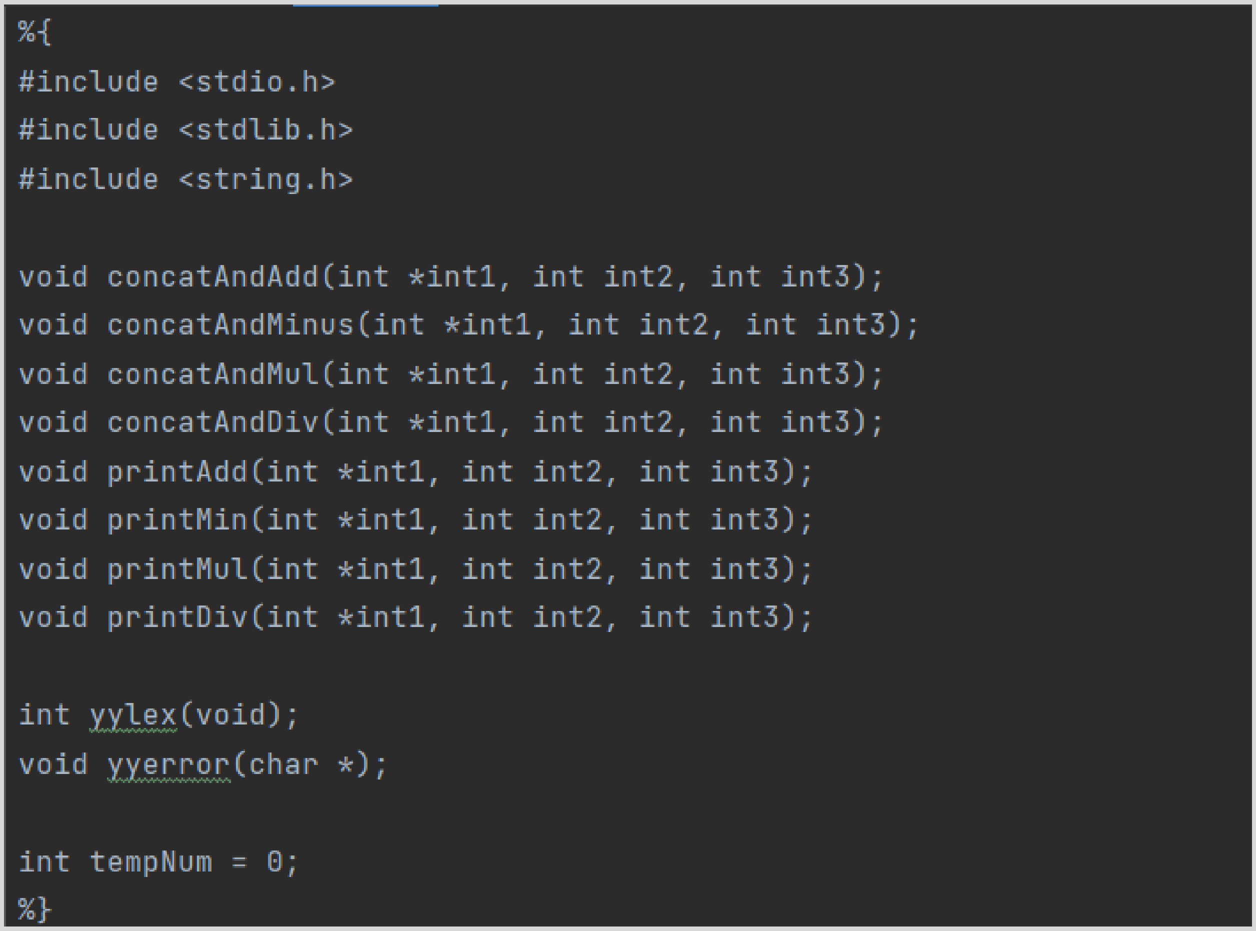
Document for Yacc (Bison) and Lex Code for Arithmetic Expressions Compiler

Overview: This document provides a detailed explanation of the Yacc (Bison) and Lex code for a simple arithmetic expressions compiler. The code consists of a parser (Yacc) and lexer (Lex) that work together to evaluate arithmetic expressions and generate intermediate code.

