南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 4

Yepang Liu

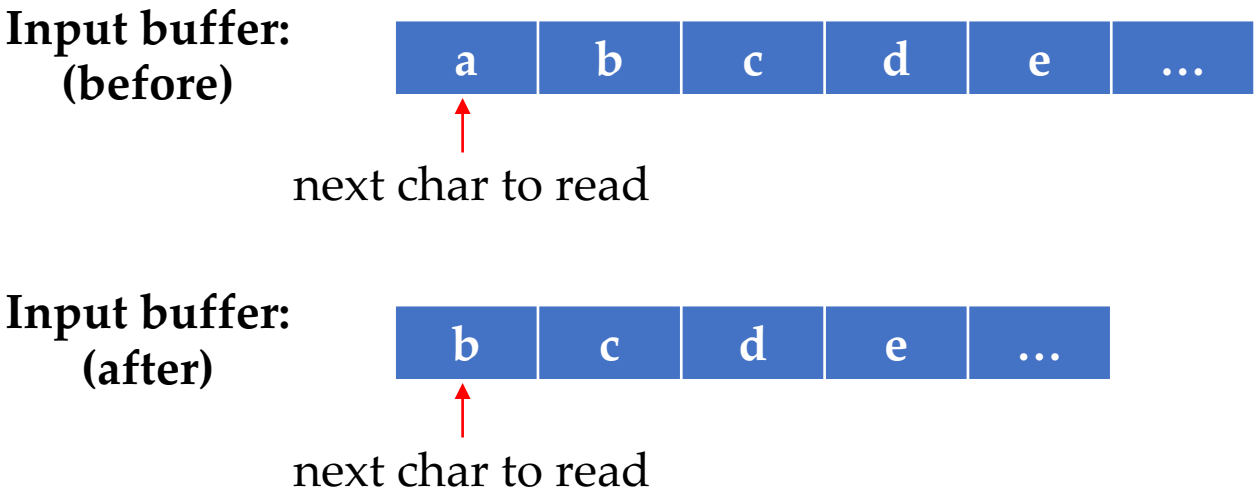
liuyp1@sustech.edu.cn

Outline

- Flex Library Functions
  - `input()`, `unput()`, `yyless()`, `yymore()`
- Grammar Design Issues

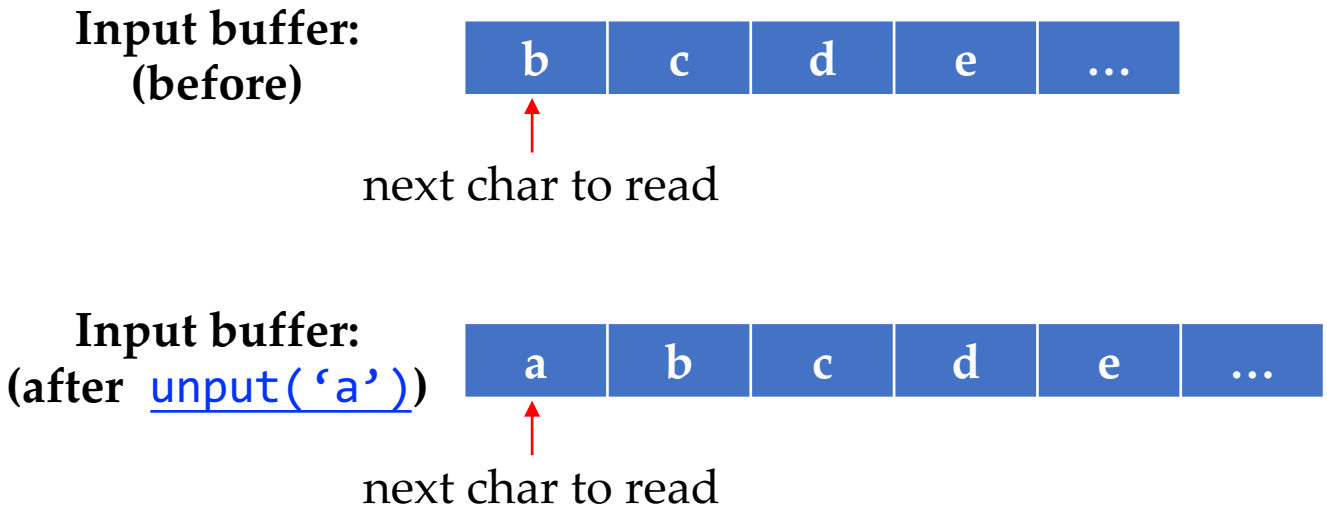
The `input()` Function

- The function takes no arguments
- When called, it reads a single character from the input buffer and return it to the caller



