) Regular Expression; (2) Finite Automata (FA); (3) Grammars. [文法]

How to Specify Syntax? [如何定义语法]

SUN YAT-SEN UNIVERSITY

- A formal language can define itself **in many ways**:(1) Regular Expression; (2) some Automata(FA); (3) **Grammars.** [文法]
- RE/FA is not powerful enough to specify a syntax.
  - the language $L = \{a^n b^n \mid n \geq 1\}$ is an example of a language that can be described by a grammar but not by a regular expression. [可以用文法描述但是不能用正则表达式描述]
- **Grammar** is a mathematical model used to define language.
  - To systematically describe the syntax of programming language constructs like expressions and statements. [文法用来定义语言/语法]
  - Grammars are most useful for describing **nested structures**. [嵌套结构]
  - Everything that can be described by a regular expression can also be described by a grammar.

Grammar[文法]

Formal definition [形式化定义] : 4 components[四元] ***G=(V_T, V_N, S, δ)***

- ***V_T*** : A set of terminal symbols. [终结符]
  - Terminals are the basic symbols from which strings are formed.
  - Essentially tokens - leaves in the parse tree.
- ***V_N***: A set of non-terminal symbols. [非终结符] $V_T \cap V_N = \emptyset$
  - Each represents a set of strings of terminals– internal nodes (statement, loop, ...)
- ***S*** : start symbol. [开始符号]
  - a non-terminal symbol (the root)
- ***δ*** : A set of productions. [产生式]
  - start symbol S must appear at least once in the left-hand-side of a production. [开始符S必须在某个产生式的左部至少出现一次]

Grammar[文法]

- **δ** : A set of productions. [产生式]
  - specify the manner in which the terminals and non-terminals can be combined to form strings
  - each production consists of
    - The **head** or **left side** of the production[产生式头/左部]
    - The symbol →, sometimes ::= [巴科斯范式 (BNF)] is used in place of the arrow. [读作 " 定义为 "]
    - The **body** or **right side** of the production. [产生式体/右部]
    - "**LHS → RHS**": left-hand-side produces right-hand-side.

Grammar[文法]

- Example Grammar:

$\boldsymbol{\delta}$:

<句子> → <主语><谓语><宾语>

<主语> → 我 | 猫

<谓语> → 喜欢 | 追

<宾语> → 巧克力 | 老鼠

$\boldsymbol{V_N}$ =
{
<句子>,
<主语>,
<谓语>,
<宾语>
}

$\boldsymbol{V_T}$ =
{
我,
猫,
喜欢,
追,
巧克力,
老鼠
}

$\boldsymbol{S}$ = 句子

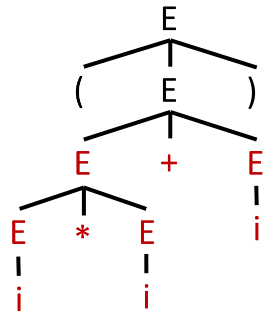
- Example Sentences (provided by 陈炜琰)
  - 猫喜欢巧克力
  - 老鼠追猫
  - 我喜欢追老鼠…

Context Free Grammar[上下文无关文法]

- To check whether a program is well-formed requires a **specification** of what is a well-formed program [语法定义]
  - The **specification** be **precise** [精确]
  - The **specification** be **complete** [完备]
    - Must cover all the syntactic details of the language
  - The **specification** must be **convenient** [便捷] to use by both language designer and the implementer
- **Context-free grammar** meets the above requirements:
- Context-free grammar has sufficient ability to describe the grammatical structure of most programming languages today.

Context Free Grammar[上下文无关文法]

- Formal definition [形式化定义] : 4 components ***$G=(V_T, V_N, S, \delta)$***
  - ◆ ***V_T*** : A set of terminal symbols. [终结符]
  - ◆ ***V_N***: A set of non-terminal symbols. [非终结符]
  - ◆ ***S*** : start symbol. [开始符号]
  - ◆ ***δ*** : is a finite set of production[有限的产生式集合] rules of the form such as $A \rightarrow \alpha$, where **A is from V_N** and **α from $(V_N \cup V_T)^*$**
- G = < {i, +,*, (,)} , {E} , E , δ > [只含*,+的算术表达式上下文无关文法]
  - δ is composed of the following production：
    - E→i; E→ E+E; E→ E*E; E→(E)



Grammar [文法]

- Usually, we can only write the δ [简写，只需写产生式]

G=< {i, +,*, (,)} ， {E} ， E ， δ >
δ is composed of the following production[只含*,+的算术表达式文法]
δ = {E→i;
E→ E+E;
E→ E*E;
E→(E)}

G = {E→i;
E→ E+E;
E→ E*E;
E→(E)}

E→i | E+E | E*E | (E)

G[E]/G(E)/G:
E→i;
E→ E+E;
E→ E*E;
E→(E)

- Sometimes, Write "G[E]/G(E)" before the production, where G is the grammar name and E is the start symbol. [文法名和开始符号]

Grammar Convention[文法的一些约定]

- Merge rules sharing the same left-hand side[规则合并]
  - $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \dots, \alpha \rightarrow \beta_n$
  - $\alpha \rightarrow \beta_1|\beta_2| \dots |\beta_n$, call β_i the alternatives[可选体/候选式] for α.

