Lex and Yacc

A Quick Tour

HW8–Use Lex/Yacc to

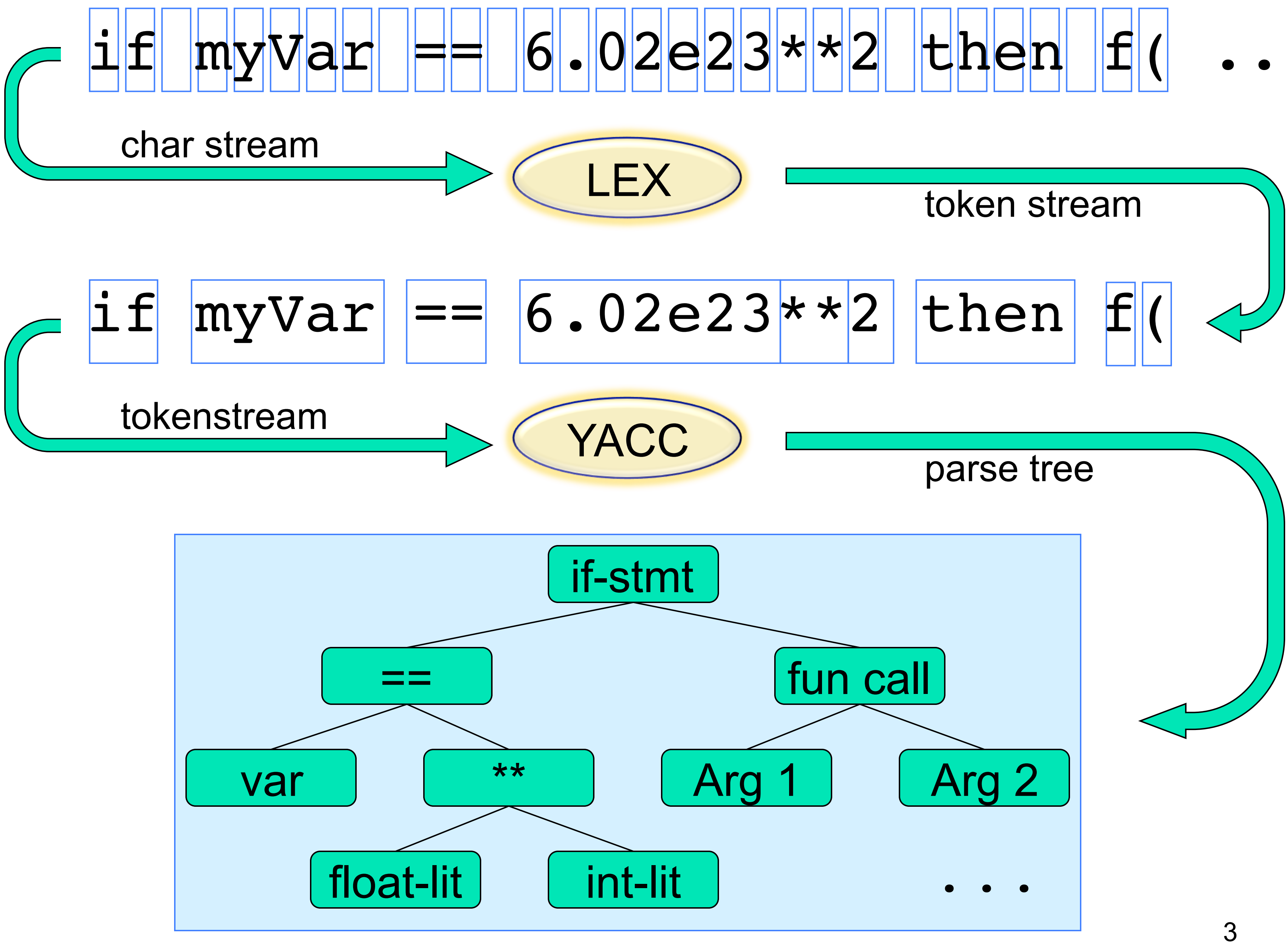
Turn this:

<P>
Here's a list:
<UL>
<LI> This is item one of a list
<LI>This is item two. Lists should be indented four spaces, with each item marked by a "*" two spaces left of four-space margin. Lists may contain nested lists, like this:<UL><LI> Hi, I'm item one of an inner list. <LI>Me two.
<LI> Item 3, inner. </UL><LI> Item 3, outer list.</UL>
This is outside both lists; should be back to no indent.
<P><P>
Final suggestions

