Yacc: A Parser Generator

Compilation Phases

- Lexical Analysis (Lex): Converts source code into tokens.
- **Syntax Analysis (Yacc): Checks token sequences against grammar rules.**
- Semantic Analysis: Ensures the meaning of statements is valid.
- Intermediate Code Generation, Optimization, and Code Generation.

Parsing

- Parsing is the process of analyzing a sequence of tokens to determine their grammatical structure according to a given formal grammar.
- It is the second phase of compilation, following lexical analysis.

Types of Parsers

- Top-Down Parsing (e.g., Recursive Descent, LL Parsing)
- Bottom-Up Parsing (e.g., Shift-Reduce, LR Parsing)

Yacc generates Bottom-Up Parsers (LR Parsers).

Parsing

- By design, every programming language has precise rules that prescribe the syntactic structure of well-formed programs.
- In C, for example, a program is made up of functions,
- a function out of declarations and statements,
- a statement out of expressions, and so on.
- The syntax of programming language constructs can be specified by context-free grammars

Grammar Rules

A grammar consists of:

- Terminals (tokens from lexical analysis)
- Non-terminals (syntactic categories)
- Production rules (how non-terminals expand)
- Start symbol

```
program --> VOID MAIN '(' ')' compound_stmt
compound_stmt --> '{' '}' | '{' stmt_list '}'
          | '{' declaration_list stmt_list '}'
stmt_list --> stmt | stmt_list stmt
stmt --> compound_stmt| expression_stmt
            | if_stmt | while_stmt
expression_stmt --> ';'| expression ';'
if_stmt --> IF '(' expression ')' stmt
   | IF '(' expression ')' stmt ELSE stmt
while_stmt --> WHILE '(' expression ')' stmt
expression --> assignment_expr
              | expression ',' assignment_expr
```

Variable Declarations

C supports different types of variable declarations, which we can define using a grammar.

decl → type IDENTIFIER ';'

type → 'int' | 'float' | 'char' | 'double'

Example:

int x;   float y;   char ch;

Arithmetic Expressions

A basic grammar for arithmetic expressions in C:

> expr → expr '+' term | expr '-' term | term
>
> term → term '*' factor | term '/' factor | factor
>
> factor → '(' expr ')' | NUMBER

Example: 3 + 5 * (2 - 1)

Conditional Statements

stmt → 'if' '(' expr ')' stmt ('else' stmt)?

expr → expr relop expr | term

relop → '<’ | '>' | '==' | '!=' | '<=' | '>='

```
if (x < y)
   z = 10;
else
   z = 20;
```

Shift-Reduce Parsing Concept

Yacc uses Shift-Reduce Parsing, which operates in four steps:

- Shift → Read a token and push it onto the stack.
- Reduce → Replace symbols on the stack using grammar rules.
- Accept → If the start symbol remains, parsing is complete.
- Error → If no valid rule applies, a syntax error occurs.

Example (for a + b * c using shift-reduce):

a * b using shift-reduce

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

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

Yacc (Yet Another Compiler Compiler)

- Yacc is a tool used to generate LALR parsers, which are a subclass of bottom-up LR parsers.
- It takes a grammar specification and produces a C program that parses input according to that grammar.
- Works in combination with Lex:
    - Lex scans tokens from input.
    - Yacc processes tokens based on the grammar and builds a parse tree.

Why Do We Need Yacc?

Without Yacc:

- Manually coding a parser.
- Implementing shift-reduce logic manually.
- Handling conflicts in grammar rules.

With Yacc, we can:

- Write grammar rules naturally → Yacc handles parsing logic.
- Automatically resolve parsing conflicts (to some extent).

Connecting Lex and Yacc

How Lex and Yacc Work Together

- Lex scans input and returns tokens to Yacc.
- Yacc reads tokens and applies grammar rules.
- If a rule matches, Yacc reduces the rule.
- If no rule matches, Yacc reports an error.

Yacc Workflow

Yacc takes a set of grammar rules and actions written in a .y file and generates a parser in C.

