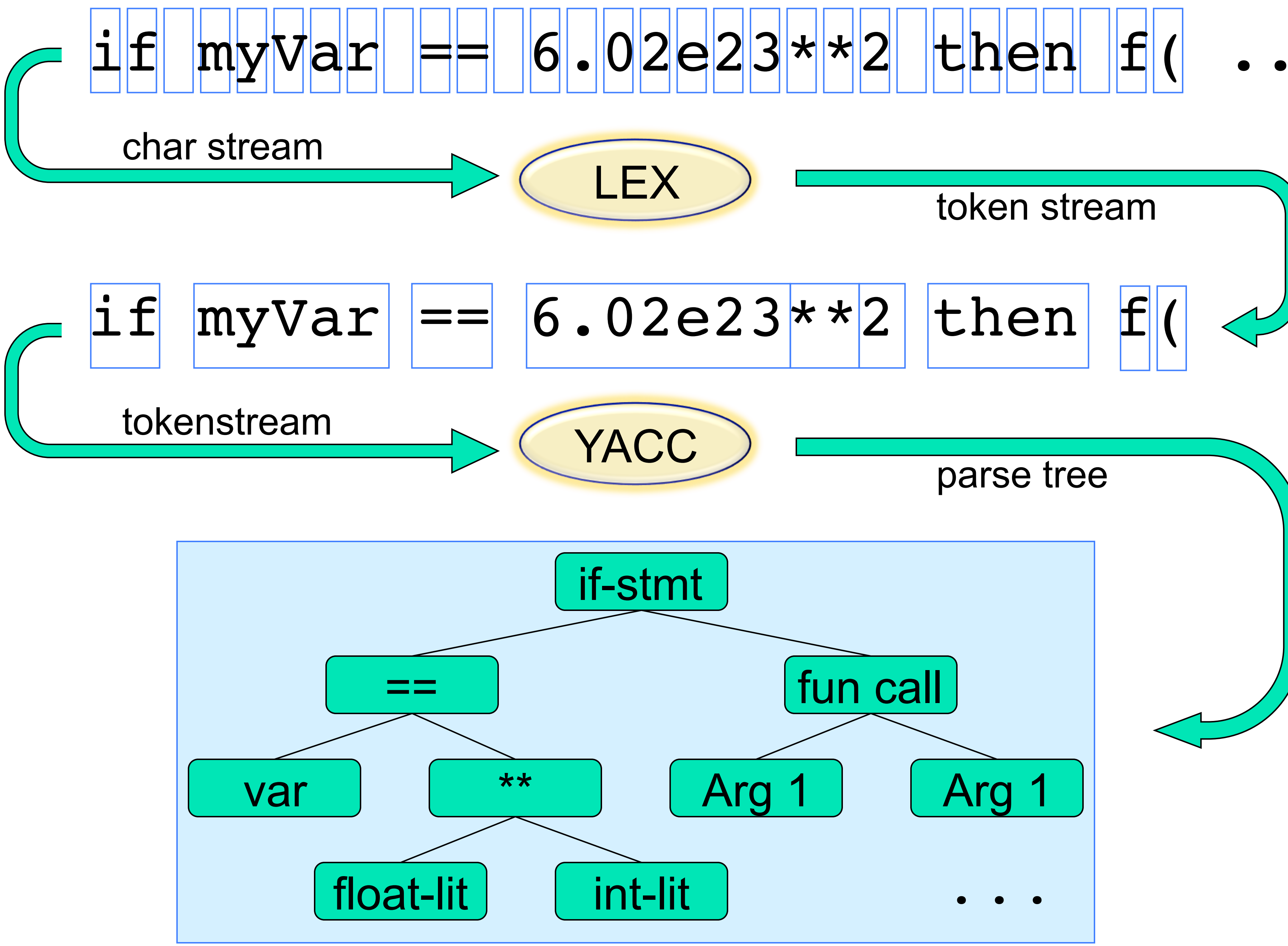
Lex and Yacc

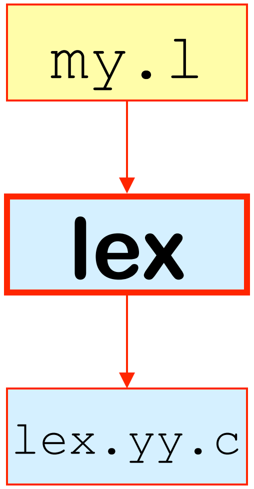
A Quick Tour



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 1
float-lit
int-lit
. . .

