LEX AND YACC

LEX – Lexical Analyzer Generator

- Help to write programs that transform structured input
  - Simple text program
  - C compiler
- Control flow is directed by instances of regular expressions
- Two tasks in programs with structured input
  - Dividing the program into meaningful units
  - Discovering the relationship among them

LEX – Lexical Analyzer Generator

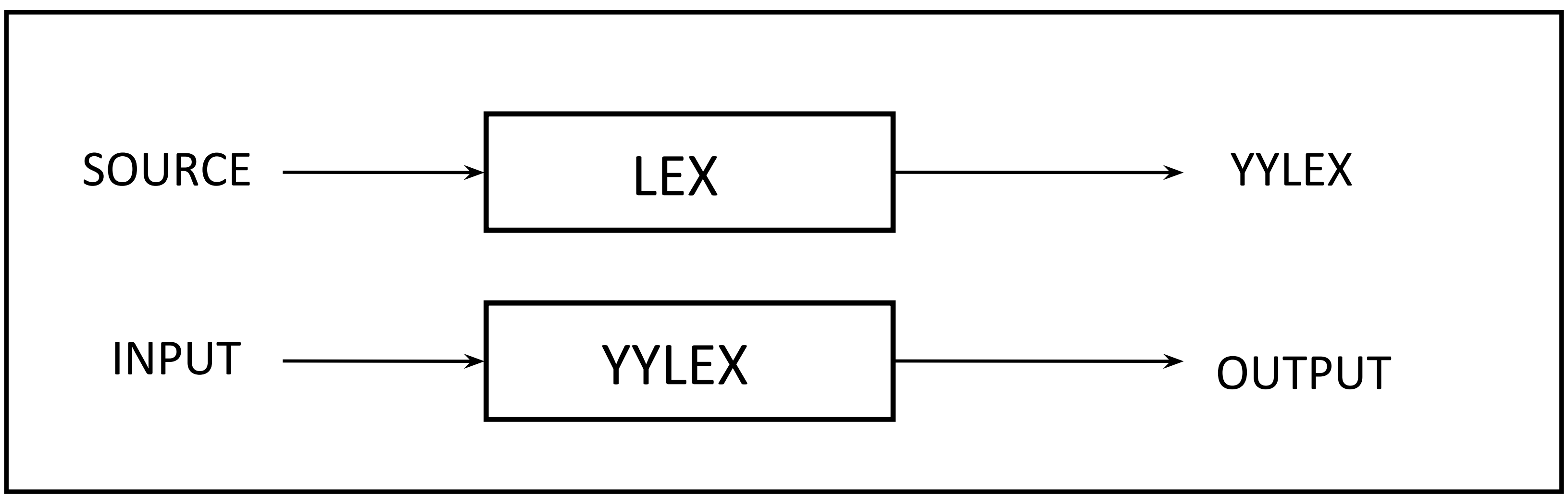
- Lexical Analysis (Scanning)
  - Division into units called tokens
- Takes a set of descriptions of tokens and produces a C routine – Lexical Analyzer
- Parsing
  - Establish relationship among tokens
  - Grammar : set of rules that define the relationships

LEX – Lexical Analyzer Generator

- Partition the input stream into strings matching the regular expressions
  - Associates regular expressions and program fragments
- Generator representing a new language feature
  - Can write code in different host languages
  - Host language used for
    - Output code generated by lex
    - Program fragments added by user

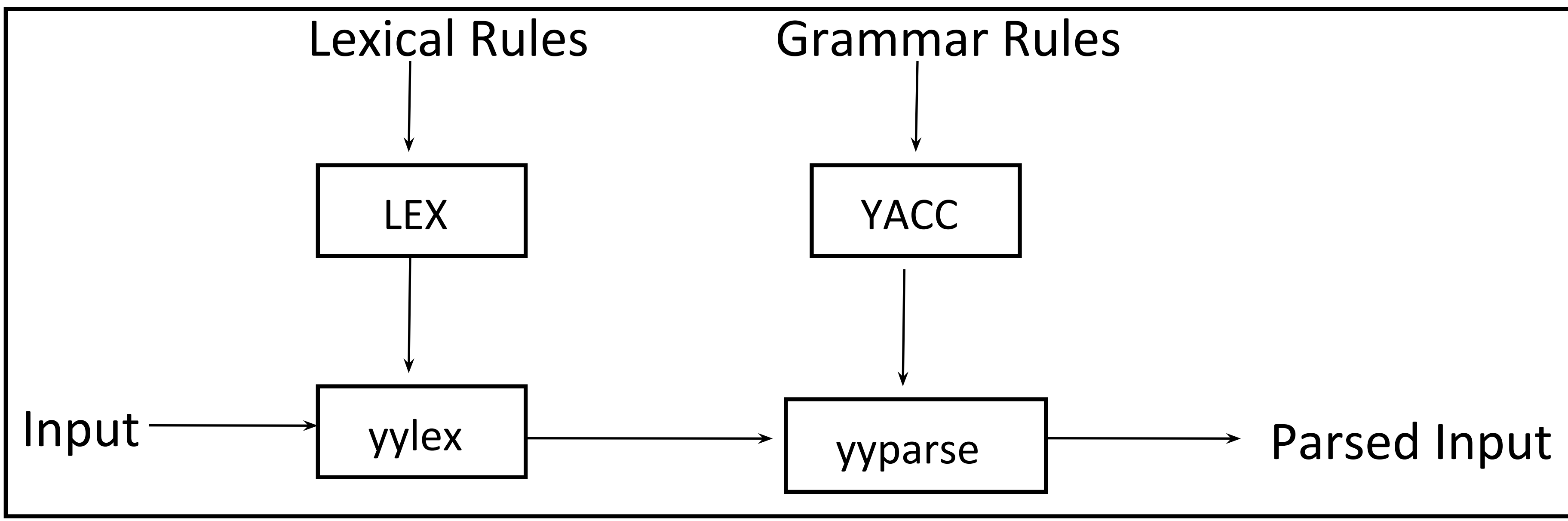
LEX – Lexical Analyzer Generator

- User's expressions and actions are converted into output general-purpose language
  - yylex : generated program
  - Recognize expressions in input and perform specified actions for each



