EECS483 D3: Project 2 Overview

Chun-Hung Hsiao

Jan 25, 2013

Announcements

- Project 1 is due at 11:59pm today
  - Submit your code earlier
  - No Michigan time!
- Homework 1 is due on next Monday
  - Hand in your answer sheet with the cover page in class
- Project 2 has will be released on CTools today
  - Due on Monday, Feb 11
  - Submission open on tomorrow

Project 1 FAQ (1/2)

- How to count the width of a TAB character?
  - You can do whatever you like (as long as you count it as a small positive number). The grader will use a special program to judge your TAB column calculation
- I redirected my output to a file, but the error messages still go to the screen
  - You need to redirect the standard output stream and the standard error stream at the same time:

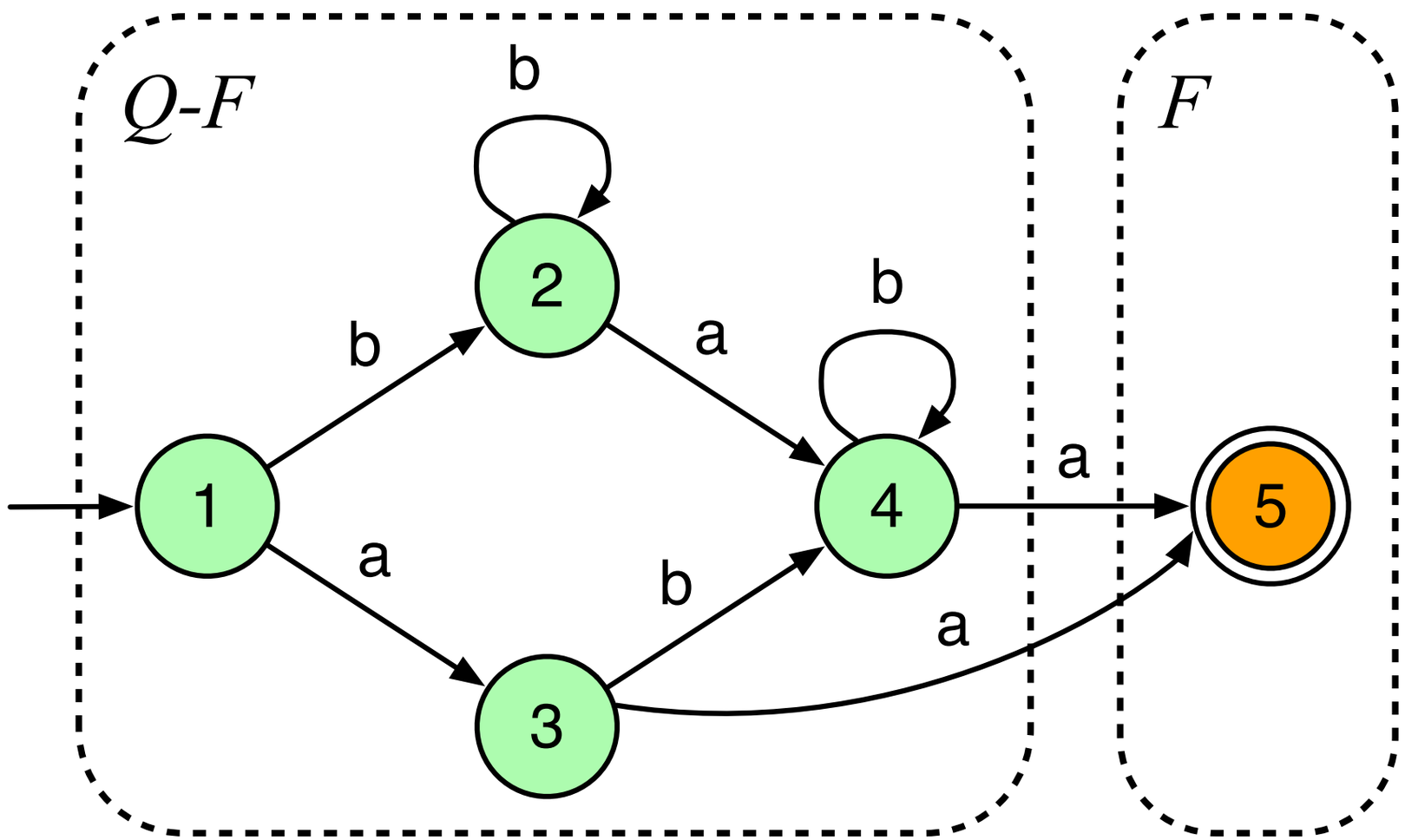
```
./dcc < input_file >& output_file
```