Yacc (Bison) Code:

Declarations (%{...%}):

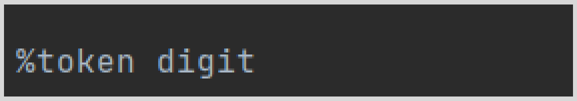


```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void concatAndAdd(int *int1, int int2, int int3);
void concatAndMinus(int *int1, int int2, int int3);
void concatAndMul(int *int1, int int2, int int3);
void concatAndDiv(int *int1, int int2, int int3);
void printAdd(int *int1, int int2, int int3);
void printMin(int *int1, int int2, int int3);
void printMul(int *int1, int int2, int int3);
void printDiv(int *int1, int int2, int int3);

int yylex(void);
void yyerror(char *);

int tempNum = 0;
%}
```

This section includes necessary C libraries and declares functions and variables used in the parser.

.

Token Declarations (%token):

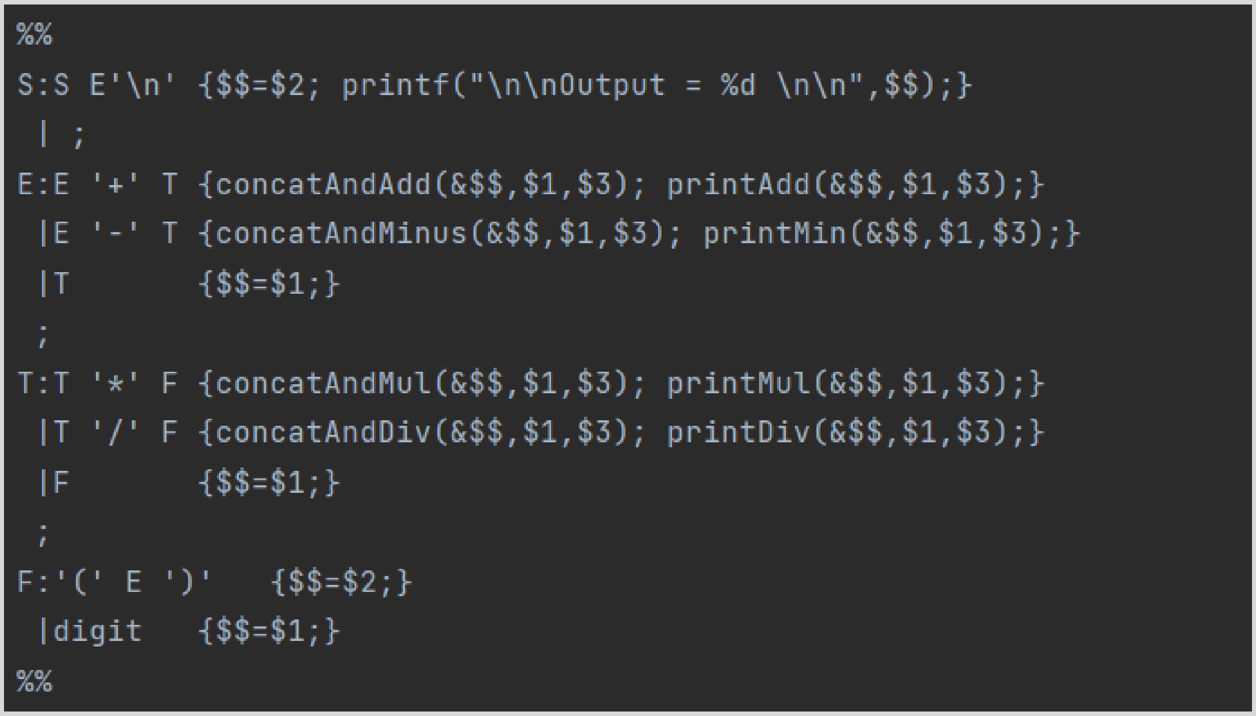


```
%token digit
```

Defines the token 'digit' to represent numeric values in the grammar.