- These symbols are ***terminals***: [使用这些符号表示终结符]
  - Lowercase letters early in the alphabet, such as *a*, *b*, *c*. [靠前的小写字母]
  - Operator symbols such as +, *, ... [运算符号]
  - Punctuation symbols such as (, , ... [标点符号]
  - Digits such as 0,1,... ,9. [数字]
  - Boldface strings such as **id** or **if**, each of which represents a single terminal symbol. [黑体字符串]

Grammar Convention[文法的一些约定]

- These symbols are ***non-terminals***: [使用这些符号表示非终结符]
  - Uppercase letters early in the alphabet, such as *A*, *B*, *C* [靠前的大写字母]
  - Letter S is usually the start symbol [使用大写字母S来表示开始符号]
  - Lowercase and italic names such as *expr* or *stmt*. [小写,斜体的名字]
  - When discussing programming constructs, uppercase letters may represent non-terminals for the constructs.
    - *E*: expression[表达式], *T*: term[项], *F*: factor[因子]

Grammar Convention[文法的一些约定]

- Uppercase letters late in the alphabet, such as ***X***, ***Y***, ***Z***, represent *grammar symbols*; that is, either non-terminals or terminals. [字母表靠后的大写字母表示文法符号,即终结符或非终结符]
- Lowercase letters late in the alphabet, chiefly ***u***, ***v***,..., ***z***, represent (possibly empty) *strings of terminals*.[靠后的小写字母表示可能为空的终结符号串]
- Lowercase Greek letters, such as **α**, **β**, **γ**, represent (possibly empty) strings of grammar symbols. [希腊字母表示可能为空的文法符号串]
- Unless stated otherwise, the head of the first production is the start symbol.[第一个产生式的头就是开始符号]

Grammar Convention[文法的一些约定]

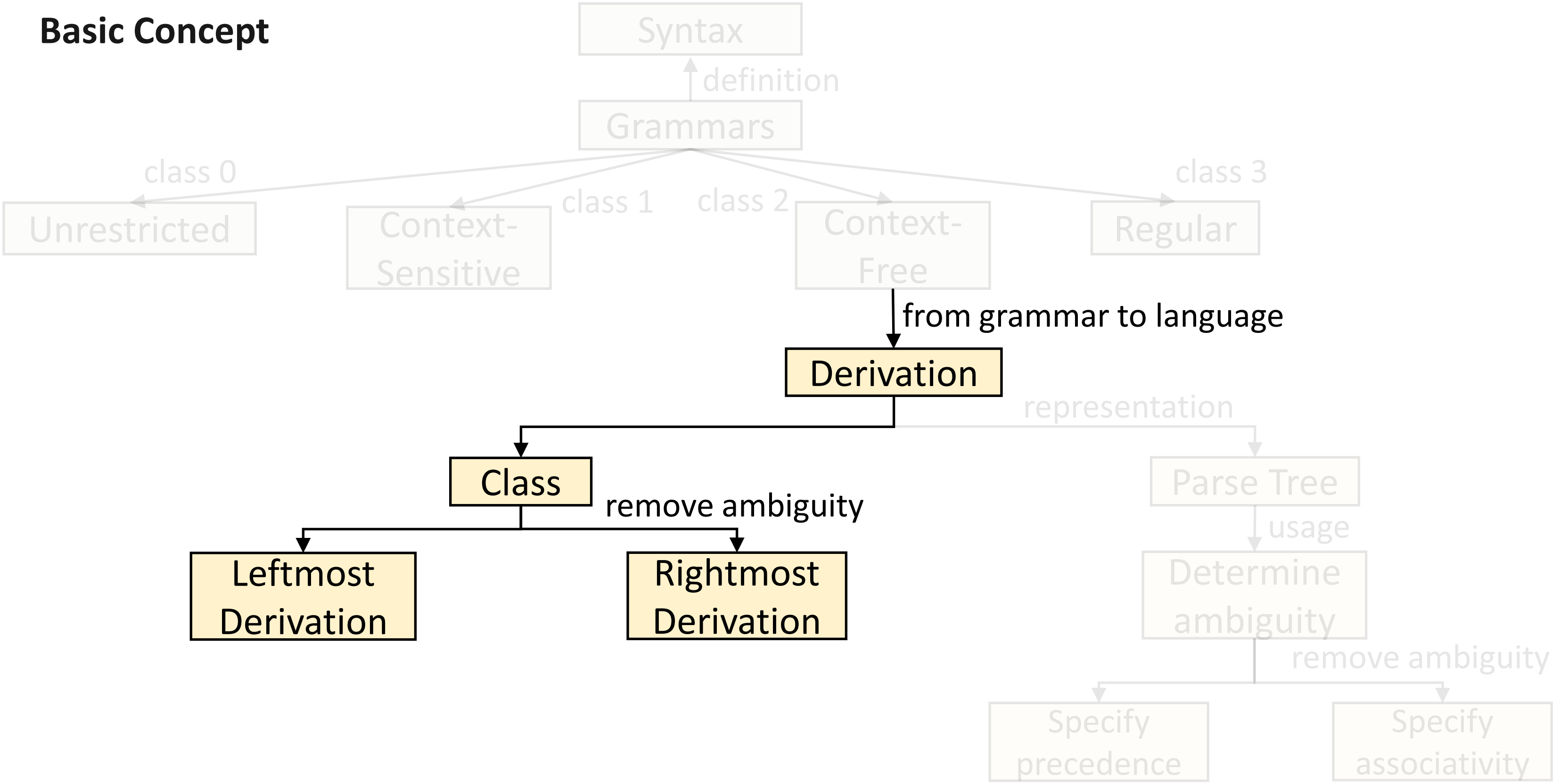
- Example:
- G[E]/G(E)/G:

$$E \to E + T \mid E - T \mid T$$
$$T \to T * F \mid T \,/\, F \mid F$$
$$F \to (E) \mid \mathbf{id}$$

- Start symbol: E
- Non-terminals: E, T and F
- Terminals: everything else

Mind Map[思维导图]

Basic Concept



Derivation[推导]

- **Production rule**[产生式规则]: $A \rightarrow \alpha$, which means that A can be constructed (or replaced) with α.
- **Derivation** [推导] : a series of applications of production rules.
  - consider a non-terminal A in the middle of a sequence of grammar symbols $\alpha A\beta$, and $A \rightarrow \gamma$ is a production. Then, we write $\alpha A\beta \Rightarrow \alpha\gamma\beta$, the symbol $\Rightarrow$ means **"derives in one step"**. [通过一步推导出]
  - when a sequence of derivation steps $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ rewrites α_1 to α_n , we say α_1 *derives* α_n[推导出], we can use the symbol $\overset{*}{\Rightarrow}$ to represents **"derives in zero or more steps"**. [经过零步或多步推导出]
  - the symbol $\overset{+}{\Rightarrow}$ means **"derives in one or more steps"**. [经过一步或多步推导出]
  - for any string α, $\alpha \overset{*}{\Rightarrow} \alpha$;   If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$ then $\alpha \overset{*}{\Rightarrow} \gamma$.

