南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Chapter 2: Regular Expressions & Context-Free Grammars

Yepang Liu

liuyp1@sustech.edu.cn

The chapter numbering in lecture notes does not follow that in the textbook.

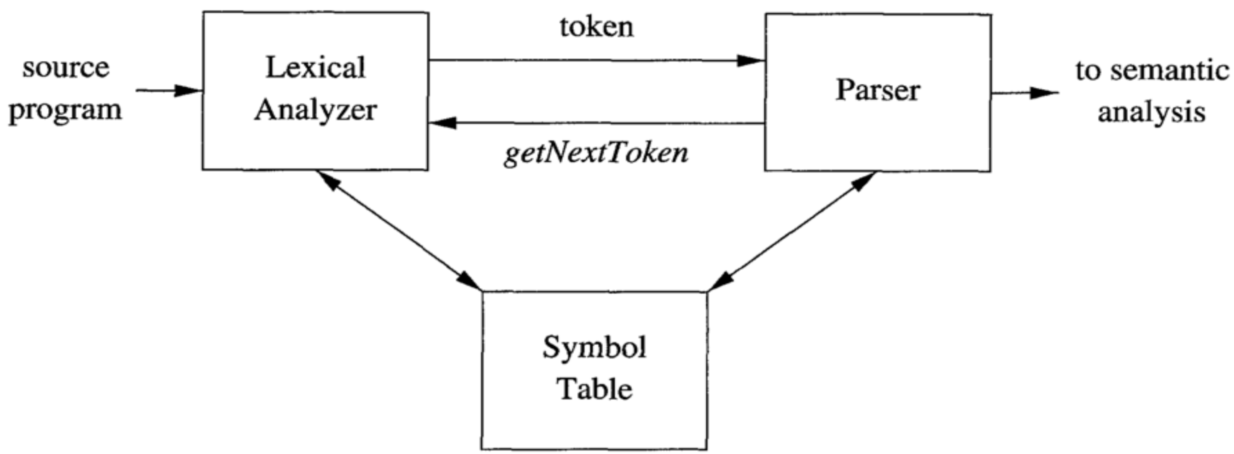
Outline

- The Role of Lexers: Recognizing Tokens
- Regular Expressions (for specifying tokens)
- The Role of Parsers: Finding Syntax Errors
- Context-Free Grammars (for describing syntax)

The Role of Lexical Analyzer

- Read the input characters of the source program, group them into lexemes, and produces a sequence of tokens

- Add lexemes into the symbol table when necessary



The Role of Lexical Analyzer

```
position = initial + rate * 60
```

Lexical Analyzer

```
<id, 1>
<=>
<id, 2>
<+>
<id, 3>
<*>
<60>
```

SYMBOL TABLE

1	position	...
2	initial	...
3	rate	...

Tokens, Patterns, and Lexemes

- A ***lexeme*** is a string of characters that is a lowest-level syntactic unit in programming languages
- A ***token*** is a syntactic category representing a class of lexemes. Formally, it is a pair <token name, attribute value>
  - Token name: an abstract symbol representing the kind of the token
  - Attribute value (optional) points to the symbol table
- Each token has a particular ***pattern***: a description of the form that the lexemes of the token may take

Examples

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Consider the C statement: `printf("Total = %d\n", score);`

Lexeme	printf	score	"Total = %d\n"	(	...
Token	**id**	**id**	**literal**	**left_parenthesis**	**...**

Attributes for Tokens

- When more than one lexeme match a pattern, the lexical analyzer must provide additional information, named *attribute values*, to the subsequent compiler phases
  - Token names influence parsing decisions
  - Attribute values influence semantic analysis, code generation etc.

- For example, an **id** token is often associated with: (1) its lexeme, (2) type, and (3) the location at which it is first found. Token attributes are stored in the symbol table.

```
A = B * 2   →   <id, pointer to symbol-table entry for A>
                <assign_op>
                <id, pointer to symbol-table entry for B>
                <mult_op> <number, integer value 2>
```

Lexical Errors

- When none of the patterns for tokens match any prefix of the remaining input
- Example: `int 3a = a * 3;`

Outline

- The Role of Lexers: Recognizing Tokens
- Specification of Tokens (Regular Expressions)
- The Role of Parsers: Finding Syntax Errors
- Specification of Syntax (Context-Free Grammars)

Specification of Tokens

- **Regular expression (正则表达式, regexp for short)** is an important notation for specifying lexeme patterns
- Content of this part
  - Strings and Languages (串和语言)
  - Operations on Languages (语言上的运算)
  - Regular Expressions
  - Regular Definitions (正则定义)
  - Extensions of Regular Expressions

