Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 11

- Syntax-Directed Translation (cont'd)
- Parser generators: Yacc/Bison

Summary

- Syntax-Directed Definitions (Attribute Grammars)
  - Enrich CFG's with Attributes and Semantic Rules
  - S-attributed SDD's: synthesized attributes only
    - Attributes computed with postorder traversal
  - L-attributed SDD's: also (constrained) inherited attributes
    - Attributes computed with left-to-right, depth-first traversal
- Syntax-Directed Translation Schemes
  - Embed semantic actions of SDD's in productions
  - Sometimes allow to compute the translation without building the whole parsing tree
- Implementation of **S-attributed SDD's** for **LR grammars** with bottom-up (LR) parsing: semantic actions are placed at the end of the corresponding production

Using Translation Schemes for L-Attributed Definitions

- An L-attributed SDD for a grammar that can be parsed top-down (LL) can be implemented using Translation Schemes

1. Embed actions that compute **inherited attributes** for nonterminal A immediately before A
2. Place actions that compute a **synthesized attribute** for the head of a production at the end of the body of that production

Using Translation Schemes for L-Attributed Definitions

Production	Semantic Rule
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := 'integer'
$T \rightarrow$ **real**	T.type := 'real'
$L \rightarrow L_1$ **, id**	L_1.in := L.in; *addtype*(**id**.entry, L.in)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.in)

Translation Scheme

$D \rightarrow T$ { L.in := T.type } L

$T \rightarrow$ **int** { T.type := 'integer' }

$T \rightarrow$ **real** { T.type := 'real' }

$L \rightarrow$ { L_1.in := L.in } L_1 **, id** { *addtype*(**id**.entry, L.in) }

$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

Recursive-Descent Parsing (Recap)

- Grammar must be LL(1)
- Every nonterminal has one (recursive) procedure
- When a nonterminal has multiple productions, the input look-ahead is used to choose one
- Note: the procedures have no parameters and no result

expr → *term rest*
rest → **+** *term rest*
  | **-** *term rest*
  | ε
term → **id**

procedure *rest*();
begin
  if *lookahead* in FIRST(**+** *term rest*) **then**
    match('**+**'); *term*(); *rest*()
  else if *lookahead* in FIRST(**-** *term rest*) **then**
    match('**-**'); *term*(); *rest*()
  else if *lookahead* in FOLLOW(*rest*) **then**
    return
  else error()
end;

Implementing L-Attributed Definitions in Recursive-Descent Parsers

- Attributes are passed as arguments to procedures (*inherited*) or returned (*synthesized*)
- Procedures store computed attributes in local variables

D → *T* { *L*.in := *T*.type } *L*
T → **int** { *T*.type := 'integer' }
T → **real** { *T*.type := 'real' }

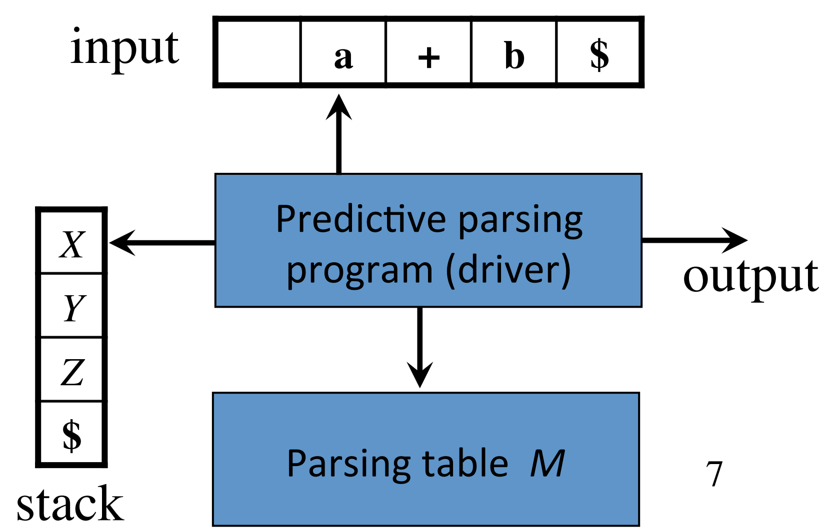
```
void D()
{ Type Ttype = T();
  Type Lin = Ttype;
  L(Lin);
}
Type T()
{ Type Ttype;
  if (lookahead == INT)
  { Ttype = TYPE_INT;
    match(INT);
  } else if (lookahead == REAL)
  { Ttype = TYPE_REAL;
    match(REAL);
  } else error();
  return Ttype;
}
void L(Type Lin)
{ … }
```