Sentential form, Sentence, Language[句型&句子&语言]

- If $S \stackrel{*}{\Rightarrow} \alpha$, where S is the start symbol of a grammar G, we say that α is a **sentential form** of G. [句型]
  - a sentential form may contain both terminals and non-terminals, and may be empty.
- A sentential form *with NO non-terminals* is called a **sentence**.[不包含非终结符的句型被称为句子]
- The **language** generated by a grammar is its set of sentences. [一个文法所产生的句子全体是一个语言（由文法生成）]
  - $L(G)=\{w \mid S \stackrel{*}{\Rightarrow} w, w \in V_T^*\}$.
  - a string of terminals w is in L(G), **if and only if** w is a sentence of G (i.e., $S \stackrel{*}{\Rightarrow} w$).

Sentential form, Sentence, Language[句型&句子&语言]

- **Example:**
- G[A]: A→ c | Ab
- Derivation: from grammar to language[文法到语言]
  - $A \Rightarrow \boxed{c}$
  - $A \Rightarrow Ab \Rightarrow \boxed{cb}$
  - $A \Rightarrow Ab \Rightarrow Abb \Rightarrow \boxed{cbb}$
  - $A \Rightarrow Ab \Rightarrow Abb \Rightarrow Abbb \Rightarrow \boxed{cbbb}$
  - $A \Rightarrow Ab \Rightarrow Abb \Rightarrow Abbb \Rightarrow Abbbb \Rightarrow \boxed{cbbbb}$
  - ……

sentence

Sentential form

Grammar and Derivation [文法与推导]

- **Grammar** is used to derive string or construct parser
- **derivation** is a sequence of applications of grammar rules
  - The process of derivation will start from start symbol.
  - In each step of derivation, the following choices need to be made:
    - <u>choice of the non-terminal to be replaced. [替换哪个非终结符]</u>
    - choice of a rule for the non-terminal. [使用文法中哪个规则来替换]

G[E]/G(E)/G:

$$
\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T \,/\, F \mid F \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

$$E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (E+T) \Rightarrow ?$$

Grammar and Derivation [文法与推导]

- **Leftmost derivations** [最左推导]:
  - the leftmost non-terminal in each sentential is always chosen.
- **Rightmost derivations** [最右推导]:
  - the rightmost non-terminal in each sentential is always chosen.

G[E]/G(E)/G:

$$
\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T / F \mid F \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

$$E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (E+T) \Rightarrow ?$$

- For a non-terminal, which rule shall we apply?
  - **Button-up** parsing
  - **Top-down** parsing

Leftmost/Rightmost derivations [最左/最右推导]

- G[E]: E→T|E+T ; T→F|T*F ; F→(E)|i

Leftmost Derivation:

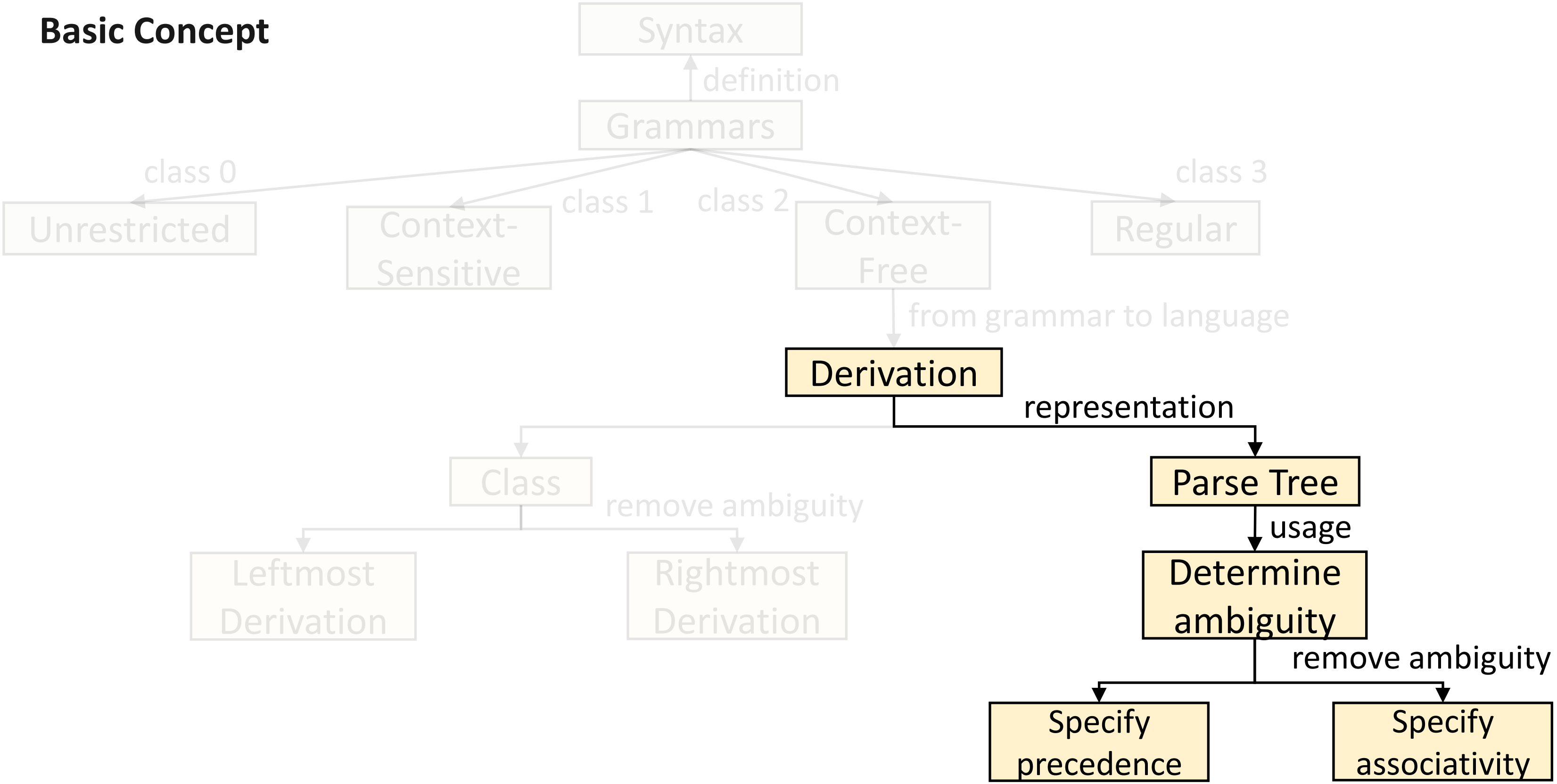
$E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (E+T)$
$\Rightarrow (T+T)$
$\Rightarrow (T*F+T)$
$\Rightarrow (F*F+T)$
$\Rightarrow (i*F+T)$
$\Rightarrow (i*i+T)$
$\Rightarrow (i*i+F)$
$\Rightarrow (i*i+i)$

