Listing 1: C program for testing

```

/************************************************
                  INCLUDES.HPP
************************************************/

#ifndef I_INCLUDES_H_
#define I_INCLUDES_H_

#include <string>
#include <vector>
#include <map>
#include <utility>
#include <stack>
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cctype>
#include <cstdlib>
#include <set>

using namespace std;

typedef unsigned long long bit64;
typedef char bit8;
typedef unsigned int bit32;
typedef unsigned short bit16;

extern bit64 line_no;
extern bit64 col_no;
extern int indentLevel;

class I_error{
    public:
        I_error(const char *msg):
            error_message_(msg), line_no_(line_no), col_no_(col_no) {}

        void print(){
            cout<<"ERROR␣::␣LINE="<<line_no_<<"␣CHARACTER="<<col_no_
                <<"␣::␣"<<error_message_<<endl;
            exit(0);
        }

    private:
        const char *error_message_;
        int line_no_;
        int col_no_;
};

#endif

/*************************************************
                       LEXER.HPP
*************************************************/

#ifndef I_LEXER_H_
#define I_LEXER_H_

#include "includes.hpp"
using namespace std;


enum TokenType{ SYMBOL = 1536, INTEGER, REAL, OPERATOR, STRING , _NONE, VAR_MARKER };
const char *typeString(TokenType t);

enum Operator{  PLUS = 7654, MINUS, MULT, DIV, MOD, POW,
                AND, OR, NOT, XOR,
                GT, LT, GTE, LTE, EQ, NEQ ,
                COND, ELSE, CONSTRAINT,
                LPAREN, RPAREN, ERROR,
                LCURL, RCURL, LBRACKET, RBRACKET,
                ESCAPE, ASSIGN, COMMA, VAR };

const char *opString(Operator op);
Operator str2op(const char *str);

union TokenHolder{
    int integer;
    double real;
    char str[1024];
    char sym[1024];
    Operator op;
};

class Token{
    public:
        TokenType type_;
        TokenHolder token_;

        Token():type_(_NONE){}
        Token(TokenType type, const char *tok_str){
            makeToken(type, tok_str);
        }

        Token (const Token &oldToken){
            type_ = oldToken.type_;
            token_ = oldToken.token_;
        }

        void makeToken(TokenType type, const char *tok_str);

        inline TokenType getType(){ return type_; }
        inline void print(){
            cout<<"<"<<typeString(type_)<<">";
            switch(type_){
                case INTEGER:   cout<<token_.integer;           break;
                case REAL:      cout<<token_.real;              break;
                case SYMBOL:    cout<<token_.sym;               break;
                case STRING:    cout<<token_.str;               break;
                case VAR_MARKER:
                case OPERATOR:  cout<<opString(token_.op);      break;
                case _NONE:     cout<<"␣";                           break;
                default:        cout<<"␣";                           break;
            }
            cout<<"</"<<typeString(type_)<<">"<<endl;
        }
};

class Tokenizer{
    public:
        Tokenizer():expr_(NULL),type_(_NONE){}
        Tokenizer(const char *expr):expr_(expr){}

        inline bool next(Token &t){
            bool ret = getNextToken();
            t.makeToken(type_, token_);
            return ret;
        }

        inline const char* getExpr(){ return expr_; }

    private:
        const char *expr_;
        char token_[1024];
        TokenType type_;

        inline bool isOperator(char ch){
            if(strchr("+-*/%^=(){}[]&|~><?:@,\\", ch)) return true;
            return false;
        }

        bool getNextToken();
};

#endif



/*************************************************
                      LEXER.CPP
*************************************************/

#include "lexer.hpp"
using namespace std;

const char *typeString(TokenType t){
    switch(t){
        case SYMBOL:
            return "SYMBOL";
        case INTEGER:
            return "INTEGER";
        case REAL:
            return "REAL";
        case OPERATOR:
            return "OPERATOR";
        case STRING:
            return "STRING";
        case _NONE:
            return "_NONE";
        case VAR_MARKER:
            return "VAR_MARKER";
        default:
            return "ERROR ...";
    }
}

const char *opString(Operator op){
    switch(op){
        case PLUS:  return "PLUS";
        case MINUS: return "MINUS";
        case MULT:  return "MULT";
        case DIV:   return "DIV";
        case MOD:   return "MOD";
        case POW:   return "POW";
        case AND:   return "AND";
        case OR:    return "OR";
        case NOT:   return "NOT";
        case XOR:   return "XOR";
        case GT:    return "GT";
        case LT:    return "LT";
