CS 4300: Compiler Theory

Chapter 4
Syntax Analysis

Dr. Xuejun Liang

Quick Review of Last Lecture

- Design of a Lexical-Analyzer Generator
  - Construct and simulate an NFA from a Lex Program
  - Convert the NFA to a DFA and simulate the DFA
- From RE to DFA Directly
  - Simulate a DFA that recognizes L(r) given a regular expression r.
    - (Annotated) Syntax Tree of a regular expression
    - nullable(n)
    - firstpos(n)
    - lastpos(n)
    - followpos(p)
    - Algorithm: Construct Dstates, and Dtran

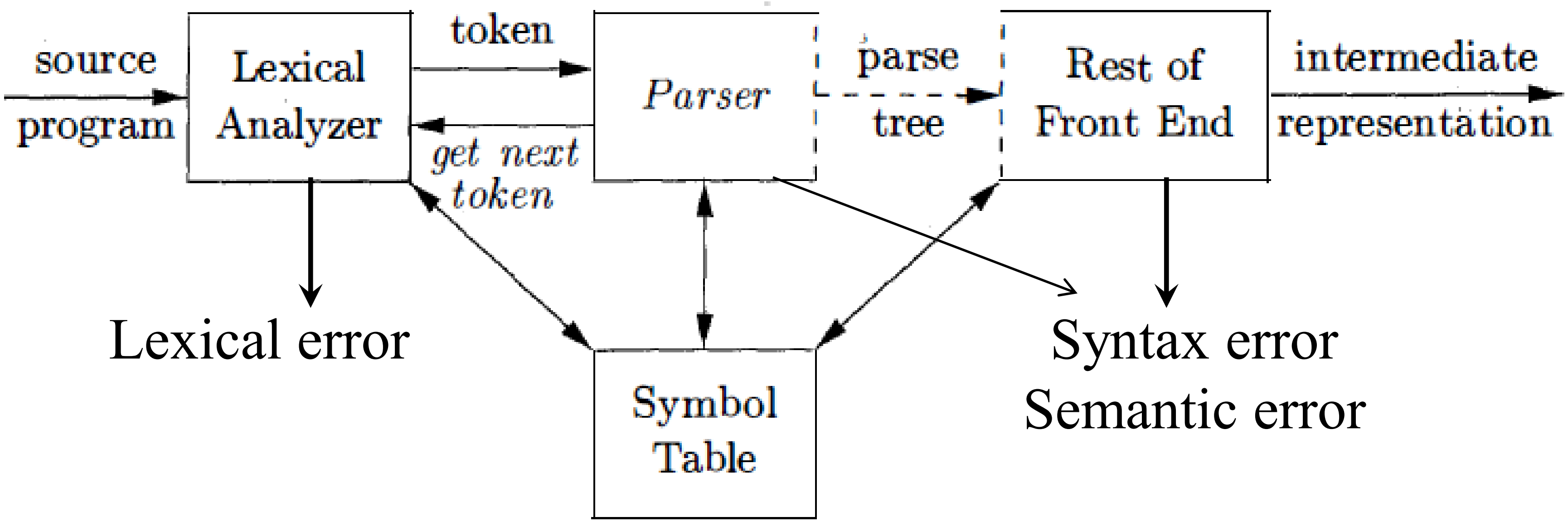
Outlines (Sections)

1. Introduction
2. Context-Free Grammars
3. Writing a Grammar
4. Top-Down Parsing
5. Bottom-Up Parsing
6. Introduction to LR Parsing: Simple LR
7. More Powerful LR Parsers
8. Using Ambiguous Grammars
9. Parser Generators

1. The role of the Parser

- A parser implements a Context-Free grammar as a recognizer of strings
- The role of the parser in a compiler is twofold:
  - To check syntax (= string recognizer)
    - And to report syntax errors accurately
  - To invoke semantic actions
    - For static semantics checking, e.g. type checking of expressions, functions, etc.
    - For syntax-directed translation of the source code to an intermediate representation

Position of Parser in Compiler Model



Error Handling

- A good compiler should be able to identify and locate errors and able to recover from errors
- Common programming errors can occur at many different levels
    - *Lexical errors*: important, compiler can easily recover and continue
    - *Syntax errors*: most important for compiler, can almost always recover
    - *Static semantic errors*: important, can sometimes recover
    - *Dynamic semantic errors*: hard or impossible to detect at compile time, runtime checks are required
    - *Logical errors*: hard or impossible to detect

Viable-Prefix Property

- The *viable-prefix property* of parsers allows early detection of syntax errors
  - Goal: detection of an error *as soon as possible* without further consuming unnecessary input
  - How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language

Error is detected here

Prefix

```
…
for (;)
…
```

Error Recovery Strategies

- *Panic mode*
  - Discard input until a token in a set of designated synchronizing tokens (such as ;) is found.
- *Phrase-level recovery*
  - Perform local correction on the input to repair the error
- *Error productions*
  - Augment grammar with productions for erroneous constructs
- *Global correction*
  - Choose a minimal sequence of changes to obtain a global least-cost correction