The `unput(char c)` Function

- The function puts `c` back into the input buffer



Example

- End-of-line comments sanitizer

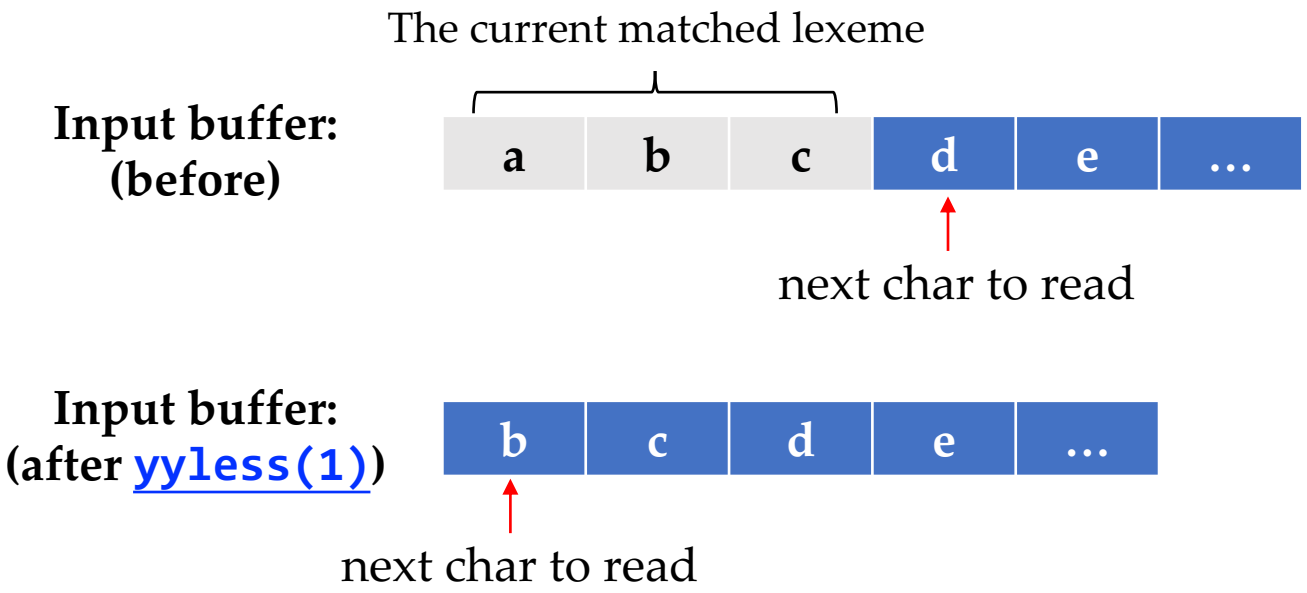
```
"//" {
  // ignore the following chars until seeing a newline character
  while((c = input()) != '\n');
  // put the newline character back to the input buffer
  unput(c);
}
```

Steps:

- Go to the "`lab4/comment_sanitizer`" directory
- Run command "`make sanitizer`"
- Run command "`./sanitizer.out test.c`" and observe the effect (compare the output after running the command with the original C program in test.c)

The `yyless(int n)` Function

- The function returns the `yyleng-n` characters in the postfix of the current lexeme back to the input buffer



The `yymore()` Function

- The function causes the next matched token's `yytext` to be appended to the current `yytext`

```
abc { yymore(); }

def { printf("%s\n", yytext); }
```

Input string:

When matching "def", `yytext` will be "abcdef" instead of "def".