.

Grammar Rules (%%):



```
%%
S:S E'\n' {$$=$2; printf("\n\nOutput = %d \n\n",$$);}
 | ;
E:E '+' T {concatAndAdd(&$$,$1,$3); printAdd(&$$,$1,$3);}
 |E '-' T {concatAndMinus(&$$,$1,$3); printMin(&$$,$1,$3);}
 |T       {$$=$1;}
 ;
T:T '*' F {concatAndMul(&$$,$1,$3); printMul(&$$,$1,$3);}
 |T '/' F {concatAndDiv(&$$,$1,$3); printDiv(&$$,$1,$3);}
 |F       {$$=$1;}
 ;
F:'(' E ')'   {$$=$2;}
 |digit   {$$=$1;}
%%
```

S -> S E '\n':

Represents the start symbol 'S' producing an expression 'E' followed by a newline character. Prints the final output.

E -> E '+' T | E '-' T | T:

Defines rules for arithmetic expressions with addition and subtraction operations. Calls corresponding functions for concatenation and prints the intermediate code.

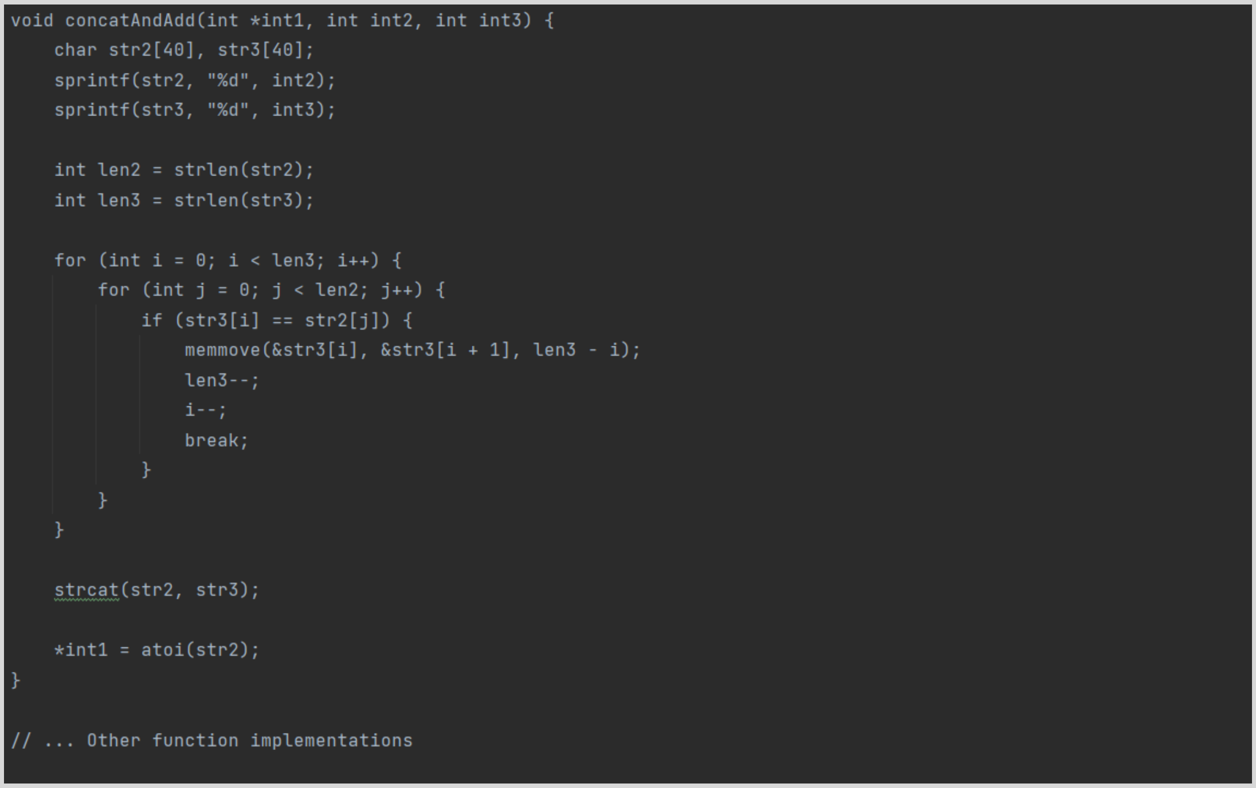
T -> T '*' F | T '/' F | F:

Defines rules for arithmetic expressions with multiplication and division operations. Calls corresponding functions for concatenation and prints the intermediate code.

F -> '(' E ')' | digit:

Defines rules for arithmetic expressions enclosed in parentheses or individual digits.

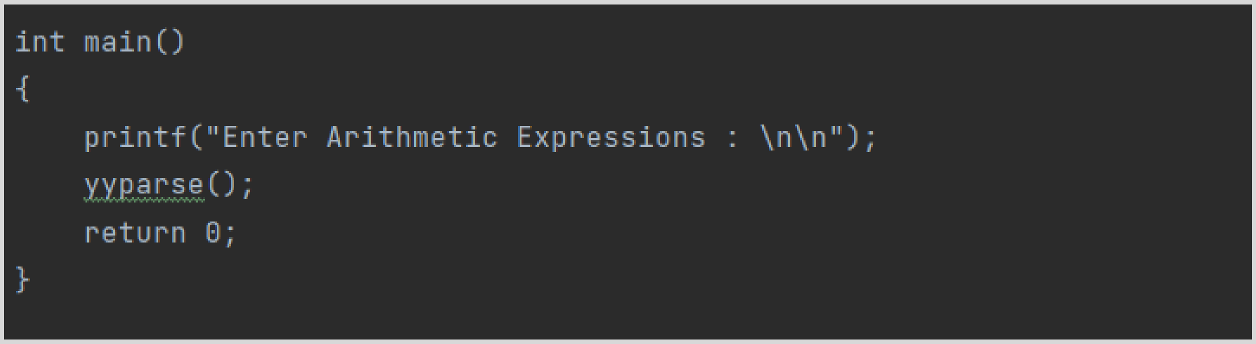
Functions:



```c
void concatAndAdd(int *int1, int int2, int int3) {
    char str2[40], str3[40];
    sprintf(str2, "%d", int2);
    sprintf(str3, "%d", int3);

    int len2 = strlen(str2);
    int len3 = strlen(str3);

    for (int i = 0; i < len3; i++) {
        for (int j = 0; j < len2; j++) {
            if (str3[i] == str2[j]) {
                memmove(&str3[i], &str3[i + 1], len3 - i);
                len3--;
                i--;
                break;
            }
        }
    }

    strcat(str2, str3);

    *int1 = atoi(str2);
}

// ... Other function implementations
```

Functions for concatenating integers with various arithmetic operations and printing the corresponding three-address code.

Main Function:



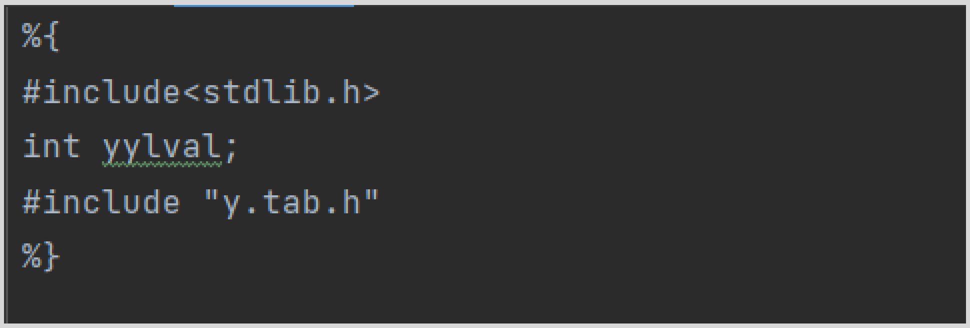
```c
int main()
{
    printf("Enter Arithmetic Expressions : \n\n");
    yyparse();
    return 0;
}
```

