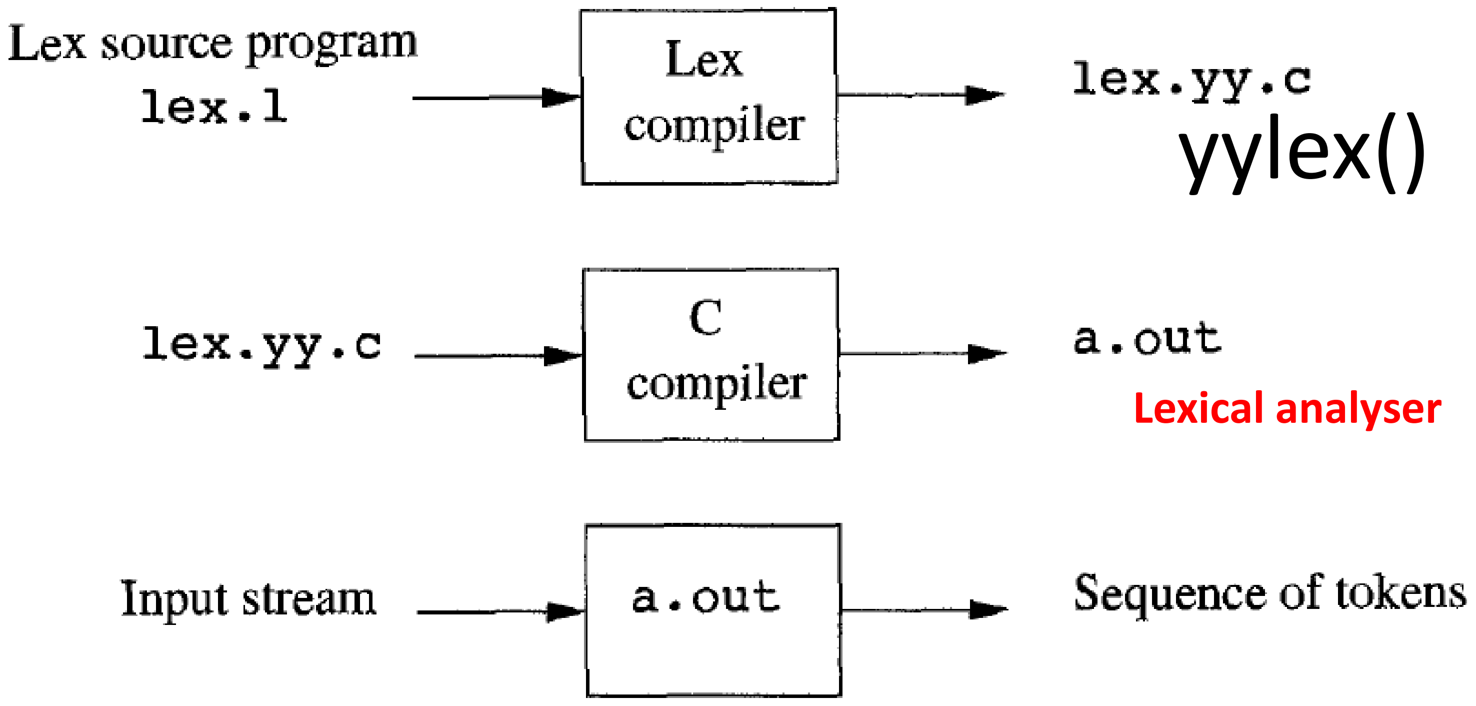
