南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 6

Yepang Liu

liuyp1@sustech.edu.cn

Outline

Bison tutorials:

- Conflict resolution
- Error recovery

Conflicts During Shift-Reduce Parsing

- There exist some grammars (e.g., ambiguous ones) for which shift-reduce parsers will encounter conflicts during parsing:
    - shift/reduce conflicts, 移入/归约冲突
    - reduce/reduce conflicts, 归约/归约冲突

Shift/Reduce Conflict Example

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

STACK	INPUT
$\cdots$ **if** *expr* **then** *stmt*	**else** $\cdots \$$

Reduce or shift? What if there is a *stmt* after **else**?

Reduce/Reduce Conflict Example

- Parsing input **id(id, id)**

STACK	INPUT
$id(id	**, id$**

(1) $stmt \rightarrow \textbf{id}\ (\ parameter\_list\ )$

(2) $stmt \rightarrow expr\ := expr$

(3) $parameter\_list \rightarrow parameter\_list\ ,\ parameter$

(4) $parameter\_list \rightarrow parameter$

(5) $\boxed{parameter \rightarrow \textbf{id}}$

(6) $expr \rightarrow \textbf{id}\ (\ expr\_list\ )$

(7) $\boxed{expr \rightarrow \textbf{id}}$

(8) $expr\_list \rightarrow expr\_list\ ,\ expr$

(9) $expr\_list \rightarrow expr$

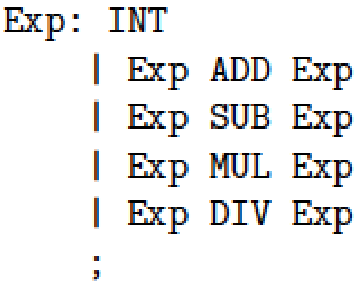
Reduce by which production?

How does Bison deal with conflicts?

- The default strategy:
  - For a shift/reduce conflict, always choose to shift
  - For a reduce/reduce conflict, reduce with the rule declared first

It is not recommended to adopt the default strategy.

Example



```
Exp: INT
    | Exp ADD Exp
    | Exp SUB Exp
    | Exp MUL Exp
    | Exp DIV Exp
    ;
```

- When we compile the above grammar, Bison will report a shift/reduce conflict

- Consider input string 3 * 4 + 5
  - During shift-reduce parsing, when we see “3 * 4” on stack[1] and the next symbol in the input is +, shall we reduce “3*4” or shift +?

- If we follow Bison’s default strategy, we will shift
  - After shifting “+5”, “4 + 5” will be reduced and the expression will evaluate to 27

[1] Here, we use lexemes instead of tokens for ease of understanding

Addressing Conflicts

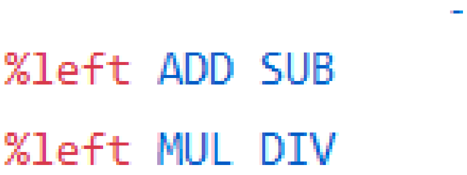
- Possible solution: rewriting grammar

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

- However, rewriting grammars is hard and can lead to less understandable productions; Sometimes, it is convenient to use ambiguous grammars.

Addressing Conflicts

- More practical solution: use **precedence** and **associativity**

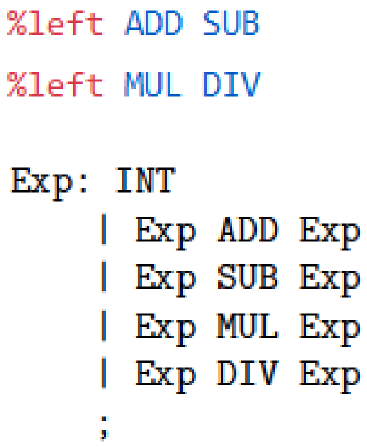


```
%left ADD SUB
%left MUL DIV
```

Token defined in front has lower precedence.

`%left`, `%right` and `%nonassoc` define associativity.

Addressing Conflicts



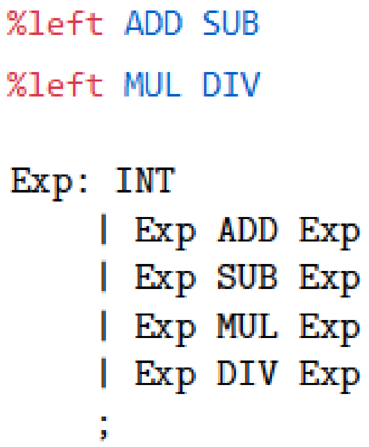
```
%left ADD SUB
%left MUL DIV

Exp: INT
        | Exp ADD Exp
        | Exp SUB Exp
        | Exp MUL Exp
        | Exp DIV Exp
        ;
```

- Handling input string 3 * 4 + 5
  - When 3 * 4 is on stack and + is the next symbol, we choose to reduce because Exp -> Exp MUL Exp[1] has a higher precedence than that of the token ADD

[1] The precedence of a rule by default is determined by the precedence of the rightmost terminal of the production body.