Exercise

- Dealing with nested quotation marks when recognizing string literals

```
printf("This is a string literal without nested quotation marks");
printf("And God said, \"Let there be light,\" and there was light.");
```

If we have the following translation rules:

```
\"[^\"]*\" { printf("Matched a string literal: %s\n", yytext); }
\n {}
. {}
```

When processing the above print statements, we will see this output:

```
Matched a string literal: "This is a string literal without nested quotation marks"
Matched a string literal: "And God said, \"
Matched a string literal: " and there was light."
```

Exercise

- Please modify the translation rule for string literals such that when processing the previous two print statements, we will see the correct output (hint: use `yyless` and `yymore` to manipulate the input buffer and `yytext`)

```
Matched a string literal: "This is a string literal without nested quotation marks"
Matched a string literal: "And God said, \"Let there be light,\" and there was light."
```

Go to the "`lab4/nested_quotation_marks`" directory. We have provided the `lex.l` file and the input program file `test.c`. The build target "`nest`" can be used. You only need to modify the `lex.l` file and then try to run the command "`./nest test.c`".

Outline

- Flex Library Functions
  - `input()`, `unput()`, `yyless()`, `yymore()`
- Grammar Design Issues

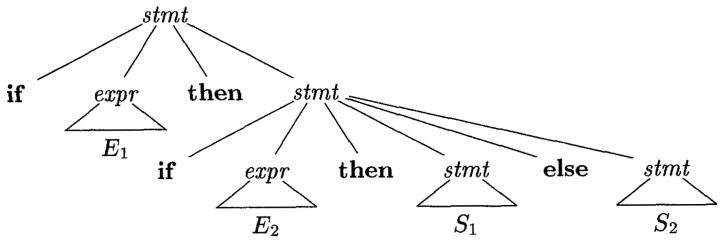
Grammar Design

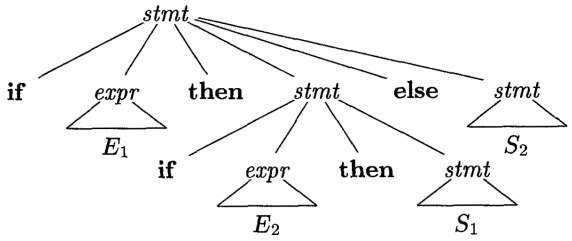
- CFGs are capable of describing most, but not all, of the syntax of programming languages
  - "Identifiers should be declared before use" cannot be described by a CFG
  - Subsequent phases must analyze the output of the parser to ensure compliance with such rules
- Before parsing, we typically apply several transformations to a grammar to make it more suitable for parsing
  - Eliminating ambiguity (消除二义性)
  - Eliminating left recursion (消除左递归)
  - Left factoring (提取左公因子)

Eliminating Ambiguity (1)

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

Two parse trees for **if** E_1 **then if** E_2 **then** S_1 **else** S_2





Which parse tree is preferred in programming?
(i.e., else matches which then?)

Eliminating Ambiguity (2)

- **Principle of proximity:** match each `else` with the closest unmatched `then`
  - **Idea of rewriting:** A statement appearing between a `then` and an `else` must be matched (must not end with an unmatched `then`)

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

Rewriting grammars to eliminate ambiguity is difficult. There are no general rules to guide the process.

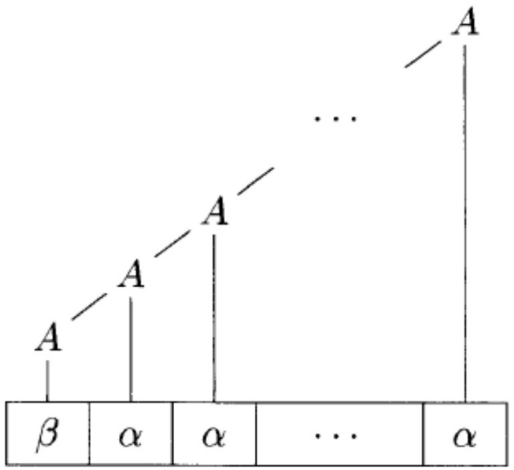
* open_stmt means the last then may not have matching else

