南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 6

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Grammar Design Issues
- Introduction to Bison

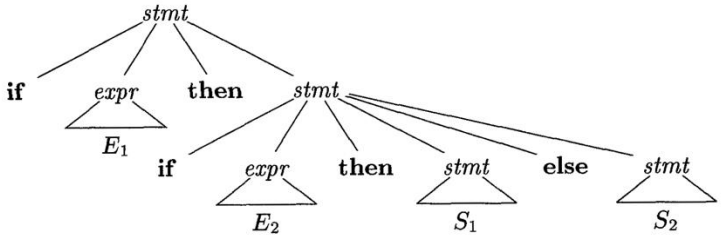
Grammar Design

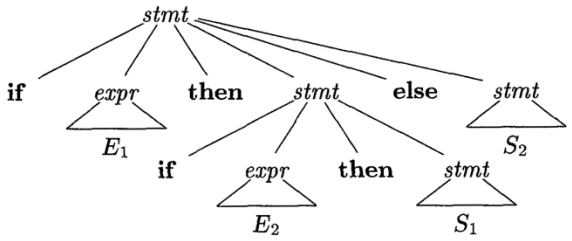
- CFGs are capable of describing most, but not all, of the syntax of programming languages
  - "Identifiers should be declared before use" cannot be described by a CFG
  - Subsequent phases must analyze the output of the parser to ensure compliance with such rules
- Before parsing, we typically apply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

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

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

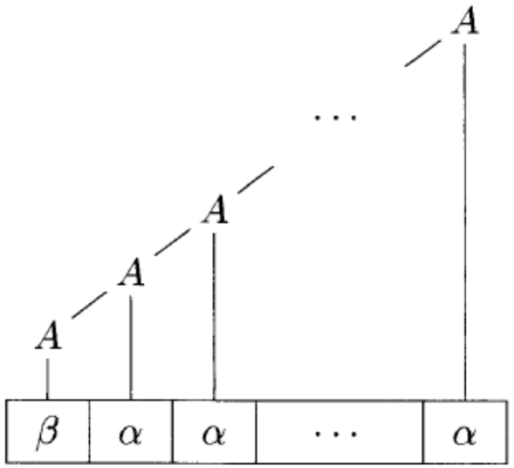
* open_stmt means the last then may not have matching else

Eliminating Left Recursion

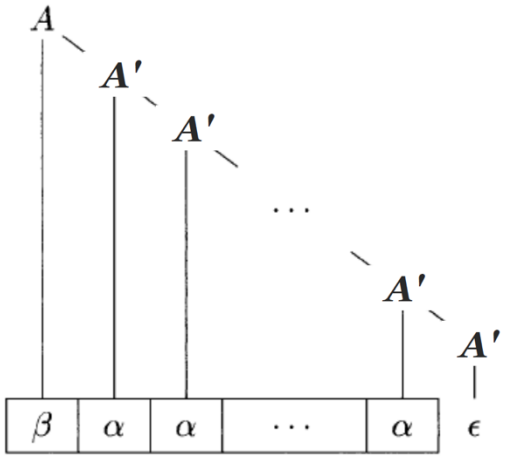
- A grammar is **left recursive** if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α
  - $S \rightarrow Aa \mid b$
  - $A \rightarrow Ac \mid Sd \mid \epsilon$
  - Because $S \Longrightarrow Aa \Longrightarrow Sda$
- **Immediate left recursion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