Into this:

```
Here's a list:
  * This is item one of a list
  * This is item two. Lists should be
    indented four spaces, with each item
    marked by a "*" two spaces left of
    four-space margin. Lists may contain
    nested lists, like this:
     * Hi, I'm item one of an inner list.
     * Me two.
     * Item 3, inner.
  * Item 3, outer list.
This is outside both lists; should be back
to no indent.

Final suggestions:
```



if myVar == 6.02e23**2 then f(..
char stream
LEX
token stream
if myVar == 6.02e23**2 then f(
tokenstream
YACC
parse tree
if-stmt
==
fun call
var
**
Arg 1
Arg 2
float-lit
int-lit
. . .

Lex / Yacc History

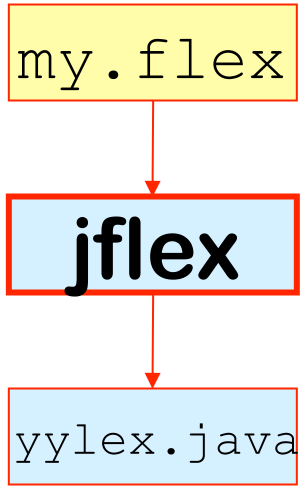
- Origin – early 1970's at Bell Labs
- Many versions & many similar tools
  - Lex, flex, jflex, posix, …
  - Yacc, bison, byacc, CUP, posix, …
  - Targets C, C++, C#, Python, Ruby, ML, …
- We'll use jflex & byacc/j, targeting java
  (but for simplicity, I usually just say lex/yacc)

Uses

- "Front end" of many real compilers
  - E.g., gcc
- "Little languages":
  - Many special purpose utilities evolve some clumsy, *ad hoc*, syntax
  - Often easier, simpler, cleaner and more flexible to use lex/yacc or similar tools from the start

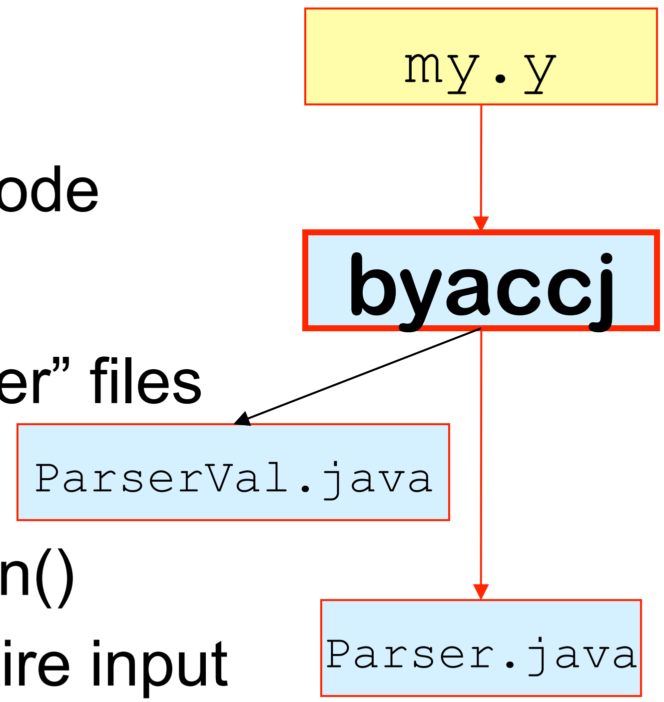
Lex:
A Lexical Analyzer Generator

- Input:
  - Regular exprs defining "tokens"
  - Fragments of declarations & code
- Output:
  - A java program “yylex.java”
- Use:
  - Compile & link with your main()
  - Calls to yylex() read chars & return successive tokens.



yacc:
A Parser Generator

- Input:
  - A context-free grammar
  - Fragments of declarations & code
- Output:
  - A java program & some “header” files
- Use:
  - Compile & link it with your main()
  - Call yyparse() to parse the entire input
  - yyparse() calls yylex() to get successive tokens



Lex Input: "mylexer.flex"

%: Lex section delims

```
// java stuff
%%
%byaccj
%{
 public foo()…
%}
%%
[a-zA-Z]+    {foo(); return(42); }
[ \t\n]      {; /* skip whitespace */}
…
```

Declarations & code: most copied verbatim to java pgm

Rules/ regexps + {Actions}

Token code

No action

Lex Regular Expressions

Letters & numbers match themselves

Ditto \n, \t, \r

Punctuation often has special meaning

- But can be escaped: * matches "*"