Rightmost Derivation:

$E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (E+T)$
$\Rightarrow (E+F)$
$\Rightarrow (E+i)$
$\Rightarrow (T+i)$
$\Rightarrow (T*F+i)$
$\Rightarrow (T*i+i)$
$\Rightarrow (F*i+i)$
$\Rightarrow (i*i+i)$

Mind Map[思维导图]

Basic Concept

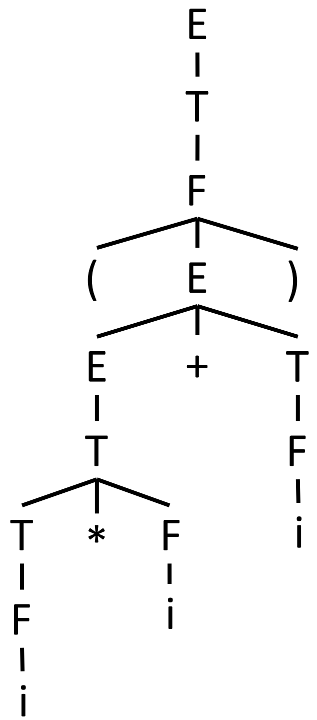


Parse Trees [分析树]

- Derivations can be summarized as a parse tree [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 to replace non-terminals. [过滤掉推导过程中对非终结符应用产生式的顺序]

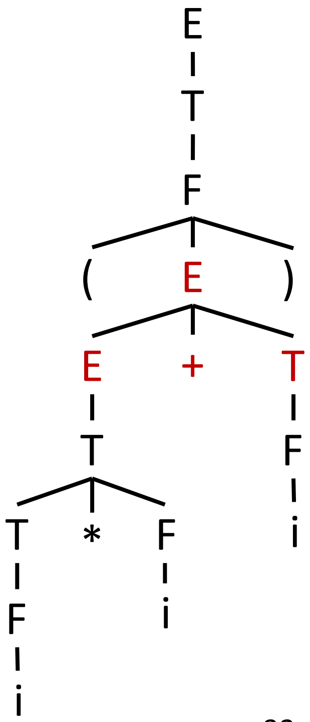
- Both previous derivations result in the same parse tree.



Parse Trees [分析树]

- Each **interior node**[内部结点] of a parse tree represents the application of a production. [产生式的应用]
  - **The interior node** is labeled with the non-terminal A **in the head of the production** [内部节点代表产生式左部]
  - **the children of the node** are labeled, from left to right, by the symbols **in the body of the production** by which this A was replaced during the derivation. [内部节点的子结点代表产生式右部]

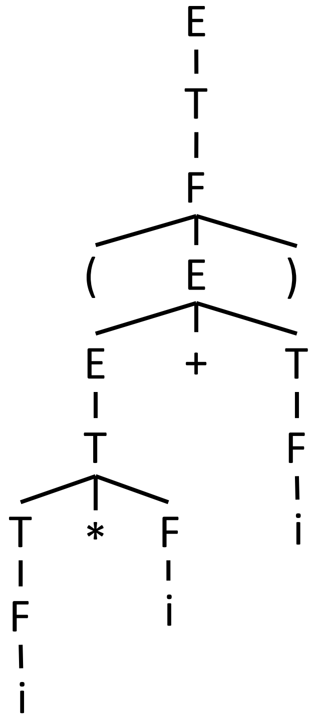
$E \Rightarrow E+T$



Parse Trees [分析树]

- **The leaves of a parse tree**[叶结点] are labeled by *non-terminals or terminals* and, **read from left to right**, constitute a sentential form [从左至右排列符号构成句型], called the *yield*[产出] or *frontier* [边缘] of the tree.

- Leftmost derivation order: builds tree left to right
- Rightmost derivation order: builds tree right to left
    - There is a one-to-one relationship between parse trees and either leftmost or rightmost derivations.[最左或最右推导与分析树具有一对一对应关系]



Parse Trees [分析树]

- <u>Leftmost / rightmost</u> derivations
  - can be summarized as a parse tree [语法分析树]
  - parse tree *filters out the order*

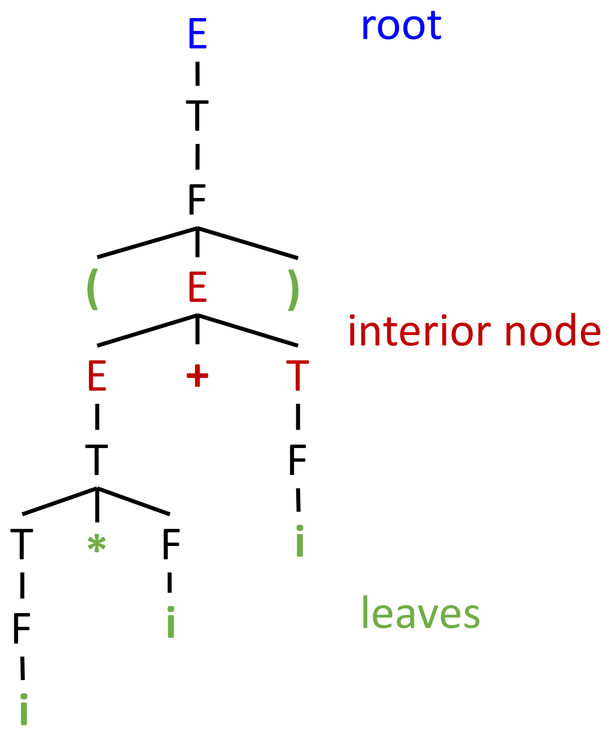
G[E]: E→T|E+T ; T→F|T*F ; F→(E)|i

Leftmost:

$E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (E+T)$
$\Rightarrow (T+T)$
$\Rightarrow (T*F+T)$
$\Rightarrow (F*F+T)$
$\Rightarrow (i*F+T)$
$\Rightarrow (i*i+T)$
$\Rightarrow (i*i+F)$
$\Rightarrow (i*i+i)$

Rightmost:

$E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (E+T)$
$\Rightarrow (E+F)$
$\Rightarrow (E+i)$
$\Rightarrow (T+i)$
$\Rightarrow (T*F+i)$
$\Rightarrow (T*i+i)$
$\Rightarrow (F*i+i)$
$\Rightarrow (i*i+i)$



Ambiguity[二义性]

- Whether a sentential form corresponds to only one grammar tree?
- Consider:
    - grammar: G(E): E → i | E+E | E*E | (E)
    - sentential form: (i*i+i)

Ambiguity[二义性]

- Unambiguous grammars are preferred for most parsers [文法最好没有二义性]

- Ambiguity of the grammar implies that at least **some strings in its language have different structures** (parse trees).

- This is unlikely to be useful for a programming language,
  - two structures for the same string (program) implies two different meanings of this program.

Ambiguity[二义性]

- **Ambiguity for grammar:** A grammar that produces more than one parse tree for a sentence.[如果一个文法存在某个句子对应两颗不同的语法树，则说这个文法是二义的]

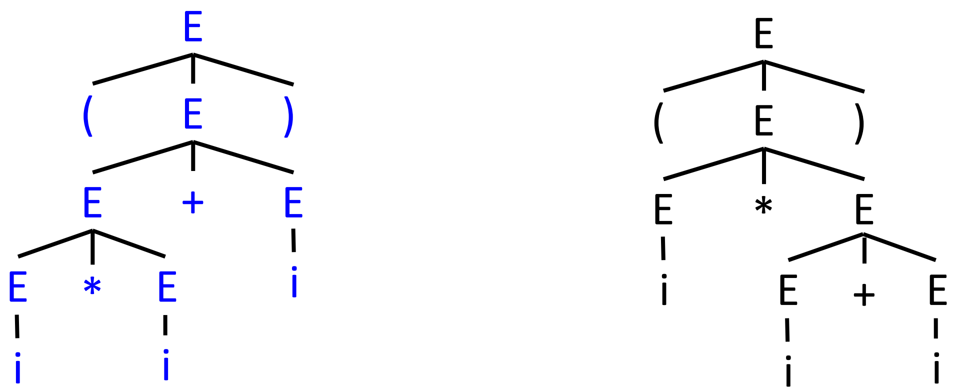
- **Ambiguity of language:** A language that has no unambiguous grammar.[一个语言是二义性的，如果对它不存在无二义性的文法]
  - There may be G and G', one is ambiguous and the other is unambiguous. But L(G) = L(G’).
  - The Ambiguity of a grammar G does not necessarily mean that its language L(G) is inherently ambiguous.

Ambiguity[二义性]

- Ambiguity is an undecidable problem[不可判定问题],
  - No algorithm exists that can accurately determine whether a grammar is ambiguous in a limited number of steps.[不存在一个算法，它能在有限步骤内，确切地判定一个文法是否是二义的]
- It is impossible to convert an ambiguous grammar to unambiguous automatically.
  - It is (often) possible to rewrite grammar to remove ambiguity
  - Or, use ambiguous grammar, along with disambiguating rules to "throw away" undesirable parse trees, **leaving only one tree for each sentence**. (as in YACC)
- There exist a set of sufficient conditions for unambiguous grammar. [可以找到一组判定无二义文法的充分条件]（但不是必要条件）

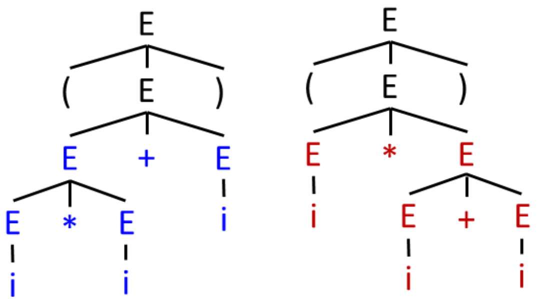
Remove Ambiguity[消除二义性]

- Consider the example again:
  - Grammar: G(E): E → i | E+E | E*E | (E)
  - sentential form: (i*i+i)
- An unambiguous grammar can be constructed, if：
  - specify the precedence of ‘+’ and ‘*’ [指定优先级], + for example
  - specify the associativity[指定结合性] , e.g., left associative



