CS323 Lab 4

Yepang Liu

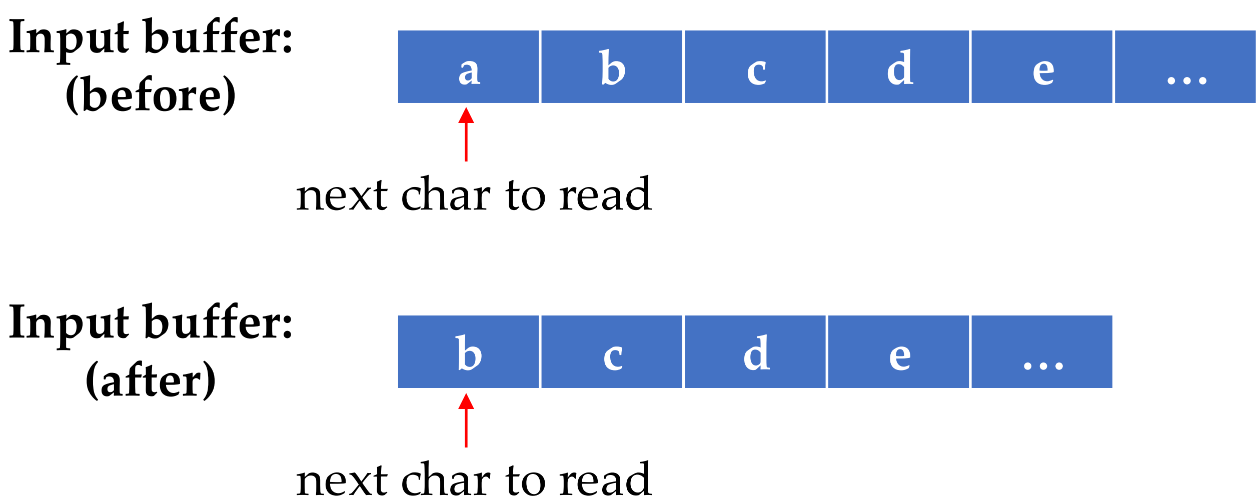
liuyp1@sustech.edu.cn

Outline

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

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

Input buffer: (before)

b	**c**	**d**	**e**	**…**

next char to read

Input buffer: (after `unput('a')`)

a	**b**	**c**	**d**	**e**	**…**

next char to read

Example

- End-of-line comments sanitizer

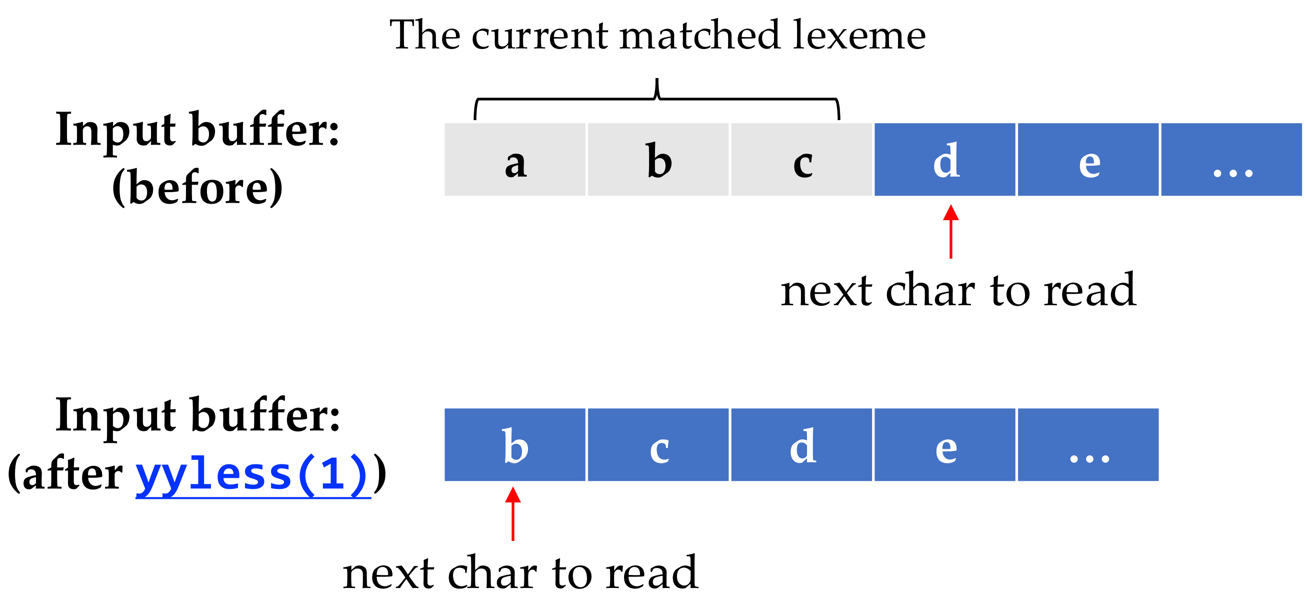
```
"//" {
  // ignore the following chars until seeing a newline character
  while((c = input()) != '\n');
  // put the newline character back to the input buffer
  unput(c);
}
```

