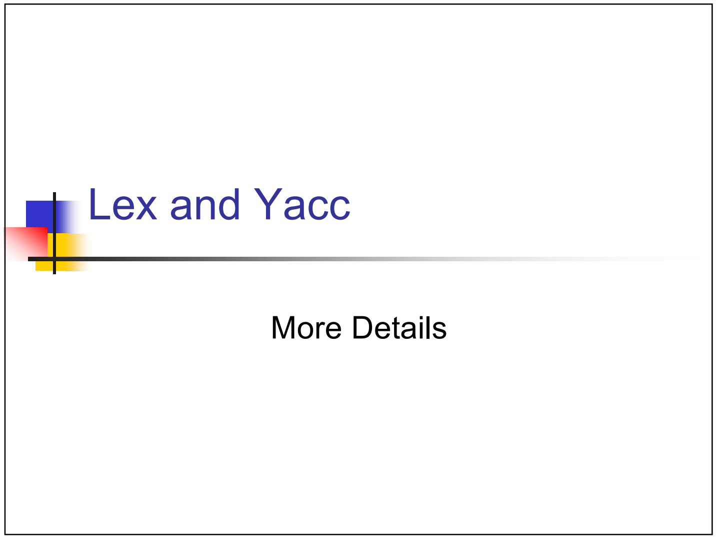
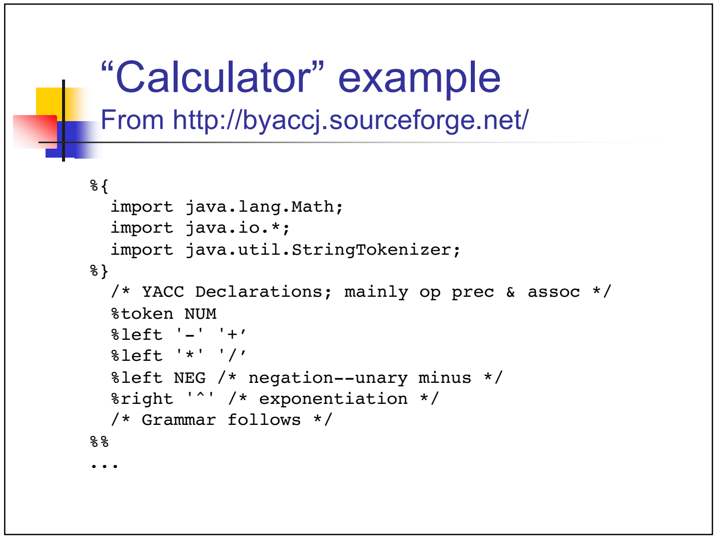