Output: synthesized attribute

Input: inherited attribute

Implementing L-Attributed Definitions in Top-Down Table-Driven Parsers

- The stack will contain, besides grammar symbols, *action-records* and *synthesize-records*
- Inherited attributes of *A* are placed in *A*'s record
  - The code computing them is in a record above *A*
- Synthesized attributes of *A* are placed in a record just below *A*
- It may be necessary to make copies of attributes to avoid that they are popped when still needed



Implementing L-Attributed Definitions for **LL grammars** in Bottom-Up Parsers

- Remove any embedded action with *marking nonterminal:* $A \rightarrow \alpha \{ act \} \beta$ becomes

  $A \rightarrow \alpha N \beta$

  $N \rightarrow \varepsilon \{ act' \}$

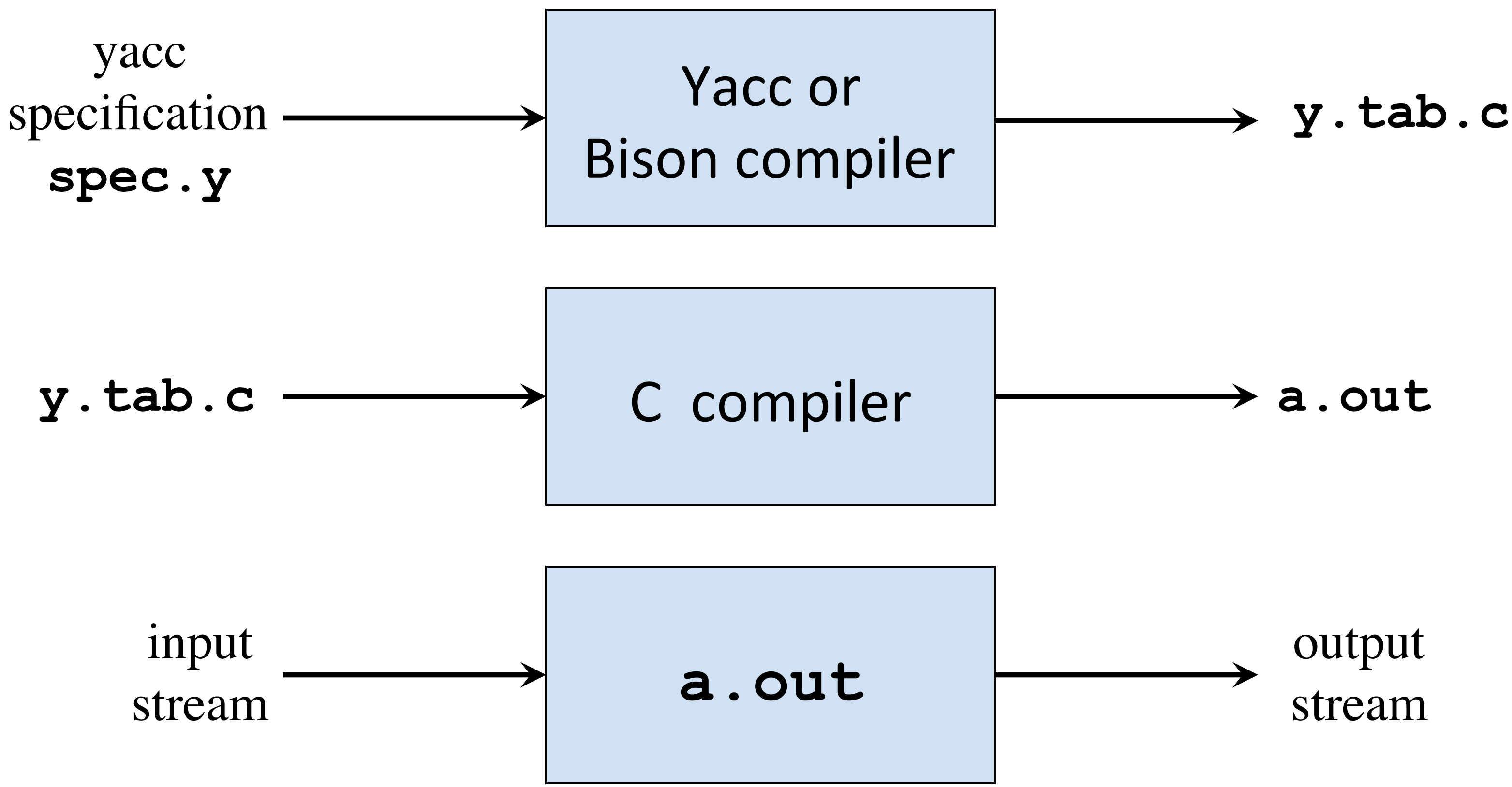
  where act':