        case GTE:   return "GTE";
        case LTE:   return "LTE";
        case EQ:    return "EQ";
        case NEQ:   return "NEQ";
        case LPAREN:return "LPAREN";
        case RPAREN:return "RPAREN";
        case ERROR: return "ERROR";
        case ESCAPE:return "ESCAPE";
        case ASSIGN:return "ASSIGN";
        case LCURL: return "LCURL";
        case RCURL: return "RCURL";
        case COMMA: return "COMMA";
        case VAR:   return "VAR";
        case LBRACKET:  return "LBRACKET";
        case RBRACKET:  return "RBRACKET";
        case CONSTRAINT:return "CONSTRAINT";
        case COND:  return "COND";
        case ELSE:  return "ELSE";
        default:    return "\0";
    }
}

Operator str2op(const char *str){
    switch(str[0]){
        case '+': return PLUS;
        case '-': return MINUS;
        case '*':
            if( str[1]=='*') return POW;
            return MULT;
        case '/': return DIV;
        case '%': return MOD;
        case '&': return AND;
        case '|': return OR;
        case '~':
            if( str[1]=='=') return NEQ;
            return NOT;
        case '^': return XOR;
        case '>':
            if( str[1]=='=') return GTE;
            return GT;
        case '<':
            if( str[1]=='=') return LTE;
            return LT;
        case '=':
            if(str[1]=='=') return EQ;
            return ASSIGN;
        case '(': return LPAREN;
        case ')': return RPAREN;
        case '{': return LCURL;
        case '}': return RCURL;
        case '\\':return ESCAPE;
        case ',': return COMMA;
        case '$': return VAR;
        case '[': return LBRACKET;
        case ']': return RBRACKET;
        case '?': return COND;
        case ':': return ELSE;
        case '@': return CONSTRAINT;
        default:
            return ERROR;
    }
}


void Token::makeToken(TokenType type, const char *tok_str){
    type_ = type;
    switch(type_){
        case SYMBOL:
            strcpy( token_.sym, tok_str);
            break;
        case INTEGER:
            token_.integer = atoi(tok_str);
            break;
        case REAL:
            token_.real = atof(tok_str);
            break;
        case STRING:
            strcpy(token_.str, tok_str);
            break;
        case VAR_MARKER:
```

```
        case OPERATOR:
            token_.op = str2op(tok_str);
            break;
        case _NONE:
            break;
        default:
            cout<<"ERROR␣OCCURED␣IN␣LEXICAL␣ANALYSIS"<<endl;
            break;
    }
}


bool Tokenizer::getNextToken(){
    char *temp;
    type_ = _NONE;
    temp = token_;
    *temp = '\0';

    if(!*expr_) return false;
    while(isspace(*expr_)) {
        expr_++ ;
        col_no++;
    }

    if( isOperator(*expr_)){
        switch(*expr_){
            case '*':
                *temp++ = *expr_++;
                col_no++;
                if(*(expr_+1)=='*'){
                    *temp++ = *expr_++;
                    col_no++;
                }
                break;
            case '~':
            case '>':
            case '<':
            case '=':
                *temp++ = *expr_++;
                col_no++;
                if(*(expr_+1)=='='){
                    *temp++ = *expr_++;
                    col_no++;
                }
                break;
            default:
                *temp++ = *expr_++;
                col_no++;
                break;
        }
        type_ = OPERATOR;
    }
    else if(*expr_ == '$'){
        *temp++ = *expr_++;
        col_no++;
        type_ = VAR_MARKER;
    }
    else if(isalpha(*expr_)){
        while(isalpha(*expr_) || *expr_ == '_'){
            *temp++ = *expr_++;
            col_no++;
        }
        type_ = SYMBOL;
    }
    else if(isdigit(*expr_)){
        while(isdigit(*expr_)){
            *temp++ = *expr_++;
            col_no++;
        }
        if(*expr_ == '.'){
            *temp++ = *expr_++;
            col_no++;
            while(isdigit(*expr_)){
                *temp++ = *expr_++;
                col_no++;
            }
            type_ = REAL;
        }
        else type_ = INTEGER;
    }
    else if(*expr_ == '\"'){
        *expr_++;
        col_no++;
        while(*expr_ != '\"'){
            *temp++ = *expr_++;
            col_no++;
        }
        *expr_++;
        col_no++;
        type_ = STRING;
    }
    else{
        while(!isspace(*expr_)){
            *temp++ = *expr_++;
            col_no++;
        }
        type_ = SYMBOL;
    }

    *temp = '\0';
    return true;
}


/**************************************************
                      PARSER.HPP
**************************************************/

#ifndef I_PARSER_H_
#define I_PARSER_H_

#include "lexer.hpp"