Lex and Yacc

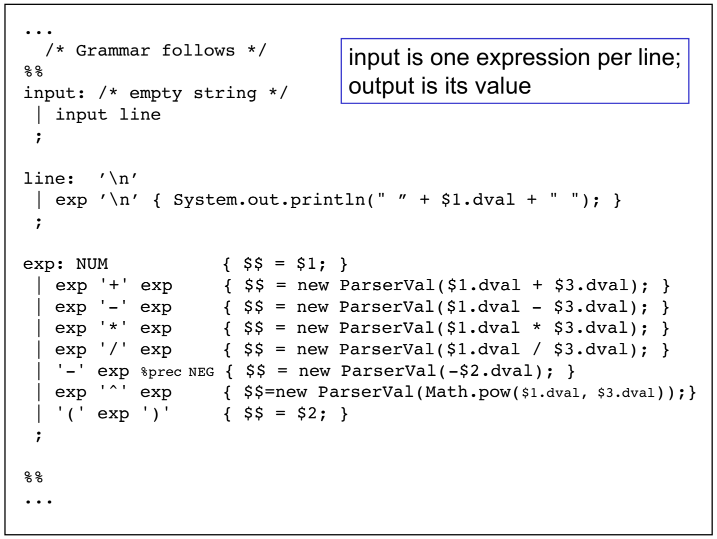
More Details



“Calculator” example

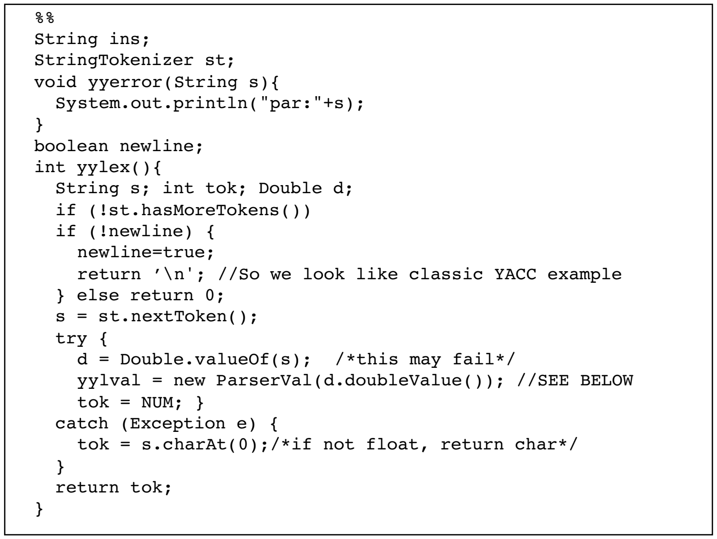
From http://byaccj.sourceforge.net/

```
%{
  import java.lang.Math;
  import java.io.*;
  import java.util.StringTokenizer;
%}
  /* YACC Declarations; mainly op prec & assoc */
  %token NUM
  %left '-' '+’
  %left '*' '/’
  %left NEG /* negation--unary minus */
  %right '^' /* exponentiation */
  /* Grammar follows */
%%
...
```



input is one expression per line; output is its value

```
...
  /* Grammar follows */
%%
input: /* empty string */
 | input line
 ;

line:  ’\n’
 | exp ’\n’ { System.out.println(" ” + $1.dval + " "); }
 ;

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

%%
...
```



```
%%
String ins;
StringTokenizer st;
void yyerror(String s){
  System.out.println("par:"+s);
}
boolean newline;
int yylex(){
  String s; int tok; Double d;
  if (!st.hasMoreTokens())
  if (!newline) {
    newline=true;
    return ’\n’; //So we look like classic YACC example
  } else return 0;
  s = st.nextToken();
  try {
    d = Double.valueOf(s);  /*this may fail*/
    yylval = new ParserVal(d.doubleValue()); //SEE BELOW
    tok = NUM; }
  catch (Exception e) {
    tok = s.charAt(0);/*if not float, return char*/
  }
  return tok;
}
```

```
void dotest(){
  BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
  System.out.println("BYACC/J Calculator Demo");
  System.out.println("Note: Since this example uses the StringTokenizer");
  System.out.println("for simplicity, you will need to separate the items");
  System.out.println("with spaces, i.e.: '( 3 + 5 ) * 2'");
  while (true) {
    System.out.print("expression:");
    try {
      ins = in.readLine();
    }
    catch (Exception e) { }
    st = new StringTokenizer(ins);
    newline=false;
    yyparse();
  }
}

public static void main(String args[]){
  Parser par = new Parser(false);
  par.dotest();
}
```

Parser “states”

- Not exactly elements of PDA’s “Q”, but similar
- A yacc "state" is a set of "dotted rules" – a grammar rules with a "dot” somewhere in the right hand side. (In some yacc printouts, "_" is the dot.)
- Intuitively, "A → α_β" in a state means this rule, up to and including α is consistent with input seen so far; next terminal in the input might derive from the left end of β. E.g., before reading any input, "S → _ β" is consistent, for every rule S → β " (S = start symbol)
- Yacc deduces legal shift/goto actions from terminals/ nonterminals following dot; reduce actions from rules with dot at rightmost end. See examples below

Yacc Output: Random Example

```
0   $accept : S $end
1   S : 'a' 'b' C 'd'
2     | 'a' 'e' F 'g'
3   C : 'h' C
4     | 'h'
5   F : 'h' F
6     | 'h'
```