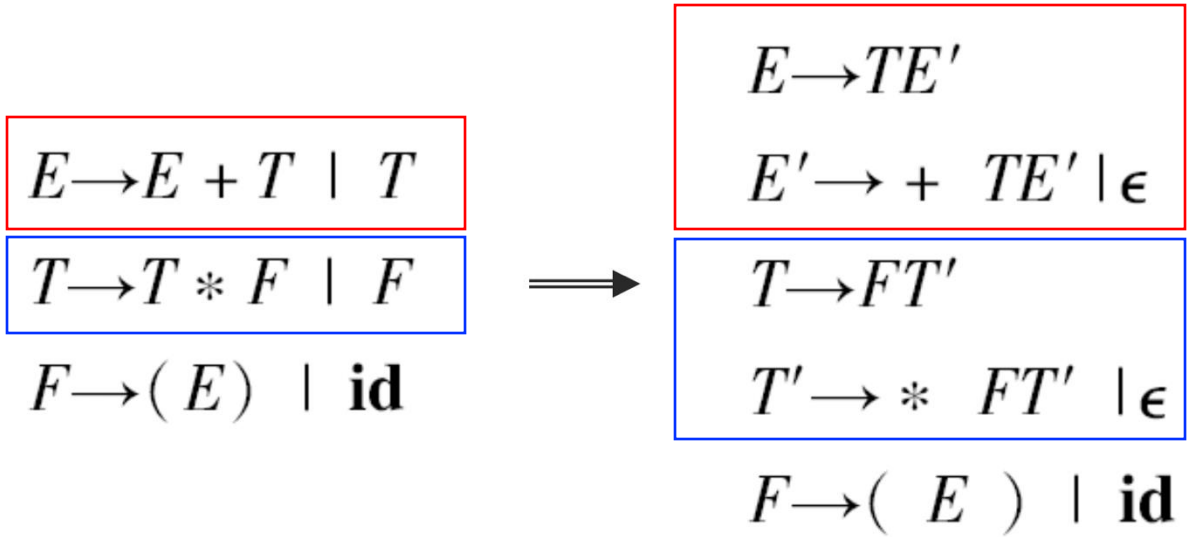
- Replace the grammar by:
  - $A \rightarrow \beta A'$
  - $A' \rightarrow \alpha A' \mid \epsilon$
  - It is right recursive now



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


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

Left Factoring (提取左公因子)

- If we have the following two productions

  $stmt \rightarrow$ **if** $expr$ **then** $stmt$ **else** $stmt$

  | **if** $expr$ **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

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

Outline

- Grammar Design Issues
- Introduction to Bison

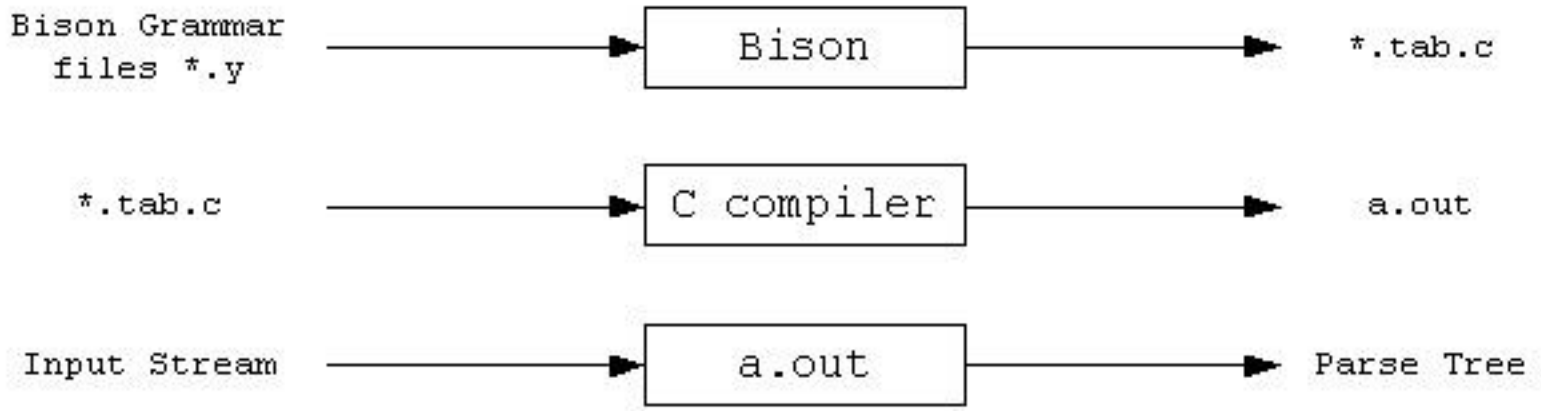
The Parser Generator Bison

- Bison generates a parser, which accepts the input token stream from Flex, to do syntax analysis, according to the specified CFG.

- Bison的前身为基于Unix的Yacc (Yet another compiler compiler)。Yacc的发布时间比Lex还要早，其采用的语法分析技术的理论基础早在20世纪50年代就已经由Knuth逐步建立了起来，而Yacc本身则是贝尔实验室的S.C. Johnson基于这些理论在1975年到1978年写成的。

- 到了1985年，当时在UC Berkeley的一个研究生Bob Corbett在BSD下重写了Yacc，取名为Bison (美洲野牛，yak牦牛的近亲)，后来GNU Project接管了这个项目，为其增加了许多新的特性，于是就有了我们今天所用的GNU Bison。

Input/Output of Bison (Yacc)