LEX – Lexical Analyzer Generator

- Can also be used with a parser generator
- Lex programs recognize only regular expressions
- Yacc creates parsers that accepts a large class of context-free grammars



LEX – Lexical Analyzer Generator

- Generates a deterministic finite automaton from regular expressions
  - Time taken to recognize and partition an input stream is proportional to the length of the input
  - Size of automaton increases with the number and complexity of rules
- Users can insert declarations or additional statements in the routine containing actions

Lex Source

- General format

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

- Minimum Lex program is
  %% : copies input to output unchanged

Lex Source

- Rules represent user's control decisions
  - Left column contains regular expressions
  - Right column contains program fragments (or actions) to be executed
  - Ex: `integer    printf("Found keyword INT");`
  - 'C' is the host procedural language
    - Blank or tab indicates end of expression
    - If action is single C expression, can be just given on right side
    - If action is compound it must be enclosed in braces

Regular Expressions

- A pattern description using a “meta language”, a language to describe particular patterns
- RE’s specify a set of strings to be matched
- Contains
  - Text characters that match corresponding characters in the string being compared
  - Operator characters that specify repetitions and choices

Regular Expressions

<table>
<tr><th>RE CHARACTERS</th><th>DESCRIPTION</th></tr>
<tr><td>.</td><td>Matches any single character except newline character</td></tr>
<tr><td>*</td><td>Matches zero or more copies of preceding expression</td></tr>
<tr><td>+</td><td>Matches one or more occurrences of preceding expression</td></tr>
<tr><td>^</td><td>Matches the beginning of a line as first character of a RE</td></tr>
<tr><td>$</td><td>Matches the end of a line as the last character of a RE</td></tr>
<tr><td>{ }</td><td>Indicates how many times the previous pattern is allowed to match</td></tr>
<tr><td>\</td><td>Used to escape meta-characters, as part of C escape sequences</td></tr>
<tr><td>?</td><td>Matches zero or one occurrence of the preceding expression</td></tr>
<tr><td>|</td><td>Matches either the preceding RE or the following expression</td></tr>
<tr><td>[ ]</td><td>Character class, matches any character within the brackets</td></tr>
<tr><td>“……”</td><td>Interprets everything within quotation marks literally</td></tr>
<tr><td>/</td><td>Matches the preceding RE but only if followed by the following RE</td></tr>
<tr><td>( )</td><td>Groups a series of RE’s together into a new RE</td></tr>
</table>

Examples of Regular Expressions

<table>
<tr><th>Expression</th><th>Matches</th></tr>
<tr><td>abc</td><td>abc</td></tr>
<tr><td>abc*</td><td>ab abc abcc abccc ...</td></tr>
<tr><td>abc+</td><td>abc abcc abccc ...</td></tr>
<tr><td>a(bc)+</td><td>abc abcbc abcbcbc ...</td></tr>
<tr><td>a(bc)?</td><td>a abc</td></tr>
<tr><td>[abc]</td><td>one of: a, b, c</td></tr>
<tr><td>[a-z]</td><td>any letter, a-z</td></tr>
<tr><td>[a\-z]</td><td>one of: a, -, z</td></tr>
<tr><td>[-az]</td><td>one of: -, a, z</td></tr>
<tr><td>[A-Za-z0-9]+</td><td>one or more alphanumeric characters</td></tr>
<tr><td>[\t\n]+</td><td>whitespace</td></tr>
<tr><td>[^ab]</td><td>anything except: a, b</td></tr>
<tr><td>[a^b]</td><td>one of: a, ^, b</td></tr>
<tr><td>[a|b]</td><td>one of: a, |, b</td></tr>
<tr><td>a|b</td><td>one of: a, b</td></tr>
</table>

Lex Actions

- An action is executed when a RE is matched
- Default action : copying input to output
- Blank, tab and newline ignored

    [\t \n]+ ;

- **yytext**
  - Array containing the text that matched the RE
  - Rule : [a-z]+ printf("%s",yytext);
- **ECHO**
  - Places matched string on the output

Lex Actions

- Why require rules with the default action?
  - Required to avoid matching some other rule
  - ‘read’ will also match instances of ‘read’ in ‘bread’ and ‘readjust’
- **yyleng**
  - Count of the number of characters matched
  - Count number of words and characters in input
    - [a-zA-Z]+ {words++ ; chars += yyleng;}
  - Last character in the string matched

yytext [yyleng-1]

Lex Actions

- yymore()
  - Next time a rule is matched, the token should be appended to current value of yytext

```
%%
mega-      ECHO; yymore();
kludge     ECHO;
```

Input : mega-kludge

Output: mega-mega-kludge

- yyless(n)
  - Returns all but the first 'n' characters of the current token back to input stream

```
%%
foobar     ECHO; yyless(3);
[a-z]+ECHO;
```