Project 1 FAQ (2/2)

- I got a feedback saying “can’t build submission,” why?
  - The folder you passed to the submission script should be the folder that contains the Makefile and other stuff
  - I’ve updated the script so it will run a building test before submitting
- The submission script told me that I have an unsuccessful submission, but I still got the feedback
  - The file system on CAEN sometimes disconnects temporarily and thus makes your submissions be partially uploaded
  - I’ve updated the script to reduce the chance but contact me if you encounter any problem

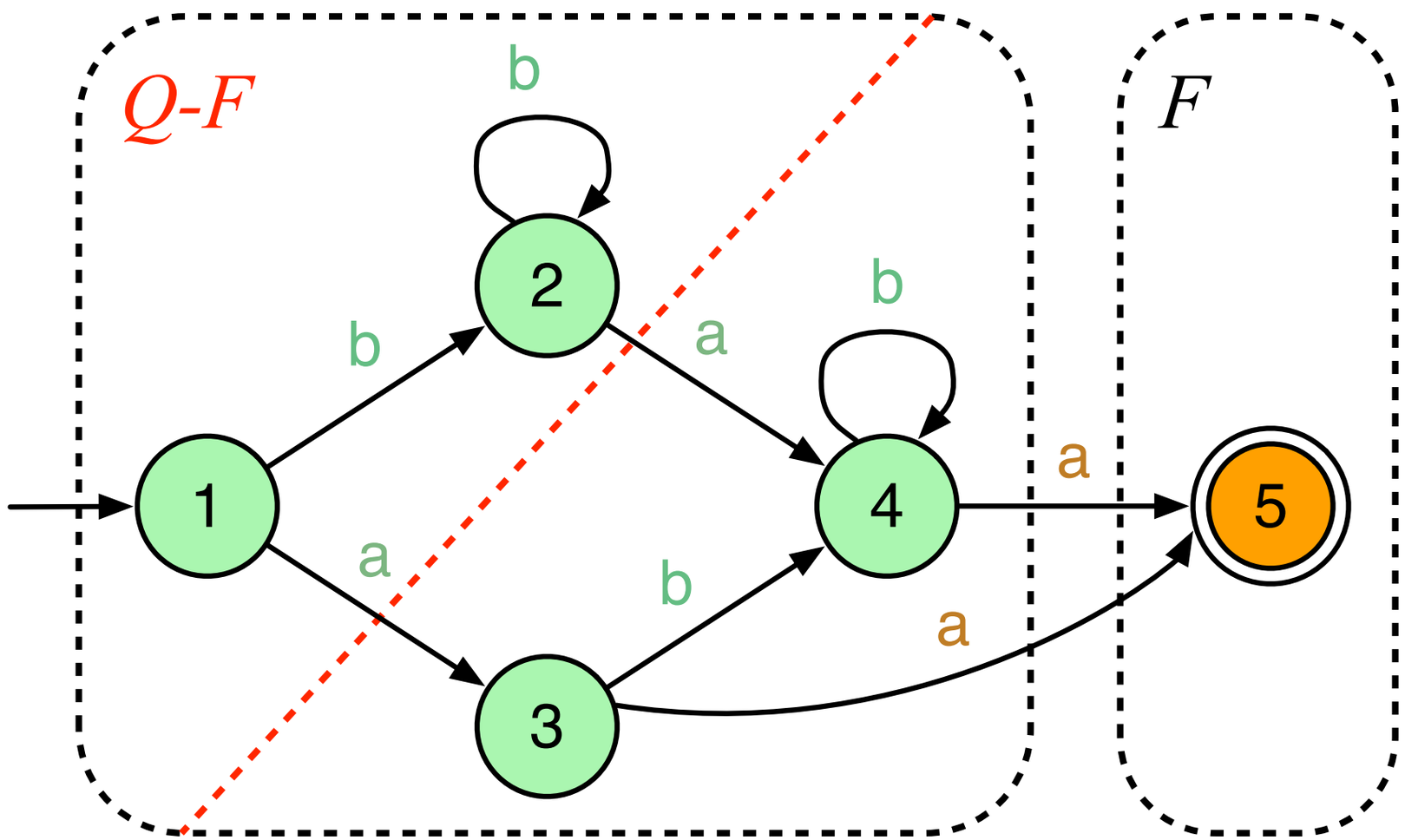
Exercise: DFA Minimization (1/4)