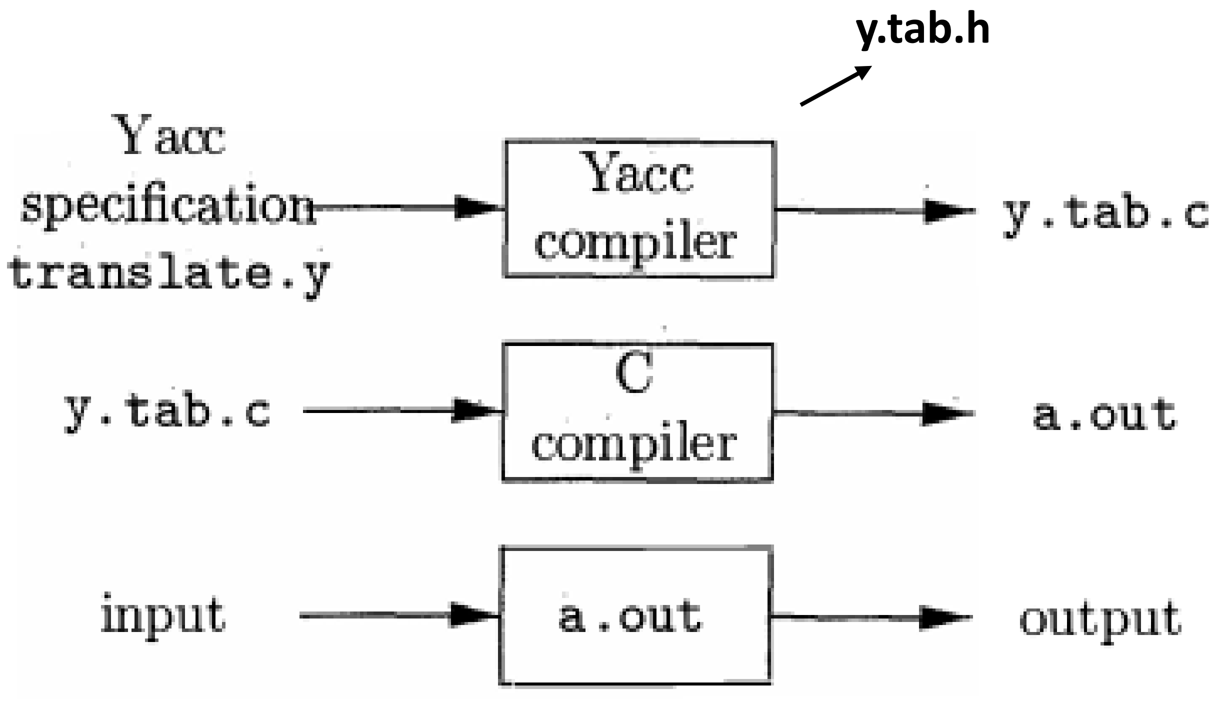
YACC/BISON

