Parser Generator

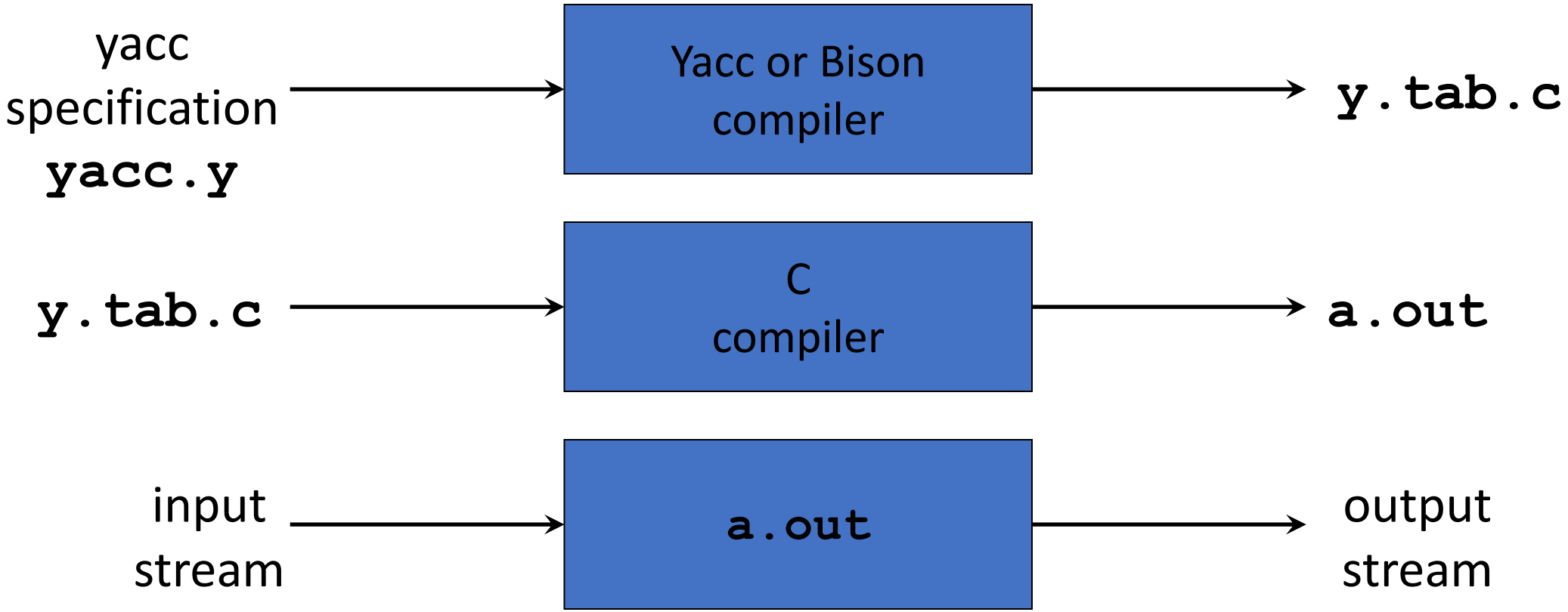
Parser Generator

- There is enough integration between the lexer and the parser
- Lexer has already been implemented using a tool LEX
- Parser could also be done with a tool so as to help speed up

ANTLR, Yacc, and Bison

- *ANTLR* tool generates LL(*k*) parsers
- *Yacc* (Yet Another Compiler Compiler) generates LALR(1) parsers
- *Bison* (Yacc improved)
- As bottom up parsers are preferred, YACC is preferred and used tool

Creating an LALR(1) Parser with Yacc/Bison



Lex v.s. Yacc

- Lex
  - Lex generates C code for a lexical analyzer, or scanner
  - Lex uses patterns that match strings in the input and converts the strings to tokens

- Yacc
  - Yacc generates C code for syntax analyzer, or parser.
  - Yacc uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree.

Yacc Specification

- A *yacc specification* consists of three parts:
  yacc declarations, and C declarations in **%{ %}**
  %%
  translation rules
  %%
  user-defined auxiliary procedures
- *Translation rules* are grammar productions and actions:
  $production_1$ { $semantic\ action_1$ }
  $production_2$ { $semantic\ action_2$ }
  …
  $production_n$ { $semantic\ action_n$ }

Writing a Grammar in Yacc

E → E+T | T

E : E '+' T { } | T { }

- Productions in Yacc are of the form

Nonterminal : tokens/nonterminals { *action* }
| tokens/nonterminals { *action* }
…
;

Grammar in YACC

- Tokens that are single characters can be used directly within productions, e.g. **'+'**
- Named tokens must be declared first in the declaration part using

