Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Compiler Theory: CS31003
3rd year CSE, 5th Semester

Laboratory Quiz - 2 : Bison *Marks: 15*
Date: October 15, 2020

1. Identify the correct matching in Bison specification of a grammar: [1]

   (A) %type
   (B) %token
   (C) The character '|'
   (D) The character ':'

   (i) Symbolized terminals
   (ii) Separator to specify multiple alternate rules
   (iii) Production rule
   (iv) Non-terminal symbols

   a) (A)-(i), (B)-(iv), (C)-(ii), (D)-(iii)
   b) (A)-(i), (B)-(iv), (C)-(iii), (D)-(ii)
   c) (A)-(iv), (B)-(i), (C)-(ii), (D)-(iii)
   d) (A)-(iv), (B)-(i), (C)-(iii), (D)-(ii)
   e) None of the options are correct

   Answer: c)

2. Consider the Bison specification of a simple calculator : [1]

```
%{
#include <string.h>
#include <iostream>
extern int yylex();
void yyerror(char *s);
%}

%union {
int intval;
}
```

```
%token <intval> NUMBER
%type <intval> expression
%type <intval> term
%type <intval> factor

%%
statement: expression { printf("= %d\n", $1); };

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

expression: expression '+' term { $$ = $1 + $3; }
  | expression '-' term { $$ = $1 - $3; }
  | term ;
factor: '(' expression ')' { $$ = $2; }
  | '-' factor { $$ = -$2; }
  | NUMBER ;
%%

void yyerror(char *s) {
std::cout << s << std::endl;
}

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

What does $$ and $2 (w.r.t. an expression 35 + 23) indicate in the above specification?

a) Attribute of the LHS non-terminal, the token '23'

b) Attribute of the LHS non-terminal, the token '+'

c) Address in symbol table corresponding to the LHS non-terminal, the token '23'

d) Address in symbol table corresponding to the LHS non-terminal, the token '+'

e) None of these

Answer: b)

3. Consider the Bison specification of a simple calculator: [1]

```
%{
#include <string.h>
#include <iostream>
extern int yylex();
void yyerror(char *s);
%}
```

```
%union {
int intval;
}

%token <intval> NUMBER
%type <intval> expression
%type <intval> term
%type <intval> factor

%%
statement: expression
{ printf("= %d\n", $1); }
;
term: term '*' factor
{ $$ = $1 * $3; }
| factor
;
expression: expression '+' term
{ $$ = $1 + $3; }
| expression '-' term
{ $$ = $1 - $3; }
| term
;
factor: '(' expression ')'
{ $$ = $2; }
| '-' factor
{ $$ = -$2; }
| NUMBER
;
%%

void yyerror(char *s) {
std::cout << s << std::endl;
}

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

What value will be printed for the expression: (12 * 3 + 10) 20 * 5

a) 54

b) -54

c) 680

d) -56

Answer: b)

4. Suppose you write a Bison specification for a given grammar in a file **myspec.y**. Which of the following statements is/are true? [1]

   a) Flex generates a file called **y.tab.h**, which specifies the token constants and attribute types.
   b) Bison generates a file called **y.tab.h**, which specifies the actions corresponding to every production rule.
   c) **yytext** is a pre-defined global variable of type **YYSTYPE**.
   d) None of the statements are correct.

   Answer: d)

5. Consider the typical Flex-Bison usage flow, where the following commands have to be executed :

```
(i) flex   test.l

(ii) g++   -c   y.tab.c

(iii) g++  -c   lex.yy.c

(iv) yacc  -dtv  test.y

(v) g++    lex.yy.o   y.tab.o   -lfl
```

   The correct order of execution of the commands will be : [1]

   a) (i), (iv), (iii), (ii), (v)
   b) (i), (ii), (iv), (iii), (v)
   c) (i), (iii), (iv), (ii), (v)
   d) None of the options are correct

   Answer: a)