  - Copies as inherited attributes of *N* any attribute of *A, α* needed by *act*
  - Computes attributes like *act*, making them synthesized for N
- Fact: if the start grammar was LL, the new one is LR
- Note: *act'* accesses attributes out of its production! This works, as they are (deeper) in the LR stack

Parser Generators: ANTLR, Yacc, and Bison

- *ANTLR* tool
  - Generates LL(*k*) parsers
- *Yacc* (Yet Another Compiler Compiler)
  - Generates LALR parsers
- *Bison*
  - Improved version of Yacc (GNU project)

Creating an LALR(1) Parser with Yacc/Bison



Yacc Specification

- A **yacc specification** consists of three parts:

- ***yacc declarations****, and C declarations within* **%{ %}**
  %%
  translation rules *(productions + semantic actions)*
  %%
  user-defined ***auxiliary procedures***

- The *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

- Production *head* -> $body_1$ | $body_2$ | … | $body_n$ | ε
  becomes in Yacc

```
head :  body1   { semantic action1 }
     |  body2   { semantic action2 }
     …
     |  /* empty */
     ;
```

- Tokens (terminals) can be:
  - Quoted single characters, e.g. `'+'`, with corresponding ASCII code
  - Identifiers declared as tokens in the declaration part using
    `%token` *TokenName*
- Nonterminals:
  - Arbitrary strings of letters and digits (not declared as tokens)

Semantic Actions and Synthesized Attributes

- **Semantic actions** are sequences of C statements, and may refer to values of the *synthesized attributes* of terminals and nonterminals in a production:

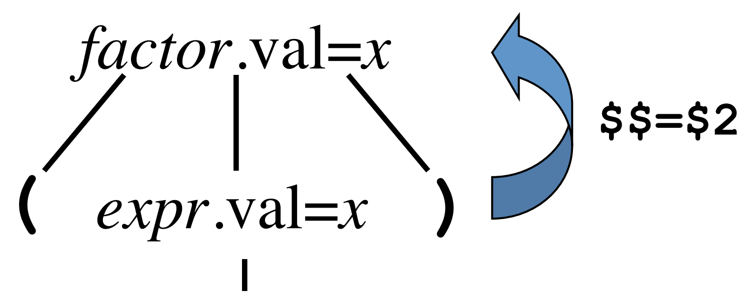
  $X : Y_1\ Y_2\ Y_3 \ldots Y_n$ { *action* }