- Write a grammar specification (parser.y)
- Run Yacc to generate y.tab.c (C source code for parser)
- Compile with Lex output (lex.yy.c)
- Execute the parser on an input

Structure of a Yacc Program

A Yacc program also consists of three sections, separated by %%

```
%{
    /* C Declarations */
%}
%token TOKEN_NAME
%%
/* Grammar Rules */
start_symbol : rule1 { Action; } | rule2 { Action; } ;
%%
/* Auxiliary C functions (main, yyerror, etc.) */
```

Compiling and Running the Parser

Generate Lex and Yacc output:

lex lex.l

yacc -d parser.y

Compiling and Running the Parser

Generate Lex and Yacc output:

```
lex lex.l
Yacc -d parser.y
```

Compile and link:

```
gcc lex.yy.c y.tab.c -o parser -lm
```

Run the parser:

```
./parser
```

Yacc Keywords and Functions

%token (Token Declaration)

- %token is used to declare terminal symbols (tokens) that are received from the Lex scanner.
- Tokens are essentially identifiers for lexical elements (e.g., keywords, operators, numbers, etc.)

%token NUMBER PLUS MINUS MULTIPLY DIVIDE

Yacc Keywords and Functions

%start (Defining the Start Symbol)

- Defines the starting symbol of the grammar.
- By default, Yacc uses the first non-terminal in the rules as the start symbol.
- %start allows explicitly defining it.

%start *program*

This tells Yacc that *program* is the root of the parse tree.

Yacc Keywords and Functions

%left, %right, %nonassoc (Operator Precedence and Associativity)

- These directives define precedence and associativity for operators.
- %left → Left-associative operators (e.g., +, -).
- %right → Right-associative operators (e.g., = in assignment a = b = 5).
- %nonassoc → Operators that cannot be chained (e.g., <, > in comparisons).

```
%left PLUS MINUS

%left MULTIPLY DIVIDE
```

This means * and / have higher precedence than + and -.

Yacc Keywords and Functions

yyparse() (Parser Execution Function)

- yyparse() is the main function generated by Yacc to parse input.
- It calls Lex (yylex()) to get tokens and applies grammar rules.

```
int main() {
    printf("Enter expression:\n");
    yyparse();  // Calls the parser
    return 0;
}
```

Yacc Keywords and Functions

yyerror(char *s) (Error Handling Function)

- A function that gets called when a syntax error is found.
- To print meaningful error messages when the input does not match the grammar.

```
int yyerror(char *s) {
    printf("Syntax Error: %s\n", s);
    return 0;
}
```

Whenever Yacc encounters an invalid expression, yyerror() is triggered.

Yacc Keywords and Functions

$$, $1, $2, $3, ... (Semantic Values and Attributes)

- These are value placeholders used inside grammar rules:
  - $1 → Left operand.
  - $3 → Right operand.
  - $$ → Stores the result of the entire rule.
- To store and pass values while parsing.
- To perform computations inside Yacc.

Yacc Keywords and Functions

yylval (Lex-Yacc Value Communication)

- yylval is a global variable used to pass values from Lex to Yacc.
- To send integer values or structures from Lex to Yacc.

Yacc Keywords and Functions

yylval (Lex-Yacc Value Communication)

```
%{    #include "y.tab.h"                %}
%%
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
%%
```

The yylval variable stores the numeric value of yytext and sends it as NUMBER to Yacc.

```
%token NUMBER
%%
expr: NUMBER { printf("Received number: %d\n", $1); };
%%
```

Handling Operator Precedence and Associativity

- In programming languages, expressions often contain operators like +, -, *, /, etc.
- The order in which these operators are evaluated is determined by precedence and associativity.
- Yacc allows us to define these rules explicitly using precedence and associativity directives (%left, %right, %nonassoc).

Operator Precedence

- Operator precedence determines which operator is evaluated first in an expression.
- Operators with higher precedence are evaluated before those with lower precedence.

```
3 + 4 * 5   // Evaluates as 3 + (4 * 5), not (3 + 4) * 5
```

* has higher precedence than +, so 4 * 5 is evaluated first.

Operator Associativity

Associativity determines how operators of the same precedence level are grouped.