Recall Lex – Generates lexical analyser



YACC/Bison – Generates Parser, and ++



Includes the parser yyparse()

Outcome of Parser

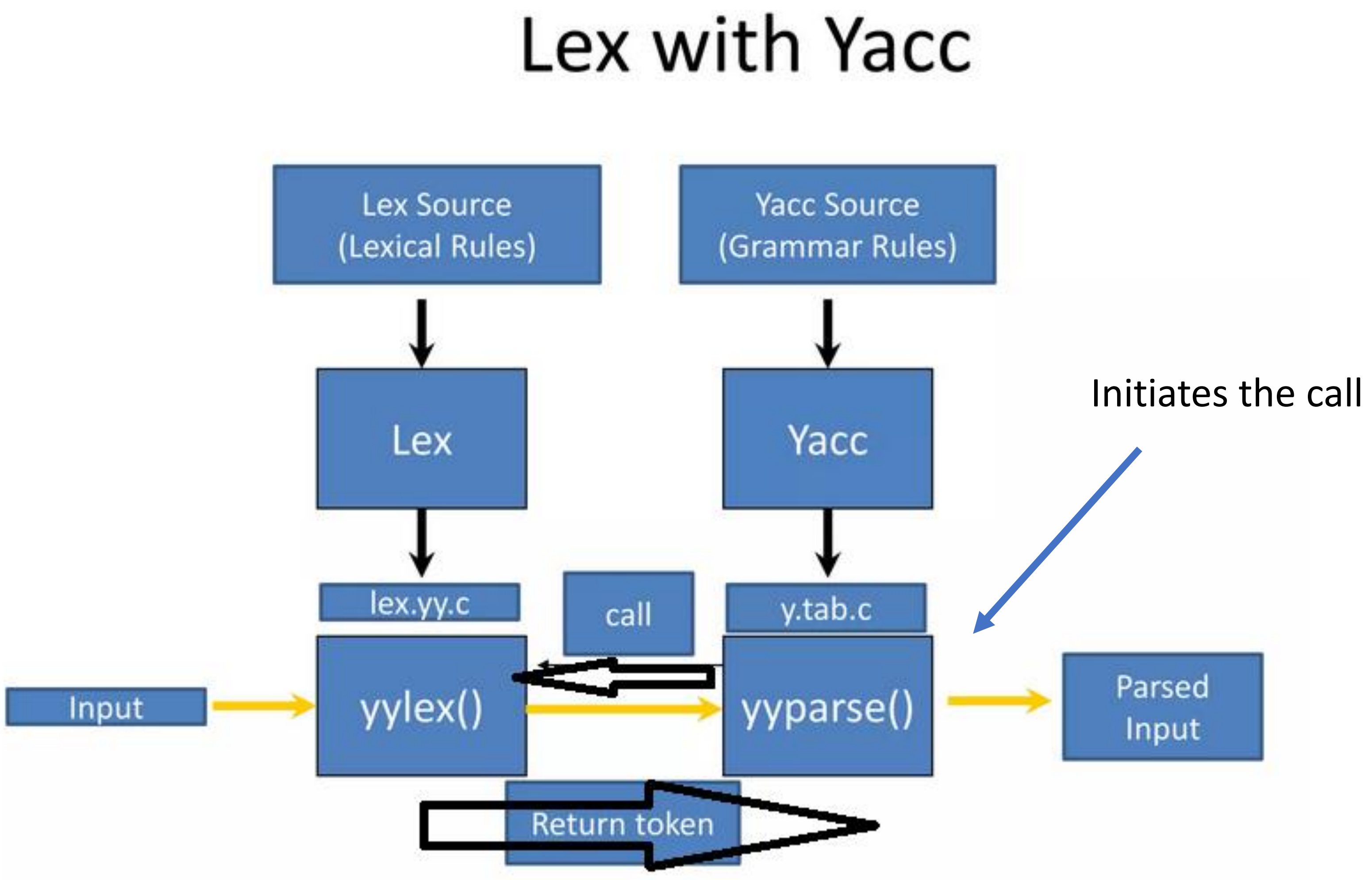
Parse Tree!

Sequence of productions!

Something else!!

Implements bottom up parser (LALR)

How to execute YACC, with Lex



Lex – example

Macro definitions of terminals

Auxiliary declarations

```
%{
#include<stdio.h>
#define IF 1
#define ELSE 2
#define NUM 3
#define OP 4
#define ERR 5

%}
```

Regular expressions

```
/* Declarations*/
%%
if      return (IF);
else    return (ELSE);
[0-9]+  return(NUM);
[+-]    return(OP);
.       return(ERR);
%%
```

Any char

Generates yylex()

Lex – example

Macro definitions of terminals/tokens

```
%{
#include<stdio.h>
#define IF 1
#define ELSE 2
#define NUM 3
#define OP 4
#define ERR 5

%}
```

Auxiliary declarations

y.tab.h keeps the macros for terminals/tokens

```
/* Declarations*/
%%
if      return (IF);
else    return (ELSE);
[0-9]+  return(NUM);
[+-]    return(OP);
.       return(ERR);
%%
```

Regular expressions

Any char

Generates yylex()

First compile with YACC

Structure of YACC program

```
declarations
%%
translation rules
%%
supporting C routines
```

Optional (declarations)

declarations:
(i) C declarations
(ii) YACC declarations

translation rules:
Grammar
(i) Productions rules
(ii) Semantic actions

supporting C routines:
Additional supporting functions
--- say main() etc

Structure of YACC program – Declaration

(a) C declarations
(i) declaration of **variable, functions**
(ii) inclusion of **header file**,
(iii)Defining **macro**

- Enclosed within %{ and %}

- Auxiliary declarations are copied as such by YACC to the output y.tab.c file.
    - Not processed by the YACC tool.