6. Consider the following ambiguous grammar for arithmetic expressions:

```
S -> E | E + E | E  E | E * E | E / E | (E) | -E |
E ^ E | num
```

   Here the terminal **num** denotes a number. The arithmetic operators +, -, * and / are left-associative, whereas the exponentiation operator ^ is right associative. The priority of evaluation is as follows (from highest to lowest):

   - Parentheses
   - Unary minus (consecutive applications are not allowed)
   - Exponentiation

- Multiplication and division
- Addition and subtraction

Which of the following Bison code segments will correctly model the above where **UMINUS** stands for unary minus (to be specified with the rule **E: -E**)? [1]

a)
```
%nonassoc UMINUS
%right ‘^’
%left ‘*’ ‘/’
%left ‘+‘ ‘-’
```

b)
```
%left ‘+’ ‘-’
%left ‘*’ ‘/’
%right ‘^’
%nonassoc UMINUS
```

c)
```
%leftassoc ‘+’ ‘-’
%leftassoc ‘*’ ‘/’
%rightassoc ‘^’
%nonassoc UMINUS
```

d)
```
%right UMINUS
%right ‘^’
%left ‘*’ ‘/’
%left ‘+’ ‘-’
```

Answer : a)

7. Consider the following segment of C language grammar that specifies the **“if”** control construct:

```
S -> IF (B) THEN S
S -> IF (B) THEN S ELSE S
```

Which of the following Bison specifications correctly models nested **IF** statements, where an **ELSE** statement is tagged with the nearest **IF**? [1]

a)
```
%token IF THEN ELSE variable
%%
stmt: expr
| if_stmt
;
if_stmt:  IF expr THEN stmt
| IF expr THEN stmt ELSE stmt
;
expr:     variable
;
```

b)
```
%token IF THEN ELSE variable
%%
stmt:     expr
| if_stmt
;

if_stmt:
IF expr THEN stmt
| ELSE stmt
;
expr:     variable
;
```

c)
```
%token IF THEN ELSE variable
%%
stmt:
IF expr THEN stmt
| IF expr THEN stmt ELSE stmt
;
expr:     variable
;
```

d)
```
%token IF THEN ELSE variable
%%
stmt:     expr
| if_stmt
;

if_stmt:
IF expr THEN stmt ELSE stmt
;
expr:     variable
;
```

Answer : a)

8. Consider the following grammar that specifies some of the C control constructs:

```
S -> { L }
S -> id = E;
S -> if (B) S
S -> if (B) S else S
S -> while (B) S
S -> do S while (B)
L -> L S
L -> S
E -> id
E -> num
```

Which of the following represents the correct grammar after back-patching using dummy non-terminals? [2]

a)
```
S -> { L }
S -> id = E;
S -> if (B) M S
S -> if (B) M1 S N else M2 S
S -> while M1 (B) M2 S
S -> do M1 S while M2 (B)
L -> L M S
L -> S
E -> id
E -> num
M -> eps
N -> eps
```

b)
```
S -> { L }
S -> id = E;
S -> if (B) M S
S -> if (B) M1 S else M2 S
S -> while M1 (B) M2 S
S -> do M1 S while M2 (B)
L -> L M S
L -> S
E -> id
E -> num
M -> eps
N -> eps
```

c)
```
S -> { L }
S -> id = E;
S -> if M1 (B) M2 S
S -> if N1 (B) M1 S N2 else M2 S
S -> while M1 (B) M2 S
S -> do M1 S while M2 (B)
L -> L M S
L -> S
E -> id
E -> num
M -> eps
N -> eps
```

d)
```
S -> { L }
S -> id = E;
S -> if (B) M S
S -> if (B) M1 S N else M2 S
S -> N while M1 (B) M2 S
S -> do M1 S N while M2 (B)
L -> L M S
L -> S
E -> id
E -> num
M -> eps
```

```
N -> eps
```

Answer : a)