  %token *TokenName*

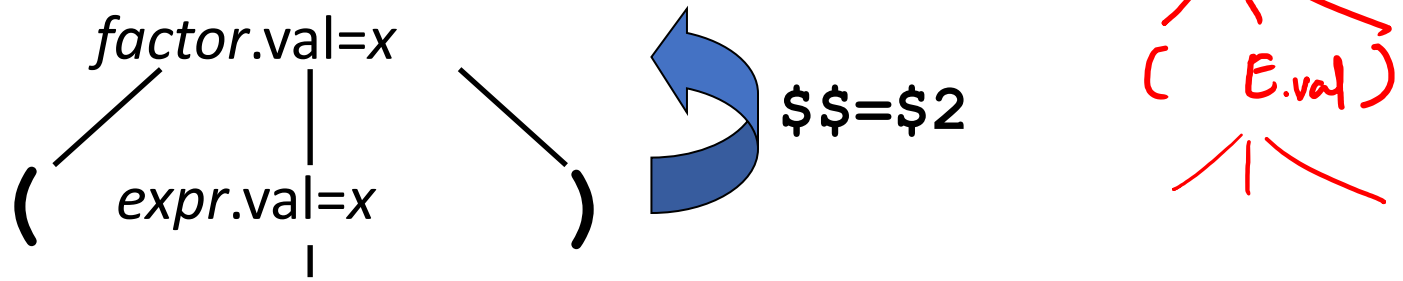
Synthesized Attributes

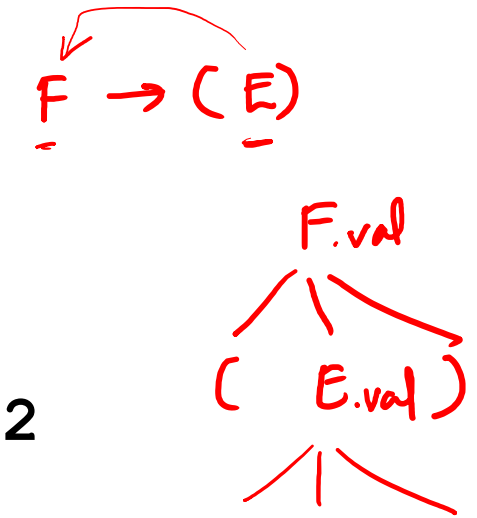
- Semantic actions may refer to values of the *synthesized attributes* of terminals and nonterminals in a production:

  $X : Y_1\ Y_2\ Y_3 \ldots Y_n \quad \{\ action\ \}$

  - **$$** refers to the value of the attribute of X
  - **$***i*** refers to the value of the attribute of Y_i
- For example

```
factor : '(' expr ')' { $$=$2; }
```





yyparse()

- Called once from main() [*user-supplied*]
- Repeatedly calls yylex() until done:
  - On syntax error, calls yyerror() [*user-supplied*]
  - Returns 0 if all of the input was processed;
  - Returns 1 if syntax error

Example:

```
int main() { return yyparse(); }
```

Definitions

- Information about tokens:
  - token names:
    - declared using '**%token**'
    - single-character tokens don't have to be declared
    - any name not declared as a token is assumed to be a nonterminal.
  - start symbol of grammar, using '**%start**' [optional]
  - operator info:
    - precedence, associativity
  - stuff to be copied verbatim into the output (e.g., declarations, **#include**s): enclosed in **%{** … **%}**

Definitions Section

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM
%start expr
```

YACC Rules

Grammar production	yacc rule
$A \rightarrow B_1\ B_2\ \ldots\ B_m$	$A : B_1\ B_2\ \ldots\ B_m$ { }
$A \rightarrow C_1\ C_2\ \ldots\ C_n$	| $C_1\ C_2\ \ldots\ C_n$ { }
$A \rightarrow D_1\ D_2\ \ldots\ D_k$	| $D_1\ D_2\ \ldots\ D_k$
	;

$A \rightarrow B_1 | B_2 | B_3$

- Rule RHS can have arbitrary C code embedded, within { … }. E.g.:

  A : B1 { printf("after B1\n"); x = 0; } B2 { x++; } B3 { } . . . $A \rightarrow B_1 B_2 B_3$ { }

- Left-recursion more efficient than right-recursion:
  - A : A x | … rather than A : x A | …

The Position of Rules

