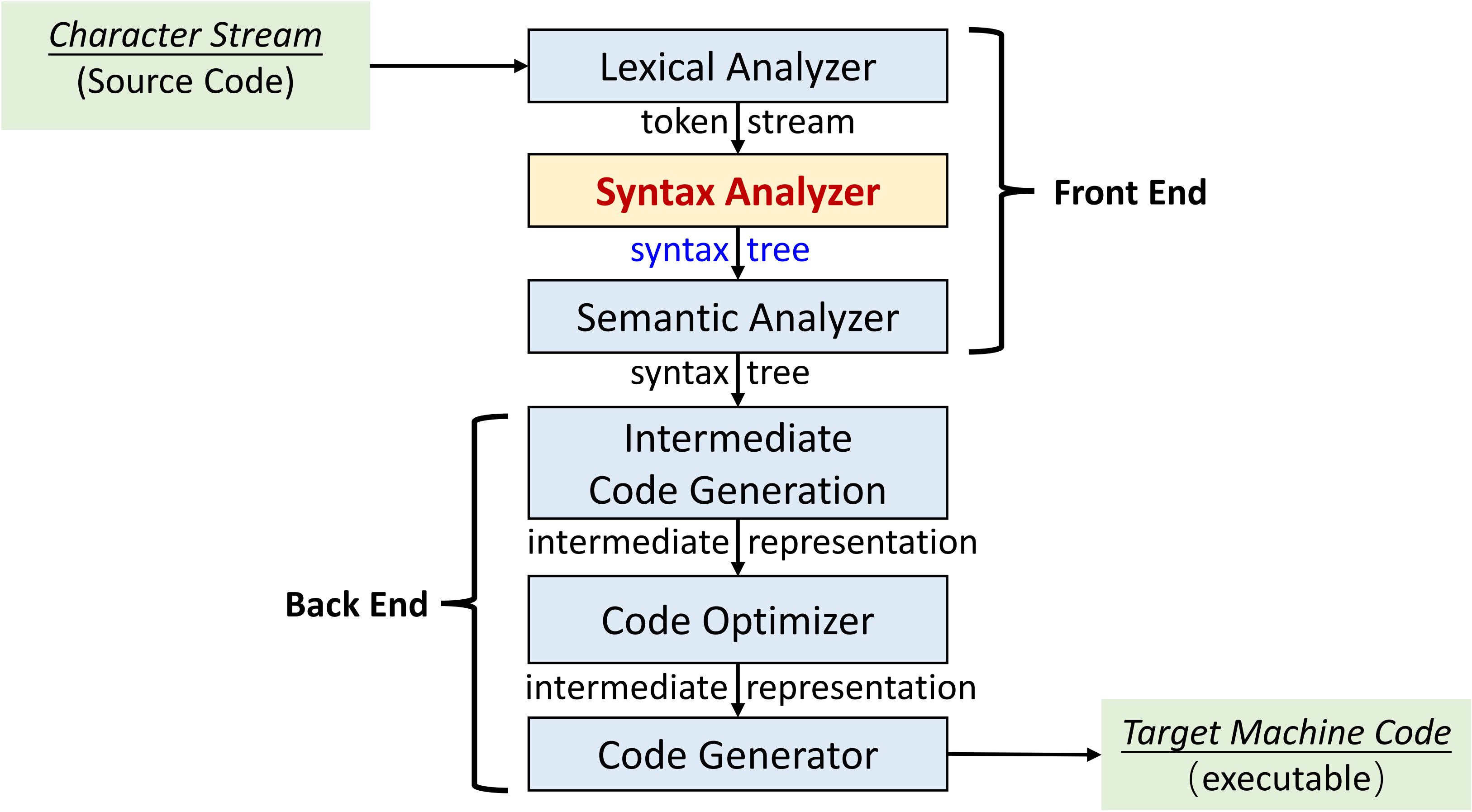
编译原理
Complier Principles

Lecture 3
Syntax Analysis: Intro & Parser & CFG

赵帅

计算机学院
中山大学

Compilation Phases[编译阶段]



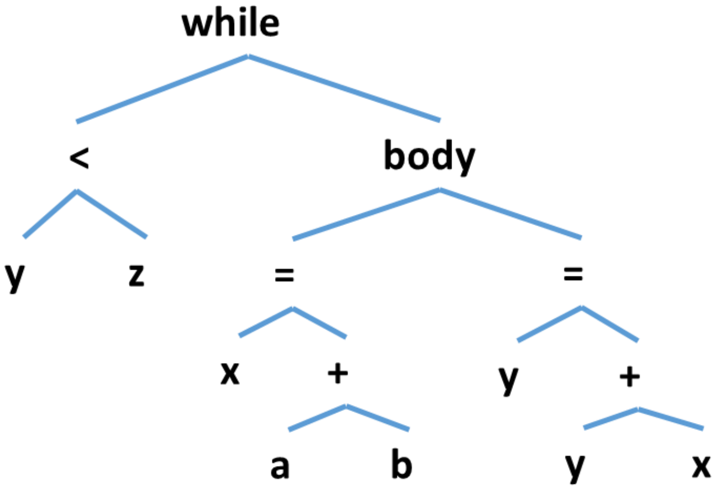
Compilation Phases[编译阶段]

```
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)

Syntax Analysis [语法分析]

- 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. [语法分析就是用来验证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.

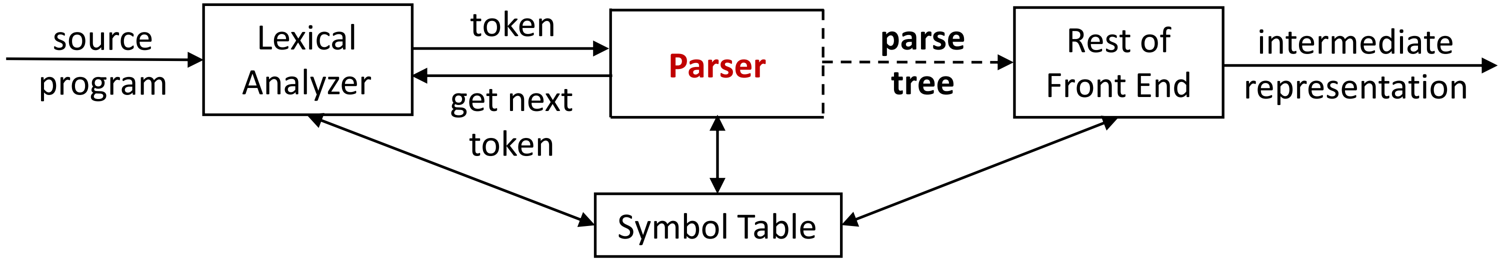
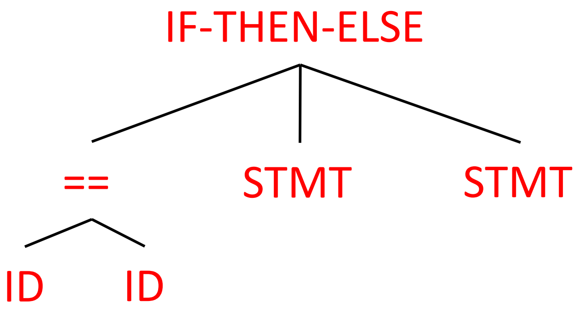


Fig. Position of parser in compiler model

Parsing Example[语法分析举例]

- **Example1:** Input: if x == y then 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串]

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) some Automata(FA); (3) Grammars. [文法]

How to Specify Syntax? [如何定义语法]

- 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] $G=(V_T, V_N, S, \delta)$
- 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.