(b) YACC declarations

```
declarations
%%
translation rules
%%
supporting C routines
```

Structure of YACC program – Declarations

(a) C declarations
(i) declaration of **variable, functions**
(ii) inclusion of **header file**,
(iii)Defining **macro**

- Enclosed within %{ and %}
- Auxiliary declarations are copied as such by LEX to the output lex.yy.c file.
  - Not processed by the LEX tool.

(b) YACC declarations

```
declarations
%%
translation rules
%%
supporting C routines
```

```
%{
  #include <stdio.h>
  int yyerror();
  int yylex();
%}

%union {int num;}
%start start
%token <num> DIGIT
%type <num> start expr

%%

start : expr '\n'  { printf("Expression value = %d",$1);}
        ;

expr:  expr '+' expr           {$$ = $1 + $3;}
       | expr '*' expr          {$$ = $1 * $3;}
       | '(' expr ')'           {$$ = $2;}
       | DIGIT                  {$$ = $1;}
       ;

%%
```

Structure of YACC program– Declarations

1. Start: Specifies **start** non-terminal
2. Token: Specifies expected terminals/tokens from Lex
   - Tokens which gets **multiple lexemes** – identifies, numbers etc
   - Generates **macros in y.tab.h**
   - No need to specify the literal tokens such as +, - etc

- Type: Specifies the Non-terminals

```
%union {int num;}
%start start
%token <num> DIGIT
%type <num> start expr
```

Structure of YACC program– Translation rules
Grammar

Each rule consists of a
(i) grammar production and
(ii) the associated semantic action.
A set of productions that we have been writing:

$$\langle\text{head}\rangle \rightarrow \langle\text{body}\rangle_1 \mid \langle\text{body}\rangle_2 \mid \cdots \mid \langle\text{body}\rangle_n$$

$$\begin{array}{lcll} \langle\text{head}\rangle & : & \langle\text{body}\rangle_1 & \{\ \langle\text{semantic action}\rangle_1\ \} \\ & \mid & \langle\text{body}\rangle_2 & \{\ \langle\text{semantic action}\rangle_2\ \} \\ & & \cdots & \\ & \mid & \langle\text{body}\rangle_n & \{\ \langle\text{semantic action}\rangle_n\ \} \\ & ; & & \end{array}$$

declarations
%%
translation rules
%%
supporting C routines

C code snippet

- **Head of the production** is followed by a **colon**, then **body** of the production
- **Multiple right side** may be separated by **|**
- **Actions** associated with each rule are entered within **{}**

1. **Literal terminals** are quoted ‘+’, ‘-’
2. **Separate productions** with **;**
3. **unquoted strings** of letters and digits not declared to be tokens are taken to be **nonterminals**

Structure of YACC program– Translation rules
Grammar

Semantic rules

```
statements: statement
     {printf("statement");}
              | statement statements
     {printf("Statements\n");}
statement: identifier '+' identifier
     {printf("plus\n");}
```

;

- **Head of the production** is followed by a **colon**, then **body** of the production
- **Multiple right side** may be separated by **|**
- **Actions** associated with each rule are entered within **{}**

1. **Literal terminals** are quoted '+', '-'
2. **Separate productions** with **;**
3. **unquoted strings** of letters and digits not declared to be tokens are taken to be **nonterminals**

Structure of YACC program– Translation rules

1. **Literal terminals** are quoted ‘+’, ‘-’
2. **Separate productions** with **;**
3. **unquoted strings** of letters and digits not declared to be tokens are taken to be **nonterminals**

```
%union {int num;}
%start line
%token <num> DIGIT
%type <num> line expr term
```