Union, Concatenation and Star

- r|s, rs, r*; also r+, r?; parens for grouping

Character groups

- [ab*c] == [*cab], [a-z2648AEIOU], [^abc]
- "^" for "not" *only* in char groups, not complementation

S → E
E → E+n | E-n | n

Yacc Input: “expr.y”

Java decls

```
%{
 import java.io.*;…
%}
```

→ Parser.java

Yacc decls

```
%token NUM VAR
```

→ Parser.java

Rules and {Actions}

```
%%
stmt: exp                  { printf(”%d\n”,$1);}
 ;
exp : exp ’+’ NUM          { $$ = $1 + $3;     }
 |    exp ’-’ NUM          { $$ = $1 - $3;     }
 |    NUM                  { $$ = $1;          }
 ;
%%
```

C code; java ex later

Java code

```
public static void main(…
```

→ Parser.java

Expression lexer: “expr.l”

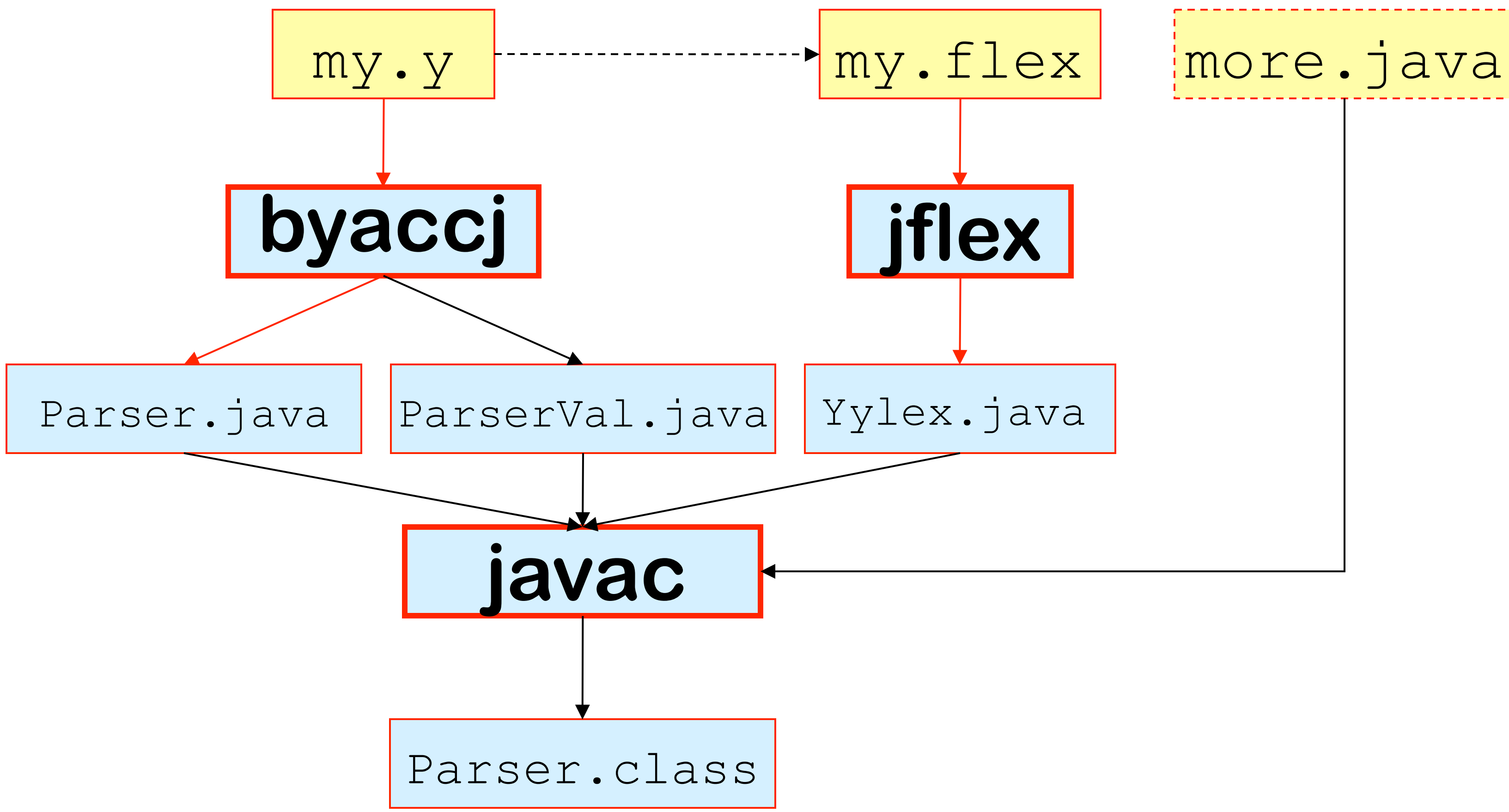
```
%{
#include "y.tab.h"

%}
%%
[0-9]+        { yylval = atoi(yytext); return NUM;}
[ \t]         {           /* ignore whitespace */ }
\n            { return 0;       /* logical EOF */ }
.             { return yytext[0]; /* +-*, etc. */ }
%%
yyerror(char *msg){printf("%s,%s\n",msg,yytext);}
int yywrap(){return 1;}
```