9. Consider the following Boolean expression grammar:

```
B -> B1 || B2
B -> B1 && B2
B -> ! B1
B -> (B1)
B -> E1 relop E2
B -> true
B -> false
```

Which of the following Bison specifications correctly model the grammar, with same precedence rules used in **tinyC**. [2]

a)
```
%left OR AND NOT LT LE EQ NE GT GE
expr:
BOOLEAN
| VARIABLE        { $$ = sym[$1];}
| expr OR expr   { if($1==1||$3 ==1){$$=1;}else{$$=0;} }
| expr AND expr   { $$ = $1 * $3;}
| NOT expr        { if($2==1){ $$=0; }else{ $$=1;} }
| '(' expr ')'    { $$ = $2; }

|  expr LT expr { if(  $1 < $3){ $$=1; }else{$$=0 ;} }
|  expr LE expr { if(  $1 <=$3){ $$=1; }else{$$=0 ;} }
|  expr EQ expr { if(  $1 ==$3){ $$=1; }else{$$=0 ;} }
|  expr NE expr { if(  $1 <>$3){ $$=1; }else{$$=0 ;} }
|  expr GT expr { if(  $1 >$3){ $$=1; }else{$$=0 ;} }
|  expr GE expr { if(  $1 >=$3){ $$=1; }else{$$=0 ;} }
;
```

b)
```
%right OR AND NOT LT LE EQ NE GT GE
expr:
BOOLEAN
| VARIABLE
| expr OR expr   { if($1==1||$3 ==1){$$=1;}else{$$=0;} }
| expr AND expr   { $$ = $1 * $3;}
| NOT expr        { if($2==1){ $$=0; }else{ $$=1;} }
| '(' expr ')'    { $$ = $2; }

|  expr LT expr { if(  $1 < $3){ $$=1; }else{$$=0 ;} }
|  expr LE expr { if(  $1 <=$3){ $$=1; }else{$$=0 ;} }
|  expr EQ expr { if(  $1 ==$3){ $$=1; }else{$$=0 ;} }
|  expr NE expr { if(  $1 <>$3){ $$=1; }else{$$=0 ;} }
|  expr GT expr { if(  $1 >$3){ $$=1; }else{$$=0 ;} }
|  expr GE expr { if(  $1 >=$3){ $$=1; }else{$$=0 ;} }
```

```
| TRUE{$$=$1;}
| FALSE{$$=$0;}
;
```

c)
```
%left OR AND NOT LT LE EQ NE GT GE
expr:
BOOLEAN
| VARIABLE        { $$ = sym[$1];}
| expr OR expr   { if($1==1||$3 ==1){$$=1;}else{$$=0;} }
| expr AND expr   { $$ = $1 + $3;}
| NOT expr        { if($2==1){ $$=0; }else{ $$=1;} }
| ‘(’ expr ‘)’    { $$ = $2; }

|  expr LT expr { if(  $1 < $3){ $$=1; }else{$$=0 ;} }
|  expr LE expr { if(  $1 <=$3){ $$=1; }else{$$=0 ;} }
|  expr EQ expr { if(  $1 ==$3){ $$=1; }else{$$=0 ;} }
|  expr NE expr { if(  $1 <>$3){ $$=1; }else{$$=0 ;} }
|  expr GT expr { if(  $1 >$3){ $$=1; }else{$$=0 ;} }
|  expr GE expr { if(  $1 >=$3){ $$=1; }else{$$=0 ;} }
| TRUE{$$=$1;}
| FALSE{$$=$0;}
;
```