Semantic rules

```
%%
line    : expr ';'      {printf("done %d\n",$1);}
        ;
expr    : term
        | expr '+' term        {printf("add %d %d\n",$1,$3); $$=$1+$3;}
        | expr '-' term    {printf("sub %d %d\n",$1, $3); $$=$1-$3;}
        ;
term        : DIGIT   {printf("digit %d\n",$1); $$=$1;}
        ;

%%
```

- **Head of the production** is followed by a **colon**, then **body** of the production
- **Multiple right side** may be separated by **|**
- **Actions** associated with each rule are entered within **{}**

Semantic actions and attributes

Each of the Terminal and Non-terminals has an **attribute**, called ***val***

$$
\begin{array}{lll}
\langle \text{head} \rangle & : \; \langle \text{body} \rangle_1 & \{ \; \langle \text{semantic action} \rangle_1 \; \} \\
 & | \; \langle \text{body} \rangle_2 & \{ \; \langle \text{semantic action} \rangle_2 \; \} \\
 & \quad \cdots & \\
 & | \; \langle \text{body} \rangle_n & \{ \; \langle \text{semantic action} \rangle_n \; \} \\
 & ; &
\end{array}
$$

	Production	Semantic Rules
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

LR parsing

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	accept

Apply the action, as soon as you reduce!

LR parsing

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \,\$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \,\$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2 \,\$$	shift
$\$\,T *$	$\mathbf{id}_2 \,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

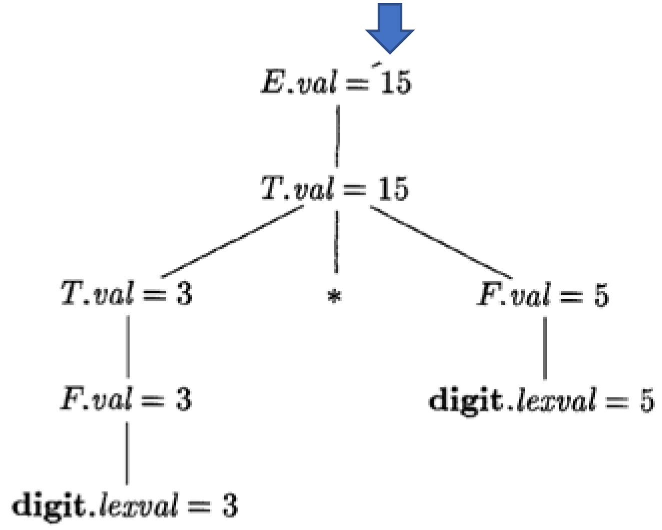
PRODUCTION	SEMANTIC RULES
2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
3) $E \to T$	$E.val = T.val$
4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \to F$	$T.val = F.val$
6) $F \to (E)$	$F.val = E.val$
7) $F \to \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

Done by Lex

Apply the action, as soon as you reduce!

Input string: **3 * 5**

Output of the parser



Structure of YACC program– Translation rules

- A Yacc **semantic action** is a sequence of **C statements**.
- In a semantic action, the **symbol $$** refers to the **attribute** value associated with the nonterminal of the **head**
- **$i** refers to the value associated with the **i^th grammar symbol** (terminal or nonterminal) of the **body**.
- The **semantic action** is performed when ever we **reduce by the associated production**,
  - Normally the semantic action computes a value for $$ in terms of the $i's.
- Default action
  $$=$1

```
⟨head⟩   :   ⟨body⟩1   { ⟨semantic action⟩1 }
         |   ⟨body⟩2   { ⟨semantic action⟩2 }
               ...
         |   ⟨body⟩n   { ⟨semantic action⟩n }
         ;
```

Structure of YACC program– Translation rules

```
%%

start : expr '\n'  { printf("Expression value = %d",$1);}
        ;

expr:  expr '+' expr            {$$ = $1 + $3;}
        | expr '*' expr          {$$ = $1 * $3;}
        | '(' expr ')'           {$$ = $2;}
        | DIGIT                  {$$ = $1;}
        ;

%%
```