Structure of YACC Source Programs

- **Declarations (声明)**
  - Ordinary C declarations
  - Grammar tokens
- **Translation rules (翻译规则)**
  - Rule = a production + semantic action
- **Supporting C routines (辅助性C语言例程)**
  - Directly copied to `y.tab.c`
  - Can be invoked in the semantic actions
  - Other procedures such as error recovery routines may be provided

declarations

%%

translation rules

%%

supporting C routines

Translation Rules

$$
\begin{array}{lcll}
\langle \text{head} \rangle & : & \langle \text{body} \rangle_1 & \{\ \langle \text{semantic action} \rangle_1\ \} \\
& | & \langle \text{body} \rangle_2 & \{\ \langle \text{semantic action} \rangle_2\ \} \\
& & \cdots & \\
& | & \langle \text{body} \rangle_n & \{\ \langle \text{semantic action} \rangle_n\ \} \\
& ; & &
\end{array}
$$

DO NOT miss it

- The first head in the list of rules is taken as the start symbol of the grammar
- A semantic action is a sequence of C statements
  - `$$` is a Bison's internal variable that holds the semantic value of the left-hand side of a production rule (i.e., the whole construct)
  - `$i` holds the semantic value of *i*th grammar symbol of the body
- A semantic action is performed when we apply the associated production for reduction (归约，the reverse of rewrite)
  - Most of the time, the purpose of an action is to compute the semantic value of the whole construct from the semantic values of its parts, i.e., compute `$$` using `$i`'s

Example: A Parser for Calculation

- When processing arithmetic expressions, lexical anlayzer will recognize the following types of tokens: INT, ADD, SUB, MUL, DIV
  - When analyzing the input string "3 + 6 / 2", the lexer will output this string of tokens: INT ADD INT DIV INT
- After lexical analysis, the parser will check if the string of tokens produced by lexer has a valid structure or not according to the synatatic specification described by the following context-free grammar

```
Calc -> Exp
Exp -> Factor | Exp ADD Factor | Exp SUB Factor
Factor -> Term | Factor MUL Term | Factor DIV Term
Term -> INT
```

*Red for non-terminals, Blue for terminals, Calc is the start symbol

Valid and Invalid Inputs

Valid input expressions:

- 3
- 3 + 5 * 4
- 3 + 6 / 2
- 3 + 2 – 1
- …

Invalid input expressions:

- -3
- 3 + 5 *
- …

Flex/Bison Code for Calculator

lex.l

```
%{
    #include "syntax.tab.h"
    #include "stdlib.h"
%}
%%
[0-9]+ { yylval = atoi(yytext); return INT; }
"+" { return ADD; }
"-" { return SUB; }
"*" { return MUL; }
"/" { return DIV; }
[ \n\r\t] {}
. { fprintf(stderr, "unknown symbol: %s\n", yytext);
    exit(1); }
```