  – **$$** 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; }
```



- The values associated with tokens (terminals) are those returned by the lexer

An S-attributed Grammar for a simple desk calculator

The grammar
line → expr '\n'
expr → expr + term | term
term → term * factor | factor
factor → (expr) | **DIGIT**

```
%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; }
     ;
%%
```

Also results in definition of **`#define DIGIT xxx`**

Attribute of **`term`** (parent)

Attribute of **`factor`** (child)

Attribute of token

A simple desk calculator

```
%{ #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 token (stored in **`yylval`**)

Very simple lexical analyzer invoked by the parser

The grammar

line → expr ‘\n’

expr → expr + term | term

term → term * factor | factor

factor → (expr) | **DIGIT**

Bottom-up Evaluation of S-Attributed Definitions in Yacc

Stack	val	Input	Action	Semantic Rule
$	_	**3*5+4n$**	shift	
$ 3	*3*	***5+4n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *F*	*3*	***5+4n$**	reduce $T \rightarrow F$	$$ = $1
$ *T*	*3*	***5+4n$**	shift	
$ *T* *	*3 _*	**5+4n$**	shift	
$ *T* * **5**	*3 _ 5*	**+4n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *T* * *F*	*3 _ 5*	**+4n$**	reduce $T \rightarrow T * F$	$$ = $1 * $3
$ *T*	*15*	**+4n$**	reduce $E \rightarrow T$	$$ = $1
$ *E*	*15*	**+4n$**	shift	
$ *E* **+**	*15 _*	**4n$**	shift	
$ *E* **+ 4**	*15 _ 4*	**n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *E* **+** *F*	*15 _ 4*	**n$**	reduce $T \rightarrow F$	$$ = $1
$ *E* **+** *T*	*15 _ 4*	**n$**	reduce $E \rightarrow E$ **+** T	$$ = $1 + $3
$ *E*	*19*	**n$**	shift	
$ *E* **n**	*19 _*	**$**	reduce $L \rightarrow E$ **n**	*print* $1
$ *L*	*19*	**$**	accept	

Dealing With Ambiguous Grammars

- By defining operator precedence levels and left/right associativity of the operators, we can specify ambiguous grammars in Yacc, such as

$E \rightarrow E\texttt{+}E \mid E\texttt{-}E \mid E\texttt{*}E \mid E\texttt{/}E \mid \texttt{(}E\texttt{)} \mid \texttt{-}E \mid \texttt{num}$

- Yacc resolves conflicts, by default, as follows:
  - **Reduce/reduce** conflict: precedence to first production in the specification
  - **Shift/reduce** conflict: precedence to shift
    - ok for *if-then-else*
    - infix binary operators are handled as right-associative

Example: PlusTimesCalculator-flat

```
%token NUMBER
%%
lines : expr '\n'     { printf("= %g\n", $1); }
expr  : expr '+' expr          { $$ = $1 + $3; }
      | expr '*' expr          { $$ = $1 * $3; }
      | NUMBER
      ;
%%
```

- bison's warning:
  conflicts: 4 shift/reduce

```
State 8 conflicts: 2 shift/reduce
State 9 conflicts: 2 shift/reduce
…
state 8
   2 expr: expr . '+' expr
   2     | expr '+' expr .
   3     | expr . '*' expr

   '+'  shift, and go to state 6
   '*'  shift, and go to state 7

   '+'      [reduce using rule 2 (expr)]
   '*'      [reduce using rule 2 (expr)]
   $default  reduce using rule 2 (expr)
```

```
> ./PlusTimesCalculator-flat
1+2*3+4*5
= 47   /* right associate, no precedence */
```

Dealing With Ambiguous Grammars

- To define precedence levels and associativity in Yacc's declaration part, list tokens in order of increasing precedence, prefixed by **`%left`** or **`%right`**:

```
%left '+' '-'  //same precedence, associate left
%left '*' '/'
%right UMINUS
```

- If tokens have precedence, productions also have, equal to that of the rightmost terminal in the body. In this case:
  - **Shift/reduce** conflict are resolved with **reduce** if the production has higher precedence than the input symbol, or if they are equal and are left-associative.