Done by Lex

	PRODUCTION	SEMANTIC RULES
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to \textbf{digit}$	$F.val = \textbf{digit}.\text{lexval}$

Recall:
Structure of YACC program – Declarations

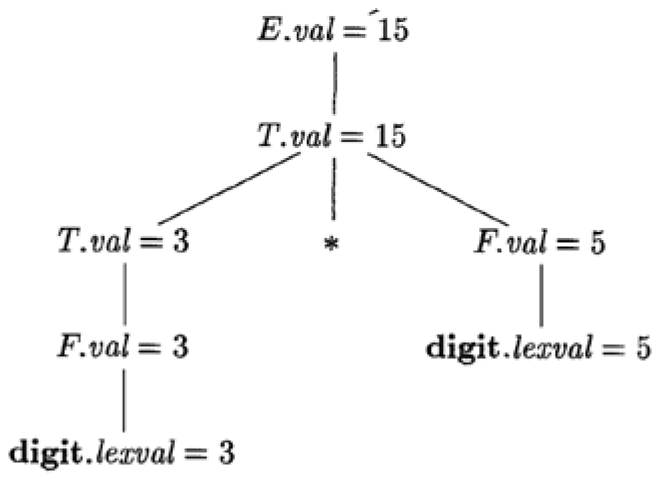
- Specifies different **attribute types** that **lexical analyzer** may return for **tokens**
- Attribute values

```
%union {int num; char id;}
%start line
%token print
%token exit_command
%token <num> number
%token <id> identifier
%type <num> line exp term
%type <id> assignment
```

The value of the token **number** gets stored in the variable **num**

- Specifies the **attribute type** of **nonterminals.**
- **Type-checking** is performed

Input string: **3 * 5**



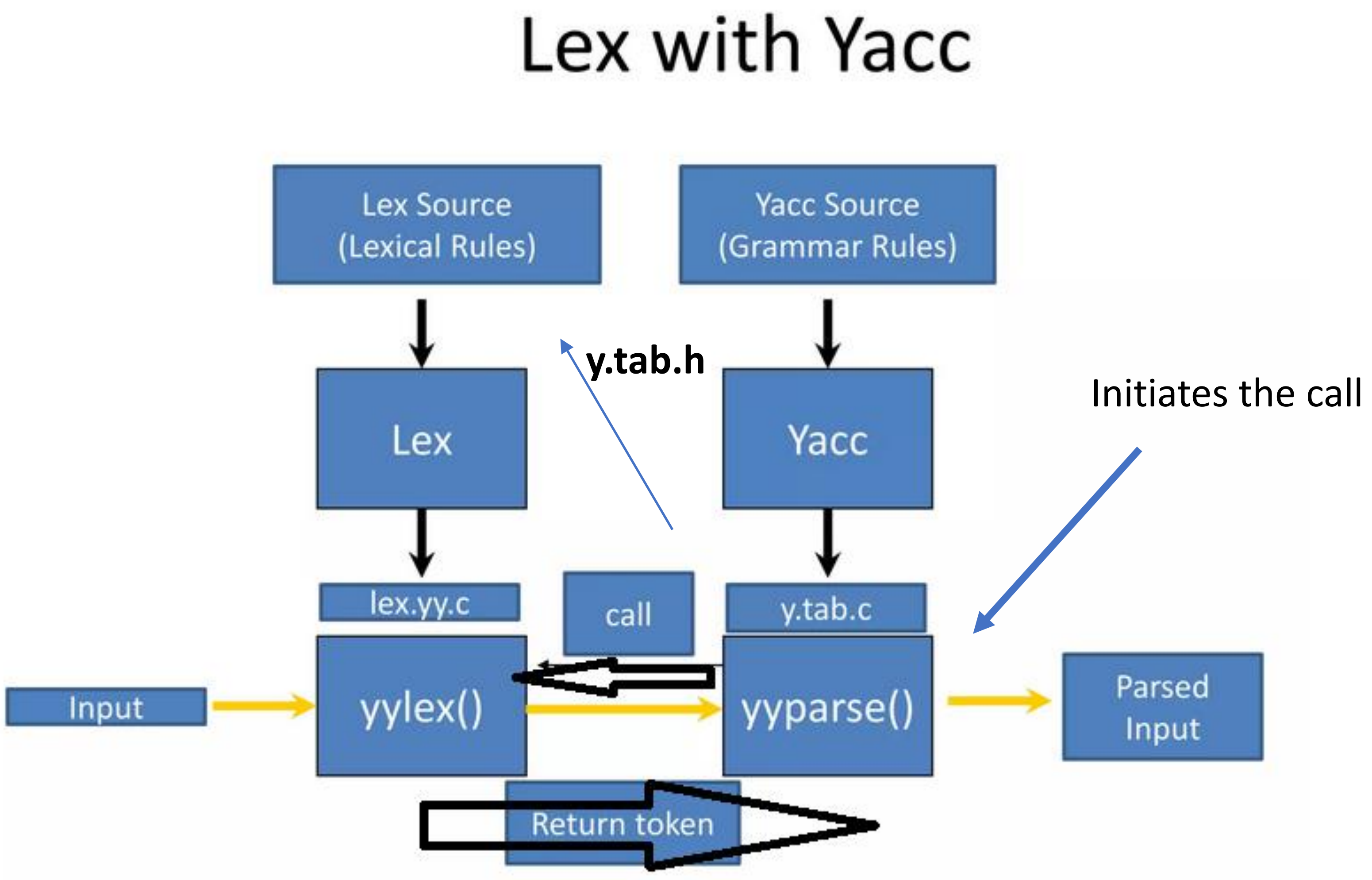
Structure of YACC program– Supporting C functions

```
declarations        ⇐ Optional
%%
translation rules
%%
supporting C routines
```