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse(); // will invoke yylex()
}
```

yylval:

- Flex internal variable that is used to store the attribute of a recognized token
- Its data type is YYSTYPE (int by default)*
- After storing values to `yylval` in Flex code, the values will be propagated to Bison (i.e., the syntax anlayzer part) and can be retrieved using `$n`

*Can be customized by putting command like `#define YYSTYPE char*` at the beginning of .l and .y files.

A Further Look at Semantic Actions

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

When appyling this rule for reduction, the semantic analysis is successful (i.e., the start symbol calc can generate the input expression), we output the calculation result.

A Further Look at Semantic Actions

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Intermediate calculation of semantic values of a language construct represented by the head symbol

A Further Look at Semantic Actions

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Although no action is specified, Bison will still propagate the attribute of the only symbol `INT` to the head `Term`

Same as writing: `{$$ = $1;}`

Illustrative Example

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Input: 3 + 5
Token string: INT ADD INT

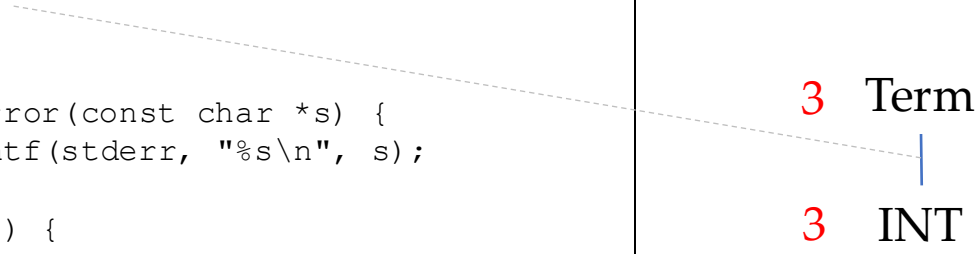
3 INT

Illustrative Example

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Input: 3 + 5
Token string: INT ADD INT

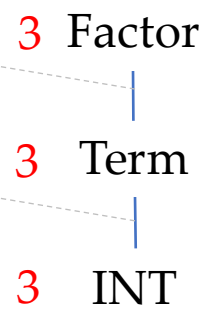


Illustrative Example

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Input: 3 + 5
Token string: INT ADD INT

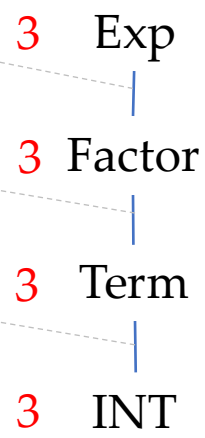


Illustrative Example

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Input: 3 + 5
Token string: INT ADD INT

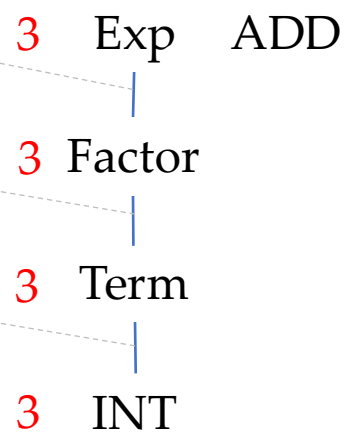


Illustrative Example

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Input: 3 + 5
Token string: INT ADD INT

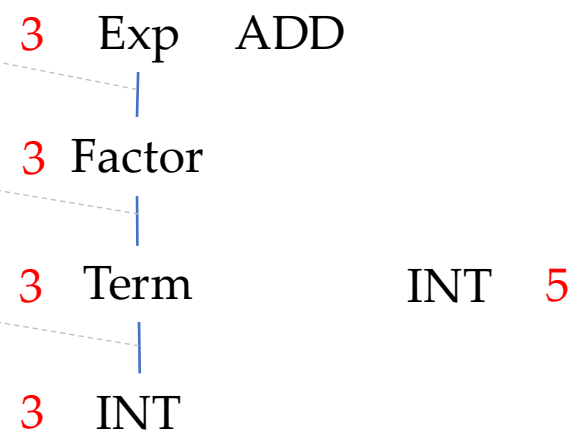


Illustrative Example

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Input: 3 + 5
Token string: INT ADD INT

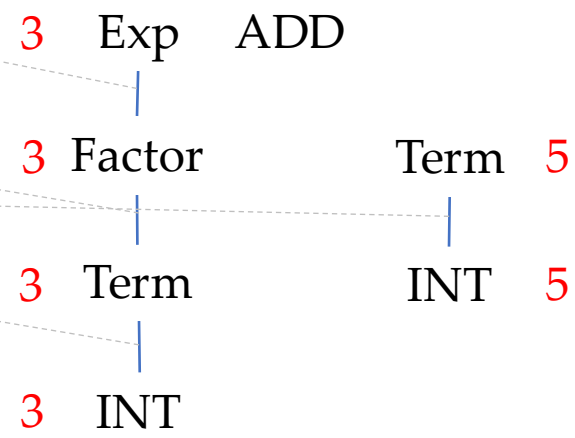


Illustrative Example

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Input: 3 + 5
Token string: INT ADD INT

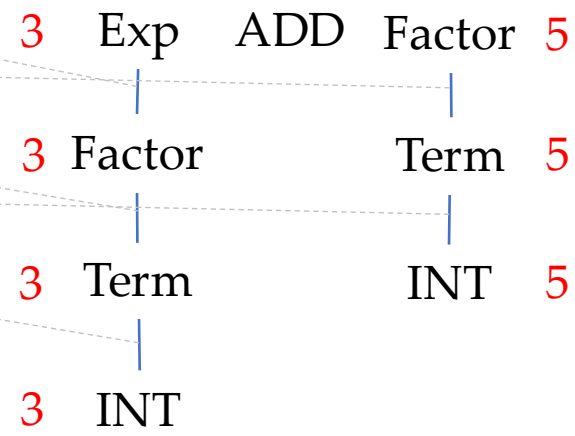


Illustrative Example

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Input: 3 + 5
Token string: INT ADD INT

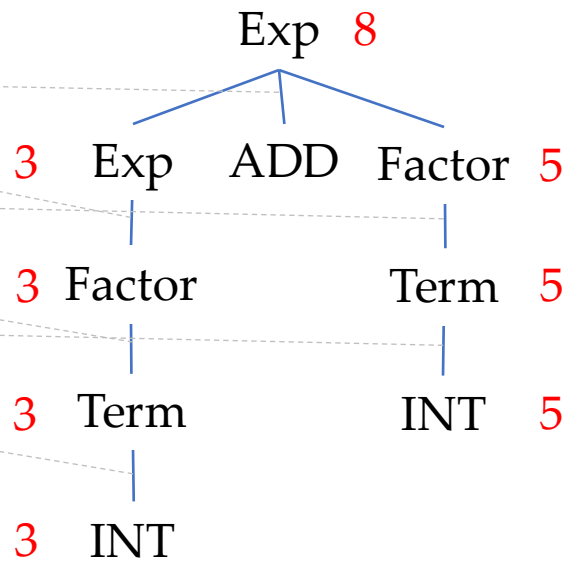


Illustrative Example

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Input: 3 + 5
Token string: INT ADD INT

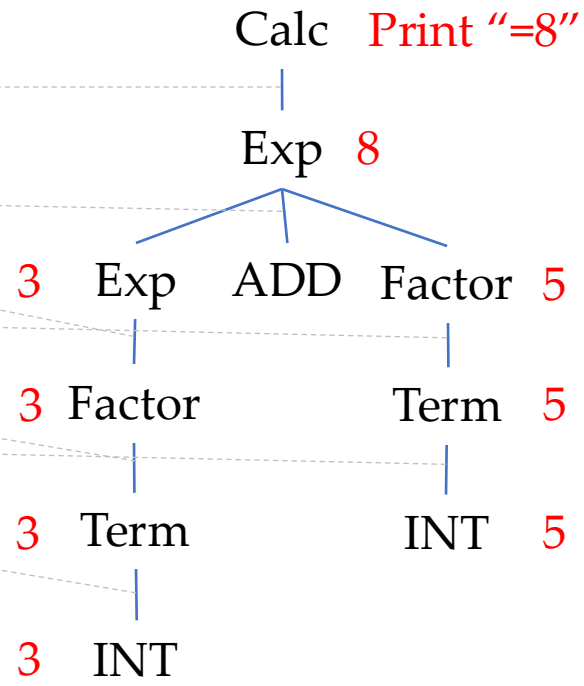


Illustrative Example

syntax.y