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** [正确]
  - 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]

SUN YAT-SEN UNIVERSITY

- 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 $A \to \alpha$, where **A is from V_N** and **α from $(V_N \cup V_T)^*$**

- $G=<\{i, +, *, (,)\}, \quad \{E\}, \quad E, \quad \delta>$ [只含*,+的算术表达式上下文无关文法]
  - δ is composed of the following production：
    - $E \to i$;　　$E \to E+E$;　　$E \to E*E$;　　$E \to (E)$

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]
  - $\alpha \to \beta_1, \alpha \to \beta_2, \dots, \alpha \to \beta_n$
  - $\alpha \to \beta_1 | \beta_2 | \dots | \beta_n$, 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]
  - Digits such as 0,1,... ,9. [数字]
  - When discussing programming constructs, uppercase letters may represent non-terminals for the constructs.
    - E.g., *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:

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T \,/\, F \mid F$$

$$F \rightarrow (E) \mid \mathbf{id}$$

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

Derivation[推导]

- **Production rule**[产生式规则]: A → α, 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 αAβ, and A → γ is a production. Then, we write $\alpha A\beta \Rightarrow \alpha\gamma\beta$, the symbol ⇒ 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 rules
  - The process of derivation will start from start symbol.
  - In each step of derivation, the following choices need to be made:
    - choice of the non-terminal to be replaced. [替换哪个非终结符]
    - 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:

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid id$$

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

- 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]: $E \rightarrow T \mid E+T$; $T \rightarrow F \mid T*F$; $F \rightarrow (E) \mid i$

Leftmost Derivation:

$$
\begin{aligned}
E &\Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (E+T) \\
&\Rightarrow (T+T) \\
&\Rightarrow (T*F+T) \\
&\Rightarrow (F*F+T) \\
&\Rightarrow (i*F+T) \\
&\Rightarrow (i*i+T) \\
&\Rightarrow (i*i+F) \\
&\Rightarrow (i*i+i)
\end{aligned}
$$

Rightmost Derivation:

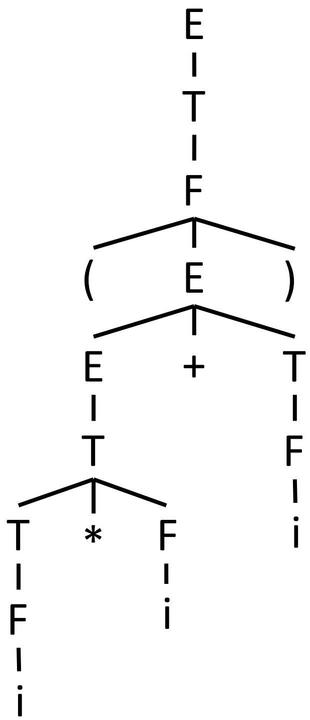
$$
\begin{aligned}
E &\Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (E+T) \\
&\Rightarrow (E+F) \\
&\Rightarrow (E+i) \\
&\Rightarrow (T+i) \\
&\Rightarrow (T*F+i) \\
&\Rightarrow (T*i+i) \\
&\Rightarrow (i*i+i)
\end{aligned}
$$

Parse Trees [分析树]

- Derivations can be summarized as a parse tree [语法分析树]

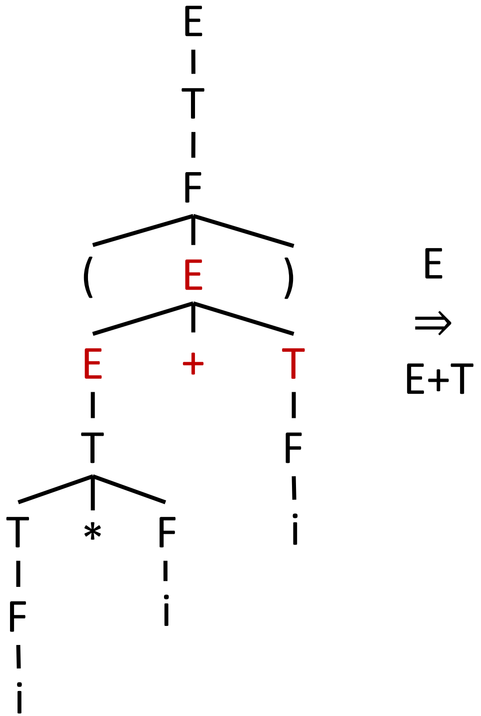
- A parse tree is a graphical representation of a **derivation** that *filters out the order* in which productions are applied to replace non-terminals. [过滤掉推导过程中对非终结符应用产生式的顺序]

- Both previous derivations result in the same parse tree.



