Chapter 4
Syntax Analysis

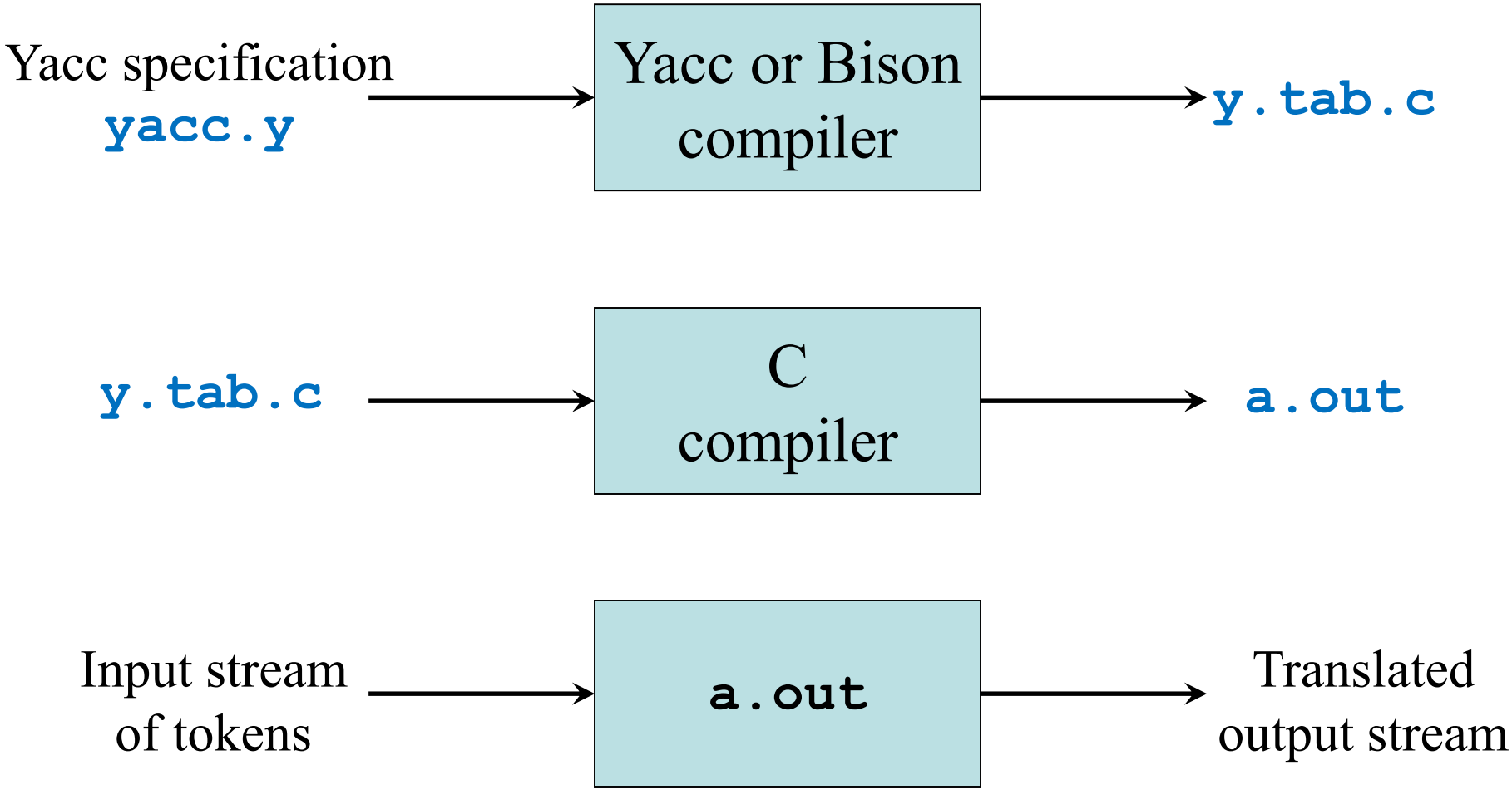
Part 3
PARSER GENERATORS

Parser Generator

- How a parser generator can be used to facilitate the construction of front end of a compiler
- Using LALR parser generator Yacc
  - Yet another compiler-compiler
  - Created by S. C. Johnson, early 1970s
  - Available as a command on UNIX
  - Bison: improved version of Yacc
- ANTLR tool
  - Generates LL parsers

Creating LALR Parser with Yacc/Bison

mansoori@shirazu.ac.ir



Yacc Specification

- A Yacc specification consists of three parts:

  Yacc declarations, and C declarations within `%{` and `%}`

  `%%`

  Translation rules

  `%%`

  User-defined auxiliary procedures

- Translation rules are productions with actions:

  $production_1$ { $semantic\ action_1$ }
  $production_2$ { $semantic\ action_2$ }
  …
  $production_n$ { $semantic\ action_n$ }

Writing a Grammar in Yacc

- Productions in Yacc have form of:

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

- Tokens that are single characters can be used directly within productions

```
factor : '(' expr ')'
```

- Named tokens must be declared first in declaration part as:

```
%token TokenName
```

```
%token DIGIT

factor : DIGIT
```

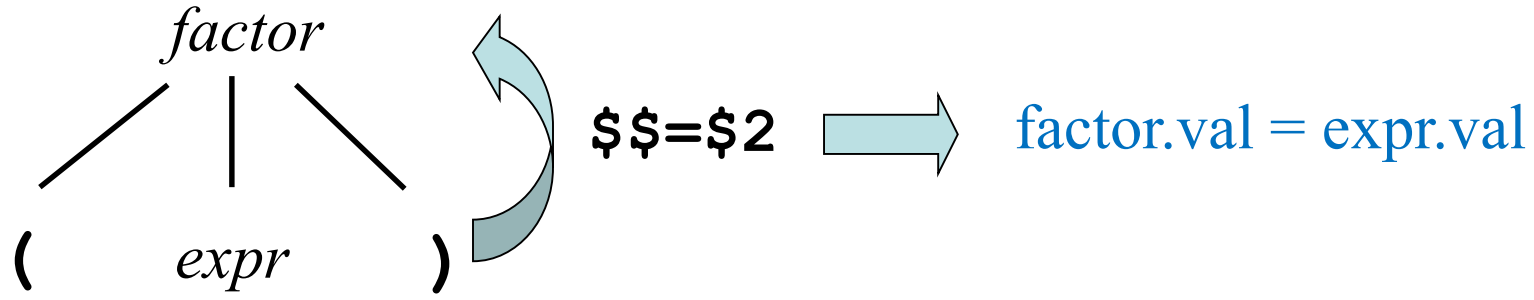
Synthesized Attributes

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

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

  - `$$` refers to value of attribute of X
  - `$i` refers to value of attribute of Y_i

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



Example1 Yacc

Also results in definition of
#define DIGIT xxx

```
%{ #include <ctype.h> %}
%token DIGIT
%%
line    : expr '\n'              { printf("%d\n", $1); }
        ;
expr    : expr '+' term          { $$ = $1 + $3; }
        | term                   { $$ = $1; }
        ;
term    : term '*' factor        { $$ = $1 * $3; }
        | factor                 { $$ = $1; }
        ;
factor  : '(' expr ')'           { $$ = $2; }
        | DIGIT                  { $$ = $1; }
        ;
%%
int yylex()
{ int c = getchar();
  if (isdigit(c))
  { yylval = c-'0';
    return DIGIT;
  }
  return c;
}
```

Attribute of **term** (child)

Attribute of **expr** (parent)

Attribute of **DIGIT** (stored in **yylval**)

Example of a very crude lexical analyzer invoked by parser

Dealing with Ambiguous Grammars

- By defining operator precedence levels and left/right associativity of operators, ambiguous grammars can be specified in Yacc:

  E → E+E | E-E | E*E | E/E | (E) | -E | **num**

- To define precedence levels and associativity in declaration part of Yacc:

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

Example2 Yacc


```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines   : lines expr '\n'        { printf("%g\n", $2); }
        | lines '\n'
        | /* empty */
        ;
expr    : expr '+' expr          { $$ = $1 + $3; }
        | expr '-' expr          { $$ = $1 - $3; }
        | expr '*' expr          { $$ = $1 * $3; }
        | expr '/' expr          { $$ = $1 / $3; }
        | '(' expr ')'           { $$ = $2; }
        | '-' expr %prec UMINUS  { $$ = -$2; }
        | NUMBER
        ;
%%
```