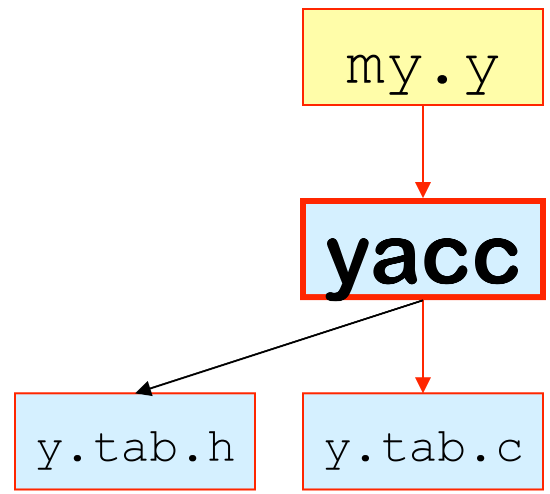
Lex (& Flex): A Lexical Analyzer Generator

- Input:
  - Regular exprs defining "tokens"
  - Fragments of C decls & code
- Output:
  - A C program "lex.yy.c"
- Use:
  - Compile & link with your main()
  - Calls to yylex() read chars & return successive tokens.



Yacc (& Bison & Byacc…): A Parser Generator

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



Lex Input: "mylexer.l"

```
%{
    #include …
    int myglobal;
    …
%}
%%
[a-zA-Z]+     {handleit(); return 42; }
[ \t\n]       {; /* skip whitespace */}
…
%%
void handleit() {…}
…
```

Declarations: To front of C program

Token code

Rules and Actions

Subroutines: To end of C program

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]

Yacc Input: “expr.y”

S → E
E → E+n | E-n | n

C Decls

```
%{
    #include …
%}
```

→ y.tab.c

Yacc Decls

```
%token NUM VAR
```

→ y.tab.h

Rules and Actions

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

Subrs

```
…
```

→ y.tab.c

Expression lexer: “expr.l”

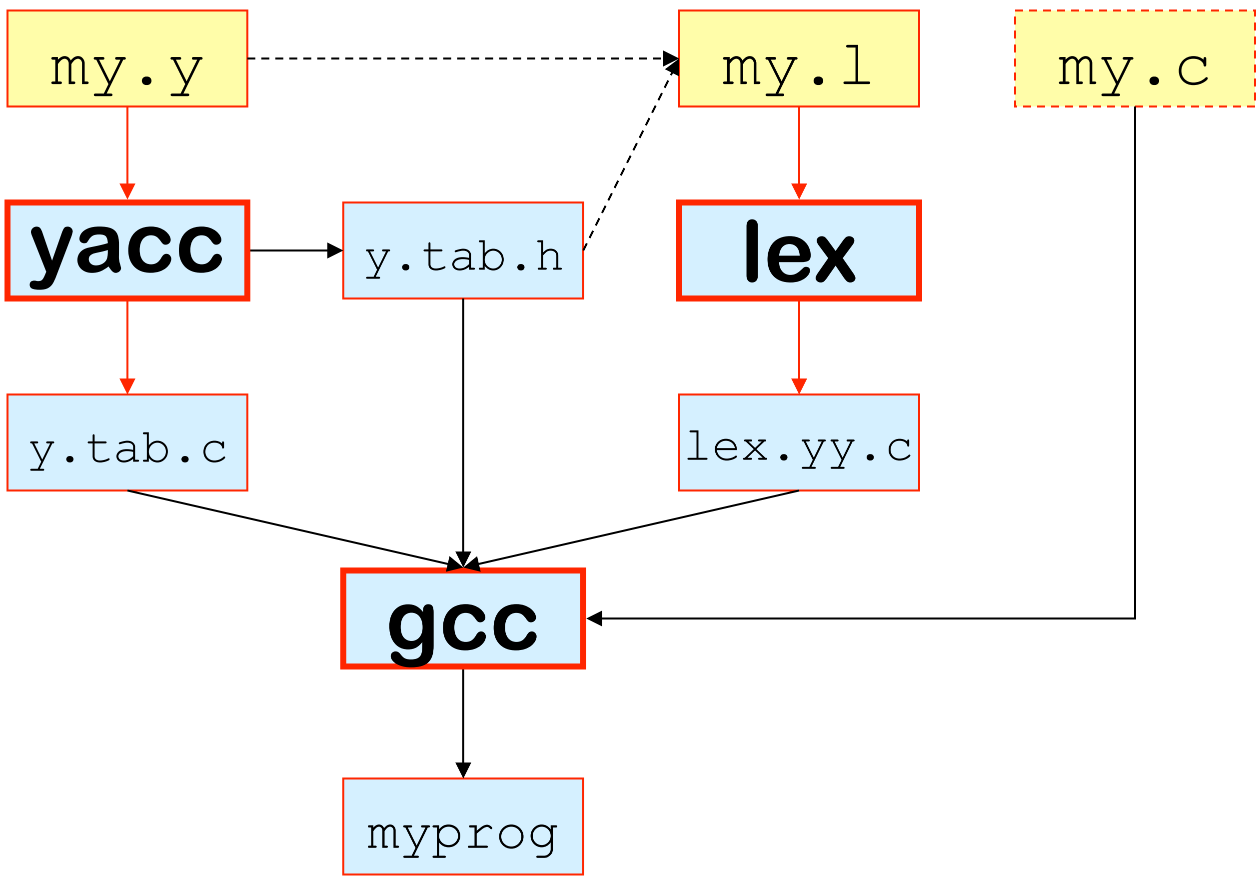
```
%{
#include "y.tab.h"

%}
%%
[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;}
```