Input : foobar

Output: foobarbar

Lex Actions

- Access to I/O routines
  - Input() : returns next input character
  - Output (c) : writes character ‘c’ on the output
  - Unput (c) : pushes character ‘c’ back onto the input stream to be read later by input()
- yywrap()
  - Called whenever lex reaches end-of-file
  - Returns 1 : Lex continues with normal wrapup on end of input

Ambiguous Source Rules

- More than one expression matches current input :
  1) Longest match is preferred
  2) Among rules that matched the same number of characters, rule given first is preferred

```
integer     keyword action…..;
[a-z]+      identifier action….;
```

Ambiguous Source Rules

- Lex does not search for all possible matches of each expression
- Each character is accounted for once and only once
- Count occurrences of both ‘she’ and ‘he’ in input text

```
she     s++;
he      h++;
.|\n    ;
```

Ambiguous Source Rules

- Action REJECT
  - Go do the next alternative
  - Goes to the rule which was the second choice after the current rule
  - Used to detect all examples of some items in the input

```
she        {s++; REJECT;}
he         {h++; REJECT;}
.| \n           ;
```

Ambiguous Source Rules

- Definitions are given before the first %%
- Lex substitution strings
  - Format : name followed by the translation
  - Associated the translation string with the name

```
D   [0-9]
E   [Dede][+-]?{D}+
%%
{D}+                    printf(“Integer\n”);
{D}*”.”{D}+({E})?               printf(“Decimal number\n”)
```

A Word Counting Example

```
%{
        unsigned int charCount = 0, wordCount = 0, lineCount = 0;
%}

word            [^ \t \n]+
eol             \n

%%

{word}          {wordCount++; charCount+=yyleng;}

{eol}           {charCount++; lineCount++;}

  .             charCount++;

%%

main()
{
        yylex();
        printf("%d %d %d\n", lineCount, charCount, wordCount);
}
```

Substitution Definition

A Word Counting Example

```
main(argc, argv)
int argc;
char **argv;
{
        if(argc > 1)  {
                        FILE *file;
                        file = fopen(argv[1] , "r");
                        if(! file)
                        {
                                fprintf(stderr, "could not open %s\n",argv[1]);
                                exit(1);
                        }
                        yyin = file;
        }
        yylex();
        printf("%u %u %u\n", wordCount, lineCount, charCount);
        return 0;
}
```

Parsing a Command Line

```
%{
        unsigned verbose;
        char *progName;
%}
%%
-h      |
"-?"    |
-help           { printf("Usage is : %s [-help | -h | -?] [-verbose | -v]"
                  "[(-file | -f) filename] \n", progName);
                }
-v      |
-verbose        {printf("Verbose mode is on \n"); verbose =1;}
%%
main(int argc, char **argv)
{
        progName = *argv;
        yylex();
}
```

Start States

- Limit scope of certain rules or change the way lexer treats part of a file
- Tagging rules with start state
  - Tells lexer to recognize rules when the start state is in effect
- Add a –file switch to recognize a filename
  - Use a start state to look for a filename

```
%s      start_state
```

Start States

```
%{
        unsigned verbose;
        unsigned fname;
        char *progName;
%}
%s      FNAME

%%
[  ]+                   ;
-h       |
"-?"     |
-help                   { printf("Usage is : %s [-help | -h | -?] [-verbose | -v]"
                          "[(-file | -f) filename] \n", progName);
                        }
-v       |
-verbose                {printf("Verbose mode is on \n"); verbose =1;}
-f       |
-file                   {BEGIN FNAME;  fname =1;}
<FNAME>.+               {printf(USe file : %s\n", yytext); BEGIN 0;
                        fname = 2;}
[^  ]+                  ECHO;
```

START STATE

RULE WITH START STATE

Start States

```
%%
char **targv;         /*remember arguments*/
char **arglim;   /*end of arguments*/

main(int argc, char **argv)
{
        progName = *argv;
        targv     = argv+1;
        arglim    = argv + argc;
        yylex();
        if(fname < 2)
                printf("No filename given");
}
```

Start States

- Rule without explicit start state will match regardless of what start state is active

```
%s      MAGIC

%%

<MAGIC>.+  {BEGIN 0;  printf("Magic: ");  ECHO;}

magic        BEGIN MAGIC;

%%

main()
{
        yylex();
}
```

Input :
magic, two, three

Output:
Magic: two
three

Start States

```
%s      MAGIC

%%

magic           BEGIN MAGIC;

. +             ECHO;

<MAGIC>.+  {BEGIN 0;  printf("Magic: ");  ECHO;}

%%

main()
{
        yylex();
}
```

Input :
magic, two, three

Output:
two
three

Parser-Lexer Communication

- Parser calls the lexer yylex() when it need a token from input
- Parser is not interested in all tokens – like comments and whitespace
- Yacc defines the token codes in "y.tab.h"

```
#define    NOUN          257
#define    PRONOUN       258
#define    VERB     259
#define    ADVERB        260
#define    ADJECTIVE     261
#define    PREPOSITION   262
```

Token Name

Token Code

```
%{
        #include "y.tab.h"
        #define LOOKUP  0
        int state;
%}
%%
\n      {state = LOOKUP;}
\.\n    {state = LOOKUP; return 0; /* end of sentence*/}

^verb   {state = VERB;}
^adj    {state = ADJECTIVE;}
^adv    {state = ADVERB;}
^noun   {state = NOUN;}
^prep   {state = PREPOSOTION;}
^pron   {state = PRONOUN;}

[a-zA-Z]+       {
                        if(state != LOOKUP)   {add_word(state, yytext);}
                        else {
                                switch(lookup_word(yytext)){
                                        case VERB               : return (VERB);
                                        case ADJECTIVE          : return (ADJECTIVE);
                                        case ADVERB             : return (ADVERB);
                                        case NOUN               : return (NOUN);
                                        case PREPOSITION        : return (PREPOSITION);
                                        case PRONOUN            : return (PRONOUN);
                                        default :
                                                printf("%s: dont recognize\n", yytext);
                                        }
                                }
                }

.               ;
%%
```