Strings and Languages

- **Alphabet (字母表)**: any finite set of symbols
  - Examples of symbols: letters, digits, and punctuations
  - Examples of alphabets: {1, 0}, ASCII, Unicode
- A **string (串)** over an alphabet is a finite sequence of symbols drawn from the alphabet
  - The length of a string s, denoted $|s|$, is the number of symbols in s (i.e., cardinality)
  - Empty string (空串): the string of length 0, ϵ

Terms (using banana for illustration)

- **Prefix (前缀) of string *s*:** any string obtained by removing 0 or more symbols from the end of *s* (ban, banana, ϵ)
- **Proper prefix (真前缀):** a prefix that is not ϵ and not *s* itself (ban)
- **Suffix (后缀):** any string obtained by removing 0 or more symbols from the beginning of *s* (nana, banana, ϵ).
- **Proper suffix (真后缀):** a suffix that is not ϵ and not equal to *s* itself (nana)

Terms Cont.

- **Substring (子串) of s:** any string obtained by removing any prefix and any suffix from s (banana, nan, ϵ)
- **Proper substring (真子串)**: a substring that is not ϵ and not equal to s itself (nan)
- **Subsequence (子序列)**: any string formed by removing 0 or more not necessarily consecutive symbols from s (bnn)

How many substrings & subsequences does banana have?

(Two substrings are different as long as they have different start/end index)

String Operations (串的运算)

- **Concatenation (连接)**: the concatenation of two strings x and y, denoted xy, is the string formed by appending y to x
  - x = dog, y = house, xy = doghouse

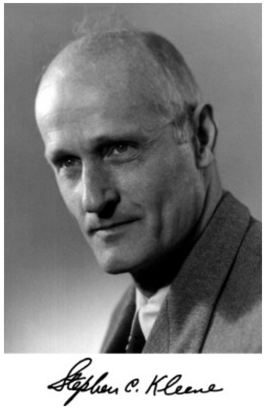
- **Exponentiation (幂/指数运算):** $s^0 = \epsilon$, $s^1 = s$, $s^i = s^{i-1}s$
  - x = dog, $x^0 = \epsilon$, x^1 = dog, x^3 = dogdogdog

Language (语言)

- A **language** is any **countable set**[1] of strings over some fixed alphabet
  - The set containing only the empty string, that is $\{\epsilon\}$, is a language, denoted $\emptyset$
  - The set of all grammatically correct English sentences
  - The set of all syntactically well-formed C programs

[1] In mathematics, a countable set is a set with the same cardinality (number of elements) as some subset of the set of natural numbers. A countable set is either a finite set or a countably infinite set.

Operations on Languages (语言的运算)



- 并，连接，Kleene闭包，正闭包

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

The exponentiation of L can be defined using concatenation. L^n means concatenating L n times.

https://en.wikipedia.org/wiki/Stephen_Cole_Kleene

Examples

- $L = \{A, B, \ldots, Z, a, b, \ldots, z\}$
- $D = \{0, 1, \ldots, 9\}$

$L \cup D$	$\{A, B, \ldots, Z, a, b, \ldots, z, 0, 1, \ldots, 9\}$
LD	the set of 520 strings of length two, each consisting of one letter followed by one digit
L^4	the set of all 4-letter strings
L^*	the set of all strings of letters, including ϵ
$L(L \cup D)^*$	?
D^+	?

Note: L, D might seem to be the alphabets of letters and digits. We define them to be languages: all strings happen to be of length one.

Regular Expressions - For Describing Languages/Patterns

Rules that define regexps over an alphabet Σ:

- **BASIS**: two rules form the basis:
  - ϵ is a regexp, $L(\epsilon) = \{\epsilon\}$
  - If a is a symbol in Σ, then a is a regexp, and $L(a) = \{a\}$
- **INDUCTION:** Suppose r and s are regexps denoting the languages $L(r)$ and $L(s)$
  - $(r)|(s)$ is a regexp denoting the language $L(r) \cup L(s)$
  - $(r)(s)$ is a regexp denoting the language $L(r)L(s)$
  - $(r)^*$ is a regexp denoting $(L(r))^*$
  - (r) is a regexp denoting $L(r)$. Additional parentheses do not change the language an expression denotes.

Regular Expressions Cont.

- Following the rules, regexps often contain unnecessary pairs of parentheses. We may drop some if we adopt the conventions:
  - **Precedence:** closure * > concatenation > union |
  - **Associativity:** All three operators are left associative, meaning that operations are grouped from the left, e.g., a | b | c would be interpreted as (a | b) | c
- Example: $(a) \mid ((b)^*(c)) = a \mid b^*c$

Regular Expressions Cont.

