- # Study about Lex and Yacc Tools

Lex and Yacc are essential tools for compiler design that help in lexical analysis and parsing.

1. Structure of Lex

Lex is a lexical analyzer generator that converts input patterns into tokens for further processing by a parser.

Lex Program Structure

A Lex program consists of three sections:

1. **Definitions** - Contains macros and header files.
2. **Rules** - Defines regular expressions and corresponding actions.
3. **User Code** - Contains the main function and additional logic.

Example Lex Code Structure:

```
%{
#include <stdio.h>
%}
%%
[0-9]+  { printf("NUMBER: %s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]* { printf("IDENTIFIER: %s\n", yytext); }
.    { printf("UNKNOWN: %s\n", yytext); }
%%
int main() {
    yylex();
    return 0;
}
```

2. Structure of Yacc

Yacc (Yet Another Compiler Compiler) is a parser generator that processes the tokens from Lex and applies grammar rules to construct a syntax tree.

Yacc Program Structure

A Yacc program consists of three sections:

1. **Definitions** - Includes header files and token declarations.
2. **Rules** - Defines grammar rules and corresponding actions.
3. **User Code** - Contains the main function.

Example Yacc Code Structure:

```
%{
#include <stdio.h>
%}
%token NUMBER
%%
expression: expression '+' expression { printf("Addition\n"); };
%%
int main() {
    yyparse();
    return 0;
}
```

3. Predefined Patterns in Regular Expressions

Lex uses regular expressions to define token patterns. Some common patterns include:

- [0-9]+ → Matches numbers
- [a-zA-Z_][a-zA-Z0-9_]* → Matches identifiers
- "[^"]*" → Matches string literals

4. Auxiliary Functions in Lex and Yacc

- yytext → Stores the matched token text.
- yylex() → Called by Yacc to get the next token from Lex.
- yyparse() → Main parsing function in Yacc.

Conclusion

Lex and Yacc work together to convert source code into structured tokens and parse them into a syntax tree, forming the foundation of a compiler.

Metacharacter	Matches
`.`	any character except newline
`\n`	newline
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`?`	zero or one copy of the preceding expression
`^`	beginning of line
`$`	end of line
`a|b`	`a` or `b`
`(ab)+`	one or more copies of `ab` (grouping)
`"a+b"`	literal "`a+b`" (C escapes still work)
`[]`	character class

Table 1: Pattern Matching Primitives

Expression	Matches
`abc`	`abc`
`abc*`	`ab abc abcc abccc ...`
`abc+`	`abc abcc abccc ...`
`a(bc)+`	`abc abcbc abcbcbc ...`
`a(bc)?`	`a abc`
`[abc]`	one of: `a`, `b`, `c`
`[a-z]`	any letter, a-z
`[a\-z]`	one of: `a`, `-`, `z`
`[-az]`	one of: `-`, `a`, `z`
`[A-Za-z0-9]+`	one or more alphanumeric characters
`[ \t\n]+`	whitespace
`[^ab]`	anything except: `a`, `b`
`[a^b]`	one of: `a`, `^`, `b`
`[a|b]`	one of: `a`, `|`, `b`
`a|b`	one of: `a`, `b`

Table 2: Pattern Matching Examples

expressions in lex are composed of metacharacters (Table 1). Pattern-ma
s are shown in Table 2. Within a character class normal operators lose their me