- Remove dead and unreachable states first
- Initial partition: $\{Q\text{-}F, F\}$



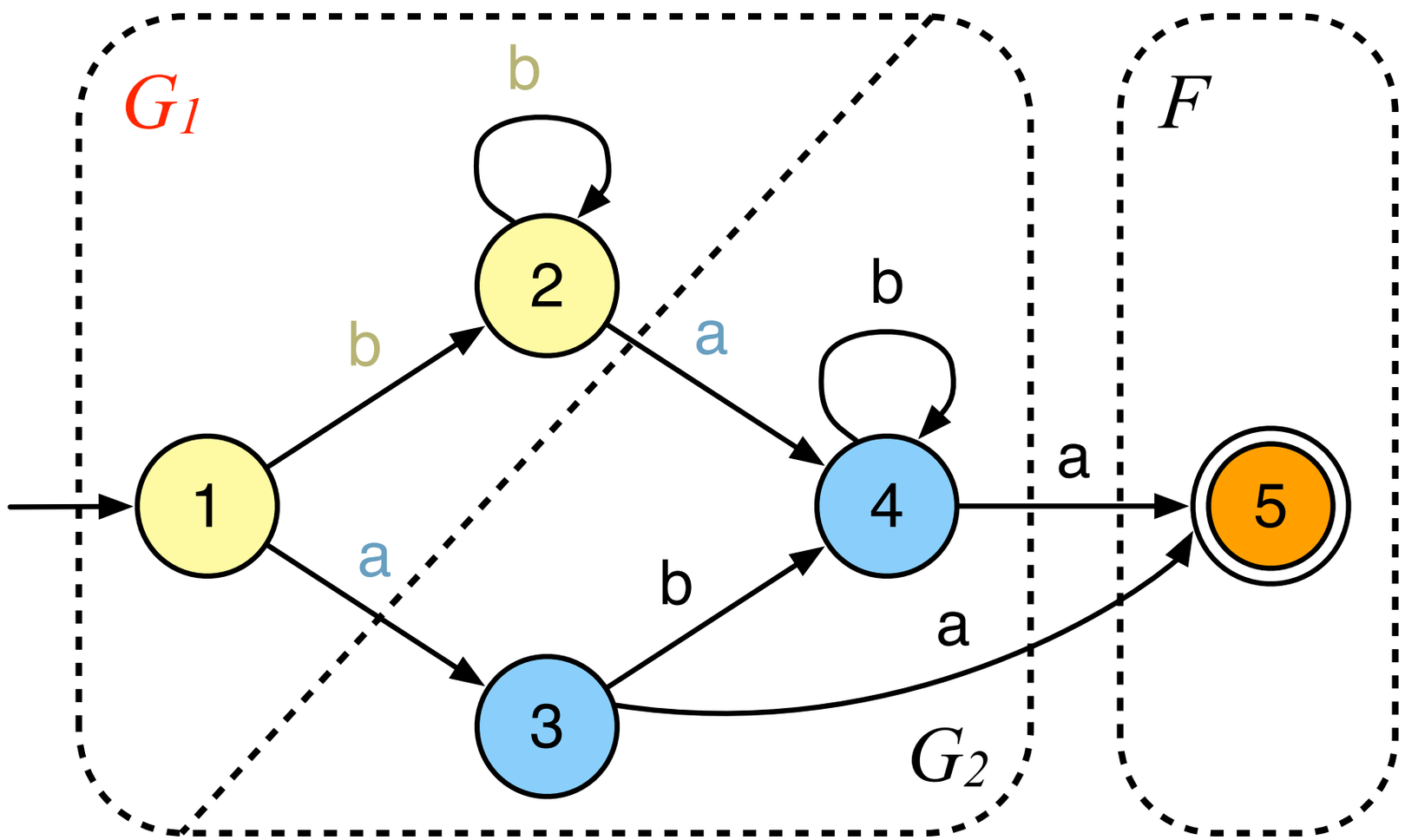
Exercise: DFA Minimization (2/4)

- Iteratively refine each group in the partition by finding distinguishable transitions