Grammars

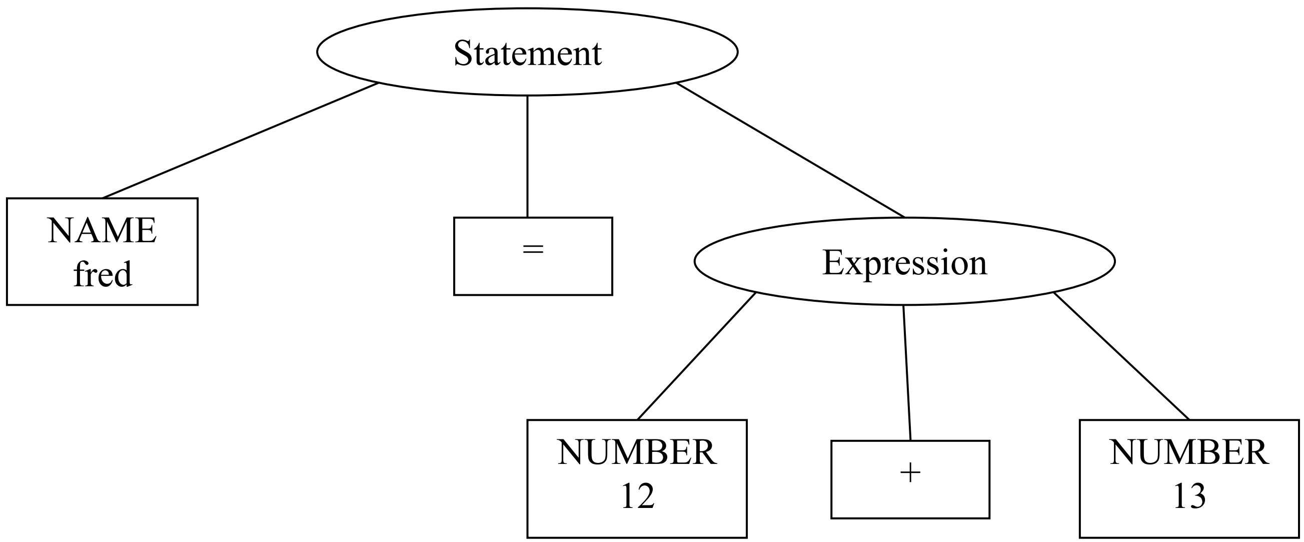
- A series of rules that the parser uses to recognize syntactically valid input.
- Grammar to build a calculator:

```
statement   ⍰   NAME = expression
expression  ⍰   NUMBER + NUMBER | NUMBER - NUMBER
```

- Non-terminal symbols
  - Symbols on LHS of rule. Ex: statement
- Terminal symbols
  - Symbols appearing in the input. Ex: NAME

Grammars

- Any parsed sentence or input is represented as a parse tree.



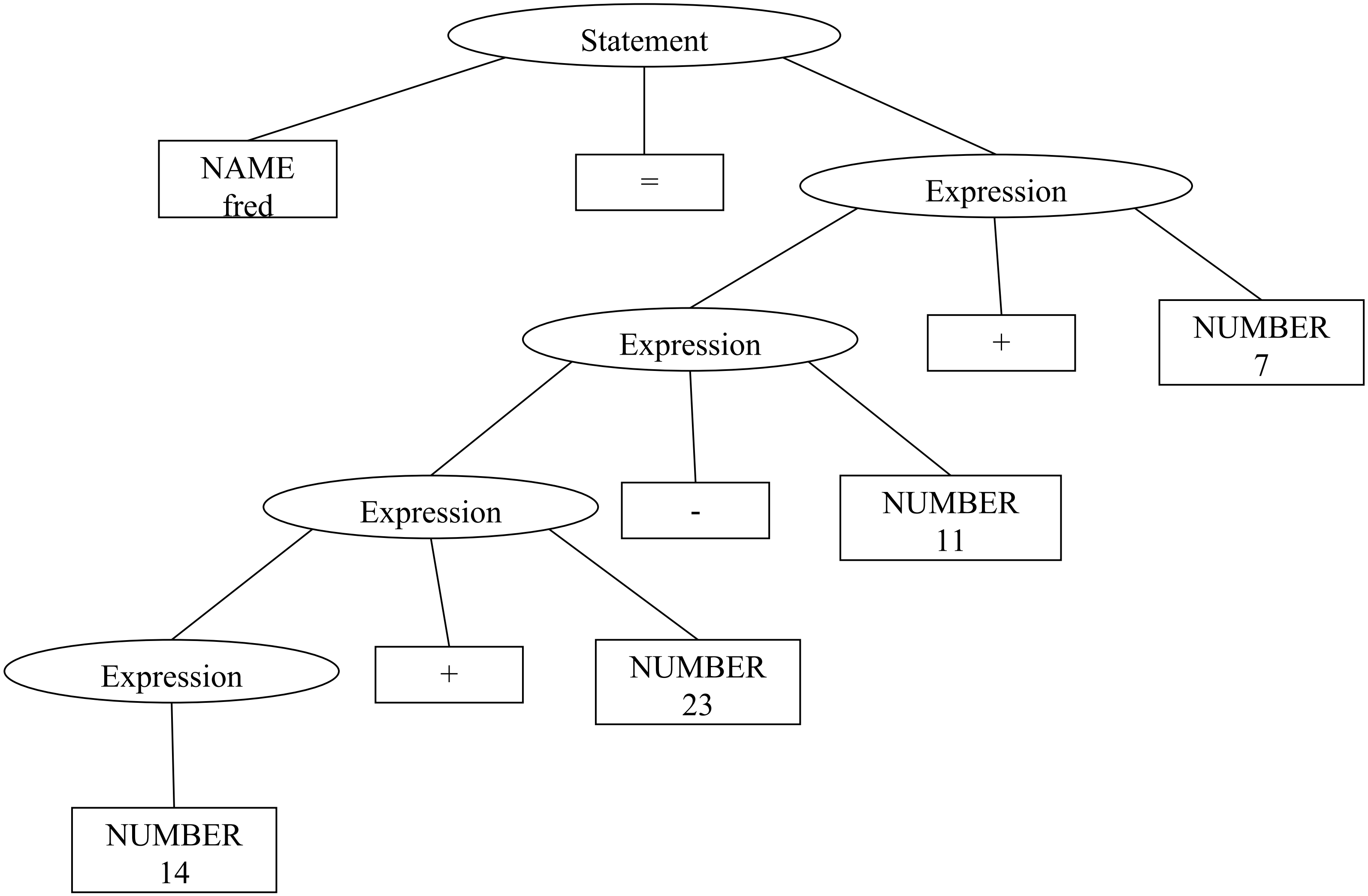
Parse tree for the input "fred = 12 + 13"

