Practical 14

Title: Prepare a report on YACC and generate Calculator Program using YACC.

Report on YACC

Introduction to YACC

YACC, or Yet Another Compiler Compiler, is a powerful tool used for generating parsers based on a predefined grammar. Developed in the 1970s for the UNIX operating system, YACC plays a crucial role in compiler design, language processing, and syntax analysis tasks. It converts context-free grammar (CFG) rules into C code, helping ensure input or code is syntactically correct.

Key Features of YACC

Context-Free Grammar (CFG): YACC relies on CFG to define the structure of input, specifying the valid sequences of tokens the parser can recognize.

Integration with Lexical Analyzer: YACC works with a lexical analyzer (like Lex or Flex) that converts input into tokens, which are then passed to YACC for parsing.

C Actions for Grammar Rules: For each grammar rule, developers can define actions written in C, executed when the rule is matched. These actions enable tasks like building syntax trees or performing calculations.

How YACC Works

Grammar Specification: Developers define a set of grammar rules that describe the valid language structure. YACC then uses these rules to generate a parser.

Parsing Process: The parser reads tokens produced by the lexical analyzer and attempts to match them to the grammar rules. If the input adheres to the rules, YACC executes the corresponding actions.

Action Execution: For each matched rule, custom actions are executed, enabling the parser to perform tasks like evaluating expressions or building abstract syntax trees.

Components of YACC

Tokens: Basic units like numbers, operators, or identifiers recognized by the lexical analyzer and used by YACC during parsing.

Non-Terminals: Higher-level structures, such as expressions or statements, defined by combinations of tokens and non-terminals.

Precedence and Associativity: YACC allows specifying precedence and associativity rules to resolve grammar ambiguities, such as operator precedence.

Applications of YACC

Compiler Development: YACC is used to build parsers for compilers, converting source code into syntax trees or intermediate code.

Interpreters: It helps create interpreters that parse and directly execute commands or evaluate expressions.

Expression Evaluators: YACC is commonly used in calculators or expression evaluators, parsing and computing arithmetic expressions.

Configuration Processing: It can also be applied to process configuration files or command-line arguments, ensuring correct syntax.

Advantages of YACC

Efficiency: YACC generates efficient parsers that handle complex input structures with speed.

Ease of Use: YACC's syntax for defining grammar rules is simple, and integrating the parser into C programs is straightforward.

C Integration: YACC's output is C code, making it easily embeddable in C-based systems.

Limitations of YACC

Context-Free Grammar Only: YACC is limited to context-free grammars and cannot handle more complex, context-sensitive languages.

Manual Error Recovery: While basic error handling is provided, more advanced error recovery requires custom code.

Left Recursion Issues: YACC doesn't handle left-recursive rules well, requiring developers to convert left-recursive grammars to right-recursive formats.

Conclusion

YACC remains a valuable tool in compiler construction and language processing. Despite some limitations, such as handling only context-free grammars and requiring manual error recovery, its efficiency, ease of integration with C, and ability to generate fast parsers make it indispensable in many programming and language design contexts.

Calculator Program

Code

Parser.y

```
%{
#include <stdio.h>
#include <stdlib.h>
int flag = 0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
/* Rule Section */
%%
ArithmeticExpression:
  E {
    printf("\nResult = %d\n", $$);
    return 0;
```

```
    }
;
E:  E '+' E { $$ = $1 + $3; }
  | E '-' E { $$ = $1 - $3; }
  | E '*' E { $$ = $1 * $3; }
  | E '/' E { $$ = $1 / $3; }
  | E '%' E { $$ = $1 % $3; }
  | '(' E ')' { $$ = $2; }
  | NUMBER { $$ = $1; }
;
%%
// Driver code
int main() {
    printf("\nEnter any arithmetic expression (supports +, -, *, /, % and
parentheses):\n");

    yyparse();
    if (flag == 0) {
        printf("\nEntered arithmetic expression is valid\n\n");
    }
    return 0;
}
void yyerror() {
    printf("\nEntered arithmetic expression is invalid\n\n");
```

```
        flag = 1;
    }
```

Scan.l

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "parser.tab.h"
extern int yylval;
%}
/* Rule Section */
%%
[0-9]+ {
   yylval = atoi(yytext);
   return NUMBER;
}
[\t] ;  /* Ignore tabs */
[\n] return 0;  /* End of line */
. return yytext[0];  /* Return any other character */
%%
int yywrap() {
   return 1;
}
```

Output:

```
C:\Users\Dell\Desktop\COMPILER DESIGN>scan.l

C:\Users\Dell\Desktop\COMPILER DESIGN>parser.y -d

C:\Users\Dell\Desktop\COMPILER DESIGN>lex scan.l

C:\Users\Dell\Desktop\COMPILER DESIGN>yacc parser.y -d

C:\Users\Dell\Desktop\COMPILER DESIGN>gcc lex.yy.c parser.tab.c -w

C:\Users\Dell\Desktop\COMPILER DESIGN>a.exe

Enter any arithmetic expression (supports +, -, *, /,  0x1.420000p-1020nd parenthese
5+9 (2*1)

Result = 14

Entered arithmetic expression is valid


C:\Users\Dell\Desktop\COMPILER DESIGN>_
```

```
C:\Users\Dell\Desktop\COMPILER DESIGN>a.exe

Enter any arithmetic expression (supports +, -, *, /,  0x1.ca0000p-1020nd parentheses):
a+b

Entered arithmetic expression is invalid


C:\Users\Dell\Desktop\COMPILER DESIGN>a.exe

Enter any arithmetic expression (supports +, -, *, /,  0x1.410000p-1020nd parentheses):
9*2*(2+1)

Result = 54

Entered arithmetic expression is valid
```

```
C:\Windows\System32\cmd.exe
C:\Users\Dell\Desktop\COMPILER DESIGN>a.exe

Enter any arithmetic expression (supports +, -, *, /,  0x1.f70000p-1020nd parentheses):
8/2

Result = 4

Entered arithmetic expression is valid


C:\Users\Dell\Desktop\COMPILER DESIGN>a.exe

Enter any arithmetic expression (supports +, -, *, /,  0x1.870000p-1020nd parentheses):
9/3*(9/3)

Result = 9

Entered arithmetic expression is valid
```