Parse Trees [分析树]

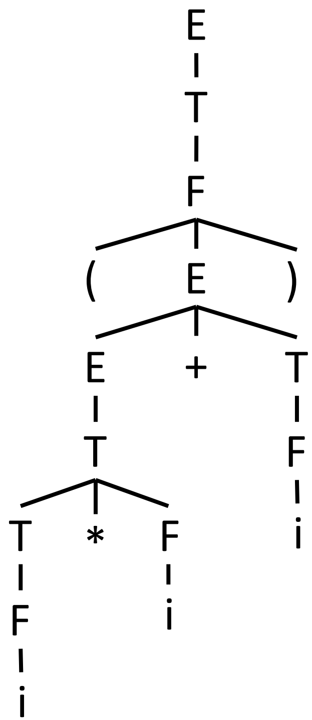
- Each **interior node**[内部结点] of a parse tree represents the application of a production. [产生式的应用]
  - **The interior node** is labeled with the non-terminal A **in the head of the production** [内部节点代表产生式左部]
  - **the children of the node** are labeled, from left to right, by the symbols **in the body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



Parse Trees [分析树]

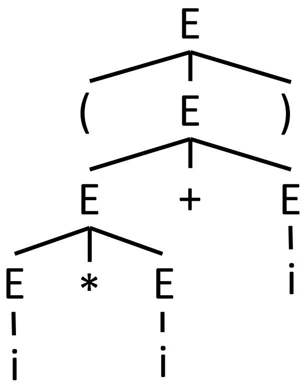
- **The leaves of a parse tree**[叶结点] are labeled by *non-terminals or terminals* and, **read from left to right**, constitute a sentential form [从左至右排列符号构成句型], called the *yield*[产出] or *frontier* [边缘] of the tree.

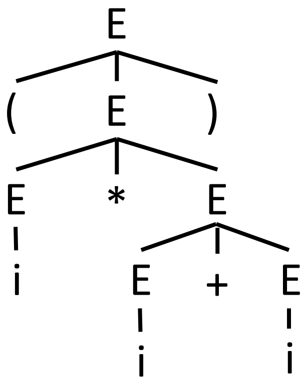
- Leftmost derivation order: builds tree left to right
- Rightmost derivation order: builds tree right to left
  - There is a one-to-one relationship between parse trees and either leftmost or rightmost derivations.[最左或最右推导与分析树具有一对一对应关系]



Ambiguity[二义性]

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





- If a grammar produces more than one parse tree for some sentence, it is said that the grammar is ambiguous.[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[二义性]

- Ambiguity of language: A language is ambiguous, **if there is no unambiguous grammar for it**.[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’). (Ambiguity is the property of the grammar, not the language, just because G is ambiguous, does not mean L(G) is inherently ambiguous)
- Ambiguity is an undecidable problem[不可判定问题], that is, there is no algorithm 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]
- A set of sufficient conditions for unambiguous grammar can be found. [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）

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).
  - Thus, such a grammar is unlikely to be useful for a programming language, because two structures for the same string (program) implies two different meanings (executable equivalent programs) for this program.
- Impossible to convert ambiguous to unambiguous grammar 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)

Remove Ambiguity[消除二义性]

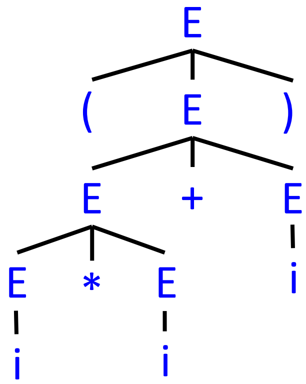
- Example:
- Grammar G(E): E → i | E+E | E*E | (E) is ambiguous.