Recursive Rules

- Rules can refer directly or indirectly to themselves
- Grammar to handle longer arithmetic expressions

```
expression →  NUMBER      |
              expression + NUMBER |
              expression - NUMBER
```

Recursive Rules



Parse tree for the input "fred = 14+23-11+7"

Shift/Reduce Parsing

- A set of states is created which reflects a possible position in the parsed rules.
- As parser reads tokens:
  - Shift Action
    - Each time it reads a token that doesn't complete a rule
      - Pushes the token onto an internal stack
      - Switches to a new state reflecting the token just read
  - Reduce Action (reduces number of items on stack)
    - When all symbols of RHS of a rule are found
      - Pops the RHS symbols off the stack
      - Pushes the LHS symbol of rule onto the stack
      - Switches to a new state reflecting the new symbol on the stack

Shift/Reduce Parsing

- Parse for the input “fred = 12 + 13”
  a) Starts by shifting tokens on to the internal stack
  b) Reduce the rule “expression □ number + number”
     - Pop 12, + and 13 from stack & replace with expression
  c) Reduce rule “statement □ NAME = expression”
     - Pop fred, = and expression & replace with statement

fred
fred =
fred = 12
fred = 12 +
fred = 12 + 13

(a)

fred = expression

(b)

statement

(c)

Stack contains start symbol

What YACC cannot parse

- Cannot deal with ambiguous grammars – same input can match more than one parse tree
- Cannot deal with grammars that need more than one token of lookahead

```
phrase → cart_animal AND CART
       |   work_animal AND PLOW
cart_animal → HORSE | GOAT
work_animal → HORSE | OX
```

<u>Input :</u> "HORSE AND CART"

Cannot tell whether HORSE is a "cart_animal" or "work_animal" until it sees a CART

YACC – Yet Another Compiler Compiler

- Tool for imposing structure on the input
- User specifies
  - Rules describing the input structure
  - Code to be invoked when rules are recognized
- Generates a function to control input process
  - Function is a parser, calls the lexical analyzer to get tokens from input
  - Tokens are organized according to input structure called grammar rules
  - When a rule is recognized, the user code for it is invoked
- Main unit of input specification – Grammar rules

Simple Yacc Sentence Parser

```
%{
        #include<stdio.h>
%}

%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION CONJUNCTION

%%
sentence :  subject VERB object             {printf("Sentence is valid\n");}
          ;
subject   :  NOUN
          |  PRONOUN
          ;
object    :  NOUN
          ;
%%
extern FILE *yyin;

main()
{
        do{
                yyparse();
          }
        while(!feof(yyin));
}

yyerror(char *s)
{
        fprintf(stderr, "%s\n",s);
}
```

Tokens to be returned by lexical analyzer

YACC start symbol

Grammar Rules

Parser is repeatedly called until the lexer's input file runs out

Running Lex and Yacc

- Lex specification : `ch1-01.l`
- Yacc specification : `ch1-01.y`
- To build the output:

```
% lex ch1-01.l
% yacc –d ch1-01.y
% cc –c lex.yy.c y.tab.c
% cc –o example-01 lex.yy.o y.tab.o -ll
```

Yacc Parser

- Definition Section
  - Declare two symbolic tokens

```
%token NAME NUMBER
```

  - Can use single quoted characters as tokens, like ‘+’
- Rules Section
  - List of grammar rules using colon (:) instead of

```
statement : NAME ‘=‘ expression
      |  expression   ;
expression : NUMBER ‘+’ NUMBER
      |   NUMBER ‘-’ NUMBER ;
```

Symbol Values and Actions

- Every symbol in the parser has a value

Symbol	Value
Number	Particular number
Literal text	Pointer to a copy of the string
Variable	Symbol table entry describing the variable

- Non-terminal symbols can have any values, created by code in the parser
- Values of different symbols use different datatypes
- If there are multiple value types
  - List all the value types so that yacc can create a C union typedef called YYSTYPE

