南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 6

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Bison introduction
- Bison tutorial

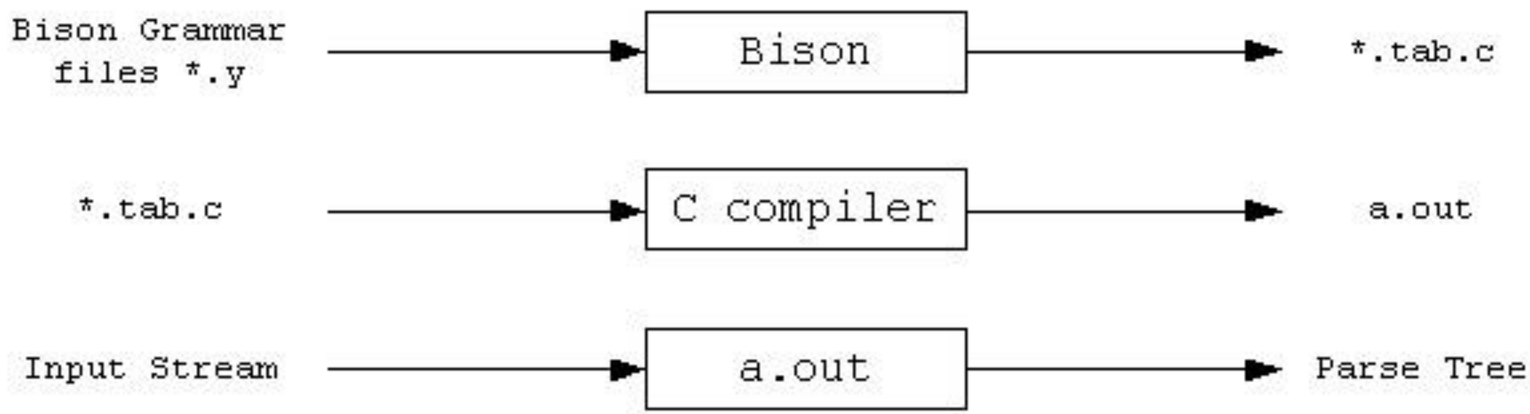
The Parser Generator Bison

- 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

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

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

Outline

- Bison introduction
- Bison tutorial

A Parser That Performs 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

3 Term

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

3 Factor

3 Term

3 INT

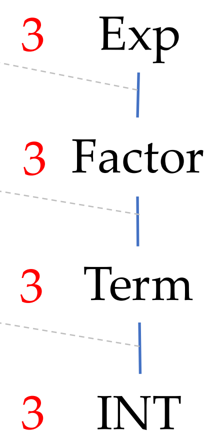
Illustrative Example

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

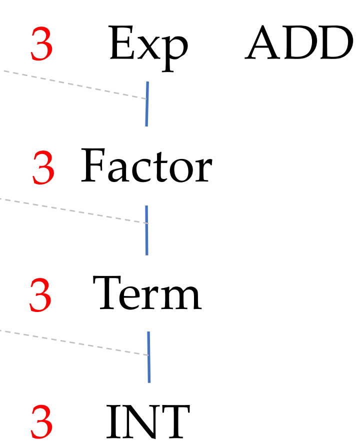


Illustrative Example

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

3 Exp ADD

3 Factor

3 Term INT 5

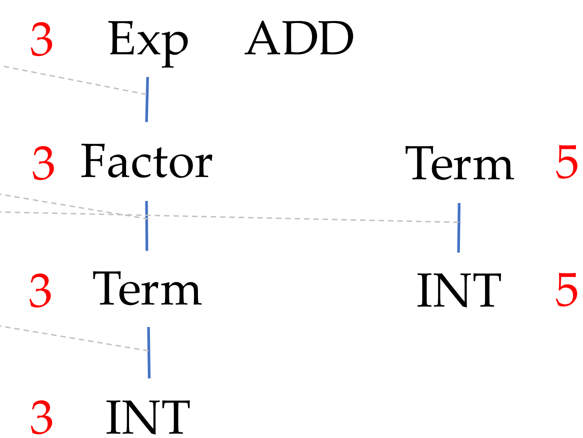
3 INT

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



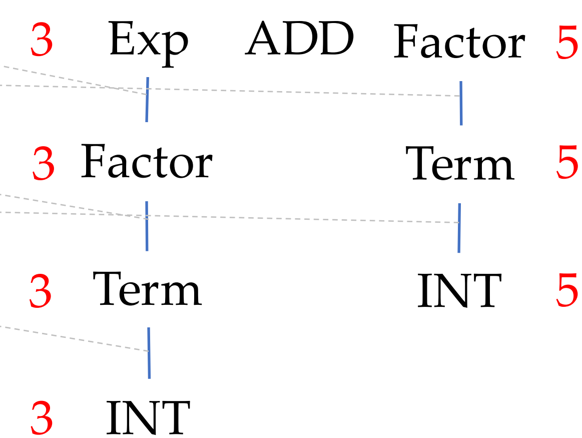
Illustrative Example

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



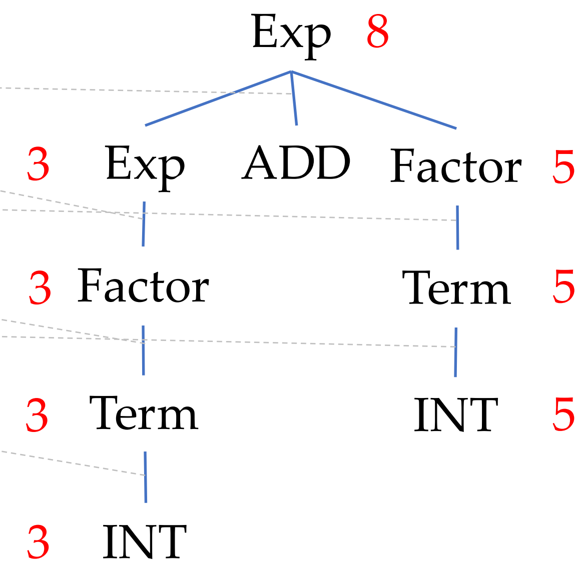
Illustrative Example

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



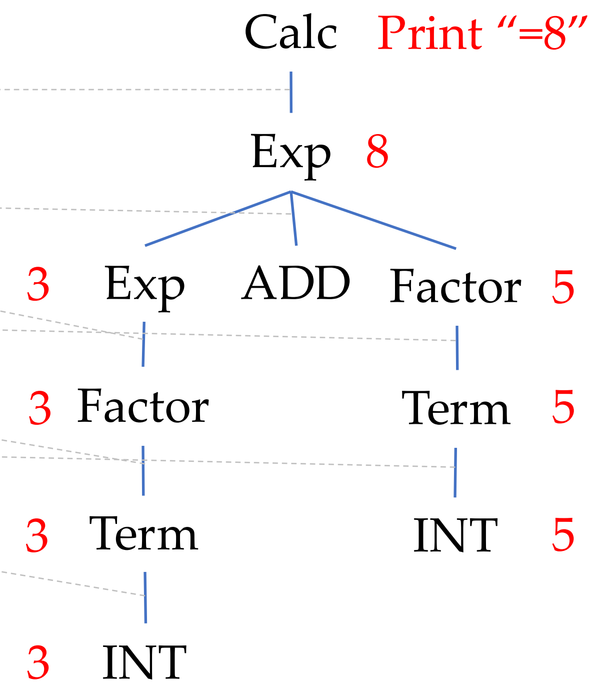
Illustrative Example

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



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 <u>`syntax.tab.h`</u> and <u>`syntax.tab.c`</u>
- Compile the flex source code
  - `flex lex.l`
  - The command produce <u>`lex.yy.c`</u>
- 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

The code is available at `lab5/calc`. We provide a build target calc to facilitate the compilation.

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

```
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab4/calc$ echo "3" | ./calc.out
= 3
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab4/calc$ echo "3+5*4" | ./calc.out
= 23
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab4/calc$ echo "3+6/2" | ./calc.out
= 6
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab4/calc$ echo "3+ 2-1" | ./calc.out
= 4
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab4/calc$ echo "-3" | ./calc.out
syntax error
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab4/calc$ echo "3+5*" | ./calc.out
syntax error
```

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

Project Reminder

- Please start to design and implement your language if you haven’t done so.
  - It is also fine if you choose to build a compiler for SPL or its simple variants.

- Milestone check: Nov. 18, during the lab session.
  - Please prepare test cases by yourself for the demo.
  - You should also prepare a report, which should at least contain: 1) the specification and core features of your language, 2) the design of your compiler, 3) the implementation progress.