using namespace std;

enum atomType{ TOKEN, SEXPR, LEXPR, VARIABLE };
struct atom{
    atomType type_;
    void *atom_;

    void print();
};

struct Logical_not{
    bool not_;
    vector<atom> logical_not_;        // Unary operator ~

    Logical_not(){
        not_ = false;
    }
    void print();
};

struct Logical_xor{
    vector<Logical_not> logical_xor_;    // values separated by ^
    void print();
```

```
};

struct Logical_and{
    vector<Logical_xor> logical_and_;    // values separated by &
    void print();
};

struct Logical_or{
    vector<Logical_and> logical_or_; // Values separated by |
    void print();
};

struct Unary{
    bool minus_;
    vector<Logical_or> unary_;  // Unary operators +, -

    Unary(){
        minus_ = false;
    }
    void print();
};

struct Exponent{
    vector<Unary> exponent_;     // Values separated by exponent op **
    void print();
};

struct Modulus{                  // Values separated by %
    vector<Exponent> modulus_;
    void print();
};

struct FactorDiv{                    // Values separated by *, /
    vector<Modulus> factor_div_;
    void print();
};

struct FactorMult{
    struct vector<FactorDiv> factor_mult_;
    void print();
};

struct TermAdd{                  //Values separated by + and -
    vector<FactorMult> term_add_;
    void print();
```

```
};

struct TermSub{
    vector<TermAdd> term_sub_;
    void print();
};

struct Component{               //The if-else operator _?_:_
    vector<TermSub> component_;
    void print();
};

struct LExpr{                   //Comma separated values
    vector<Component> lexpr_;
    void print();
};


struct Variable{
    LExpr *context_;
    char var_name_[1024];

    Variable(){
        context_ = NULL;
    }

    void print();
};


struct SExpr{
    vector<atom> sexpr_;
    void print();
};

struct Statement{
    bool store_;
    SExpr *lval_;
    SExpr *rval_;

    Statement():store_(true),lval_(NULL),rval_(NULL){}
    ~Statement(){
        delete lval_;
        delete rval_;
    }
```

```cpp

    inline void print(){
        cout<<"<STATEMENT>"<<endl;
        if(lval_ != NULL){
            cout<<"<L-VALUE>"<<endl;
            lval_->print();
            cout<<"</L-VALUE>"<<endl;
        }

        if(rval_ != NULL){
            cout<<"<R-VALUE>"<<endl;
            rval_->print();
            cout<<"</R-VALUE>"<<endl;
        }
        cout<<"</STATEMENT>"<<endl;
    }
};







class Logical_notParser{
    private: size_t parse_(size_t begin, Logical_not &expr);

    public:
        Logical_notParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        Logical_notParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        Logical_notParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }

        void destruct(){
            vector<Token>().swap(tok_stream_);        //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<Logical_not, size_t> getLogical_not();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

};

class Logical_xorParser{
    public:
        Logical_xorParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        Logical_xorParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        Logical_xorParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);      //clear and dellocate memory
        }
        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }
        pair<Logical_xor, size_t> getLogical_xor();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

        size_t parse_(size_t begin, Logical_xor &expr);

};
```

```

class Logical_andParser{
    public:
        Logical_andParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        Logical_andParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        Logical_andParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);      //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<Logical_and, size_t> getLogical_and();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

        size_t parse_(size_t begin, Logical_and &expr);

};

class Logical_orParser{
    public:
        Logical_orParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        Logical_orParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        Logical_orParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);      //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<Logical_or, size_t> getLogical_or();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

        size_t parse_(size_t begin, Logical_or &expr);

};

class UnaryParser{
    public:
        UnaryParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        UnaryParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        UnaryParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }

        void destruct(){
            vector<Token>().swap(tok_stream_);      //clear and dellocate memory
        }
        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<Unary, size_t> getUnary();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

        size_t parse_(size_t begin, Unary &expr);

};

class ExponentParser{
    public:
        ExponentParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        ExponentParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        ExponentParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);      //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<Exponent, size_t> getExponent();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

        size_t parse_(size_t begin, Exponent &expr);

```

```
};

class ModulusParser{
    public:
        ModulusParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        ModulusParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        ModulusParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);      //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<Modulus, size_t> getModulus();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

        size_t parse_(size_t begin, Modulus &expr);

};

class FactorDivParser{
    public:
        FactorDivParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        FactorDivParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        FactorDivParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);      //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<FactorDiv, size_t> getFactorDiv();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

        size_t parse_(size_t begin, FactorDiv &expr);

};

class FactorMultParser{
    public:
        FactorMultParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        FactorMultParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        FactorMultParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);      //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<FactorMult, size_t> getFactorMult();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

        size_t parse_(size_t begin, FactorMult &expr);

};

class TermAddParser{
    public:
        TermAddParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        TermAddParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        TermAddParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);      //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<TermAdd, size_t> getTermAdd();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

        size_t parse_(size_t begin, TermAdd &expr);
```

```

};

class TermSubParser{
    public:
        TermSubParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        TermSubParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        TermSubParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);      //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<TermSub, size_t> getTermSub();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

        size_t parse_(size_t begin, TermSub &expr);

};

class ComponentParser{
    public:
        ComponentParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        ComponentParser(const vector<Token> &tok_stream)
```

```
            :tok_stream_(tok_stream){}
        ComponentParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);        //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<Component, size_t> getComponent();

    private:
        vector<Token> tok_stream_;
        Operator delimiter_;

        size_t parse_(size_t begin, Component &expr);

};

class LExprParser{
    public:
        LExprParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        LExprParser(const vector<Token> &tok_stream):tok_stream_(tok_stream){}
        LExprParser(){
            delimiter_ = RCURL;
        }

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);        //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<LExpr *,size_t> getLExpr();

    private:
        const char *expr_;
        vector<Token> tok_stream_;
        Token tok_;
        LExpr *lexpr_;
        bit32 curl_match_;
        Operator delimiter_;

        size_t parse_(size_t begin, LExpr *lexpr);
};

class VariableParser{
    public:
        VariableParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        VariableParser(const vector<Token> &tok_stream):tok_stream_(tok_stream){}
        VariableParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);       //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        pair<Variable,size_t> getVariable();


    private:
        vector<Token> tok_stream_;
        Token tok_;
        Operator delimiter_;