Addressing Conflicts



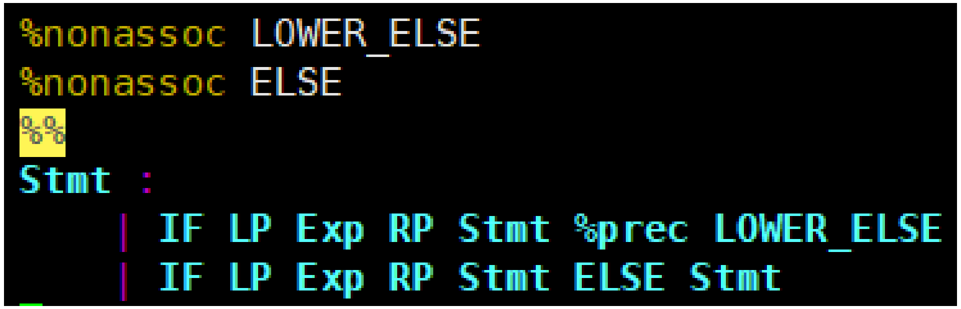
```
%left ADD SUB
%left MUL DIV

Exp: INT
    | Exp ADD Exp
    | Exp SUB Exp
    | Exp MUL Exp
    | Exp DIV Exp
    ;
```

- Handling input string 3 + 4 + 5
  - When 3 + 4 is on stack and + is the next symbol, we choose to reduce: we have a tie by only looking at the precedence, but the associativity of the token ADD helps break the tie

Addressing Conflicts

- We can also use `%prec` directive to define precedence



```
%nonassoc LOWER_ELSE
%nonassoc ELSE
%%
Stmt :
      | IF LP Exp RP Stmt %prec LOWER_ELSE
      | IF LP Exp RP Stmt ELSE Stmt
```

When the parser sees **if (exp) stmt** on stack and the next input symbol is **else**, it will choose to shift since the **else** token has a higher precedence than the first production

Exercise 1

Tip: the command "`bison -d syntax.y --report all`" will generate a file `syntax.output` containing all details about the automaton (for parsing) and conflicts.

- Given the following grammar:

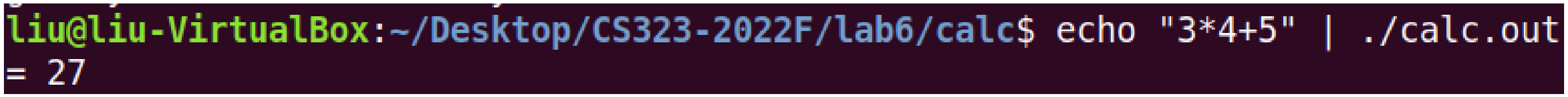
```
Calc -> Exp
Exp -> INT | LP Exp RP | Exp ADD Exp | Exp SUB Exp | Exp MUL Exp
         | Exp DIV Exp
```

- Write a program using Flex and Bison to evaluate the arithmetic expressions in the above grammar.
  - Use precedence and associativity directives to resolve conflicts
  - Think about why the following grammar has no conflicts

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

Instructions

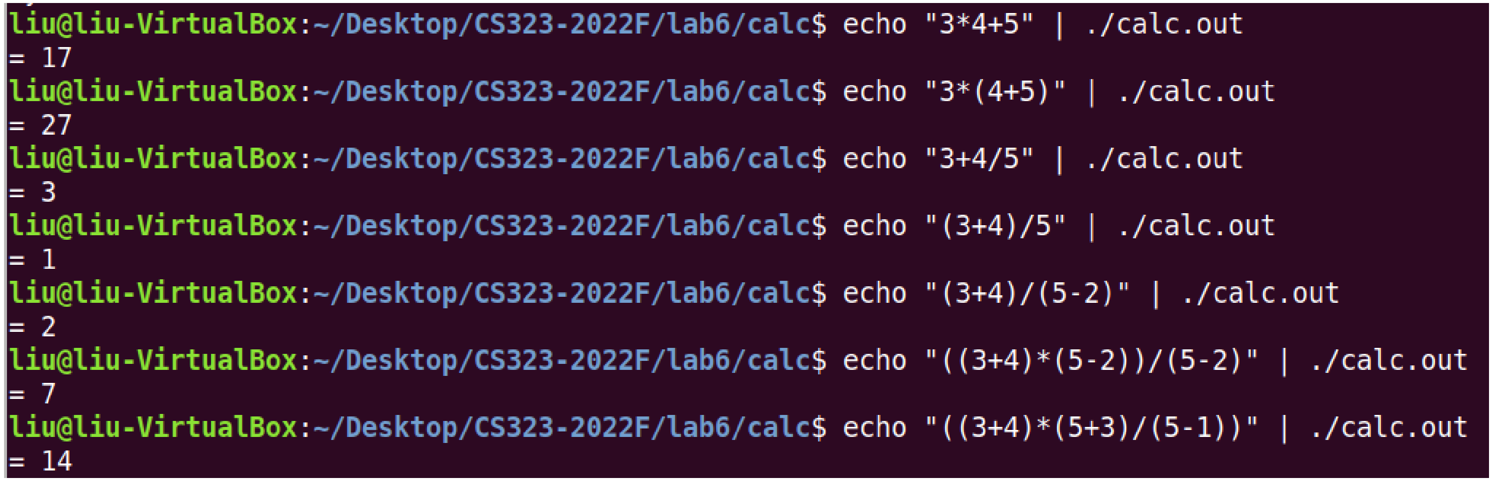
- Clone `lab6/calc` from our GitHub repo
- Run "`make calc`" to build the runnable `calc.out` (Observe that Bison will print "16 shift/reduce conflicts")
- Try to understand the conflicts (use the test case below)
  - Run "`bison -d syntax.y --report all`" to check the details about the conflicts



```
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab6/calc$ echo "3*4+5" | ./calc.out
= 27
```