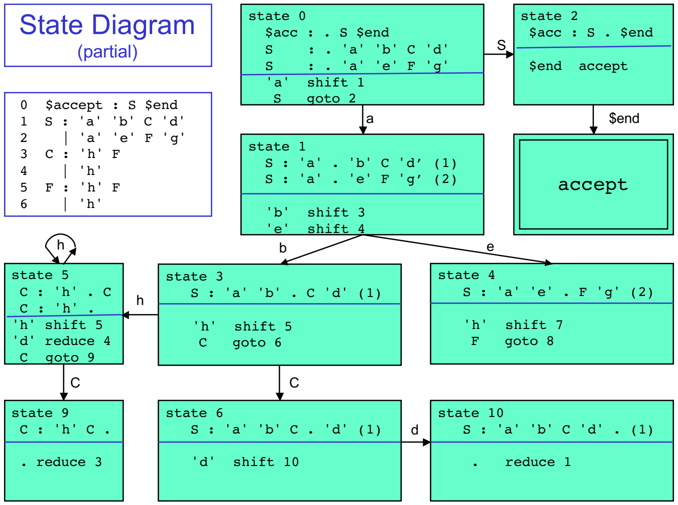
```
state 0
   $accept : . S $end  (0)

   'a'  shift 1
    .   error

    S   goto 2

state 1
   S : 'a' . 'b' C 'd' (1)
   S : 'a' . 'e' F 'g' (2)

   'b'  shift 3
   'e'  shift 4
    .   error

state 2
   $accept : S . $end  (0)

   $end  accept
```

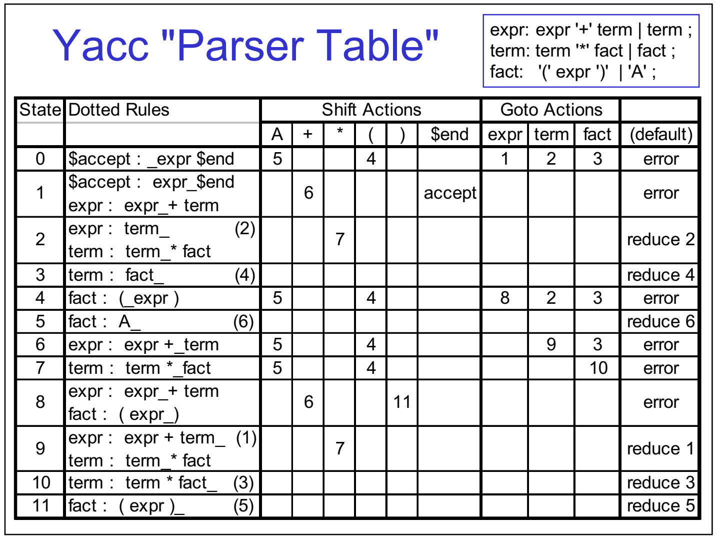
```
state 3
   S : 'a' 'b' . C 'd' (1)

   'h'  shift 5
    .   error

    C   goto 6

state 4
   S : 'a' 'e' . F 'g' (2)

   'h'  shift 7
    .   error

    F   goto 8

state 5
   C : 'h' . C  (3)
   C : 'h' .  (4)

   'h'  shift 5
   'd'  reduce 4

    C   goto 9

state 6
   S : 'a' 'b' C . 'd' (1)

   'd'  shift 10
    .   error
```

```
state 7
   F : 'h' . F  (5)
   F : 'h' .  (6)

   'h'  shift 7
   'g'  reduce 6

    F  goto 11
state 8
   S : 'a' 'e' F . 'g' (2)

   'g'  shift 12
    .   error
state 9
   C : 'h' C .  (3)

    .   reduce 3
state 10
   S : 'a' 'b' C 'd' . (1)

    .   reduce 1
state 11
   F : 'h' F .  (5)

    .   reduce 5
state 12
   S : 'a' 'e' F 'g' . (2)

    .   reduce 2
```

State Diagram
(partial)

```
0   $accept : S $end
1   S : 'a' 'b' C 'd'
2     | 'a' 'e' F 'g'
3   C : 'h' F
4     | 'h'
5   F : 'h' F
6     | 'h'
```