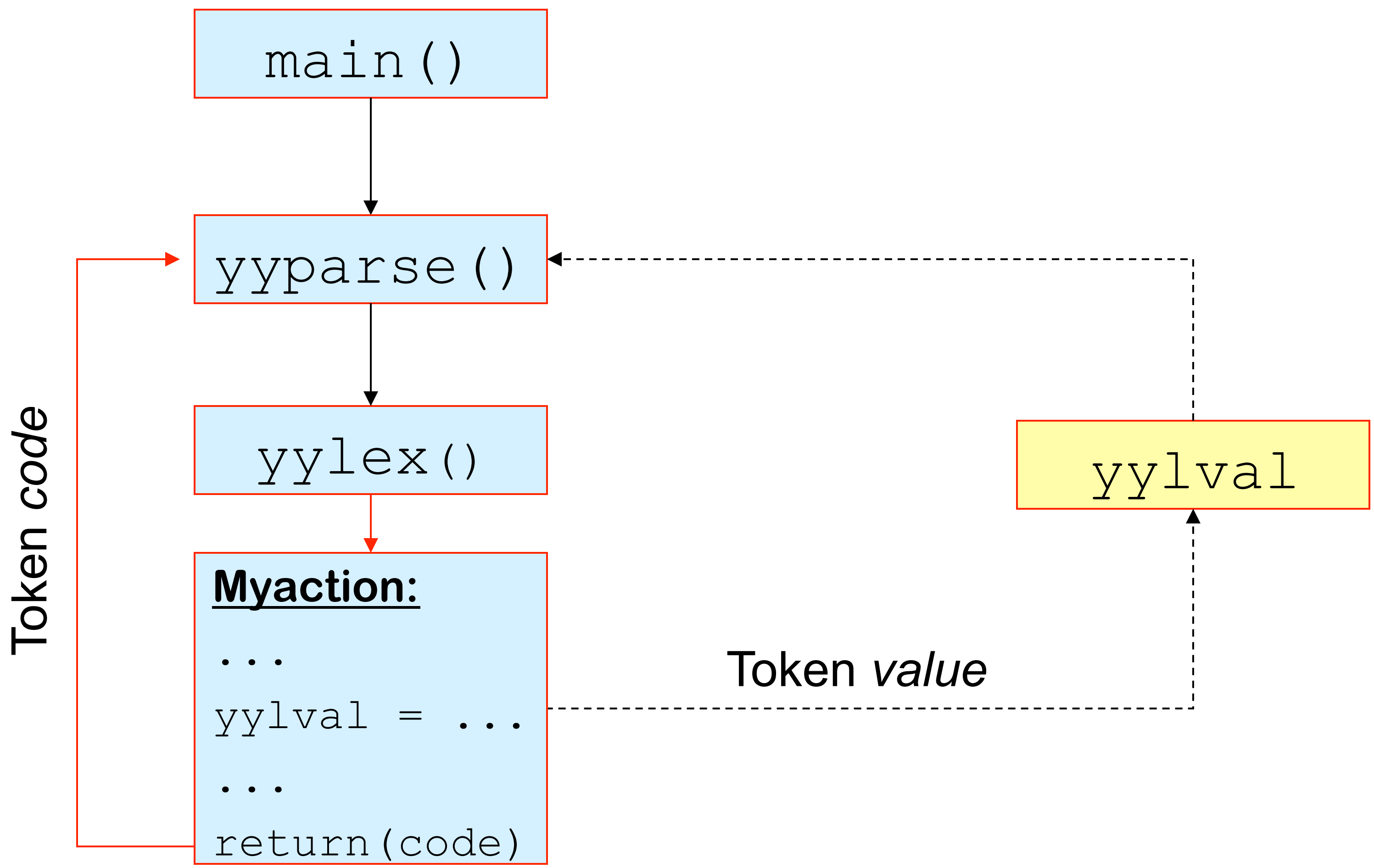
y.tab.h:

```
#define NUM     258
#define VAR     259
#define YYSTYPE int
extern  YYSTYPE yylval;
```