Additional supporting functions
--- say **main()** etc

- **YACC generates C code** for the **production rules** specified in the rules section and places this code into a single function called **yyparse()**.
- **In addition** to this YACC generated code, the programmer may wish to add **his own code** to the **y.tab.c** file.
  - **Symbol table** implementation
  - **Functions** associated with **semantic actions**
- The **auxiliary functions section** allows the programmer to achieve this.

How to execute YACC, with Lex



Example 1

```
admins@admin:~/bivas$ yacc -d trans_v1.y
admins@admin:~/bivas$ lex exp_v1.l
admins@admin:~/bivas$ gcc lex.yy.c y.tab.c
admins@admin:~/bivas$ ./a.out
5+8;
digit 5
digit 8
add 5 8
done 13

11*6;
error
admins@admin:~/bivas$
```

```
%{
#include<stdio.h>
#include<stdlib.h>
int yylex();
void yyerror();
%}

%union {int num;}
%start line
%token <num> DIGIT
%type <num> line expr term

%%
line    : expr ';'      {printf("done %d\n",$1);}
        ;
expr    : term
        | expr '+' term         {printf("add %d %d\n",$1,$3); $$=$1+$3;}
        | expr '-' term     {printf("sub %d %d\n",$1, $3); $$=$1-$3;}
        ;
term        : DIGIT    {printf("digit %d\n",$1); $$=$1;}
        ;

%%


int main()
{

        yyparse();
        return 1;
}
void yyerror()
{
        printf("error\n");
}
```

As soon as yyparse() encounters input that **does not match** any known grammatical productions,
it calls the **yyerror()** function

trans_v1.y

Example 1

exp_v1.l

```
%{
#include "y.tab.h"
void yyerror();
int yylex();
%}
%%
[0-9]+                  {yylval.num = atoi(yytext); return DIGIT;}
[+-;]                       {return yytext[0];}
.                       {printf("unexpected character");}

%%
int yywrap (void) {return 1;}
```

Example 1

```
admins@admin:~/bivas$ yacc -d trans_v1.y
admins@admin:~/bivas$ lex exp_v1.l
admins@admin:~/bivas$ gcc lex.yy.c y.tab.c
admins@admin:~/bivas$ ./a.out
5+8;
digit 5
digit 8
add 5 8
done 13

11*6;
error
admins@admin:~/bivas$
```