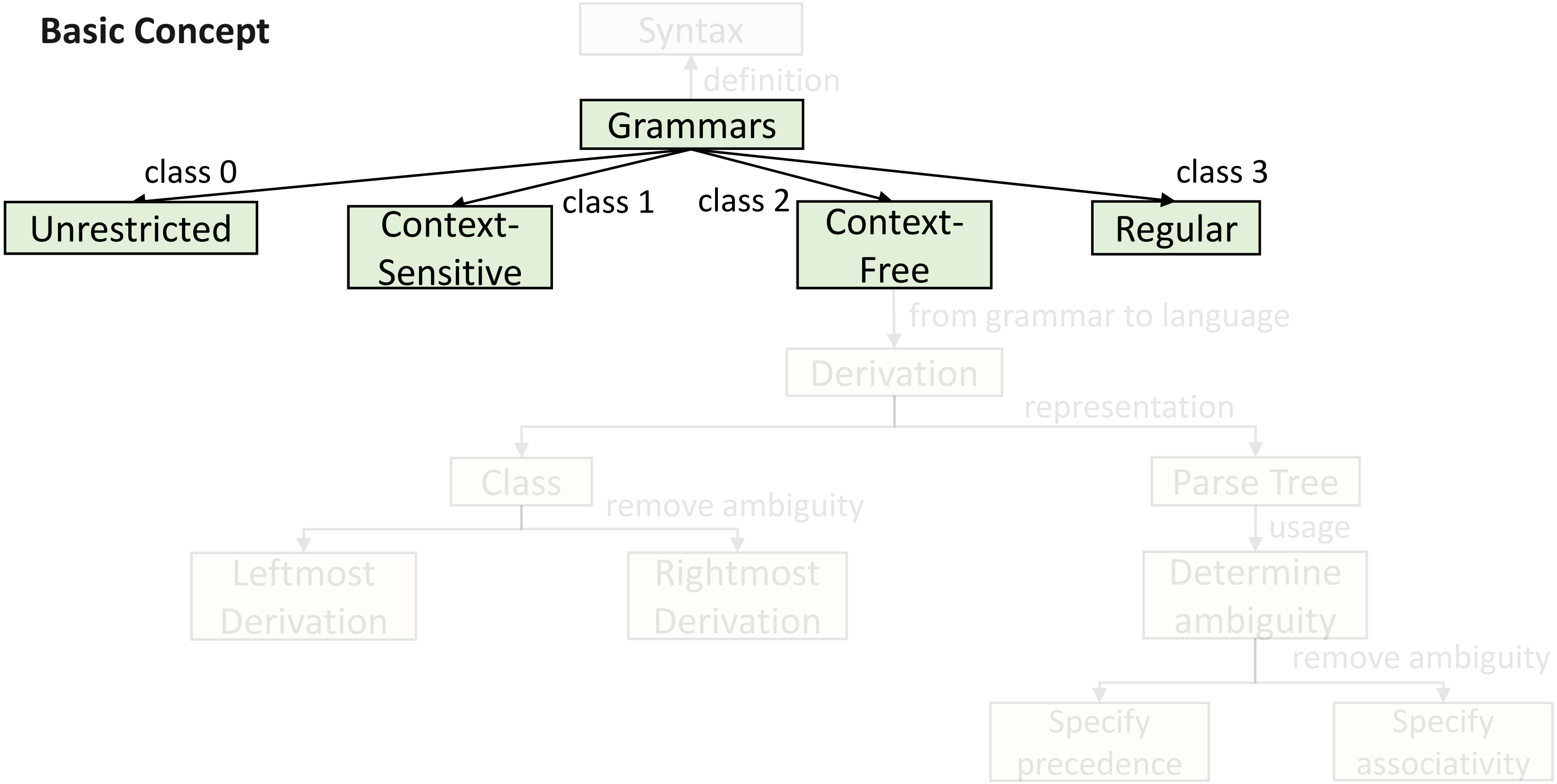
Revisit

- Syntax analysis
  - Input: takes a string of tokens
  - Process: verifies whether they can be generated by the grammar.
  - Output: Parse tree
- Grammar: $G=(V_T, V_N, S, \delta)$
  - Context Free Grammar: A→ α, A is from V_N and α from $(V_N \cup V_T)^*$
  - G(E): E → i | E+E | E*E | (E)
- Derivation: leftmost / rightmost
- Parse Tree
  - A graphical representation of derivation



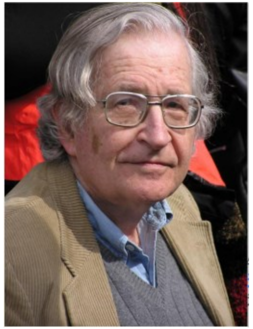
- Ambiguity:
  - specify the precedence[优先级] and associativity[结合性]

Mind Map[思维导图]



Chomsky Grammar System[乔姆斯基文法体系]

- Chomsky established the formal language system in 1956. He divided grammar into four types: 0, 1, 2, 3.
- Like context-free grammar, they are composed of 4 components ***$G=(V_T, V_N, S, \delta)$***, but with **different restrictions on productions**.



Noam Chomsky

- 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,上下文无关文法]
- Type 3 — regular grammar [3型文法,正则文法]

Type 0: Unrestricted Grammar

- A Grammar $G=(V_T, V_N, S, \delta)$ is Type 0 [无限制文法,短语结构文法], if each production $\alpha \rightarrow \beta$ of G:
  - $\alpha \in (V_N \cup V_T)^*$, $\alpha \neq \varepsilon$
  - $\beta \in (V_N \cup V_T)^*$.
- Recognized by Turing machine[与图灵机等价], example:
  - aA → aBCd: LHS is shorter than RHS;
  - aBcd → aE: LHS is longer than RHS;
  - A → ε: ε-productions are allowed;
- Derivations
  - Derivation strings may contract and expand repeatedly (since LHS may be longer or shorter than RHS)
  - Unbounded number of derivations before target string.

Type 1: Context Sensitive Grammar

- A Grammar $G=(V_T, V_N, S, \delta)$ is Type 1 [上下文有关文法], if each production $\boldsymbol{\alpha \rightarrow \beta}$ of G:
  - $\alpha \in (V_N \cup V_T)^*$, $\alpha \neq \varepsilon$
  - $\beta \in (V_N \cup V_T)^*$, and $\beta \neq \varepsilon$ unless α is the start symbol and does not appear on the right of any production
  - $|\alpha| \leq |\beta|$
- If not consider ε, it is accepted by linear bound automaton (LBA) [线性有界自动机]:
  - $\alpha A \beta \rightarrow \alpha \gamma \beta$: Only non-terminal A exists in the context of α and β, you can replace it with γ.
  - $A \rightarrow \gamma$: replace A with γ regardless of context.
- **Derivations**
  - Derivation strings may only expand
  - Bounded number of derivations before target string

Type 2: Context Free Grammar

SUN YAT-SEN UNIVERSITY 中山大學

- A Grammar $G=(V_T, V_N, S, \delta)$ is Type 2 [上下文无关文法], if each production **$A \rightarrow \alpha$** of G:
  - **$A \in V_N$, $A \neq \varepsilon$**
  - **$\alpha \in (V_N \cup V_T)^*$, $\alpha \neq \varepsilon$** but sometimes relaxed to simplify grammar, rules can always be rewritten to exclude ε-productions.
- Corresponding non-deterministic pushdown automaton [非确定下推自动机] (NDPDA)
- Example: $A \rightarrow aBc$: replace A with aBc regardless of context;

$L = \{ a^n b^n \mid n \geq 0\}$ is **NOT regular** but is a **context-free language**.