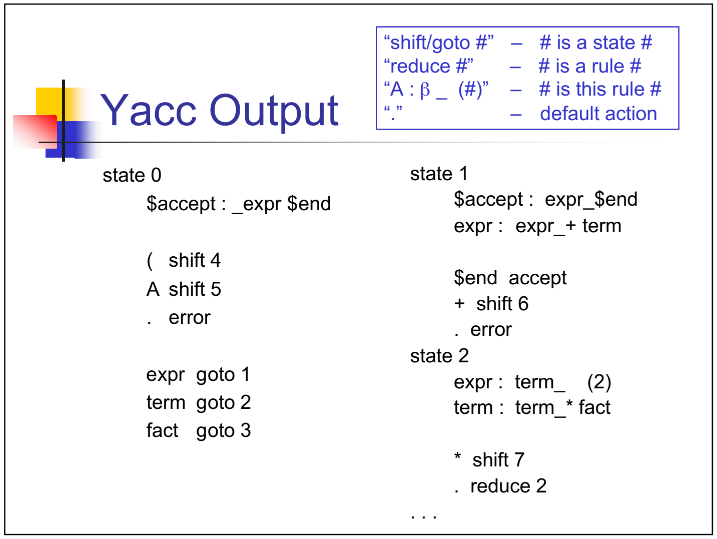

```
state 0
  $acc  : . S $end
  S     : . 'a' 'b' C 'd'
  S     : . 'a' 'e' F 'g'
  'a'   shift 1
   S    goto 2

state 2
 $acc : S . $end

 $end  accept

accept

state 1
  S : 'a' . 'b' C 'd’ (1)
  S : 'a' . 'e' F 'g’ (2)

  'b'  shift 3
  'e'  shift 4

state 5
 C : 'h' . C
 C : 'h' .
'h' shift 5
'd' reduce 4
 C  goto 9

state 3
   S : 'a' 'b' . C 'd' (1)

   'h'  shift 5
    C   goto 6

state 4
   S : 'a' 'e' . F 'g' (2)

   'h'  shift 7
    F   goto 8

state 9
 C : 'h' C .

. reduce 3

state 6
   S : 'a' 'b' C . 'd' (1)

   'd'  shift 10

state 10
   S : 'a' 'b' C 'd' . (1)

    .   reduce 1
```



Yacc "Parser Table"

expr: expr '+' term | term ;
term: term '*' fact | fact ;
fact: '(' expr ')' | 'A' ;

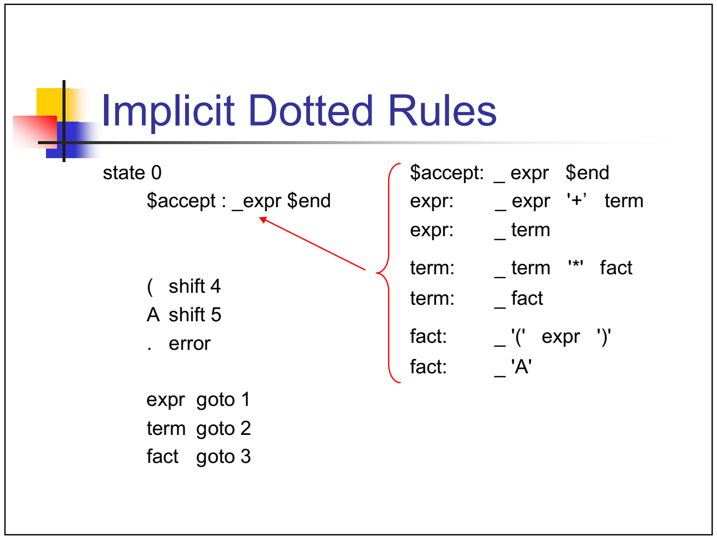
State	Dotted Rules	Shift Actions						Goto Actions			
		A	+	*	(	)	$end	expr	term	fact	(default)
0	$accept : _expr $end	5			4			1	2	3	error
1	$accept : expr_$end expr : expr_+ term		6				accept				error
2	expr : term_ (2) term : term_* fact			7							reduce 2
3	term : fact_ (4)										reduce 4
4	fact : (_expr)	5			4			8	2	3	error
5	fact : A_ (6)										reduce 6
6	expr : expr +_term	5			4				9	3	error
7	term : term *_fact	5			4					10	error
8	expr : expr_+ term fact : (expr_)		6			11					error
9	expr : expr + term_ (1) term : term_* fact			7							reduce 1
10	term : term * fact_ (3)										reduce 3
11	fact : (expr)_ (5)										reduce 5