Operators can be:

- Left-associative (%left): Evaluated left to right (e.g., +, -, *, /).
- Right-associative (%right): Evaluated right to left (e.g., = in a = b = c).
- Non-associative (%nonassoc): Operators cannot be chained (e.g., relational operators like <, >).

Defining Precedence for Arithmetic Operators

%left '+' '-'

%left '*' '/'

* and / have higher precedence than + and -.

All operators are left-associative.

Handling Right-Associative Operators (Exponentiation)

Some operators, like ** (exponentiation), are right-associative.

```
%right '**'  // Right-associative exponentiation
```

2 ** 3 ** 2 → Evaluates as 2 ** (3 ** 2), not (2 ** 3) ** 2.

The %right directive ensures the rightmost ** is evaluated first.

Handling Non-Associative Operators (<, >, ==, !=)

Some operators, like < and >, cannot be used together.

```
%nonassoc '<' '>' '==' '!='            // Prevents chaining like "a < b < c"
```

3 < 4 < 5 → Syntax error (not allowed).

3 < 4 → Allowed.

4 == 5 → Allowed.

Handling Mixed Operators

```
%right '='           // Assignment (right-associative)
%left '&&' '||'      // Logical operators (left-associative)
%nonassoc '<' '>' '==' '!=' // Non-associative comparison
%left '+' '-'        // Addition and subtraction (left-associative)
%left '*' '/'        // Multiplication and division (left-associative)
%right '^'           // Exponentiation (right-associative)
```

Rule in Yacc

- The precedence is determined by the order of appearance in the Yacc file.
- Operators declared later in the precedence section have higher precedence than those declared earlier.

```
%left '+'
%left '*'
%right '**'
```

** is evaluated first, then *, then +.

Shift/Reduce and Reduce/Reduce Conflicts in Yacc

Parsing conflicts occur when Yacc cannot determine the correct parsing action due to ambiguity in the grammar.

There are two main types of conflicts:

- Shift/Reduce Conflict
- Reduce/Reduce Conflict

Shift/Reduce Conflict

A Shift/Reduce conflict happens when Yacc is unsure whether to shift (read more input) or reduce (apply a rule) at a particular point in the parsing process.

if-else Ambiguity

```
if (x)
    statement;
if (y)
    statement;
else
    statement;
```

Shift/Reduce Conflict

- Yacc cannot decide whether to shift (read more input) or
- reduce (finalize if (y) statement;)

```
stmt: IF '(' expr ')' stmt
    | IF '(' expr ')' stmt ELSE stmt    ;
```

Shift/Reduce Conflict

To resolve this, we declare else with higher precedence than if

```
%nonassoc IFX  // A dummy token for resolving ambiguity
%nonassoc ELSE  // Ensures ELSE is resolved first
```

Then, modify the rule

```
stmt: IF '(' expr ')' stmt %prec IFX
    | IF '(' expr ')' stmt ELSE stmt    ;
```

Now, else always binds to the closest if, just like in C.

Reduce/Reduce Conflict

- A Reduce/Reduce conflict occurs when two different grammar rules apply at the same point, and Yacc cannot decide which rule to reduce.

```
list: list ',' ID    | list ',' NUMBER
    | ID    | NUMBER    ;
```

For input: x, y, 5

- Yacc doesn’t know whether to Reduce x, y into list first or Reduce y, 5 first
- Both reductions seem valid at the same point

Reduce/Reduce Conflict

Instead of separate rules for ID and NUMBER, define a single general rule

```
item: ID | NUMBER ;

list: list ',' item

    | item    ;
```

Abstract Syntax Trees (AST)

- When parsing an expression, we often need to construct a structured representation of it.
- This structured representation is called an Abstract Syntax Tree (AST).
- ASTs are crucial for semantic analysis, optimization, and code generation in a compiler.

Abstract Syntax Tree (AST)

An AST is a hierarchical tree representation of the structure of a program.

- Removes unnecessary details (e.g., parentheses, specific keywords).
- Represents the essential structure of an expression or statement.
- Used in the later compiler stages, such as semantic analysis and code generation.