Symbol Values and Actions

- Action code
  - Refers to values of the RHS symbols as $1,$2,…
  - Sets the value of LHS by setting $$

```
%token NAME NUMBER
%%
statement  :    NAME '=' expression
           |    expression              {printf("=%d\n",$1);}
           ;
expression :    expression '+' NUMBER        {$$ = $1 + $3;}
           |    expression '-' NUMBER        {$$ = $1 - $3;}
           |    NUMBER                  {$$ = $1;}
           ;
```

The Lexer

- Need a lexer to give tokens to the parser
- "y.tab.h" – include file with token number definitions

```
%{
      #include "y.tab.h"
      extern int yylval;
%}
%%

[0-9]+      {yylval = atoi(yytext);  return NUMBER;}
[ \t]         ;
\n          return 0;  /* logical EOF*/
.           return yytext[0];
%%
```

Compiling & Running a simple Parser

```
% yacc –d ch3-01.y  #makes y.tab.c and y.tab.h
% lex ch3-01.l       #makes lex.yy.c
% cc –o ch3-01 y.tab.c lex.yy.c –ly –ll
% ch3-01
99+12
= 111
% ch3-01
2 + 3-14 +33
=24
% ch3-01
100 + -50     #does not confirm to grammar
Syntax error
```

Arithmetic Expressions and Ambiguity

```
expression :    expression '+' expression        {$$ = $1 + $3;}
           |    expression '-' expression        {$$ = $1 - $3;}
           |    expression '*' expression        {$$ = $1 * $3;}
           |    expression '/' expression
                     { if ($3 == 0)
                            yyerror("Divide by zero");
                       else
                            $$ = $1 / $3;
                     }
           |    '-' expression                   {$$ = -$2;}
           |    '(' expression ')'               {$$ = $2;}
           |    NUMBER                           {$$ = $1;}
           ;
```

Arithmetic Expressions and Ambiguity

- Ambiguous Grammar
  - When an input string can be structured in two or more different ways

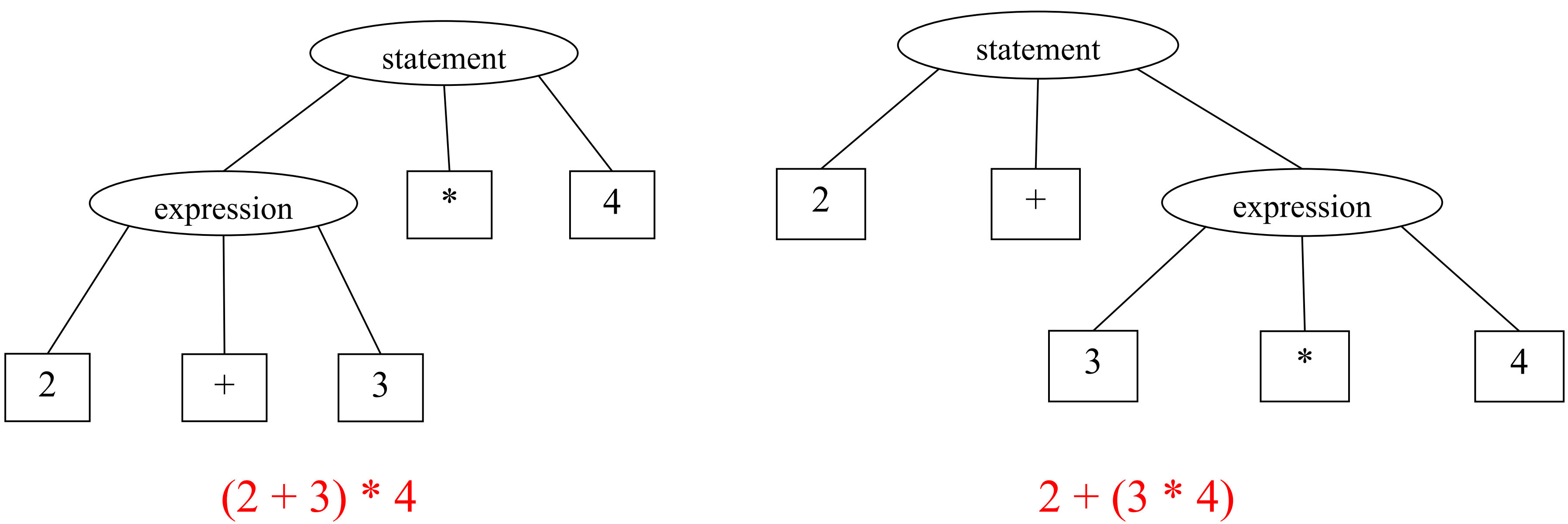


Fig : Two possible parses for 2+3*4

Arithmetic Expressions and Ambiguity

- Parsing “2 + 3 * 4”

```
2               shift NUMBER
E               reduce E --> NUMBER
E +             shift +
E + 3           shift NUMBER
E + E           reduce E--> NUMBER
```

- It sees a “*”
  - Reduce “2 + 3” to an expression
  - Shift the “*” expecting to reduce

```
expression : expression ‘*’ expression
```

Precedence and Associativity

- Precedence
  - Controls which operator is to be executed first in an expression
  - Ex : `a+b*c` ⇔ `a + (b * c)`

    `d/e-f` ⇔ `(d / e) - f`
- Associativity
  - Controls grouping of operators at same precedence level
  - Left Associativity : `a-b-c` ⇔ `(a-b)-c`
  - Right Associativity : `a=b=c` ⇔ `a=(b=c)`

Specifying Precedence & Associativity

- Implicitly
  - Rewrite the grammar using separate non-terminal symbols for each precedence level

```
expression :    expression '+' mulexp
           |    expression '-' mulexp
           |    mulexp
           ;
mulexp     :    mulexp '*' primary
           |    mulexp '/' primary
           |    primary
           ;
primary    :    '(' expression ')'
           |    '-' primary
           |    NUMBER
           ;
```