Example 1

```
admins@admin:~/bivas$ ./a.out
18+yu;
digit 18
unexpected characterunexpected charactererror
admins@admin:~/bivas$
```

Example 2

```
admins@admin:~/bivas$ yacc -d calc.y
admins@admin:~/bivas$ lex calc.l
admins@admin:~/bivas$ gcc lex.yy.c y.tab.c
admins@admin:~/bivas$ ./a.out
a=10;
print a;
Printing 10
b=17;
print b;
Printing 17
c=a+b;
print c;
Printing 27
exit
;
admins@admin:~/bivas$
```

Example 2

Calc.y

```
%{
void yyerror (char *s);
int yylex();
#include <stdio.h>     /* C declarations used in actions */
#include <stdlib.h>
#include <ctype.h>
int symbols[52];
int symbolVal(char symbol);
void updateSymbolVal(char symbol, int val);
%}

%union {int num; char id;}         /* Yacc definitions */
%start line
%token print
%token exit_command
%token <num> number
%token <id> identifier
%type <num> line exp term
%type <id> assignment
```

Calc.y

```
%%

/* descriptions of expected inputs     corresponding actions (in C) */

line    : assignment ';'                {;}
                | exit_command ';'              {exit(EXIT_SUCCESS);}
                | print exp ';'                 {printf("Printing %d\n", $2);}
                | line assignment ';'   {;}
                | line print exp ';'    {printf("Printing %d\n", $3);}
                | line exit_command ';' {exit(EXIT_SUCCESS);}
        ;

assignment : identifier '=' exp  { updateSymbolVal($1,$3); }
                        ;
exp     : term                  {$$ = $1;}
        | exp '+' term          {$$ = $1 + $3;}
        | exp '-' term          {$$ = $1 - $3;}
        ;
term    : number                {$$ = $1;}
                | identifier                    {$$ = symbolVal($1);}
        ;

%%                    /* C code */
```

```c
int computeSymbolIndex(char token)
{
	int idx = -1;
	if(islower(token)) {
		idx = token - 'a' + 26;
	} else if(isupper(token)) {
		idx = token - 'A';
	}
	return idx;
}

/* returns the value of a given symbol */
int symbolVal(char symbol)
{
	int bucket = computeSymbolIndex(symbol);
	return symbols[bucket];
}

/* updates the value of a given symbol */
void updateSymbolVal(char symbol, int val)
{
	int bucket = computeSymbolIndex(symbol);
	symbols[bucket] = val;
}

int main (void) {
	/* init symbol table */
	int i;
	for(i=0; i<52; i++) {
		symbols[i] = 0;
	}

	return yyparse ( );
}

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

Calc.l

Reading identifier

Reading number

```
%{
#include "y.tab.h"
void yyerror (char *s);
int yylex();
%}
%%
"print"                                {return print;}
"exit"                                 {return exit_command;}
[a-zA-Z]                               {yylval.id = yytext[0]; return identifier;}
[0-9]+                  {yylval.num = atoi(yytext); return number;}
[ \t\n]                 ;
[-+=;]                      {return yytext[0];}
.                       {ECHO; yyerror ("unexpected character");}

%%
int yywrap (void) {return 1;}
```

Example 2

```
admins@admin:~/bivas$ yacc -d calc.y
admins@admin:~/bivas$ lex calc.l
admins@admin:~/bivas$ gcc lex.yy.c y.tab.c
admins@admin:~/bivas$ ./a.out
a=10;
print a;
Printing 10
b=17;
print b;
Printing 17
c=a+b;
print c;
Printing 27
exit
;
admins@admin:~/bivas$
```

Example 2

text

```
a=15;
b=10;
c=a+b;
print c;
a=a-b;
print a;
print b;
exit;
```

```
admins@admin:~/bivas$ ./a.out<text
Printing 25
Printing 5
Printing 10
admins@admin:~/bivas$
```