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:	a	b	c	d	e	f

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
- SPL Grammar Rules

Grammar Design

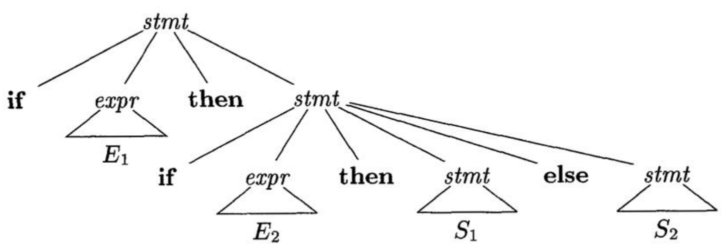
- CFGs are capable of describing most, but not all, of the syntax of programming languages
  - "Identifiers should be declared before use" cannot be described by a CFG
  - Subsequent phases must analyze the output of the parser to ensure compliance with such rules

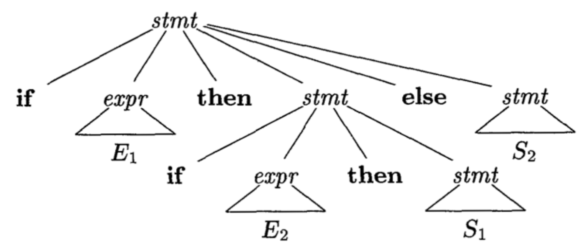
- Before parsing, we typically apply several transformations to a grammar to make it more suitable for parsing
  - Eliminating ambiguity (消除二义性)
  - Eliminating left recursion (消除左递归)
  - Left factoring (提取左公因子)

Eliminating Ambiguity (1)

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

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`**)

$$
\begin{aligned}
\mathit{stmt} &\rightarrow \mathit{matched\_stmt} \\
&\mid \mathit{open\_stmt} \\
\mathit{matched\_stmt} &\rightarrow \textbf{if } \mathit{expr} \textbf{ then } \boxed{\mathit{matched\_stmt}} \textbf{ else } \mathit{matched\_stmt} \\
&\mid \textbf{other} \\
\mathit{open\_stmt} &\rightarrow \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
&\mid \textbf{if } \mathit{expr} \textbf{ then } \boxed{\mathit{matched\_stmt}} \textbf{ else } \mathit{open\_stmt}
\end{aligned}
$$

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

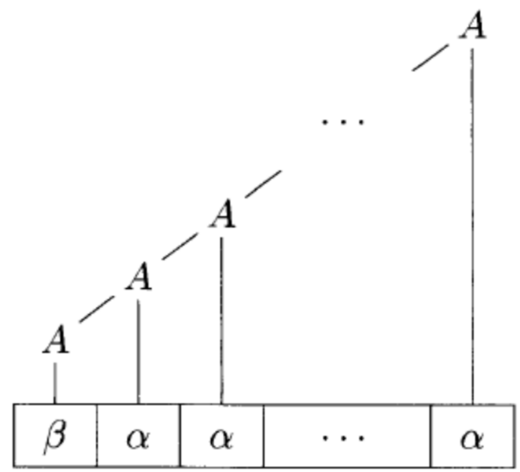
* open_stmt means the last then may not have matching else

Eliminating Left Recursion

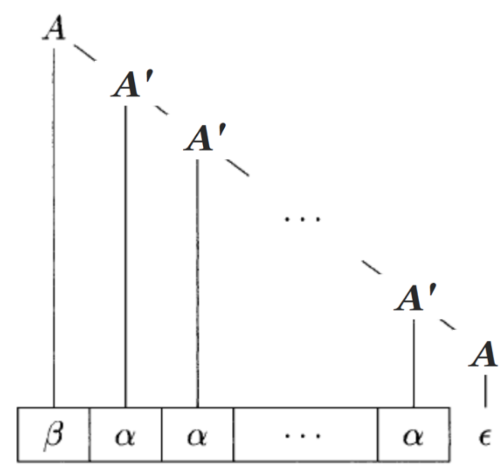
- A grammar is **left recursive** if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α
  - $S \rightarrow Aa \mid b$
  - $A \rightarrow Ac \mid Sd \mid \epsilon$
  - Because $S \Longrightarrow Aa \Longrightarrow Sda$
- **Immediate left recursion (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: $A \to A\alpha_1 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \dots \mid \beta_n$

- Replace the grammar by:
  - $A \to \beta_1 A' \mid \dots \mid \beta_n A'$
  - $A' \to \alpha_1 A' \mid \dots \mid \alpha_m A' \mid \epsilon$