```
%{
    #include "lex.yy.c"
    void yyerror(const char*);
%}
%token INT
%token ADD SUB MUL DIV
%%
Calc: /* to allow empty input */
    | Exp { printf("= %d\n", $1); }
    ;
Exp: Factor
    | Exp ADD Factor { $$ = $1 + $3; }
    | Exp SUB Factor { $$ = $1 - $3; }
    ;
Factor: Term
    | Factor MUL Term { $$ = $1 * $3; }
    | Factor DIV Term { $$ = $1 / $3; }
Term: INT
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}
int main() {
    yyparse();
}
```

Input: 3 + 5
Token string: INT ADD INT



Compile the Example

- Compile the Bison code into C code: a header file and an implementation file
  - `bison -d syntax.y` (why it is compilable without lex.yy.c?)
  - The command produces `syntax.tab.h` and `syntax.tab.c`
- Compile the flex source code
  - `flex lex.l`
  - The command produce `lex.yy.c`
- Putting things together
  - `gcc syntax.tab.c -lfl -ly -o calc.out`
  - The options `-lfl` and `-ly` tell gcc to include Flex and Bison libraries (in some environments, the options may not be needed)

Run the Calculator

- In the runs below, we use pipes* to pass the output of the first command to be the input of the second command

```
cs323@deb-cs323-compilers:~/Desktop/lab6$ echo "3" | ./calc.out
= 3
cs323@deb-cs323-compilers:~/Desktop/lab6$ echo "3+5*4" | ./calc.out
= 23
cs323@deb-cs323-compilers:~/Desktop/lab6$ echo "3+6/2" | ./calc.out
= 6
cs323@deb-cs323-compilers:~/Desktop/lab6$ echo "3+ 2-1" | ./calc.out
= 4
cs323@deb-cs323-compilers:~/Desktop/lab6$ echo "-3" | ./calc.out
syntax error
cs323@deb-cs323-compilers:~/Desktop/lab6$ echo "3+5*" | ./calc.out
syntax error
```

* https://linuxhint.com/what-is-pipe-in-linux/