# THEORY OF COMPUTATION AND COMPILERS

## Unit - II

## CONTEXT FREE GRAMMARS AND PARSING

- Introduction
- Context-Free Grammars - Derivation, Parse trees, Ambiguity
- Types of Parsers
- LL(K) grammars and LL(1) parsing
- Bottom-up Parsing - handle pruning
- LR Grammar Parsing
- LALR parsing
- Parsing ambiguous grammars
- Error Recovery in Parsing
- **YACC programming specification** 

**Dr. R. Madana Mohana**
**Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning**
**CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY**
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

# Unit-II: Syntax Analysis (or) Parser
## **Parser Generators**

*Outline:*

- The Parser Generator Yacc
- Using Yacc with Ambiguous Grammars
- Creating Yacc Lexical Analyzers with Lex

# Parser Generators

- We shall use the **LALR** parser generator **Yacc** as the basis of our discussion, since it implements many of the concepts discussed in the previous sections and it is widely available.
- **Yacc** stands for "**yet another compiler-compiler**," reflecting the popularity of **parser generators** in the early **1970s** when the first version of **Yacc** was created by **S. C. Johnson**.
- **Yacc** is available as a *command* on the **UNIX** system, and has been used to help implement many production compilers.

# The Parser Generator Yacc

- A translator can be constructed using **Yacc** in the manner illustrated in below Fig.

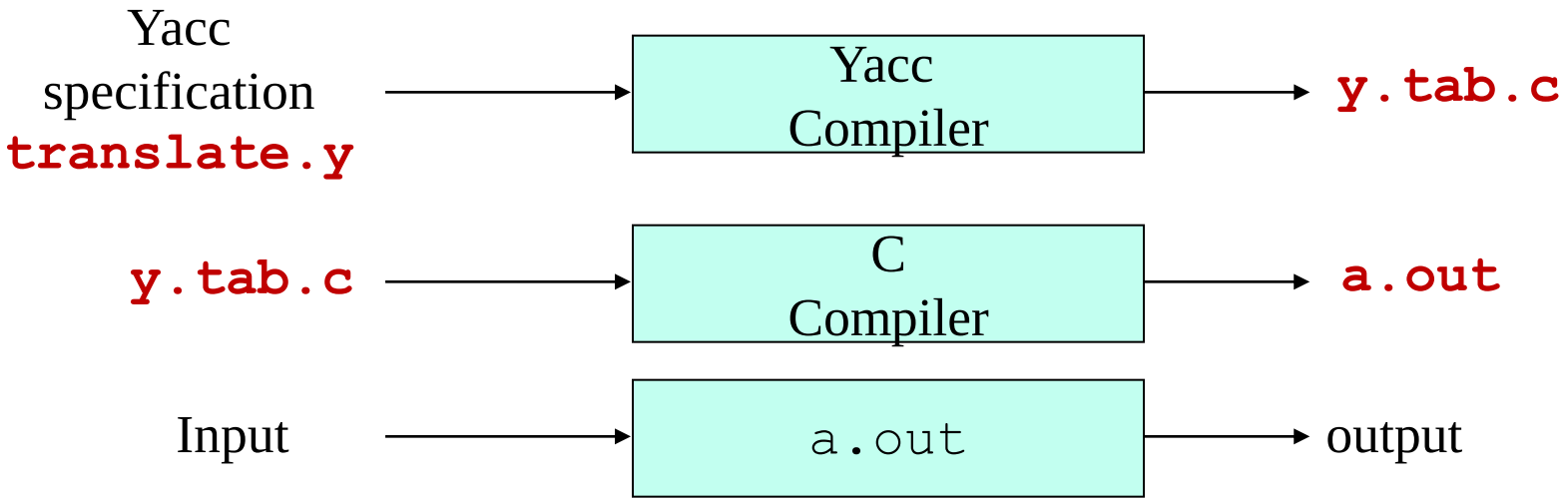


*Figure:* Creating an input/output translator with **Yacc**

# The Parser Generator Yacc

**Steps:**

1. First, a file, say **`translate.y`**, containing a **`Yacc`** specification of the translator is prepared. The **`UNIX`** system command