- the above formula can construct the unambiguous grammar, If：
  - specify the precedence of ‘+’ and ‘*’ [指定优先级], + for example
  - specify the associativity[指定结合性] , that is, the left associative

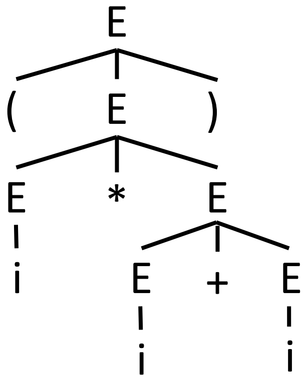
G[E]:

E→T|E+T

T→F|T*F

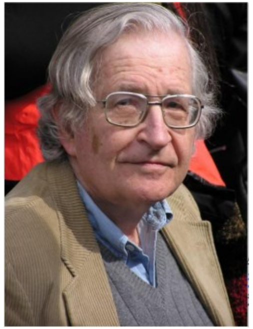
F→(E)|i





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 ***(V_T, V_N, S, δ)***, but have **different restrictions on production**.


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 Grammer $G=(V_T, V_N, S, \delta)$ is Type 0 [无限制文法,短语结构文法], if each production $\alpha \to \beta$ of G:
  - $\alpha \in (V_N \cup V_T)^*$, $\alpha \neq \varepsilon$
  - $\beta \in (V_N \cup V_T)^*$.
- Equivalent to 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 any production **$\alpha \to \beta$** 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 equivalent to linear bounded automaton (LBA) [线性有界自动机], example:
  - $\alpha A\beta \to \alpha\gamma\beta$: Only non-terminal A exists in the context of α and β, you can replace it with γ.
  - $A \to \gamma$: 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

- A Grammar $G=(V_T, V_N, S, \delta)$ is Type 2 [上下文无关文法], if any production **A → α** 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 → aBc: replace A with aBc regardless of context;

L = { $a^n b^n$ | $n \geq 0$} is **NOT regular** but **IS** a **context-free language**.
For the following CFG G=< V_T, V_N, S, δ > generates L:
V_T = { a,b }, V_N = { S } and δ = { S → aSb, S → ab }

Type 3: Regular Grammar

- A Grammar $G=(V_T, V_N, S, \delta)$ is Type 3 [正则文法], if any production **$A \rightarrow \alpha B$ or $A \rightarrow \alpha$** of G:
  - **$A, B \in V_N$**
  - **$\alpha \in V_T \cup \{\varepsilon\}$**
    - 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).
- Derivation
  - Derivation string length increases by 1 at each step
- Example
  - $A \rightarrow 1A \mid 0$
  - RE: 1*0

Type 3: Regular Grammar

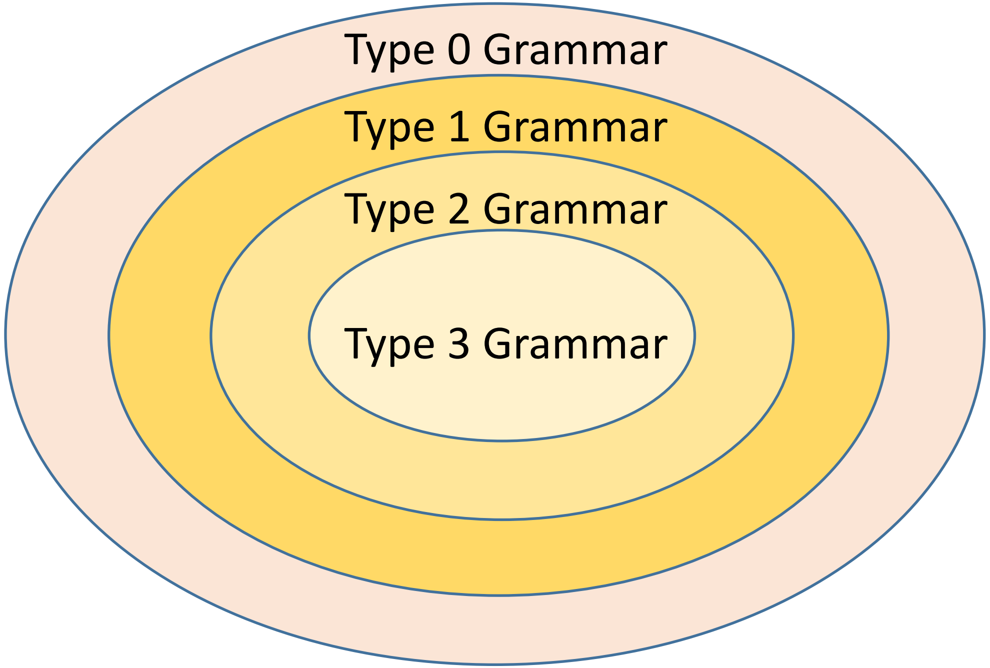
中山大學 SUN YAT-SEN UNIVERSITY

Useful in Practice

Useful in Theory

Class	Grammar	Restriction	Recognizer