Representative Grammars (Expression)

LR grammar

- Suitable for bottom-up parsing.
- Not suitable for top-down parsing
  - Because it is left recursive

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

LL grammar

- Non-left-recursive
- Suitable for top-down parsing

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & + \ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & * \ F\ T' \mid \epsilon \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

Ambiguous Grammar

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

2. Context-Free Grammars (Recap)

- Context-free grammar is a 4-tuple
  $G = (N, T, P, S)$ where
  - T is a finite set of tokens (*terminal* symbols)
  - N is a finite set of *nonterminals*
  - P is a finite set of *productions* of the form

    $$\alpha \to \beta$$

    where $\alpha \in (N \cup T)^* \, N \, (N \cup T)^*$ and $\beta \in (N \cup T)^*$
  - $S \in N$ is a designated *start symbol*

Notational Conventions

- Terminals

  $a,b,c,\ldots \in T$

  specific terminals: **0**, **1**, **id**, **+**
- Nonterminals

  $A,B,C,\ldots \in N$

  specific nonterminals: *expr*, *term*, *stmt*
- Grammar symbols

  $X,Y,Z \in (N \cup T)$
- Strings of terminals

  $u,v,w,x,y,z \in T^*$
- Strings of grammar symbols

  $\alpha,\beta,\gamma \in (N \cup T)^*$

Derivations (Recap)

- The *one-step derivation* is defined by
  $$\alpha\, A\, \beta \Rightarrow \alpha\, \gamma\, \beta$$
  where $A \to \gamma$ is a production in the grammar
- In addition, we define
  - $\Rightarrow$ is *leftmost* $\Rightarrow_{lm}$ if α does not contain a nonterminal
  - $\Rightarrow$ is *rightmost* $\Rightarrow_{rm}$ if β does not contain a nonterminal
  - Transitive closure $\Rightarrow^*$ (zero or more steps)
  - Positive closure $\Rightarrow^+$ (one or more steps)
- The *language generated by G* is defined by
  $$L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$$

Derivation (Example)

Grammar $G = (\{E\}, \{+, *, \mathbf{(}, \mathbf{)}, \mathbf{-}, \mathbf{id}\}, P, E)$ with productions $P =$

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

Example derivations:

$$E \Rightarrow \mathbf{-}\ E \Rightarrow \mathbf{-\ id}$$

$$E \Rightarrow_{rm} E + E \Rightarrow_{rm} E + \mathbf{id} \Rightarrow_{rm} \mathbf{id} + \mathbf{id}$$

$$E \Rightarrow^{*} E$$

$$E \Rightarrow^{*} \mathbf{id} + \mathbf{id}$$

$$E \Rightarrow^{+} \mathbf{id} * \mathbf{id} + \mathbf{id}$$

Language Classification

- A grammar G is said to be
  - *Regular* if it is *right linear* where each production is of the form

    $A \rightarrow w\,B$ or $A \rightarrow w$

    or *left linear* where each production is of the form

    $A \rightarrow B\,w$ or $A \rightarrow w$
  - *Context free* if each production is of the form

    $A \rightarrow \alpha$

    where $A \in N$ and $\alpha \in (N \cup T)^*$
  - *Context sensitive* if each production is of the form

    $\alpha\,A\,\beta \rightarrow \alpha\,\gamma\,\beta$

    where $A \in N$, $\alpha, \gamma, \beta \in (N \cup T)^*$, $|\gamma| > 0$
  - *Unrestricted*

Chomsky Hierarchy

$\mathbb{L}(\textit{regular}) \subset \mathbb{L}(\textit{context free}) \subset \mathbb{L}(\textit{context sensitive}) \subset \mathbb{L}(\textit{unrestricted})$

Where $\mathbb{L}(T) = \{\, L(G) \mid G \text{ is of type } T \,\}$
That is: the set of all languages generated by grammars G of type T

Examples:

Every *finite language* is regular!
(construct a FSA for strings in $L(G)$)

$L_1 = \{\, \mathbf{a}^n\mathbf{b}^n \mid n \geq 1 \,\}$ is context free, but not regular

$L_2 = \{\, \mathbf{wcw} \mid \mathbf{w} \text{ is in } \mathrm{L}(\mathbf{a}|\mathbf{b})^*\}$ is context sensitive

$L_3 = \{\, \mathbf{a}^n\mathbf{b}^m\mathbf{c}^n\mathbf{d}^m \mid n \geq 1 \,\}$ is context sensitive