   **`yacc translate.y`**

2. Transforms the file **`translate.y`** into a **`C`** program called **`y.tab.c`** using the **`LALR`** method.

# The Parser Generator Yacc

**Steps:**

3. The program **`y.tab.c`** is a representation of an **`LALR`** parser written in **`C`**, along with other **`C`** routines that the user may have prepared. The **`LALR`** parsing table is compacted. By compiling **`y.tab.c`** along with the **`ly`** library that contains the **`LR`** parsing program using the command

```
cc y.tab.c -ly
```

# The Parser Generator Yacc

**Steps:**

4. We obtain the desired object program **`a.out`** that performs the translation specified by the original **`Yacc`** program. If other procedures are needed, they can be compiled or loaded with **`y.tab.c`**, just as with any **`C`** program.

# The Parser Generator Yacc

**A `Yacc` source program has three parts:**

```
1.declarations
%%
2.translation rules
%%
3.supporting C routines
```

# The Parser Generator Yacc

**Example:**

To illustrate how to prepare a **`Yacc`** source program, let us construct a simple desk calculator that reads an arithmetic expression, evaluates it, and then prints its numeric value.

# The Parser Generator Yacc

## Example:

We shall build the desk calculator starting with the with the following grammar for arithmetic expressions:

```
1. E → E + T | T
2. T → T * F | F
3. F → ( E ) | id
```

The token digit is a single digit between `0` and `9`. A `Yacc` desk calculator program derived from this grammar is shown in Fig.

# The Parser Generator Yacc

**Example: Yacc specification of a simple desk calculator**

```
%{
#include <ctype.h>
%}
%token DIGIT
```

# The Parser Generator Yacc

**Example: Yacc specification of a simple desk calculator**

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

# The Parser Generator Yacc

**Example: Yacc specification of a simple desk calculator**

```
term : term '*' factor { $$ = $1 * $3; }
| factor
;
factor : '(' expr ')' { $$ = $2; }
| DIGIT
;
%%
```

# The Parser Generator Yacc

**Example: Yacc specification of a simple desk calculator**

```
yylex() {
      int c;
      c = getchar();
      if (isdigit(c)) {
            yylval = c-'0';
            return DIGIT;
      }
      return c;
}
```

# The Parser Generator Yacc

## The Declarations Part:

There are two sections in the declarations part of a **`Yacc program`**; both are optional.

In the first section, we put ordinary **`C`** declarations, delimited by **`%{`** and **`%}`**. Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections.

# The Parser Generator Yacc

## The Declarations Part:

In Previous Ex. program, this section contains only the include-statement

```
#include <ctype.h>
```

that causes the `C` preprocessor to include the standard header file `<ctype.h>` that contains the predicate `isdigit`.

Also in the declarations part are declarations of grammar tokens.

In Previous Ex. program, the statement

```
%token DIGIT
```

declares `DIGIT` to be a token.

# The Parser Generator Yacc

## The Translation Rules Part:

In the part of the **`Yacc`** specification after the first **`%%`** pair, we put the translation rules.

Each rule consists of a grammar production and the associated semantic action.

# The Parser Generator Yacc

## The Translation Rules Part:

A set of productions that we have been writing:

**<head> → <body>$_1$ | <body>$_2$ |… <body>$_n$**

would be written in **Yacc** as

```
<head> : <body>1 {<semantic action>1}
       | <body>2 {<semantic action>2}
                 …
       | <body>n {<semantic action>n}
```

# The Parser Generator Yacc

## The Supporting C-Routines Part:

The third part of a **Yacc** specification consists of **supporting C-routines**.

A **lexical analyzer** by the name **yylex()** must be provided.

Using **Lex** to produce **yylex()** is a common choice.