Instructions cont.

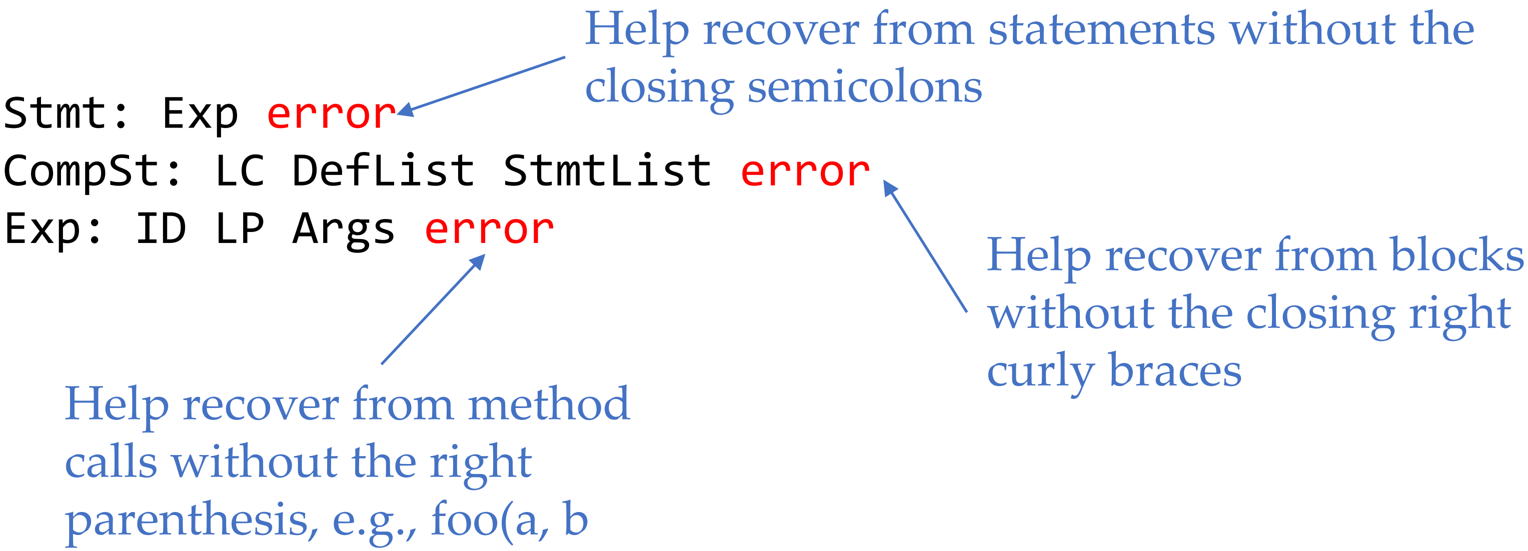
- Read the provided `syntax.y` file and try to modify it to resolve all conflicts
- After resolving the conflicts, make sure your calculator program can pass the following tests



```
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab6/calc$ echo "3*4+5" | ./calc.out
= 17
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab6/calc$ echo "3*(4+5)" | ./calc.out
= 27
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab6/calc$ echo "3+4/5" | ./calc.out
= 3
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab6/calc$ echo "(3+4)/5" | ./calc.out
= 1
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab6/calc$ echo "(3+4)/(5-2)" | ./calc.out
= 2
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab6/calc$ echo "((3+4)*(5-2))/(5-2)" | ./calc.out
= 7
liu@liu-VirtualBox:~/Desktop/CS323-2022F/lab6/calc$ echo "((3+4)*(5+3)/(5-1))" | ./calc.out
= 14
```