- Examples: Let $\Sigma = \{a, b\}$
  - $a|b$ denotes the language $\{a, b\}$
  - $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$
  - a^* denotes $\{\epsilon, a, aa, aaa, \ldots\}$
  - $(a|b)^*$ denotes the set of all strings consisting of 0 or more *a*'s or *b*'s: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$
  - $a|a^*b$ denotes the string *a* and all strings consisting of 0 or more *a*'s and ending in *b*: $\{a, b, ab, aab, aaab, \ldots\}$

Regular Language (正则语言)

- A **regular language** is a language that can be defined by a regexp
- If two regexps r and s denote the same language, they are *equivalent*, written as $r = s$

Regular Language Cont.

- Each algebraic law below asserts that expressions of two different forms are equivalent

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

Is (**a** | **b**)(**a** | **b**) = **aa** | **ab** | **ba** | **bb** true?

| can be viewed as + in arithmetics, concatenation can be viewed as ×, * can be viewed as the power operator.

Regular Definitions (正则定义)

- For notational convenience, we can give names to certain regexps and use those names in subsequent expressions

If Σ is an alphabet of basic symbols, then a ***regular definition*** is a sequence of definitions of the form:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$d_n \rightarrow r_n$$

where:

- Each d_i is a new symbol not in Σ and not the same as the other d's
- Each r_i is a regexp over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Each new symbol denotes a regular language. The second rule means that you may reuse previously-defined symbols.

Examples

- Regular definition for C identifiers

$$
\begin{array}{rcl}
letter\_ & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
id & \rightarrow & letter\_ \; (\; letter\_ \mid digit \;)^{*}
\end{array}
$$

`_hello` valid?

`3times` valid?

- Regexp for C identifiers

```
(A|B|...|Z|a|b|...|z|_)((A|B|...|Z|a|b|...|z|_)|(0|1|
...|9))*
```

Extensions of Regular Expressions

- **Basic operators:** union $|$, concatenation, and Kleene closure * (proposed by Kleene in 1950s)
- A few **notational extensions**:
  - One of more instances: the unary, postfix operator $^+$
    - $r^+ = rr^*$, $r^* = r^+ \mid \epsilon$
  - Zero or one instance: the unary postfix operator ?
    - $r? = r \mid \epsilon$
  - Character classes: shorthand for a logical sequence
    - $[a_1 a_2 \ldots a_\mathrm{n}] = a_1 \mid a_2 \mid \ldots \mid a_n$
    - $[a\text{-}e] = a \mid b \mid c \mid d \mid e$
- The extensions are only for notational convenience, they do not change the descriptive power of regexps

Outline

- The Role of Lexers: Recognizing Tokens
- Regular Expressions (for specifying tokens)
- The Role of Parsers: Finding Syntax Errors
- Context-Free Grammars (for describing syntax)

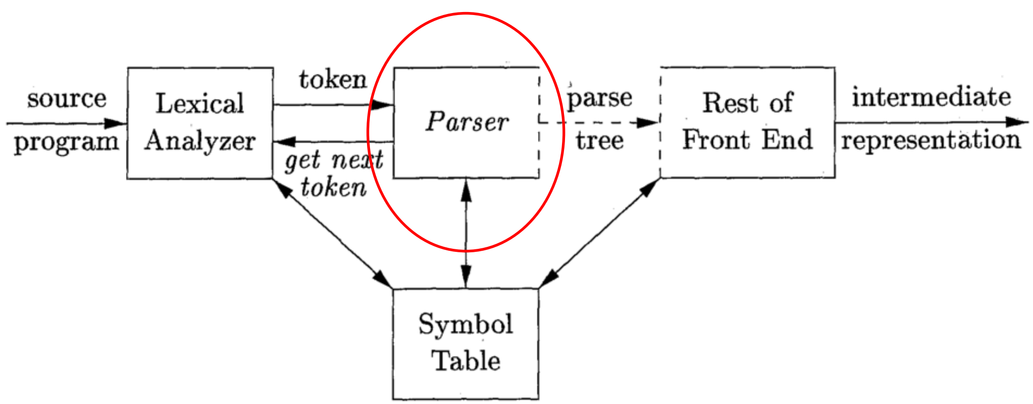
Describing Syntax

- The syntax of programming language constructs can be specified by context-free grammars[1]
  - A grammar gives a precise and easy-to-understand syntactic specification of a programming language, defining its structure
  - For certain grammars, we can automatically construct an efficient parser
  - A properly designed grammar helps translate source programs into correct object code and detect errors

[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
    - The parse tree need not be constructed explicitly



Outline

- The Role of Lexers: Recognizing Tokens
- Regular Expressions (for specifying tokens)
- The Role of Parsers: Finding Syntax Errors
- Context-Free Grammars (for describing syntax)
  - Formal definition of CFG
  - Derivation and parse tree
  - Ambiguity
  - CFG vs. regexp