**Other procedures** such as **error recovery routines** may be added as necessary.

# The Parser Generator Yacc

## The Supporting C-Routines Part:

The lexical analyzer **`yylex()`** produces tokens consisting of a token name and its associated attribute value. If a token name such as **`DIGIT`** is returned, the token name must be declared in the first section of the **`Yacc specification`**.

The attribute value associated with a token is communicated to the parser through a **`Yacc`**-defined variable **`yylval`**.

# Using Yacc with Ambiguous Grammars

Let us now modify the **Yacc specification** so that the resulting **desk calculator** becomes more useful. First, we shall allow the desk calculator to evaluate a sequence of expressions, one to a line. We shall also allow blank lines between expressions. We do so by changing the first rule to

```
lines: lines expr '\n'{ printf("%g\n", $2);}
| lines '\n'
| /* empty */
;
```

In **Yacc**, an empty alternative, as the third line is, denotes **ϵ**.

# Using Yacc with Ambiguous Grammars

*Second*, we shall enlarge the class of expressions to include numbers with a decimal point instead of single digits and to include the arithmetic operators **+**, **-**, (both binary and unary), *****, and **/**. The easiest way to specify this class of expressions is to use the ambiguous grammar

1. E → E + E
2. E → E - E
3. E → E * E
4. E → E / E
5. E → - E
6. E → ( E ) | number

# Using Yacc with Ambiguous Grammars

*The resulting* **Yacc** *specification is shown in Fig.*

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

# Using Yacc with Ambiguous Grammars

```
%%
lines : lines expr '\n' {printf("%g\n",
$2);}
      | lines '\n'
      | /* empty */
      ;
```

# Using Yacc with Ambiguous Grammars

```
expr : expr '+' expr { $$ = $1 + $3;}
     | expr '-' expr { $$ = $1 - $3; }
     | expr '*' expr { $$ = $1 * $3; }
     | expr '/' expr { $$ = $1 / $3; }
     | '(' expr ')' { $$ = $2; }
     | '-' expr %prec UMINUS {$$ = - $2;}
     | NUMBER
     ;
%%
```

# Using Yacc with Ambiguous Grammars

```
yylex() {
     int c;
     while ( ( c = getchar() ) == ' ' );
     if ( (c == '.') || (isdigit(c)) ) {
          ungetc(c, stdin);
          scanf("%lf", &yylval);
          return NUMBER;
     }
     return c;
}
```

# Creating Yacc Lexical Analyzers with Lex

- **Lex** was designed to produce **lexical analyzers** that could be used with **Yacc**.
- The **Lex** library **ll** will provide a driver program named **yylex()**, the name required by **Yacc** for its **lexical analyzer**.
- If **Lex** is used to produce the **lexical analyzer**, we replace the routine **yylex()** in the third part of the **Yacc** specification by the statement

```
#include "lex.yy.c"
```

# Creating Yacc Lexical Analyzers with Lex

- And we have each **Lex** action return a terminal known to **Yacc.**
- By using the **#include "lex.yy.c"** statement, the program **yylex** has access to **Yacc's** names for tokens, since the **Lex** output file is compiled as part of the **Yacc** output file **y.tab.c**.

# Creating Yacc Lexical Analyzers with Lex

- Under the **UNIX** system, if the **Lex** specification is in the file **first.l** and the **Yacc** specification in **second.y**, we can say

```
lex first.l

yacc second.y

cc y.tab.c -ly -ll
```

to obtain the desired translator.

# Creating Yacc Lexical Analyzers with Lex

## Lex specification for yylex():

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

# Summary…

Syntax Analysis (or) Parser: **Parser Generators**

- The Parser Generator Yacc
- Using Yacc with Ambiguous Grammars
- Creating Yacc Lexical Analyzers with Lex

*Reading:* Aho2, Section 4.9

*Next Lecture:* Unit-III: Syntax Directed Translation (SDT) & Intermediate Code Generation