Eliminating Left Recursion

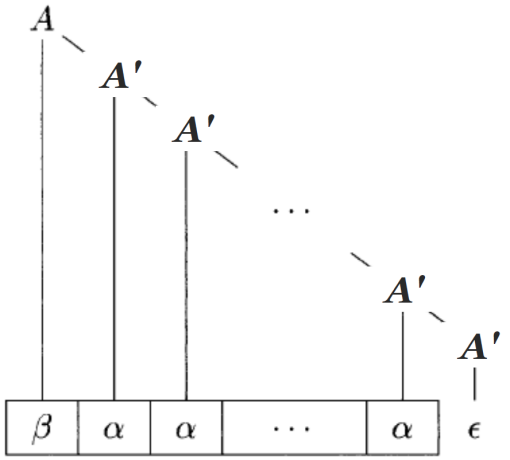
- A grammar is **left recursive** if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α
  - $S \rightarrow Aa \mid b$
  - $A \rightarrow Ac \mid Sd \mid \epsilon$
  - Because $S \Longrightarrow Aa \Longrightarrow Sda$
- **Immediate left recursion (立即左递归):** the grammar has a production of the form $A \rightarrow A\alpha$
- Top-down parsing methods cannot handle left-recursive grammars (bottom-up parsing methods can handle…)

Eliminating Immediate Left Recursion

- Simple grammar: $A \rightarrow A\alpha \mid \beta$
  - It generates sentences starting with the symbol β followed by zero or more α's



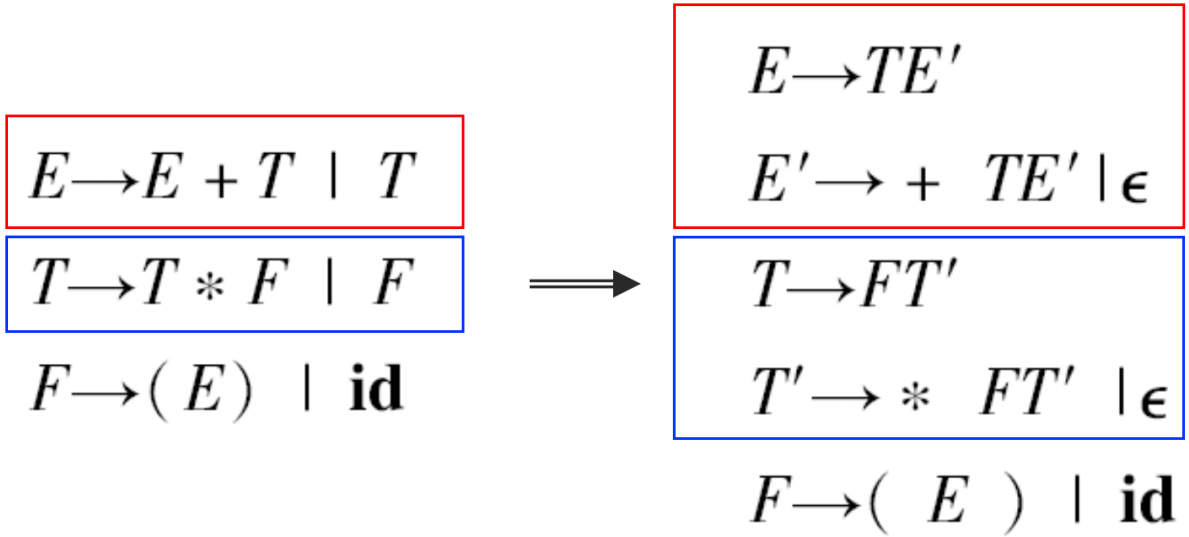
- Replace the grammar by:
  - $A \rightarrow \beta A'$
  - $A' \rightarrow \alpha A' \mid \epsilon$
  - It is right recursive now



Eliminating Immediate Left Recursion

- The general case: $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$
- Replace the grammar by:
  - $A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$
  - $A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$

Example


$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \mathbf{id}$
$E \rightarrow TE'$
$E' \rightarrow + \ TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow * \ FT' \mid \epsilon$
$F \rightarrow (\ E \ ) \mid \mathbf{id}$

Left Factoring (提取左公因子)

- If we have the following two productions

$$stmt \rightarrow \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt$$
$$\mid \quad \textbf{if } expr \textbf{ then } stmt$$

- On seeing input **if**, we cannot immediately decide which production to choose
- In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two productions, and the input begins with a nonempty string derived from α. We may defer choosing productions by expanding A to $\alpha A'$ first

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$