Lex/Yacc Interface: Compile Time



Lex/Yacc Interface: Run Time



Parser “Value” class

```
public class ParserVal
{
  public int ival;
  public double dval;
  public String sval;
  public Object obj;
  public ParserVal(int val)
  { ival=val; }
  public ParserVal(double val)
  { dval=val; }
  public ParserVal(String val)
  { sval=val; }
  public ParserVal(Object val)
  { obj=val;  }
}//end class
```

```
//then do
yylval = new ParserVal(3.14);
yylval = new ParserVal(42);
// ...or something like...
yylval = new ParserVal(new
               myTypeOfObject());
```

```
// in yacc actions, e.g.:
$$.ival = $1.ival + $2.ival;
$$.dval = $1.dval - $2.dval;
```

More Yacc Declarations

Token names & types

```
%token BHTML BHEAD BTITLE BBODY P BR LI
%token EHTML EHEAD ETITLE EBODY
%token <sval> TEXT
```

Type of yylval (if any)

Nonterm names & types

```
%type  <obj> page head title
%type  <obj> words list item items
```

Start sym

```
%start page
```

"Calculator" example
From http://byaccj.sourceforge.net/

On this & next 3 slides, some details may be missing or wrong, but the big picture is OK

```
%{
  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 */
%%
...
```

```
...
  /* 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; }
 ;

%%
...
```

input is one expression per line; output is its value

Ambiguous grammar; prec/assoc decls are a (smart) hack to fix that.

```
%%
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'; //As in classic YACC example
    } else return 0;
  s = st.nextToken();
  try {
    d = Double.valueOf(s);  /*this may fail*/
    yylval = new ParserVal(d.doubleValue());
    tok = NUM; }
  catch (Exception e) {
    tok = s.charAt(0);/*if not float, return char*/
  }
  return tok;
}
```

NOT using lex; barehanded lexer with same interface

token code via *return*

value via *yylval*

See slide 20

```
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();
}
```

Lex and Yacc

More Details

```
# set following 3 lines to the relevant paths on your system
JFLEX  = ~ruzzo/src/jflex-1.4.3/jflex-1.4.3/bin/jflex
BYACCJ = ~ruzzo/src/byaccj/yacc.macosx
JAVAC  = javac

LEXDEBUG = 0 # set to 1 for token dump

# targets:

run: Parser.class
	java Parser $(LEXDEBUG) test.ratml

Parser.class: Yylex.java Parser.java Makefile test.ratml
	$(JAVAC) Parser.java

Yylex.java: jratml.flex
	$(JFLEX) jratml.flex

Parser.java: jratml.y
	$(BYACCJ) -J jratml.y

clean:
	rm -f *~ *.class *.java
```

Makefile:

Not required, but convenient

General form

```
A: B C
(tab) D
```

Means A depends on B & C and is built by running D

Parser "states"