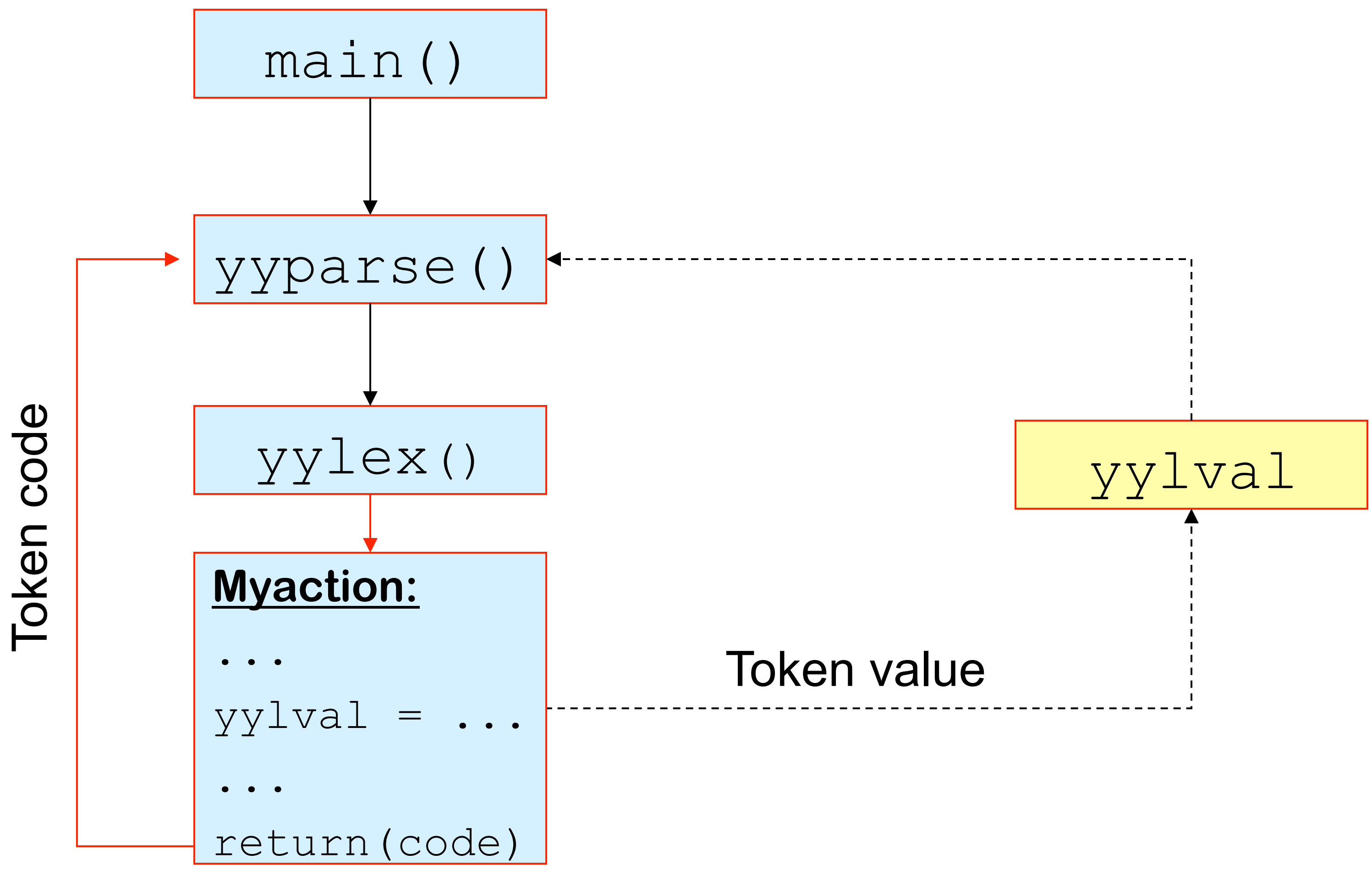
y.tab.h:

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

Lex/Yacc Interface: Compile Time



Lex/Yacc Interface: Run Time



Some C Tidbits

Enums

```
enum kind {
   title_kind, para_kind};
typedef struct node_s{
    enum kind k;
    struct node_s
      *lchild,*rchild;
    char *text;
  } node_t;
node_t root;
root.k = title_kind;
if(root.k==title_kind){…}
```

Malloc

```
root.rchild = (node_t*)
 malloc(sizeof(node_t));
```

Unions

```
typedef union _{
      double d;
      int i;
 } YYSTYPE;
extern YYSTYPE yylval;
yylval.d = 3.14;
yylval.i = 3;
```

More Yacc Declarations

Type of yylval

```
%union {
     node_t *node;
     char   *str; }
```

Token names & types

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

Nonterm names & types

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

Start sym

```
%start page
```

Makefile

```
CC = gcc -DYYDEBUG=0
test.out: test.html parser
        parser < test.html > test.out
        cat test.out
        #diff test.out test.out.std

parser: lex.yy.o y.tab.o
        $(CC) -o parser y.tab.o lex.yy.o

lex.yy.o: lex.yy.c y.tab.h

lex.yy.o y.tab.o: html.h

lex.yy.c: html.l y.tab.h Makefile
        lex html.l

y.tab.c y.tab.h: html.y Makefile
        yacc -dv html.y

# "make clean" removes all rebuildable files.
clean:
rm -f lex.yy.c lex.yy.o y.tab.c y.tab.h y.tab.o y.output \
       parser test.out
```

The classic infix calculator

```
%{
   #define YYSTYPE double
   #include <math.h>
   #include <stdio.h>
   int yylex (void);
   void yyerror (char const *);
%}

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

```
%% /* The grammar follows.  */
input:  /* empty */
      | input line

line:   '\n'
      | exp '\n'  { printf ("\t%.10g\n", $1);  }
;

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;          }
;
%%
```

Input: one expression per line
Output: its value

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

“Calculator” example

From http://byaccj.sourceforge.net/

Skim this & next 3 slides; details may be 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