The following CFG: $G=<V_T, V_N, S, \delta>$ generates L:
$V_T = \{ a,b \}$, $V_N = \{ S \}$ and $\delta = \{ S \rightarrow aSb, S \rightarrow ab \}$

Type 3: Regular Grammar

- A Grammar $G=(V_T, V_N, S, \delta)$ is Type 3 [正则文法], if each production **A→aB or A→a** of G:
  - **$A, B \in V_N$**
  - **$a \in V_T \cup \{\varepsilon\}$**
    - LHS: a single non-terminal; RHS: a terminal or a terminal followed by a non-terminal.
    - $A \rightarrow \varepsilon$ permitted if A is the start symbol and does not appear on the right of any production.
- Corresponding non-deterministic Finite Automaton [有限自动机] (FA).
- Example
  - $A \rightarrow 1A \mid 0$, $A \rightarrow A1 \mid 0$
  - RE: 1*0
- Derivation
  - Derivation string length increases by 1 at each step

Type 3: Regular Grammar

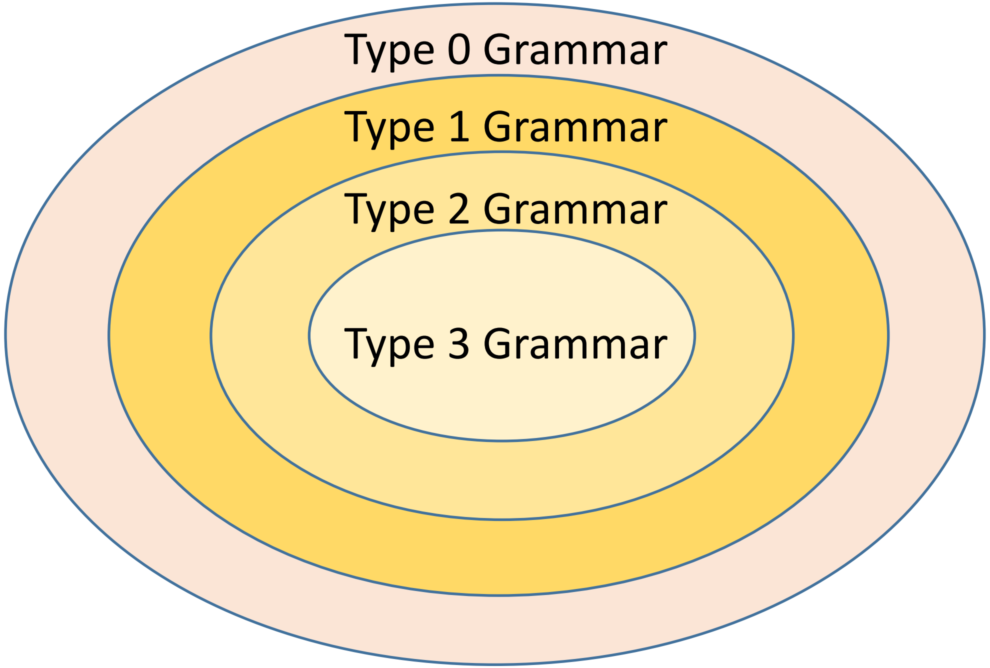
Useful in Practice

Useful in Theory

Class	Grammar	Restriction	Recognizer
3	Regular	$A \to aB$ or $A \to a$, where $A, B \in N \wedge a \in \Sigma \cup \{\varepsilon\}$. $A \to \varepsilon$ permitted if A is the start symbol and does not appear on the right of any production.	Finite-State Automaton (**FSA**)
2	Context-Free	$A \to \alpha$, where $A \in N \wedge \alpha \in (\Sigma \cup N)^*$.	Push-Down Automaton (**PDA**)
1	Context-Sensitive	$\alpha \to \beta$, where $\alpha, \beta \in (\Sigma \cup N)^* \wedge \alpha \neq \varepsilon \wedge \lvert\alpha\rvert \leq \lvert\beta\rvert$. β can't be ε, unless α is the start symbol and does not appear on the right of any production.	Linear-Bounded Automaton (**LBA**)
0	Unrestricted	$\alpha \to \beta$, where $\alpha, \beta \in (\Sigma \cup N)^* \wedge \alpha \neq \varepsilon$.	Turing Machine (**TM**)

Source: Prof Wenjun Li @ SYSU

Comparison



In Practice[实际中]

- Most programming languages are not context-free language, or even context-sensitive language.
- However, for today's programming languages, CFG is still widely used to describe the language structure in compilers.
  - ◆ Perfectly suited for describing recursive syntax of expressions and statements
  - ◆ CSG parsers are provably inefficient [CSG复杂且效率低下]
  - ◆ The construction of CFG is currently very mature and efficient. [CFG成熟且效率高]
  - ◆ The remaining context-sensitive constructs can be analyzed in semantic analysis stage
- In programming languages:
  - ◆ Regular language for lexical analysis
  - ◆ Context-free language for syntax analysis

Others[其他]

- So, what exactly is parsing, or syntax analysis?
  - To process an input string based on a given grammar, and compose the derivation if the string is in the language.
  - Two subtasks:

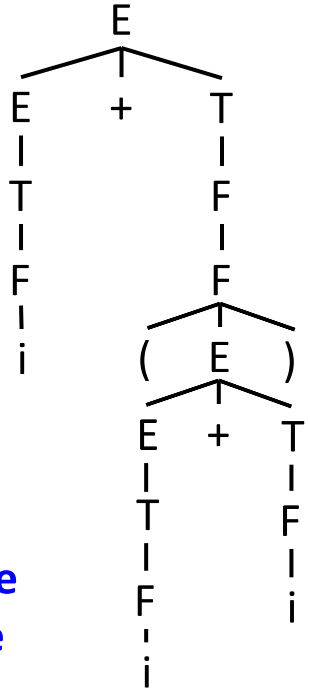
    (1) determine if a string can be derived from a grammar or not;

    (2) build a representation of derivation and pass to next phase.