Example: PlusTimesCalculator

```
%token NUMBER  /* tokens listed in increasing order of precedence */
%left '+'
%left '*'
%%
lines : expr '\n'      { printf("= %g\n", $1); }
expr  : expr '+' expr          { $$ = $1 + $3; }
      | expr '*' expr          { $$ = $1 * $3; }
      | NUMBER
      ;
%%
```

- No warnings by bison

```
state 8

   2 expr: expr . '+' expr
   2     | expr '+' expr .
   3     | expr . '*' expr

   '*'  shift, and go to state 6

   $default  reduce using rule 2 (expr)
```

```
> ./PlusTimesCalculator-flat
1+2*3+4*5
= 27    /* correct  precedence */
```

A more advanced calculator

Double type for attributes and **yylval**

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}
%token NUMBER   /* tokens listed in increasing order of precedence */
%left '+' '-'
%left '*' '/'
%right UMINUS   /* fake token with highest precedence, used below */
 %%
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;} /* rule with highest precedence */
    | NUMBER
    ;
%%
```

A more advanced calculator (cont'd)

```
%%
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

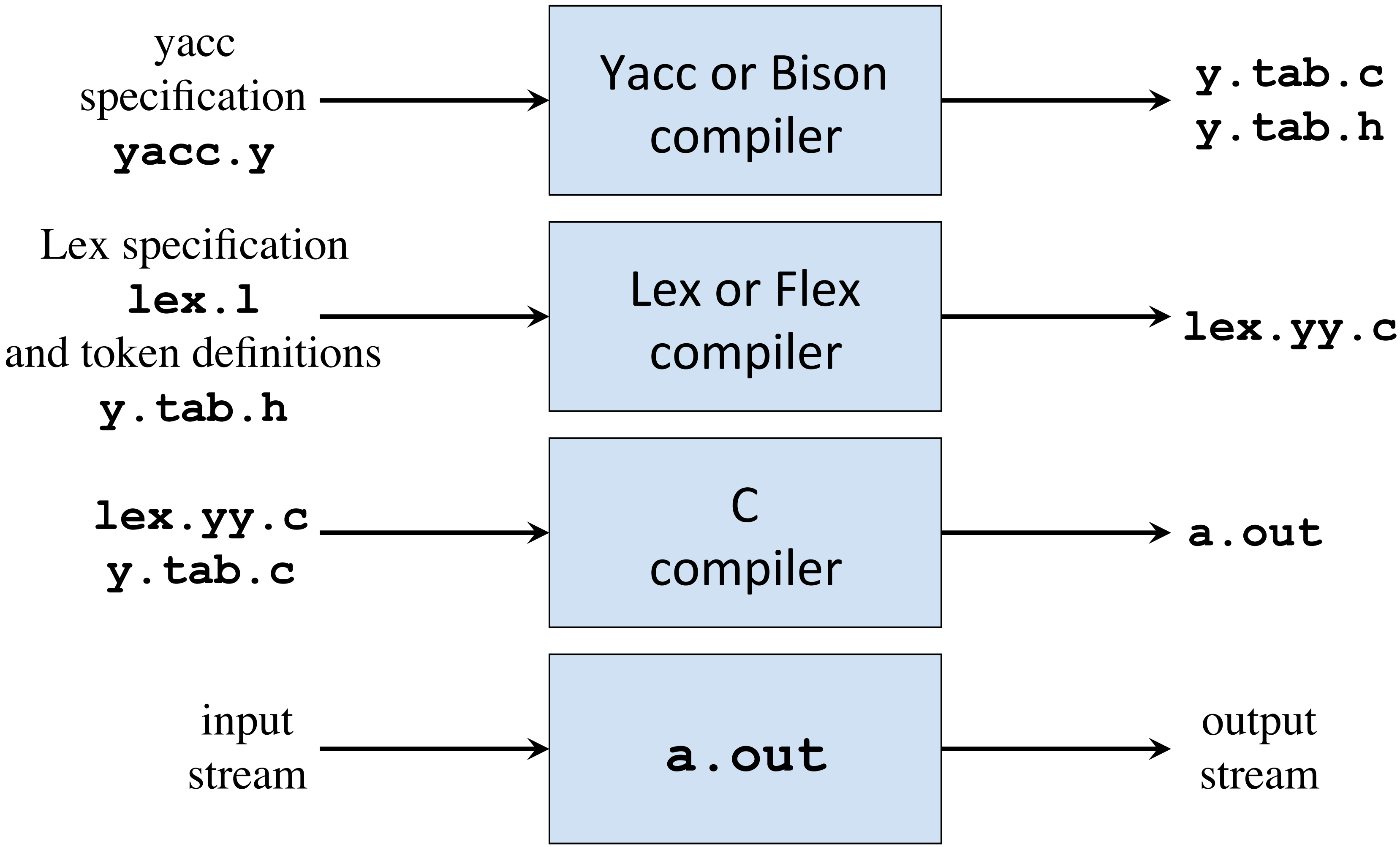
Run the parser

Invoked by parser to report parse errors

Dealing With Ambiguous Grammars (summary)

- Yacc does not report about conflicts that are solved using user-defined precedences
- It reports conflicts that are resolved with the default rules
- To visit the automaton and the LALR parsing table generated, execute Bison/Yacc with option **–v**, and read the **<filename>.output** file
- This allows to see where conflicts were generated, and if the parser resolved them correctly