```
expr : expr '+' term     { $$ = $1 + $3; }
     | term              { $$ = $1; }
     ;
term : term '*' factor   { $$ = $1 * $3; }
     | factor            { $$ = $1; }
     ;
factor : '(' expr ')'    { $$ = $2; }
       | ID
       | NUM
       ;
```

E → E+T | T

T → T*F | F

F → (E) | id | num

Conflicts

- A conflict occurs when the parser has multiple possible actions in some state for a given next token.
- Two kinds of conflicts:
  - *shift-reduce conflict*:
    - The parser can either keep reading more of the input (“shift action”), or it can mimic a derivation step using the input it has read already (“reduce action”).
  - *reduce-reduce conflict*:
    - There is more than one production that can be used for mimicking a derivation step at that point.

Conflicts

✓ shift-reduce

- Two types:
  - shift-reduce [default action: *shift*]
  - reduce-reduce [default: *reduce with the first rule listed*]
- Removing conflicts:
  - specify operator precedence, associativity;
  - restructure the grammar
    - use **y.output** to identify reasons for the conflict.

Handling Conflicts

General approach:

- Iterate as necessary:
  1. Use “yacc -v” to generate the file **y.output**.
  2. Examine **y.output** to find parser states with conflicts.
  3. For each such state, examine the items to figure why the conflict is occurring.
  4. Transform the grammar to eliminate the conflict

Handling Conflicts

Reason for conflict	Possible grammar transformation
Ambiguity with operators in expressions	Specify associativity, precedence
Error action	Remove or eliminate offending error action
Semantic action	Remove the offending semantic action
Insufficient lookahead	Expand the nonterminal involved
Other	No idea

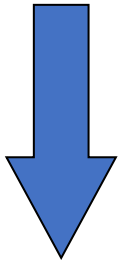
Specifying Operator Properties

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

- Binary operators: **%left**, **%right**, **%nonassoc**:

```
%left  '+'  '-'
%left  '*'  '/'
%right '^'
```

Operators in the same group have the same precedence



highest precedence

- Unary operators: **%prec**
  - Changes the precedence of a rule to be that of the token specified. E.g.:

```
%left '+'  '-'
%left  '*'  '/'
Expr: expr '+' expr
    | '–' expr   %prec '*'
    | …
```

Error Handling

- The “token” ‘**error**’ is reserved for error handling:
  - can be used in rules;
  - suggests places where errors might be detected and recovery can occur.

Example:

```
stmt  : IF '(' expr ')' stmt
      | IF '(' error ')' stmt
      | FOR …
      | …
```

Parser Behavior on Errors

- When an error occurs, the parser:
  - pops its stack until it enters a state where the token ‘error’ is legal;
  - then behaves as if it saw the token ‘error’
    - performs the action encountered;
    - resets the lookahead token to the token that caused the error.
  - If no ‘error’ rules specified, processing halts.

Controlling Error Behavior

- Parser remains in error state until three tokens are correctly read in and shifted
  - prevents cascaded error messages;
  - if an error is detected while parser in error state:
    - no error message is given;
    - input token causing the error is deleted.
- To force the parser to believe that an error has been fully recovered from:

  yyerrok;
- To clear the token that caused the error:

  yyclearin;

Placing ‘error’ tokens

- Close to the start symbol of the grammar:
  - To allow recovery without discarding all input.
- Near terminal symbols:
  - To help permit a small amount of input to be discarded due to an error.
  - Tokens like ‘**)**’, ‘**;**’, that follow non-terminals.
- Without introducing conflicts

Error Messages

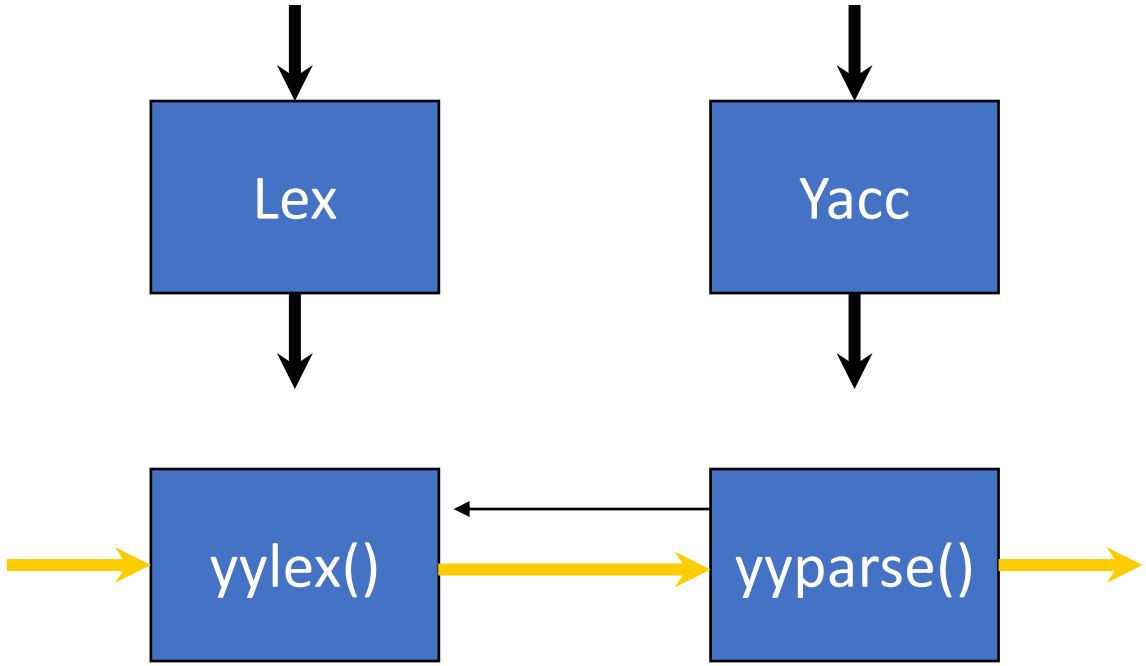
- On finding an error, the parser calls a function

  void yyerror(char *s) /* s points to an error msg */

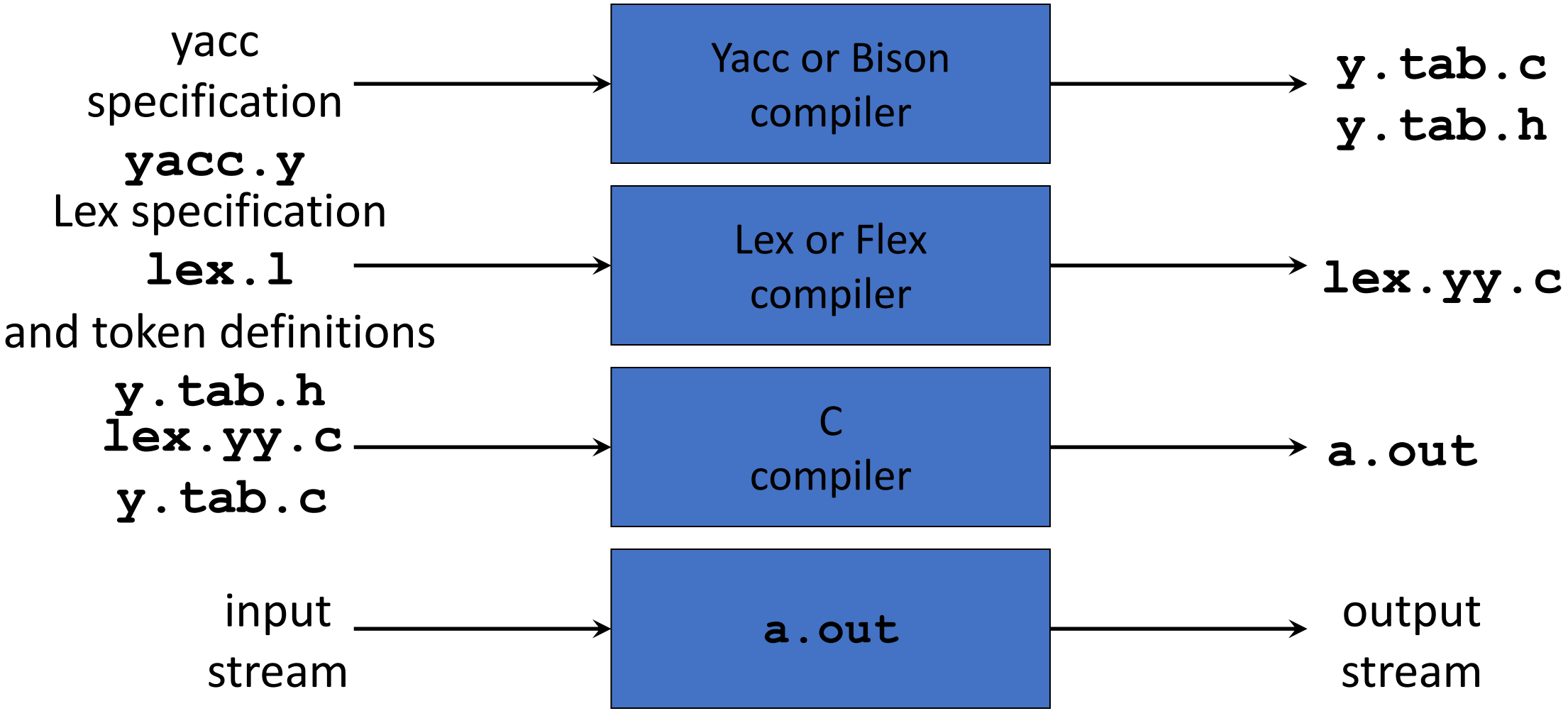
  - user-supplied, prints out error message.

- More informative error messages:
  - int yychar: token no. of token causing the error.
  - user program keeps track of line numbers, as well as any additional info desired.

Lex with Yacc



Combining Lex/Flex with Yacc/Bison



Summary

- Features of YACC and the need for YACC to perform parsing