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

- The Role of Lexers: Recognizing Tokens
- Regular Expressions (for specifying tokens)
- The Role of Parsers: Finding Syntax Errors
- Context-Free Grammars (for describing syntax)
  - Formal definition of CFG
  - Derivation and parse tree
  - Ambiguity
  - CFG vs. regexp

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"

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

- $\stackrel{+}{\Rightarrow}$ 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 ***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 with no 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

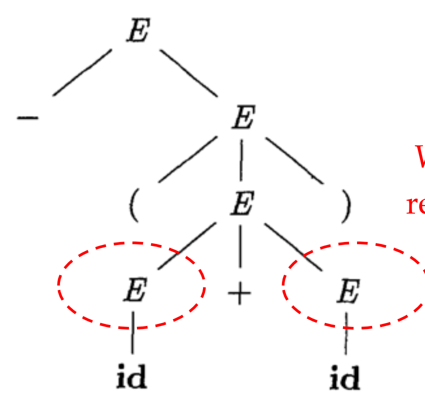
- At each step of a derivation, we need to choose which nonterminal to replace
- In **leftmost derivations (最左推导)**, the leftmost nonterminal in each sentential form is always chosen to be replaced
  - $E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$
- 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: $E \rightarrow - E \mid E + E \mid E * E \mid (E) \mid \mathbf{id}$

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



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.

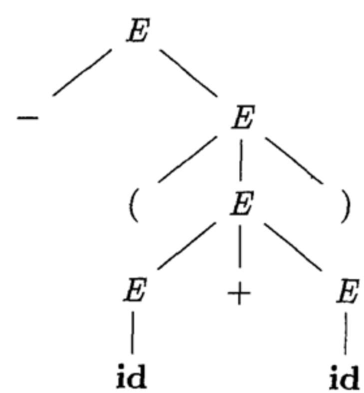
Parse Trees (语法分析树) Cont.

- 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.

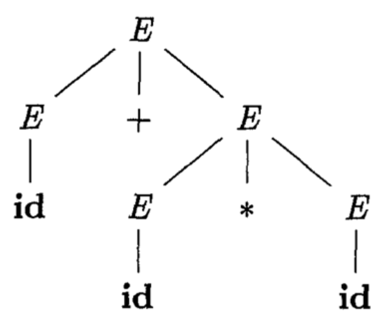
Outline

- The Role of Lexers: Recognizing Tokens
- Regular Expressions (for specifying tokens)
- The Role of Parsers: Finding Syntax Errors
- Context-Free Grammars (for describing syntax)
  - Formal definition of CFG
  - Derivation and parse tree
  - Ambiguity
  - CFG vs. regexp

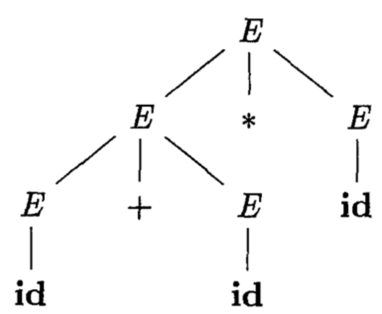
Ambiguity (二义性)

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

- The Role of Lexers: Recognizing Tokens
- Regular Expressions (for specifying tokens)
- The Role of Parsers: Finding Syntax Errors
- Context-Free Grammars (for describing syntax)
  - 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)

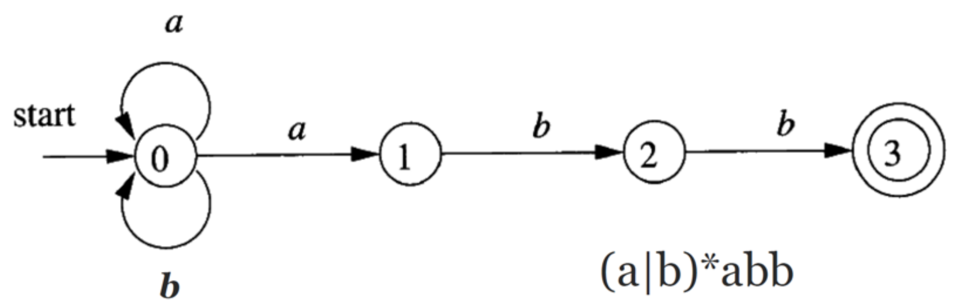
  2. Some context-free languages cannot be described using regular expressions

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

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



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

Reading Tasks

- Chapter 3 of the dragon book
  - 3.1 The role of the lexical analyzer
  - 3.3 Specification of tokens

- Chapter 4 of the dragon book
  - 4.1 Introduction
  - 4.2 Context-Free Grammars
  - 4.3 Writing a Grammar (4.3.1 – 4.3.4)