Yacc Output

"shift/goto #"	–	# is a state #
"reduce #"	–	# is a rule #
"A : β _ (#)"	–	# is this rule #
"."	–	default action

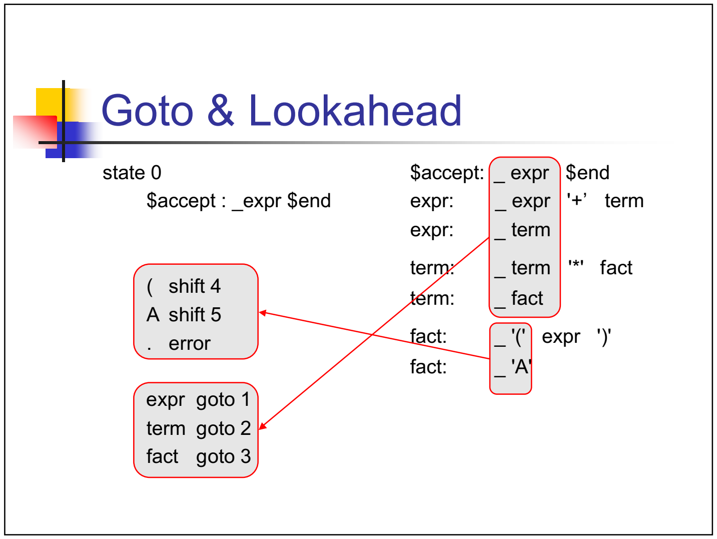
```
state 0
        $accept : _expr $end

        (  shift 4
        A  shift 5
        .  error

        expr  goto 1
        term  goto 2
        fact   goto 3

state 1
        $accept :  expr_$end
        expr :  expr_+ term

        $end  accept
        +  shift 6
        .  error
state 2
        expr :  term_    (2)
        term :  term_* fact

        *  shift 7
        .  reduce 2
. . .
```



Implicit Dotted Rules

```
state 0
        $accept : _expr $end

        (  shift 4
        A  shift 5
        .  error

        expr  goto 1
        term  goto 2
        fact   goto 3
```

```
$accept: _ expr   $end
expr:      _ expr  '+'   term
expr:      _ term
term:      _ term  '*'   fact
term:      _ fact
fact:      _ '('   expr  ')'
fact:      _ 'A'
```



Goto & Lookahead

```
state 0
        $accept : _expr $end

        (  shift 4
        A  shift 5
        .  error

        expr  goto 1
        term  goto 2
        fact   goto 3
```

```
$accept: _ expr   $end
expr:    _ expr   '+'   term
expr:    _ term
term:    _ term   '*'   fact
term:    _ fact
fact:    _ '('   expr   ')'
fact:    _ 'A'
```

using the unambiguous expression grammar

Example: input "A + A $end"

Action:	Stack:	Input:
	0	A + A $end
shift 5		
	0 A 5	+ A $end
reduce fact → A, go 3 state 5 says reduce rule 6 on +; state 0 (exposed on pop) says goto 3 on fact		
	0 fact 3	+ A $end
reduce fact → term, go 2		
	0 term 2	+ A $end
reduce expr → term, go 1		
	0 expr 1	+ A $end
shift 6		

Action:	Stack:	Input:
shift 6		
	0 expr 1 + 6	A $end
shift 5		
	0 expr 1 + 6 A 5	$end
reduce fact → A, go 3		
	0 expr 1 + 6 fact 3	$end
reduce term → fact, go 9		
	0 expr 1 + 6 term 9	$end
reduce expr → expr + term, go 1		
	0 expr 1	$end
accept		

An Error Case: "A) $end":

Action:	Stack:	Input:
	0	A) $end
shift 5		
	0 A 5	) $end
reduce fact → A, go 3		
	0 fact 3	) $end
reduce fact → term, go 2		
	0 term 2	) $end
reduce expr → term, go 1		
	0 expr 1	) $end
error		