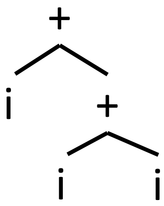
- What is the best representation of derivation? [推导表示]
  - a parse tree or an abstract syntax tree.[语法解析树或抽象语法树]

Parse Tree VS Abstract Syntax Tree

- An abstract syntax tree (AST) is abbreviated representation[缩写表示] of a parse tree
- drops details without compromising meaning [在不影响意义的情况下删除推导细节].

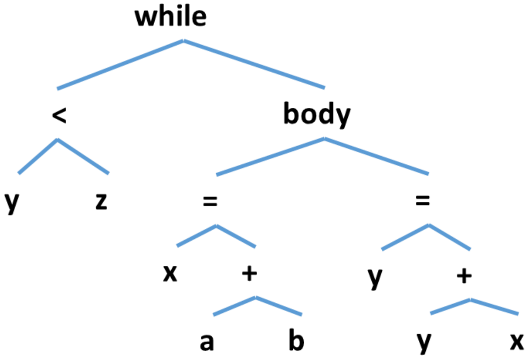


Parse tree



AST

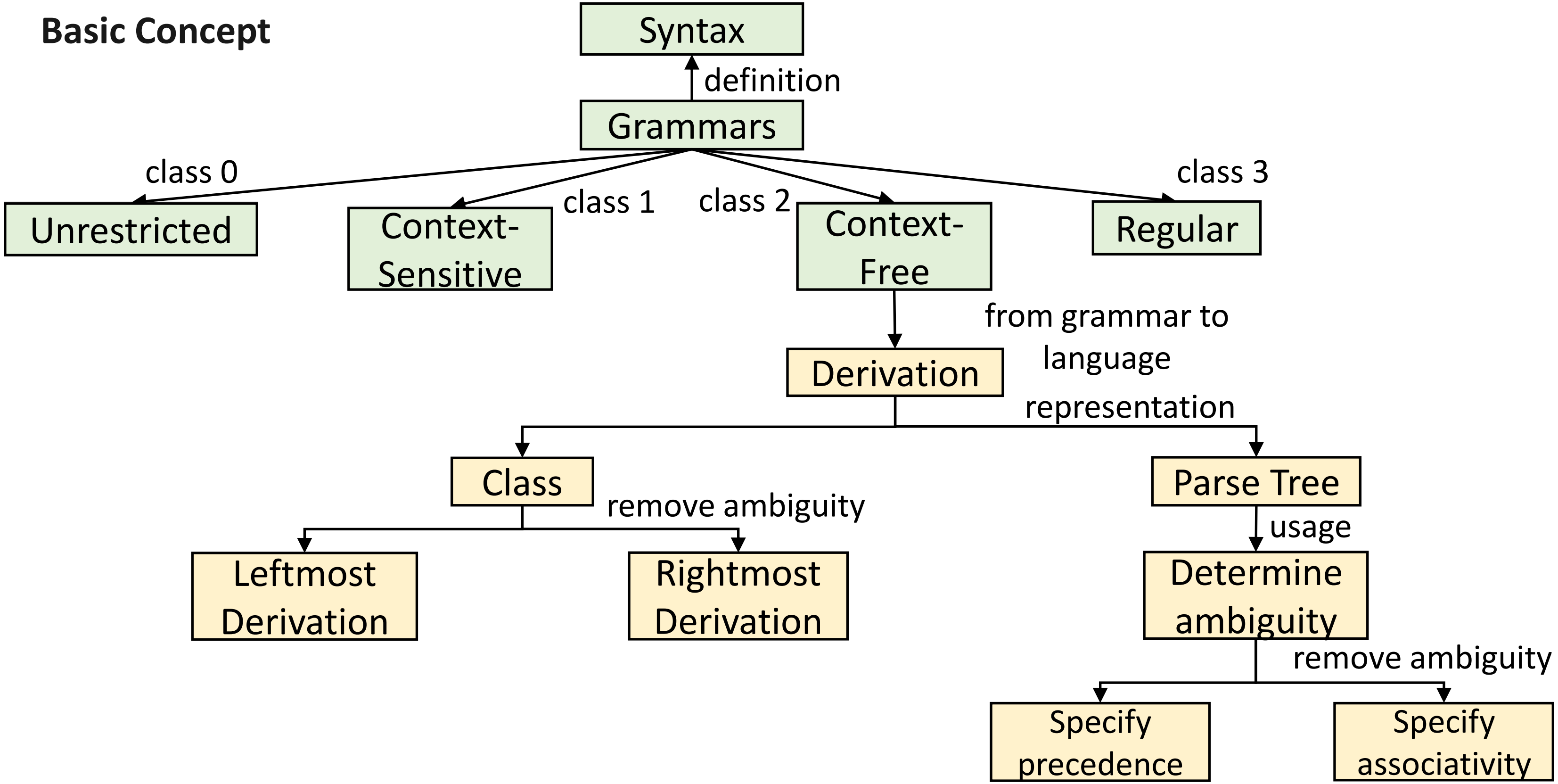
```
while (y<z){
    int x = a + b;
    y += x;
}
```



Abstract Syntax Tree(AST)

Mind Map[思维导图]

Basic Concept



Summary

- Grammar (and Chomsky Grammar System)
- Context Free Grammar, a parser uses CFG to
  - judge if an input *str* $\in$ L(G)
  - build a parse tree or AST
  - pass it to the rest of compiler or give an error message.
- Parse tree: shows how a string can be derived from a grammar.
  - A grammar is ambiguous if a string has more than one parse tree.
- Abstract syntax trees (AST): an abstract representation of a program's syntax.

Further Reading

- Dragon Book
  - Comprehensive Reading:
    - Section 2.4, 4.1.1 for the introduction to parsing.
    - Section 4.2 and 4.3 for context free grammar and grammar transformations.