Not exactly elements of PDA's "Q", but similar

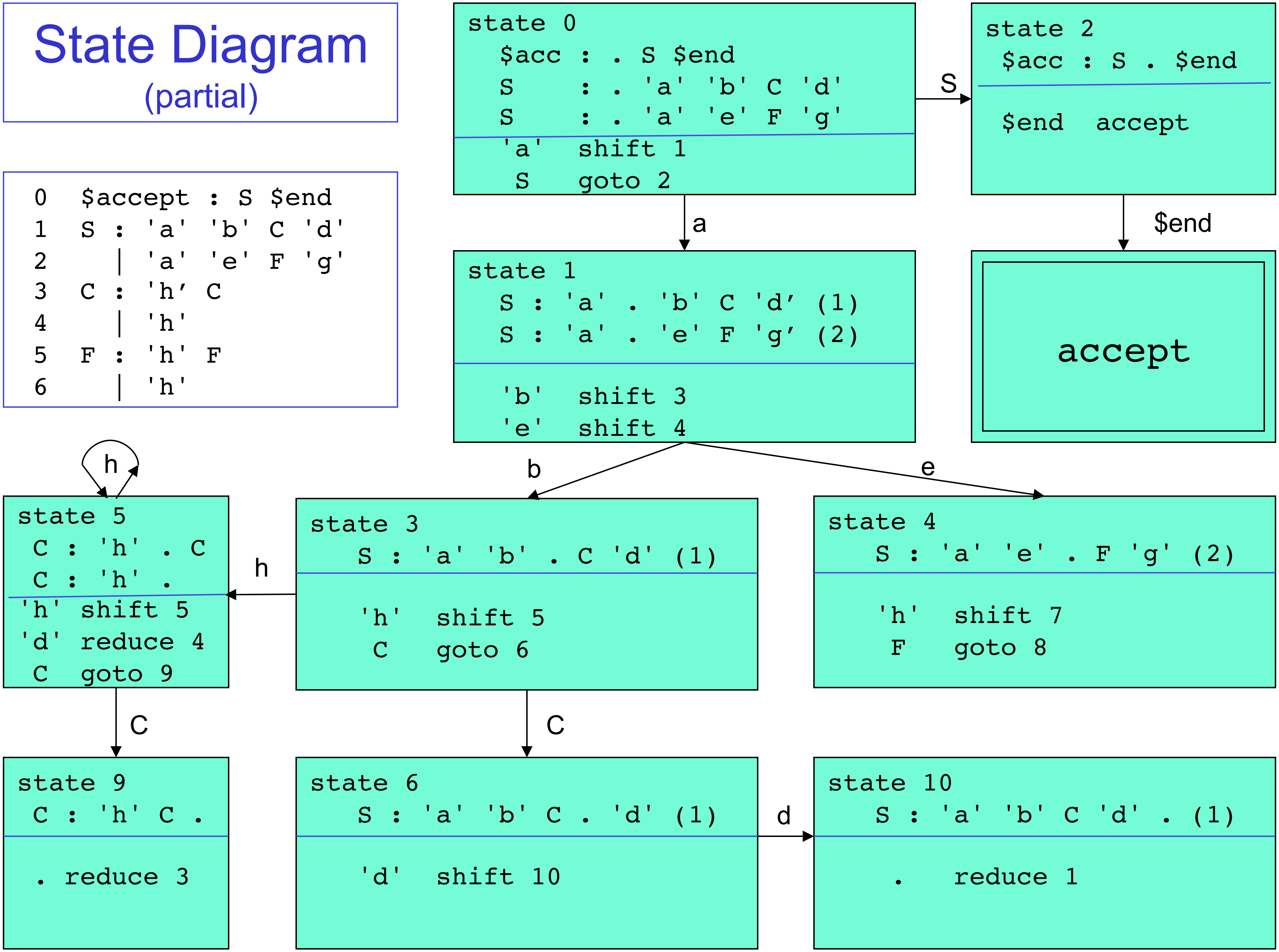
A yacc "state" is a set of "dotted rules" – rules in G with a "dot" (or "_") somewhere in the right hand side.

In a state, "$A \to \alpha\_\beta$" means this rule, up to and including α is *consistent with input seen so far;* next terminal in the input must derive from the *left end* of some such β. E.g., before reading any input, "$S \to \_ \beta$" is consistent, for every rule $S \to \beta$ " (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

State Diagram
(partial)

```
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'
```



Yacc Output: Same 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
```

Yacc In Action

PDA stack: alternates between "states" and symbols from ($V \cup \Sigma$).

```
initially, push state 0
while not done {
   let S be the state on top of the stack;
   let i in Σ be the next input symbol;
   look at the action defined in S for i:
      if "accept", halt and accept;
      if "error", halt and signal a syntax error;
      if "shift to state T", push i then T onto the stack;
      if "reduce via rule r (A → α )", then:
         pop exactly 2*|α| symbols
           (the 1st, 3rd, ... will be states, and
            the 2nd, 4th, ... will be the letters of α);
         let T = the state now exposed on top of the stack;
         T's action for A is "goto state U" for some U;
         push A, then U onto the stack.
}
```

Implementation note: given the tables, it’s deterministic, and fast -- just table lookups, push/pop.

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 here & parse table on slide 36

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

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		

Q: Do you have any advice for up-and-coming programmers?

A: ... One more piece of advice – take a theoretician to lunch...

From the end of a 2008 interview with Steve Johnson, creator of yacc

http://www.techworld.com.au/article/252319/a-z_programming_languages_yacc