Example

$$\boxed{E \rightarrow E + T \mid T}$$
$$\boxed{T \rightarrow T * F \mid F}$$
$$F \rightarrow (E) \mid \textbf{id}$$

$$\Longrightarrow$$

$$\boxed{\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +\ TE' \mid \epsilon \end{array}}$$
$$\boxed{\begin{array}{l} T \rightarrow FT' \\ T' \rightarrow *\ FT' \mid \epsilon \end{array}}$$
$$F \rightarrow (\ E\ ) \mid \textbf{id}$$

Left Factoring (提取左公因子)

- If we have the following two productions

$$\begin{aligned} stmt \rightarrow\ & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ |\ & \textbf{if } expr \textbf{ then } stmt \end{aligned}$$

- 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

$$\boxed{\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}}$$

Outline

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

SPL Grammar Rules

```
/* high-level definition */
Program -> ExtDefList
ExtDefList -> ExtDef ExtDefList
    | $
ExtDef -> Specifier ExtDecList SEMI
    | Specifier SEMI
    | Specifier FunDec CompSt
ExtDecList -> VarDec
    | VarDec COMMA ExtDecList
```

```
/* specifier */
Specifier -> TYPE
    | StructSpecifier
StructSpecifier -> STRUCT ID LC DefList RC
    | STRUCT ID
```

```
/* declarator */
VarDec -> ID
    | VarDec LB INT RB
FunDec -> ID LP VarList RP
    | ID LP RP
VarList -> ParamDec COMMA VarList
    | ParamDec
ParamDec -> Specifier VarDec
```

```
/* statement */
CompSt -> LC DefList StmtList RC
StmtList -> Stmt StmtList
    | $
Stmt -> Exp SEMI
    | CompSt
    | RETURN Exp SEMI
    | IF LP Exp RP Stmt
    | IF LP Exp RP Stmt ELSE Stmt
    | WHILE LP Exp RP Stmt
```

```
/* local definition */
DefList -> Def DefList
    | $
Def -> Specifier DecList SEMI
DecList -> Dec
    | Dec COMMA DecList
Dec -> VarDec
    | VarDec ASSIGN Exp
```

https://github.com/sqlab-sustech/CS323-2024F/blob/main/spl-spec/syntax.txt

```
/* Expression */
Exp -> Exp ASSIGN Exp
    | Exp AND Exp
    | Exp OR Exp
    | Exp LT Exp
    | Exp LE Exp
    | Exp GT Exp
    | Exp GE Exp
    | Exp NE Exp
    | Exp EQ Exp
    | Exp PLUS Exp
    | Exp MINUS Exp
    | Exp MUL Exp
    | Exp DIV Exp
    | LP Exp RP
    | MINUS Exp
    | NOT Exp
    | ID LP Args RP
    | ID LP RP
    | Exp LB Exp RB
    | Exp DOT ID
    | ID
    | INT
    | FLOAT
    | CHAR
Args -> Exp COMMA Args
    | Exp
```

Example

```
int test_1_r01(int a, int b)
{
  c = 'c';
  if (a > b)
  {
    return a;
  }
  else
  {
    return b;
  }
}
```

A syntactically valid program*

* Here, the vairable `c` is used without definition.
This error will be caught during semantic analysis.

The parse tree:

```
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        TYPE: int
      FunDec (1)
        ID: test_1_r01
        LP
        VarList (1)
          ParamDec (1)
            Specifier (1)
              TYPE: int
            VarDec (1)
              ID: a
          COMMA
          VarList (1)
            ParamDec (1)
              Specifier (1)
                TYPE: int
              VarDec (1)
                ID: b
        RP
      CompSt (2)
        LC
        StmtList (3)
          Stmt (3)
            Exp (3)
              Exp (3)
                ID: c
              ASSIGN
              Exp (3)
                CHAR: 'c'
            SEMI
          StmtList (4)
            Stmt (4)
              IF
              LP
              Exp (4)
                Exp (4)
                  ID: a
                GT
                Exp (4)
                  ID: b
              RP
              Stmt (5)
                CompSt (5)
                  LC
                  StmtList (6)
                    Stmt (6)
                      RETURN
                      Exp (6)
                        ID: a
                      SEMI
                  RC
              ELSE
              Stmt (9)
                CompSt (9)
                  LC
                  StmtList (10)
                    Stmt (10)
                      RETURN
                      Exp (10)
                        ID: b
                      SEMI
                  RC
        RC
```

Example

```
1 int test_1_r03()
2 {
3         int i = 0, j = 1;
4   float i = $;
5   if(i < 9.0){
6     return 1
7   }
8   return @;
9 }
```

```
Error type A at Line 4: unknown lexeme $
Error type B at Line 6: Missing semicolon ';'
Error type A at Line 8: unknown lexeme @
```