Takes user input for arithmetic expressions, invokes the parser, and prints the final result or an error message if the expression is invalid.

Combines the Yacc parser and Lex lexer to create a complete compiler for arithmetic expressions.

Lex Code:

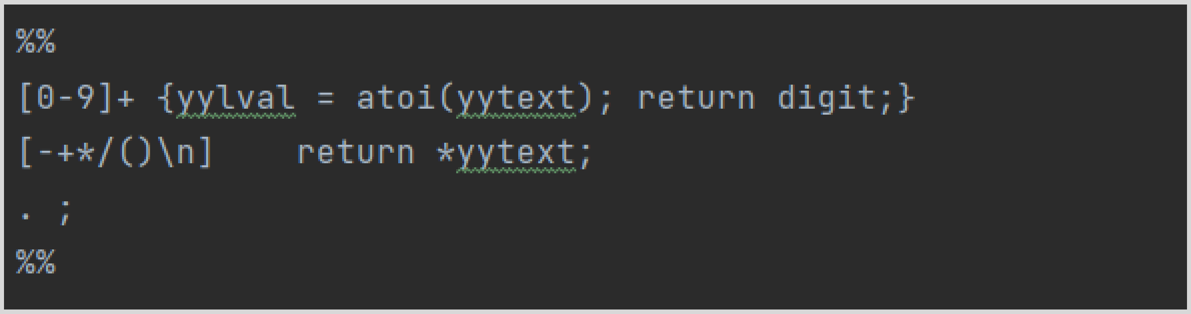
Declarations (%{...%}):



```
%{
#include<stdlib.h>
int yylval;
#include "y.tab.h"
%}
```

Includes standard C libraries, declares yylval, and includes the header file "y.tab.h" generated by Yacc.

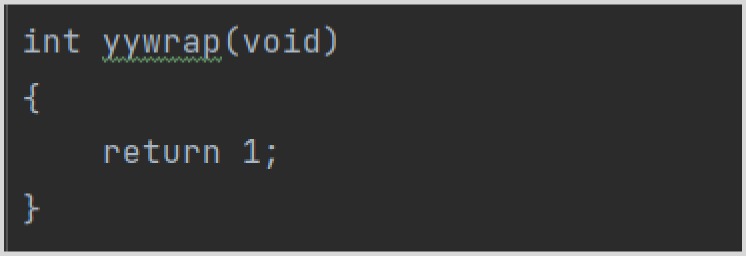
Regular Expressions and Rules (%%):



```
%%
[0-9]+ {yylval = atoi(yytext); return digit;}
[-+*/()\n]    return *yytext;
. ;
%%
```

Defines regular expressions for digits and operators and returns corresponding tokens for recognized patterns.

yywrap Function:



```
int yywrap(void)
{
    return 1;
}
```

Indicates the end of input.

Execution Flow:

User enters arithmetic expressions.

Lex tokenizes the input.

Yacc parses the tokens and generates intermediate code.

Intermediate code is printed, showing the steps involved in the arithmetic operations.

Conclusion:

The provided code demonstrates a simple compiler for arithmetic expressions, including lexical analysis, parsing, and intermediate code generation. It showcases how Yacc and Lex work together to build a compiler for a specific language. The generated intermediate code provides insight into the step-by-step execution of arithmetic operations.

Ali Bardestani