Double type for attributes and **`yylval`**

Example2 Yacc

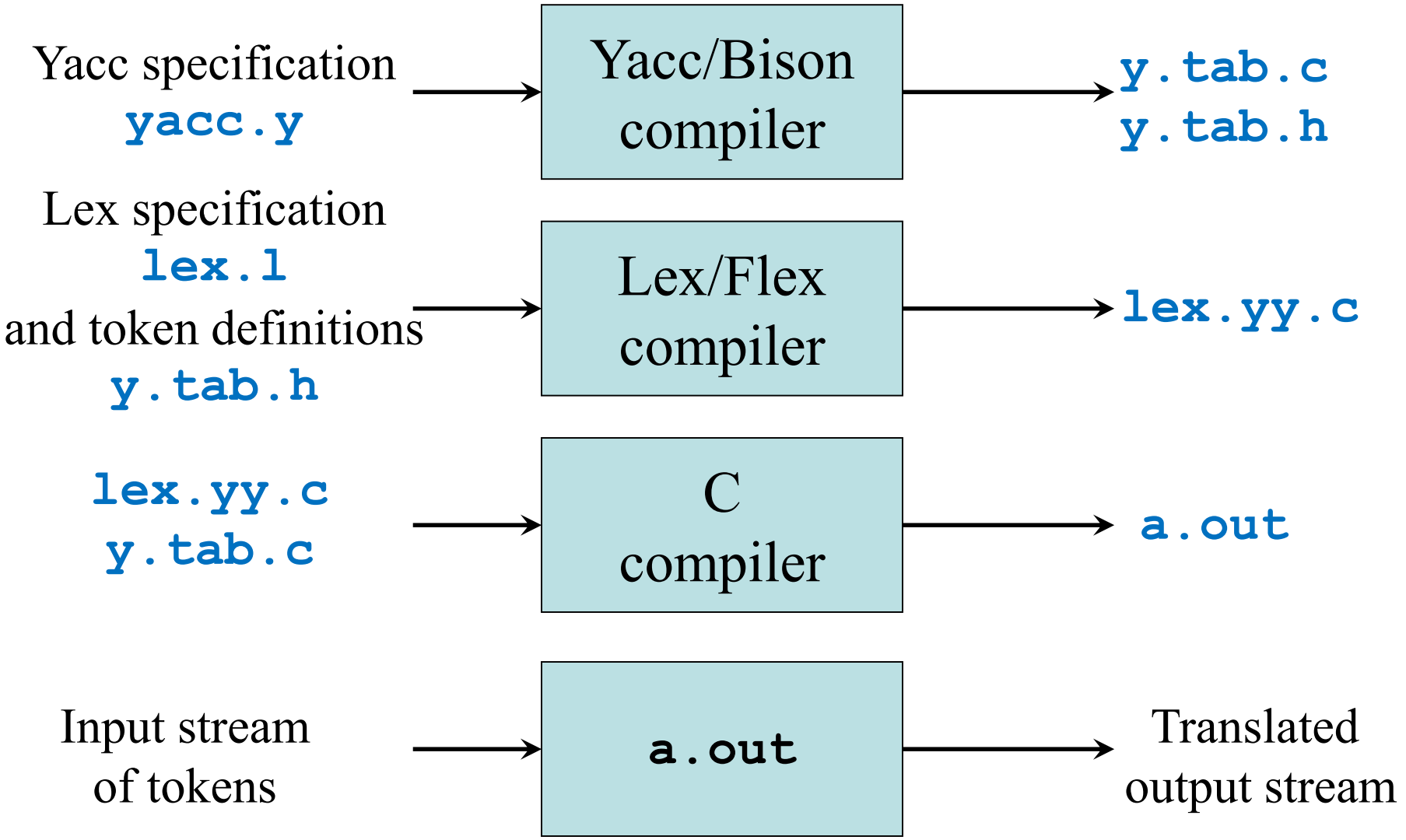

```
%%
int yylex()
{ int c;
  while ((c = getchar()) == ' ')
  ;
  if ((c == '.') || isdigit(c))
  { ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}
int main()
{ if (yyparse() != 0)
    fprintf(stderr, "Abnormal exit\n");
  return 0;
}
int yyerror(char *s)
{ fprintf(stderr, "Error: %s\n", s);
}
```

Crude lexical analyzer for fp doubles and arithmetic operators

Run parser

Invoked by parser to report parse errors

Combining Lex/Flex with Yacc/Bison



Lex Specification for Example2


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

extern double yylval;
%}
number  [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]             { /* skip blanks */ }
{number}        { sscanf(yytext, "%lf", &yylval);
                  return NUMBER;
                }
\n|.            { return yytext[0]; }
```

Generated by Yacc, contains **`#define NUMBER xxx`**

Defined in **`y.tab.c`**

```
yacc -d example2.y
lex example2.l
gcc y.tab.c lex.yy.c ./a.out
```

```
bison -d -y example2.y
flex example2.l
gcc y.tab.c lex.yy.c ./a.out
```

Error Recovery in Yacc


```
%{
…
%}
…
%%
lines   : lines expr '\n'       { printf("%g\n", $2; }
        | lines '\n'
        | /* empty */
        | error '\n'            { yyerror("reenter last line: ");
                                  yyerrok;
                                }
        ;
…
```

Error production:
set error mode and
skip input until newline

Reset parser to normal mode