Specifying Precedence & Associativity

- Explicitly

```
% left  '+' '-'
% left  '*' '/'
% nonassoc  UMINUS
```

  - “+” & “-” □left-associative and at lowest precedence
  - “*” & “/” □left-associative and at higher precedence
  - UMINUS □Token for unary minus. No associativity but is at highest precedence

Specifying Precedence & Associativity

```
%token NAME NUMBER
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS
%%
statement  :    NAME '=' expression
           |    expression                  {printf("=%d\n",$1);}
           ;
expression :    expression '+' expression        {$$ = $1 + $3;}
           |    expression '-' expression        {$$ = $1 - $3;}
           |    expression '*' expression        {$$ = $1 * $3;}
           |    expression '/' expression
                     { if ($3 == 0)
                           yyerror("Divide by zero");
                       else
                           $$ = $1 / $3;
                     }
           |    '-' expression  %prec UMINUS     {$$ = -$2;}
           |    '(' expression ')'               {$$ = $2;}
           |    NUMBER                           {$$ = $1;}
           ;
%%
```

Use precedence of UMINUS

“Dangling Else” Conflict in IF-THEN-ELSE Constructs

- Grammar for IF-THEN ELSE

Simple if rule

If-else rule

```
stmt :    IF '(' cond ')' stmt
     |    IF '(' cond ')' stmt ELSE stmt
     ;
```

- It is an ambiguous grammar. Input of form,

```
IF (C1) IF (C2) S1 ELSE S2
```

can be structured in two ways :

```
IF(C1) {                    IF(C1) {
 IF(C2)  S1          OR          IF(C2)  S1
  }                         ELSE  S2
ELSE S2                    }
```

Grammar to avoid "dangling else" Conflict

```
stmt        :     matched    |    unmatched  ;
matched     :     other_stmt
            |     IF expr THEN matched ELSE matched
            ;
unmatched   :     IF expr THEN stmt
            |     IF expr THEN matched ELSE unmatched
            ;
other_stmt : /* rules for other kind of statements*/
```

Set explicit Precedence

- Assign precedence to token (ELSE) and rule(%prec LOWER_THAN_ELSE)
- Precedence of token to shift must be higher than precedence of rule to reduce

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%%

stmt :    IF expr stmt        %prec LOWER_THAN_ELSE;
     |    IF expr stmt ELSE stmt
     ;
```

Set explicit Precedence

- Swap the precedence's of the token to shift and rule to reduce

```
if expr if expr stmt else stmt
if expr { if expr stmt else stmt }
```

Equivalent

- Allow one ELSE with a sequence of IF's, and the ELSE is associated with the first IF

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%%

stmt     :    IF expr stmt2       %prec LOWER_THAN_ELSE
         |    IF expr stmt2 ELSE stmt;
stmt2    :    IF expr stmt2;
```

Variables and Typed tokens

```
%{
    double vbltable[26];
%}

%union{
        double dval;
        int vblno;
    }
%token <vblno> NAME

%token <vblno> NUMBER

%left '-' '+'
%left '*' '/'

%nonassoc UMINUS

%type <dval> expression

%%
```

Variables and Typed tokens

```
statement_list  :    statement '\n'
                |    statement_list  statement '\n'
                ;
statement       :    NAME '=' expression   {vbltable[$1] = $3;}
                |    expression            {printf("=%g\n",$1);}
                ;
expression :    expression '+' expression        {$$ = $1 + $3;}
           |    expression '-' expression        {$$ = $1 - $3;}
           |    expression '*' expression        {$$ = $1 * $3;}
           |    expression '/' expression
                    {
                        if($3 == 0.0)
                            yyerror("divide by zero");
                        else
                            $$ = $1 / $3;
                    }
           |    '-' expression  %prec UMINUS     {$$ = -$2;}
           |    '(' expression ')'               {$$ = $2;}
           |    NUMBER
           |    NAME                    {$$ = vbltable[$1];}
           ;
%%
```

Lexer for calculator

```
%{
      #include "y.tab.h"
      #include <math.h>
      extern double vbltable[26];
%}

%%

([0-9]+ | ([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)    {

                         yylval.dval = atof(yytext); return NUMBER;
                   }
[ \t]      ;
[a-z]      {yylval.vblno = yytext[0] - 'a'; return NAME;}
"$"        {return 0;}
\n | .           return yytext[0];

%%
```

How the parser works??

- Consists of a finite state machine with a state
- Capable of reading next input token (lookahead)
- Current state is the one on top of stack
  - Initially machine is in state 0
  - No lookahead token has been read
- Machine has 4 actions available
  - Shift, reduce, accept & error

How the parser works??

- Move of the parser is as follows:
  - Based on current state parser decides
    - Whether it needs a lookahead token to decide what action is to be done
    - If it needs one, it calls yylex to obtain next token
  - Using current state and lookahead token
    - Parser decides on its next action
    - Results in states being pushed onto stack or popped off
    - Lookahead token may be processed or left alone

How the parser works??

- Shift Action
  - When shift action is taken, there is lookahead token
  - Ex: In state 56 there may be an action:

```
IF    shift 34
```

- Reduce Action
  - Used when parser has seen RHS of a rule
  - Replaces right hand side by left hand side
- Action
  - Reduce 18 □ grammar rule 18
  - Shift 34 □ state 34

How the parser works??