3	Regular	$A \rightarrow aB$ or $A \rightarrow a$, where $A, B \in N \wedge a \in \Sigma \cup \{\varepsilon\}$. $A \rightarrow \varepsilon$ permitted if A is the start symbol and does not appear on the right of any production.	Finite-State Automaton (**FSA**)
2	Context-Free	$A \rightarrow \alpha$, where $A \in N \wedge \alpha \in (\Sigma \cup N)^*$.	Push-Down Automaton (**PDA**)
1	Context-Sensitive	$\alpha \rightarrow \beta$, where $\alpha, \beta \in (\Sigma \cup N)^* \wedge \alpha \neq \varepsilon \wedge \|\alpha\| \leq \|\beta\|$. 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	$\alpha \rightarrow \beta$, where $\alpha, \beta \in (\Sigma \cup N)^* \wedge \alpha \neq \varepsilon$.	Turing Machine (**TM**)

Comparison



In Practice[实际中]

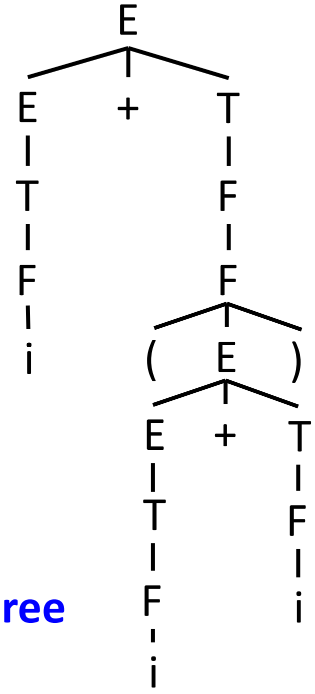
- Programming language (PL) is not context-free language, or even context-sensitive language.
- However, for today's programming languages, CFG is still used to describe the language structure in compilers.
  - ◆ Perfectly suited for describing recursive syntax of expressions and statements
  - ◆ CSG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

Others[其他]

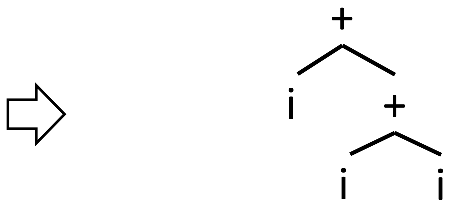
- What exactly is parsing, or syntax analysis?
  - To process an input string for a given grammar, and compose the derivation if the string is in the language.
  - Two subtasks: (1) determine if string can be derived from grammar or not; (2) build a representation of derivation and pass to next phase.
- 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 is abbreviated representation[缩写表示] of a parse tree and drops some details without compromising meaning [在不影响意义的情况下删除一些细节].



Parse tree



AST

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.
  - Although most programming languages are not context free, context sensitive analysis can easily be separated out to semantic analysis phase
- 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 (ASTs) : an abstract representation of a program's syntax.

Further Reading

- Dragon Book
  - Comprehensive Reading:
    - Section 2.4, 4.1.1–3.7, 3.9.6 for the introduction to parsing.
    - Section 4.2 and 4.3 for context free grammar and grammar transformations.