d)
```
%left OR AND
%left EQ NE
%left LT LE GT GE
%left NOT
expr:
BOOLEAN
| VARIABLE        { $$ = sym[$1];}
| expr OR expr   { if($1==1||$3 ==1){$$=1;}else{$$=0;} }
| expr AND expr   { $$ = $1 * $3;}
| NOT expr        { if($2==1){ $$=0; }else{ $$=1;} }
| ‘(’ expr ‘)’    { $$ = $2; }

|  expr LT expr { if(  $1 < $3){ $$=1; }else{$$=0 ;} }
|  expr LE expr { if(  $1 <=$3){ $$=1; }else{$$=0 ;} }
|  expr EQ expr { if(  $1 ==$3){ $$=1; }else{$$=0 ;} }
|  expr NE expr { if(  $1 <>$3){ $$=1; }else{$$=0 ;} }
|  expr GT expr { if(  $1 >$3){ $$=1; }else{$$=0 ;} }
|  expr GE expr { if(  $1 >=$3){ $$=1; }else{$$=0 ;} }
| TRUE{$$=$1;}
| FALSE{$$=$0;}
;
```

Answer : d)

10. Consider the following Bison specification :

```
%union {
```

```
int num;
}
%token <num> NUMBER
%token PLUS
%%
exp : exp PLUS exp
| NUMBER
;
%%
```

Which of the following shift/reduce conflicts is/are present in the above code (the **dot** symbol is shown inside { } brackets for better visibility)? [1]

a)
```
Current State: 5
exp: exp {.} PLUS exp
exp PLUS exp {.}

PLUS  shift, and go to state 4
PLUS  [reduce using rule 1 (exp)]
```

b)
```
Current State: 5
exp: exp PLUS {.} exp
exp PLUS exp {.}

exp  shift, and go to state 4
exp  [reduce using rule 1 (exp)]
```

c)
```
Current State 5:
exp: exp {.} PLUS exp
exp PLUS {.} exp

PLUS  shift, and go to state 4
PLUS  [reduce using rule 1 (exp)]
```

d) `No shift/reduce Conflict`

Answer: a)

11. Consider the following Bison specification:

```
%union {
char *str;
}
%token <str> ID
%token ARROW
%token EQUALS
%%
stmt : assignmentList edgeList
assignmentList : assignmentList assignment
```

```
|
;
assignment : ID EQUALS ID
;
edgeList : edgeList edge
|
;
edge : ID ARROW ID
;
%%
```

Which of the following shift/reduce conflicts is/are present in the above code (the **dot** symbol is shown inside { } brackets for better visibility)? [1]

a)
```
Current State: 2
stmt: assignmentList {.} edgeList
assignmentList: assignmentList {.} assignment

ARROW        shift, and go to state 4
ARROW        [reduce using rule 6 (edgeList)]
```

b)
```
Current State: 2
stmt: assignmentList {.} edgeList
assignmentList: assignmentList {.} assignment

ID        shift, and go to state 4
ID        [reduce using rule 6 (edgeList)]
```

c)
```
Current State: 2
stmt: assignmentList {.} edgeList
assignmentList: assignmentList {.} assignment

EQUAL        shift, and go to state 4
EQUAL        [reduce using rule 6 (edgeList)]
```

d) `No shift/reduce conflict.`

Answer : b)

12. Consider the following Bison specification :

```
%%
exp: { a(); } "b" { c(); } { d(); } "e" { f(); };
```

which is translated into:

```
%%
$@1: %empty { a(); };
```

```
$@2: %empty { c(); };
$@3: %empty { d(); };
exp: $@1 "b" $@2 $@3 "e" { f(); };
```

with new non-terminal symbols $@n, where n is a number. The action produces an error in : [1]

(a) $\{a();\}$
(b) $\{c();\}$
(c) $\{d();\}$
(d) $\{f();\}$

Answer: a)

13. Shift/reduce conflict occurs in the following situation, where the period (.) denotes the current parsing state:

```
if e1 then if
e2 then s1 . else s2
```

Select the order of precedence without any conflict for the rule **IF expr THEN stmt**: [1]

a) `%precedence THEN %precedence ELSE`
b) `%precedence ELSE %precedence THEN`
c) `%precedence IF %precedence ELSE`
d) None of the other options

Answer: a)