- Consider the rule to be reduced, $\boxed{\texttt{A : xyz;}}$
  - Reduce action depends on
    - Left hand symbol (A)
    - Number of symbols on right hand side (3)
  - To reduce
    - First pop off top 3 states from stack (No. of symbols on RHS)
    - After popping, the state uncovered is the one that parser was in before processing the rule
    - The uncovered state and symbol on LHS of rule causes a new state to be pushed onto the stack

How the parser works??

- Uncovered state contains an entry

```
A goto 20
```

- When a rule is reduced, user code for that rule is executed first
- Another stack is used for holding values returned from lexical analyzer
- When shift takes place, external variable yylval is copied onto the value stack
  - yylval is copied when goto action is done
  - Variable $1,$2,…… refer to the value stack

How the parser works??

- Accept Action
  - Indicates entire input has been seen and it matches specification
  - Appears when the lookahead token is the endmarker
- Error Action
  - A place where the parser can no longer continue parsing according to specification
  - Input tokens with lookahead token cannot be followed by anything that leads to a valid input
  - Reports error and attempts to recover the situation

Parser Conflicts

- Two kinds of conflicts
  - Reduce / reduce
  - Shift / reduce
- Categorized based upon what is happening with the other pointer when one pointer is reducing

Reduce/Reduce Conflict

- Occurs when same token can complete two different rules
- If other rule is reducing when the first is reducing it leads to a reduce/reduce conflict

```
Start  : x
     | y  ;
x    : A  ;
y    : A  ;
```

Shift/Reduce Conflict

- Occurs when there are two parses for an input
  - One completes a rule (reduce option)
  - One doesn't complete a rule (shift option)
- If other pointer is not reducing, then it is shifting

```
Start  :  x
       |  yR ;
x      :  A  R ;
y      :  A ;
```

Conflicts contd.

- Exactly when does reduction take place with respect to token lookahead

```
Start : xB | yB;
  x   : A   ;
  y   : A  ;
```

Reduce/reduce Conflict

```
Start : xB | yC;
x   : A    ;
y   : A    ;
```

No Conflict

Parser States

- Yacc tells where the conflicts are in y.output
  - Description of the state machine generated
  - Can be generated by running yacc with –v option
- Every yacc grammar has at least two states:
  - One at beginning, when no input is accepted
  - One at end, when a complete valid input has been accepted

```
start  : A <state1> B <state2> C;
```

Parser States

```
start :  a | b;
 a       :  X <state1> Y <state2> Z;
 b       :  X <state1> Y <state2> Q;


start :  threeAs;
threeAs  :  /* empty */
    |  threeAs A <state1> A <state2>
           A <state3>;
```

Contents of y.output

- Includes a listing all the parser states
- For each state, it lists
  - The rules and positions for that state
  - Shifts and reductions the parser will do when it reads tokens in that state
  - The state to be switched to after a reduction produces a non-terminal in that state

Reduce/reduce Conflict

GRAMMAR

```
start :  a Y | b Y;
 a    :  X;
 b    :  X;
```

```
            1:reduce/reduce conflict
             (reduce 3, reduce 4) on Y
```

```
State 3              State 1
  start : a . Y (1)         a :   X .   (3)
                b :   X .   (4)
   Y    shift 5
   .    Error           .   Reduce 3
```

PARSER
STATES

Reduce/reduce conflict in rules with tokens or rule names

```
start  :  Z  |  b Z ;
 a     :  X y ;
 b     :  X y ;
 y     :  Y;
```

```
6 : reduce/reduce conflict(red. 3,red. 4) on Z
State 6:
    a :   Xy .    (3)
    b :   Xy .    (4)

    .  Reduce 3
```

Reduce/reduce Conflict in rules that are not identical

```
start :  A B X Z |  y Z   ;
  X   :  C  ;
  y   :  A B C    ;
```

```
7 : reduce/reduce (reduce 3, reduce 4) on Z
State 7
   X  :  C_    (3)
   y  :  A B C_  (4)

   .  Reduce 3
```

Shift/Reduce Conflicts

- To identify the conflict
  - Find the shift/reduce error in y.output
  - Pick out the reduce rule
  - Pick out relevant shift rules
  - See where the reduce rule reduces to
  - Deduce the token stream that will produce the conflict

Shift/Reduce Conflicts

```
start :  X | y R ;
 X    :  A R     ;
 y    :  A    ;
```

```
4 : shift/reduce (shift 6, reduce 4) on R
state 4
   X  :  A_R
   y  :  A_

   R  shift 6
   .  error
```

Shift/Reduce Conflicts

```
start  :  X1 | X2 | y R ;
 X1    :  A R      ;
 X2    :  A z      ;
 y     :  A    ;
 z     :  R    ;
```

```
1:shift/reduce (shift 6, reduce 6) on R
State 1
    X1 :  A . R     (4)
    X2 :  A . z     (5)
    y  :  A .    (6)

    R  shift 6
    z  goto 7
```

Example

```
%token  DING DONG DELL
%%
rhyme :  sound place
    ;
sound :  DING DONG
    ;
place :  DELL
    ;
```

Example : y.output

```
State 0
    $accept          :    _rhyme $end

    DING shift 3
    .    error

    rhyme     goto 1
    sound     goto 2

State 1
    $accept          :    rhyme_$end

    $end accept
    .    error

State 2
    rhyme            :    sound_place

    DELL shift 5
    .    error

    place     goto 4
```

Example : y.output

```
State 3
     sound          :    DING_DONG

     DONG shift 6
     .    error

State 4
     rhyme          :    sound place_

     .    reduce 1

State 5
     place          :    DELL_

     .    reduce 3

State 6
     sound          :    DING DONG_

     .    reduce 2
```