```
%%
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;
}
```

```java
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

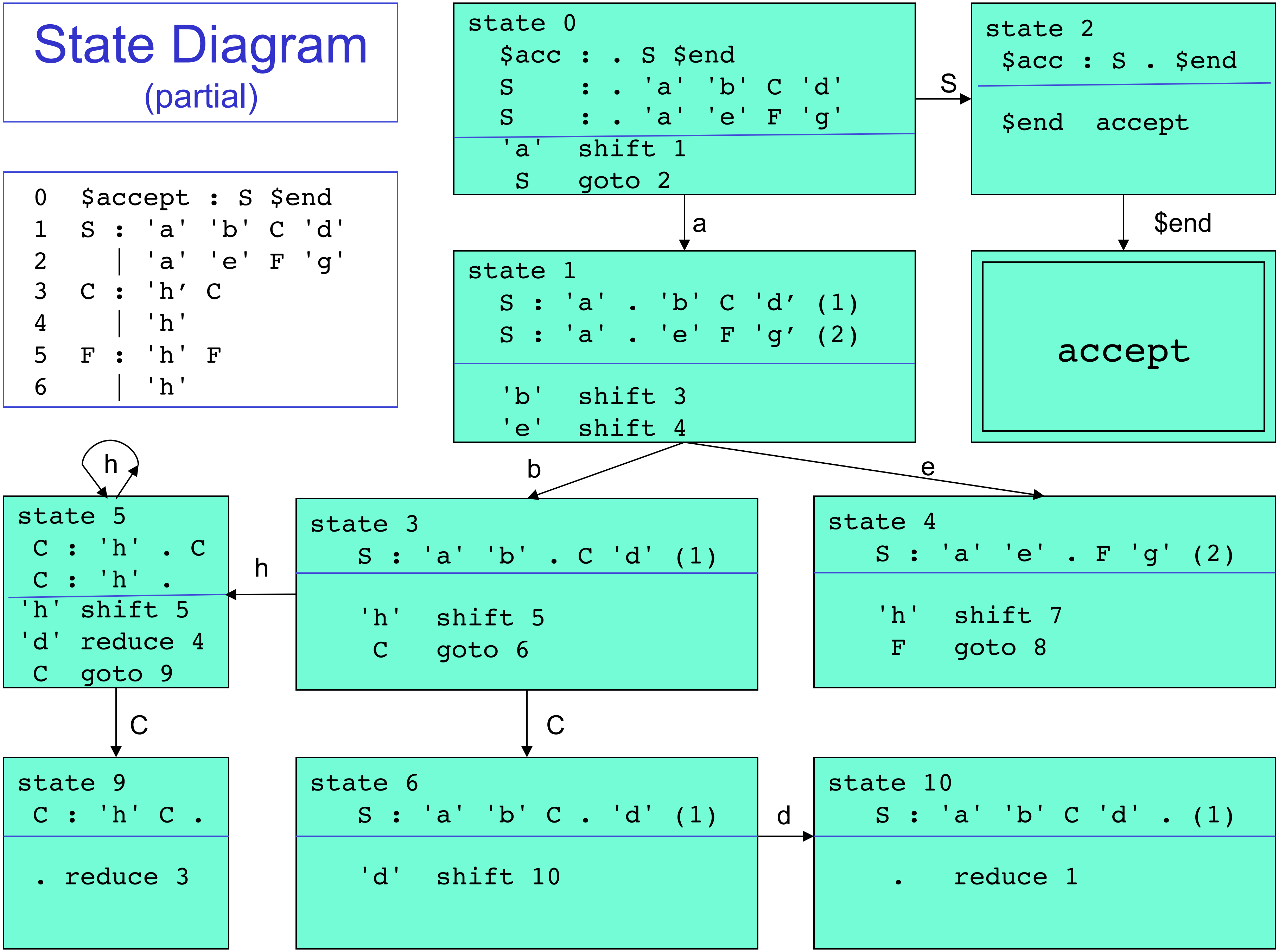
A yacc "state" is a set of "dotted rules" – rules in G with a "dot” (or “_”) somewhere in the right hand side.

In a state, "$A \rightarrow \alpha\_\beta$" 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, "$S \rightarrow \_\ \beta$" is consistent, for every rule $S \rightarrow \beta$ " (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 be the next input symbol (i in Σ);
   look at the 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

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		

More Lex: "Start States"

This lexer has two "states":
- NORMAL: input echoed to stdout
- COMMENT: all chars → "X".

Toggle on /* */ comment delimiters.

```
%{
%}
%s  CMNT NRML
%%
%{
   BEGIN NRML;
%}
<NRML> .        { printf("%s",yytext);    /* action equiv to ECHO        */}
<NRML> "/*"     { ECHO; BEGIN CMNT; }

<CMNT> "*/"     { ECHO; BEGIN NRML;  }
<CMNT> .        |
<CMNT> \n       { printf("X");            /* blot out comment text       */}
%%
```

Declare states

Start in NORMAL state

Switch states

State Names

Patterns

Actions

Lex and Yacc

More Details