3. Lexical Versus Syntactic Analysis

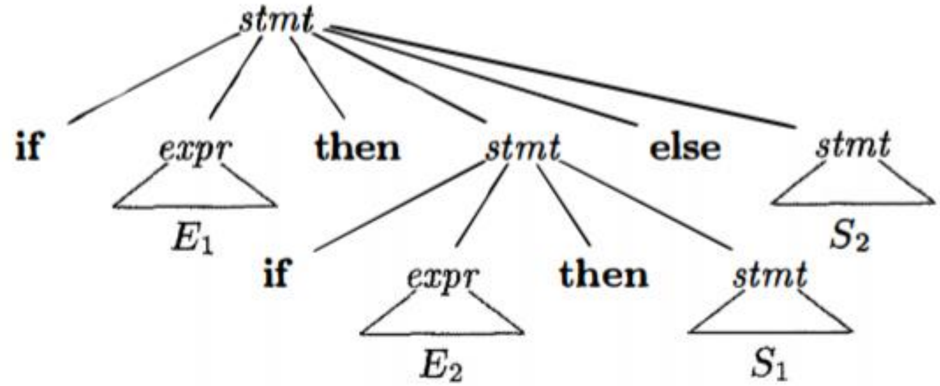
- Why use regular expressions to define the lexical syntax of a language?
  - Quite simple, more concise and easier-to-understand
  - More efficient lexical analyzers can be constructed automatically from regular expressions
  - Regular expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords, and white space.
  - Grammars are most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on.

Eliminating Ambiguity (1)

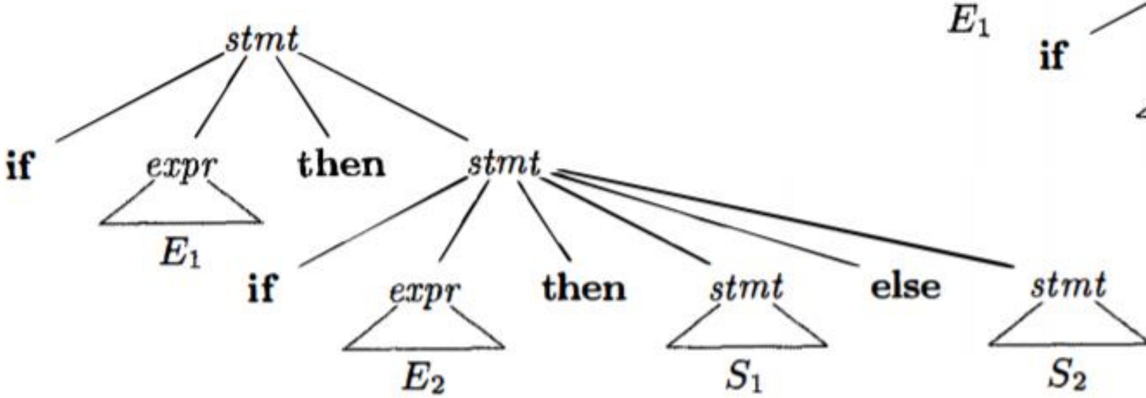
Ambiguous grammar:
"dangling else"

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

if E1 **then if** E2 **then** S1 **else** S2



(2)



(1)

Eliminating Ambiguity (2)

Ambiguous grammar: "dangling else"

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

Unambiguous grammar for if-then-else statements

$$
\begin{array}{rcl}
stmt & \rightarrow & matched\_stmt \\
 & | & open\_stmt \\
matched\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } matched\_stmt \\
 & | & \textbf{other} \\
open\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } open\_stmt
\end{array}
$$

Eliminating Ambiguity (3)

if E1 **then if** E2 **then** S1 **else** S2

Stmt => open_stmt => **if** expr **then** stmt => **if** expr **then** matched_stmt

=> **if** expr **then if** expr **then** matched_stmt **else** matched _stmt

=>* **if** E1 **then if** E2 **then** S1 **else** S2

Unambiguous grammar for if-then-else statements

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{matched\_stmt} \\
 & | & \mathit{open\_stmt} \\
\mathit{matched\_stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{matched\_stmt} \\
 & | & \textbf{other} \\
\mathit{open\_stmt} & \rightarrow & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
 & | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{open\_stmt}
\end{array}
$$

Left Recursion

- A grammar is **left recursive** if it has a nonterminal *A* such that there is a derivation $A \stackrel{+}{\Rightarrow} A\ \alpha$ for some string α.
- When a grammar is left recursive then a predictive parser loops forever on certain inputs.
- **Immediate left recursion**, where there is a production of the form $A \rightarrow A\ \alpha$.

$$\begin{aligned} A &\rightarrow A\ \alpha \\ &\mid \beta \\ &\mid \gamma \end{aligned} \quad \Longrightarrow \quad \begin{aligned} A &\rightarrow \beta\ R \\ &\mid \gamma\ R \\ R &\rightarrow \alpha\ R \\ &\mid \varepsilon \end{aligned}$$

Algorithm to eliminate left recursion

Input: Grammar G with no cycles or ε-productions

Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$
for $i = 1, \ldots, n$ {
 for $j = 1, \ldots, i\text{-}1$ {
  replace each
   $A_i \to A_j \gamma$
  with
   $A_i \to \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$
  where
   $A_j \to \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
 }
 eliminate the *immediate left recursion* in A_i
}