- Graphical representation of the automaton using Bison/Yacc with option **–g**. Output should be in **dot** format

Combining Lex/Flex with Yacc/Bison



Lex Specification for Advanced Calculator

```
%option noyywrap
%{
#define YYSTYPE double
#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

- Based on error productions of the form   ***A → error α***

```
%{
…
%}
…
%%
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

Emulating the Evaluation of L-Attributed Definitions in Yacc

$D \rightarrow T\ \{\ L.\text{in} := T.\text{type}\ \}\ L$

$T \rightarrow \textbf{int}\ \{\ T.\text{type} := \text{'integer'}\ \}$

$T \rightarrow \textbf{real}\ \{\ T.\text{type} := \text{'real'}\ \}$

$L \rightarrow \{\ L_1.\text{in} := L.\text{in}\ \}\ L_1\ \textbf{, id}$
$\quad \{\ \mathit{addtype}(\textbf{id}.\text{entry}, L.\text{in})\ \}$

$L \rightarrow \textbf{id}\ \{\ \mathit{addtype}(\textbf{id}.\text{entry}, L.\text{in})\ \}$

```
%{
Type Lin; /* global variable */
%}
%%
D   : Ts L
    ;
Ts  : T           { Lin = $1; }
    ;
T   : INT         { $$ = TYPE_INT; }
    | REAL        { $$ = TYPE_REAL; }
    ;
L   : L ',' ID { addtype($3, Lin);}
    | ID          { addtype($1, Lin);}
    ;
%%
```

Rewriting a Grammar to Avoid Inherited Attributes

Production
$D \rightarrow L : T$
$T \rightarrow$ **int**
$T \rightarrow$ **real**
$L \rightarrow L_1$ **, id**
$L \rightarrow$ **id**

Production	Semantic Rule
$D \rightarrow$ **id** L	*addtype*(**id**.entry, *L*.type)
$T \rightarrow$ **int**	*T*.type := 'integer'
$T \rightarrow$ **real**	*T*.type := 'real'
$L \rightarrow$ **, id** L_1	*addtype*(**id**.entry, *L*.type)
$L \rightarrow$ **:** T	*L*.type := *T*.type