Error Recovery

- Bison supports an error recovery mode called *panic-mode error recovery*
- The special `error` token helps recovery



```
Stmt: Exp error
CompSt: LC DefList StmtList error
Exp: ID LP Args error
```

Help recover from statements without the closing semicolons

Help recover from blocks without the closing right curly braces

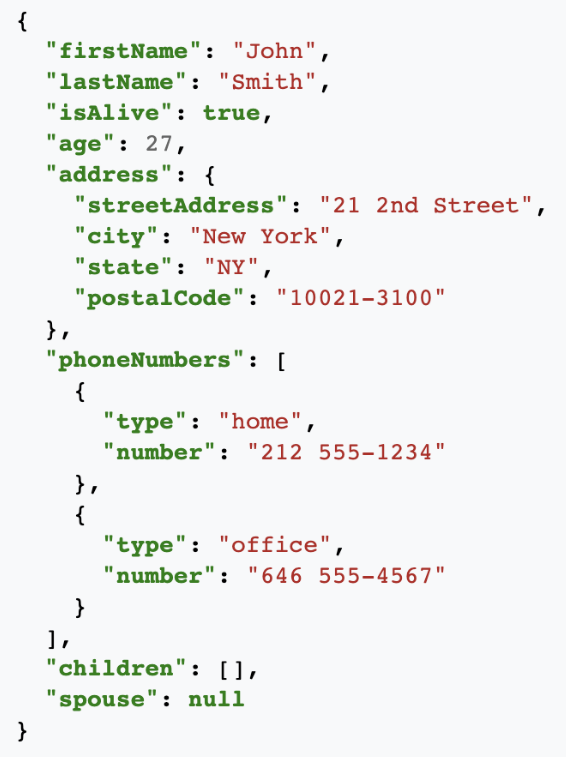
Help recover from method calls without the right parenthesis, e.g., foo(a, b

Error Recovery Routine

- When Bison-generated parsers encounter a syntax error, the following routine will be triggered:
  - Invoke `yyerror` function (you may overwrite)
  - Pop all un-reduced tokens from stack top, until the `error` token can be shifted
  - Shift `error` token to the stack
  - Pop tokens from the stack until the parser finds one on which the normal parsing can resume (discarding and re-synchronization)
  - `yyparse` resumes parsing in provisional mode (will switch to normal mode after successfully shifting three tokens)

Exercise 2

- Write a json parser with error recovery capability



```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

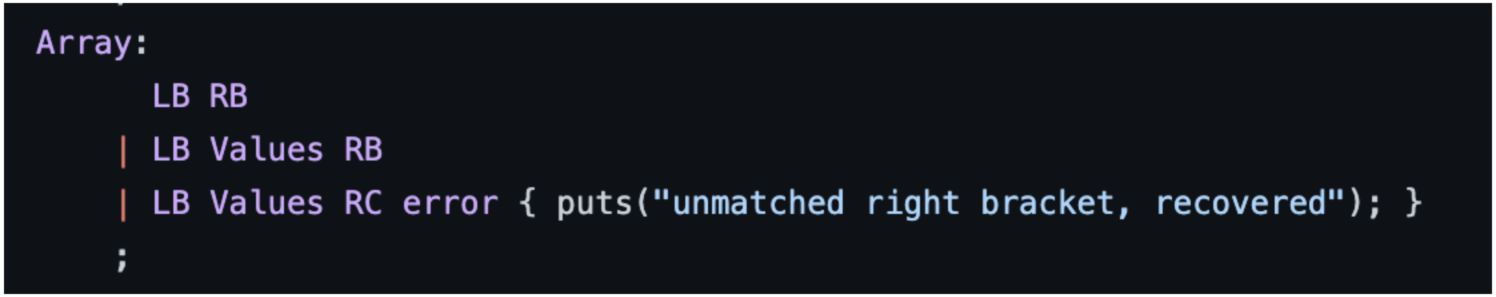
A well-formed json file

```
{"Extra value after close": true} "misplaced quoted value"
```

A malformed json file

Instructions

- Clone `lab6/jp` from our GitHub repository
- Your job is to modify the given `syntax.y` to recognize all syntax errors in our provided malformed json files (under `data/jsonchecker/`)
- Below is an example to recognize "unmatched right bracket", e.g., ["mismatch"}



```
Array:
        LB RB
      | LB Values RB
      | LB Values RC error { puts("unmatched right bracket, recovered"); }
      ;
```

Instructions cont.

- Build the executable parse with the command “`make jp`”
- Run test cases with “`python3 jsonparser_test.py`”
- You are done if the python script prints “Recovered/Total: 15/15”