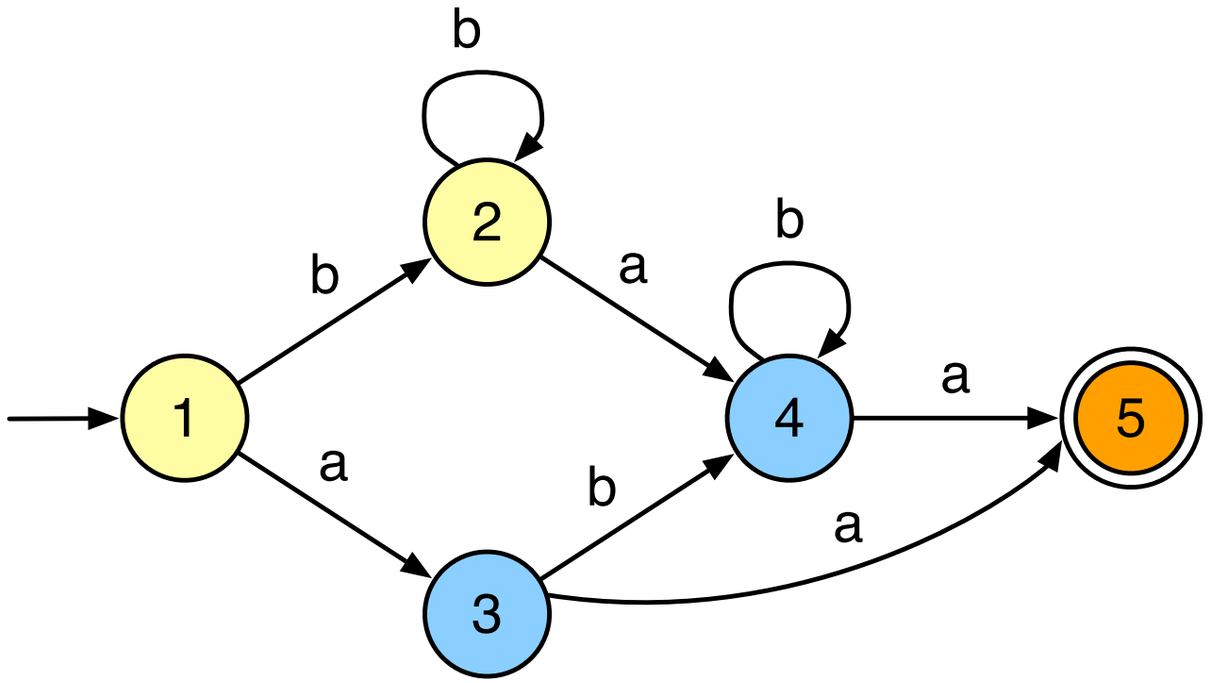


Exercise: DFA Minimization (3/4)

- Continuously examine a newly generated group to see if it needs to be further split



Exercise: DFA Minimization (4/4)



Group being examined	Transition table of each state in the group	New partition
-	-	{1,2,3,4}, {5}
{1,2,3,4}	1: a → {1,2,3,4}, b → {1,2,3,4} 2: a → {1,2,3,4}, b → {1,2,3,4} 3: a → {5}, b → {1,2,3,4} 4: a → {5}, b → {1,2,3,4}	{1,2}, {3,4}, {5}
{5}	5: a → {}, b → {}	{1,2}, {3,4}, {5}

Project 2: Overview

- Now you have a scanner, the next step is to do the syntax analysis based on Decaf's grammar
- You will learn to use the Yacc/Bison parser generator to build a parser for Decaf
  - Just like using lex/flex to generate a scanner
- Build and print the abstract syntax tree if the input program is good; report syntax errors otherwise
  - The abstract syntax tree will then be used in the semantic analysis phase in later projects

Syntax Analysis

- Given the token sequence, now we want to recognize expressions and statements
- Let's start with an English example:

  `Alice ate apples.`

  – After lexical analysis, we have got a sequence of tokens:

    N(`Alice`) V(`ate`) N(`apples`) Period(`.`)

  – The goal of syntax analysis is to know it's a statement:

    S(`Alice`) V(`ate`) O(`apples`)

- In programming language, we want to know how the code can be derived from the grammar
  – By knowing which rules we are using, we can get the meaning of the code

Yacc and Bison

- Yacc - Yet Another Compiler Compiler
  - Using a (Yacc) compiler to generate another compiler!
- Automatically generates an LALR parser given the CFG of a language
  - Requires a scanner as its front-end to recognize the terminals
- Bison: a Yacc-compatible GNU replacement
  - Just like Flex to Lex
  - In addition to LALR parser, it can also generate more kinds of LR parsers

Yacc Syntax

- Structure of a .y file:

  DECLARATIONS
  %%
  RULES
  %%
  USERCODE

Yacc Example: In-fix Calculator v1 (1/2)

Adapted from http://www.gnu.org/software/bison/manual/html_node/Infix-Calc.html#Infix-Calc

```
%{
#define YYSTYPE int
#include <math.h>
#include <stdio.h>
int yylex();
%}

/* Bison declarations.  */
%token NUM

%% /* The grammar follows.  */

input:
    /* empty */
  | '\n' input
  | exp '\n' { printf("%d\n", $1); } input
  ;
```

```
exp:
    NUM             { $$ = $1;          }
  | exp '+' exp     { $$ = $1 + $3;     }
  | exp '-' exp     { $$ = $1 - $3;     }
  | exp '*' exp     { $$ = $1 * $3;     }
  | exp '/' exp     { $$ = $1 / $3;     }
  | '-' exp         { $$ = -$2;         }
  | exp '^' exp     { $$ = pow($1, $3); }
  | '(' exp ')'     { $$ = $2;          }
  ;

%%

int main() {
    return yyparse();
}
```

`$n` : `yylval` of the n^{th} symbol	`@n` : `yylloc` of the n^{th} symbol
`$$` : `yylval` for the resulting symbol	`@$` : `yylloc` of the resulting symbol

Yacc Example: In-fix Calculator v1 (2/2)

- Header file generation:
  - bison -d calc.y
- Compilation & execution:
  - flex calc.l
  - bison calc.y
  - gcc lex.yy.c calc.tab.c -ll -ly
- 30 shift/reduce conflicts!
  - Need to resolve ambiguities
- 1+2*3 is 7 but 2*3+1 is 8?
  - Introduce new nonterminals to implement operator precedence

Yacc Example: In-fix Calculator v2 (1/2)

Adapted from http://www.gnu.org/software/bison/manual/html_node/Infix-Calc.html#Infix-Calc

```
/* Bison declarations.  */
%token NUM

%% /* The grammar follows.  */

input:
    /* empty */
  | '\n' input
  | exp '\n' { printf("%d\n", $1); } input
  ;

exp:
    term           { $$ = $1;          }
  | exp '+' exp    { $$ = $1 + $3;     }
  | exp '-' exp    { $$ = $1 - $3;     }
  ;

term:
    fact           { $$ = $1;          }
  | term '*' term  { $$ = $1 * $3;     }
  | term '/' term  { $$ = $1 / $3;     }
  ;

fact:
    pow            { $$ = $1;          }
  | '-' fact       { $$ = -$2;         }
  ;

pow:
    elm            { $$ = $1;          }
  | pow '^' pow    { $$ = pow($1, $3); }
  ;

elm:
    NUM            { $$ = $1;          }
  | '(' exp ')'    { $$ = $2;          }
  ;

%%
```

Yacc Example: In-fix Calculator v2 (2/2)

- 9 shift/reduce conflicts remaining!
- 2*6/3 is 4 but 6/3*2 is 1?
  - Use only left and right recursion in a rule to implement operator associativity

Yacc Example: In-fix Calculator v3 (1/2)

Adapted from http://www.gnu.org/software/bison/manual/html_node/Infix-Calc.html#Infix-Calc

```
/* Bison declarations.  */
%token NUM

%% /* The grammar follows.  */

input:
    /* empty */
  | '\n' input
  | exp '\n' { printf("%d\n", $1); } input
  ;

exp:
    term           { $$ = $1;          }
  | exp '+' term   { $$ = $1 + $3;     }
  | exp '-' term   { $$ = $1 - $3;     }
  ;

term:
    fact           { $$ = $1;          }
  | term '*' fact  { $$ = $1 * $3;     }
  | term '/' fact  { $$ = $1 / $3;     }
  ;

fact:
    pow            { $$ = $1;          }
  | '-' fact       { $$ = -$2;         }
  ;

pow:
    elm            { $$ = $1;          }
  | elm '^' pow    { $$ = pow($1, $3); }
  ;

elm:
    NUM            { $$ = $1;          }
  | '(' exp ')'    { $$ = $2;          }
  ;

%%
```

Yacc Example: In-fix Calculator v3 (2/2)

- Finally we got 0 conflict!
- Exercise: error handling
- It is tedious to introduce so many nonterminals and examine recursions, and the code is hard to maintain
- Yacc provides some directives to make it easier to write the grammar

Yacc Example: In-fix Calculator v4

Adapted from http://www.gnu.org/software/bison/manual/html_node/Infix-Calc.html#Infix-Calc

```
/* Bison declarations.  */
%token NUM
%left '+' '-'
%left '*' '/'
%left NEG      /* negation--unary minus */
%right '^'

%% /* The grammar follows.  */

input:
    /* empty */
  | '\n' input
  | exp '\n' { printf("%d\n", $1); } input
  ;
```

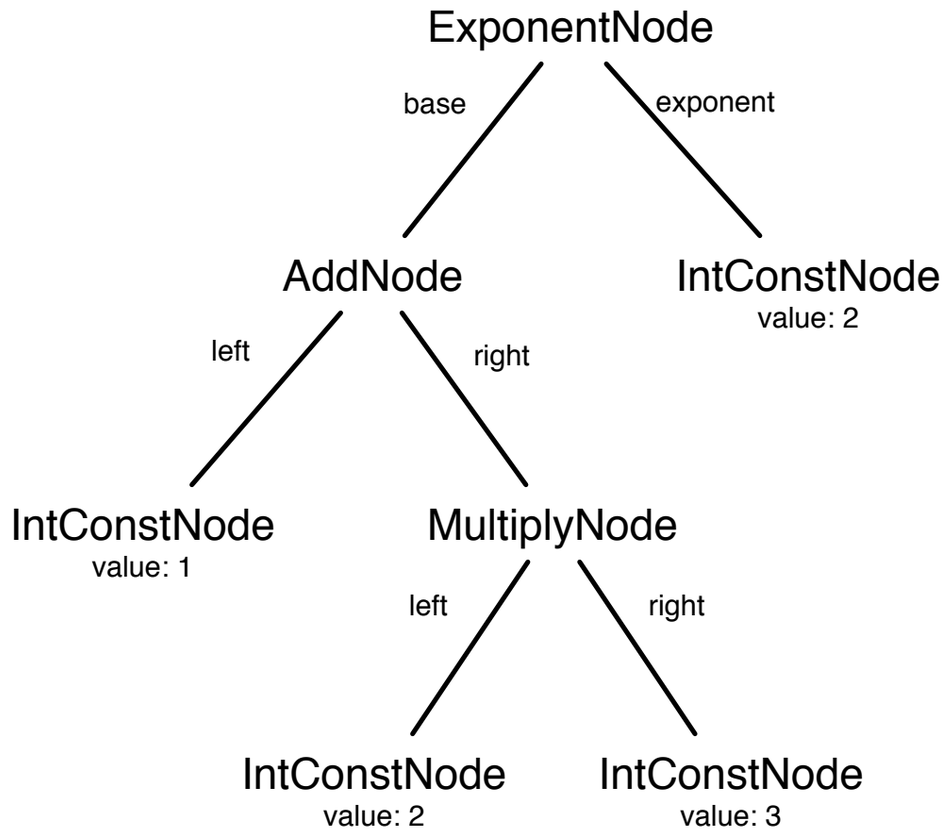
```
exp:
    NUM                { $$ = $1;          }
  | exp '+' exp        { $$ = $1 + $3;     }
  | exp '-' exp        { $$ = $1 - $3;     }
  | exp '*' exp        { $$ = $1 * $3;     }
  | exp '/' exp        { $$ = $1 / $3;     }
  | '-' exp %prec NEG  { $$ = -$2;         }
  | exp '^' exp        { $$ = pow($1, $3); }
  | '(' exp ')'        { $$ = $2;          }
  ;

%%
```

Constructing AST

- Instead of do the calculation in place, we need to return a specific AST node in each action

```
exp:
    NUM
        { $$ = new IntConstNode($1);      }
  | exp '+' exp
        { $$ = new AddNode($1, $3);       }
  | exp '-' exp
        { $$ = new SubtractNode($1, $3); }
  | exp '*' exp
        { $$ = new MultiplyNode($1, $3); }
  | exp '/' exp
        { $$ = new DivideNode($1, $3);    }
  | '-' exp %prec NEG
        { $$ = new NegateNode($2);        }
  | exp '^' exp
        { $$ = new ExponentNode($1, $3); }
  | '(' exp ')'
        { $$ = $2;                        }
;
```

AST of “(1+2*3)^2”



References

- Manual of Bison - http://www.gnu.org/software/bison/manual/html_node/index.html
- http://dinosaur.compilertools.net/#yacc
- http://dinosaur.compilertools.net/#bison
- http://epaperpress.com/lexandyacc/

Thank you & all the best