        size_t parse_(size_t begin, Variable &var);
};


class SExprParser{
    public:
        SExprParser(const char *expr):expr_(expr),lex_(expr){
            while(lex_.next(tok_)) tok_stream_.push_back(tok_);
            bareDelimited_ = false;
        }
        SExprParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
            bareDelimited_ = false;
        }
        SExprParser(const vector<Token> &tok_stream):tok_stream_(tok_stream){
            bareDelimited_ = false;
        }
        SExprParser(){
            bareDelimited_ = false;
        }

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
            delimiter_ = RPAREN;
        }
        void destruct(){
            vector<Token>().swap(tok_stream_);        //clear and dellocate memory
        }

        inline void setDelimiter(Operator op){
            delimiter_ = op;
        }

        inline void setBareDelimited(){
            bareDelimited_ = true;
        }

        pair<SExpr *,size_t> getSExpr();

    private:
        const char *expr_;
        vector<Token> tok_stream_;
        Token tok_;
        SExpr *sexpr_;
        Tokenizer lex_;
        bit32 paren_match;
        Operator delimiter_;
        bool bareDelimited_;

        size_t parse_(size_t begin, SExpr *sexpr);
};


class StatementParser{
    public:
        StatementParser(const char *expr):expr_(expr){
            Tokenizer lex_(expr);
            Token t;
            vector<Token> tok_stream_l;
            vector<Token> tok_stream_r;

            stmt_ = new Statement;

            while(lex_.next(t)){
                if(t.type_ == OPERATOR){
                    if(t.token_.op == ASSIGN){
                        break;
                    }
                    else tok_stream_l.push_back(t);
                }
                else tok_stream_l.push_back(t);
            }
            SExprParser lval_parser_(tok_stream_l);
            stmt_->lval_ = lval_parser_.getSExpr().first;
            int count = 0;
            while(lex_.next(t)){
                count++;
                tok_stream_r.push_back(t);
            }
            if(!count)stmt_->store_=false;
            else{
                SExprParser rval_parser_(tok_stream_r);
                stmt_->rval_ = rval_parser_.getSExpr().first;
            }
        }

        inline Statement *getStmt(){
            return stmt_;
        }


    private:
        const char *expr_;
        vector <Token> tok_stream_;
        Statement *stmt_;
};

#endif

/***************************************************
                    PARSER.CPP
***************************************************/

#include "parser.hpp"
using namespace std;
size_t SExprParser::parse_(size_t begin, SExpr *sexpr){
    size_t i;
    atom myAtom;
    SExpr *newSexpr;
    LExprParser lParser;
    VariableParser vParser;
    pair<Variable, size_t> vres;
    Variable *var;
    SExprParser sParser;
    pair<SExpr *, size_t> sres;
    pair<LExpr *,size_t> lres;


    for(i=begin; i<tok_stream_.size(); i++){
        //tok_stream_[i].print();
        if(tok_stream_[i].type_ == OPERATOR){
            switch(tok_stream_[i].token_.op){
                case LPAREN:
                    paren_match++;
                    sParser.construct(tok_stream_, ++i);
                    sParser.setDelimiter(RPAREN);
                    sres = sParser.getSExpr();
                    i += sres.second;
                    newSexpr = sres.first;
                    myAtom.type_=SEXPR;
                    myAtom.atom_ = (void *)newSexpr;
                    sexpr->sexpr_.push_back(myAtom);
                    break;
                case LCURL:
                    lParser.construct(tok_stream_, ++i);
                    lParser.setDelimiter(RCURL);
                    lres = lParser.getLExpr();
                    i += lres.second;
                    myAtom.type_ = LEXPR;
                    myAtom.atom_ = (void *)lres.first;
                    sexpr->sexpr_.push_back(myAtom);
                    break;
                default:
                    if(bareDelimited_){
                        return i;
                    }
                    if(tok_stream_[i].token_.op == delimiter_){
                        return ++i;
                    }
                    myAtom.type_=TOKEN;
                    Token *myToken = new Token(tok_stream_[i]);
                    myAtom.atom_ = (void *)myToken;
                    sexpr->sexpr_.push_back(myAtom);
                    break;
            }
        }
        else if(tok_stream_[i].type_ == VAR_MARKER){
            vParser.construct(tok_stream_, ++i);
            vres = vParser.getVariable();
            i += vres.second;
            var = new Variable(vres.first);
            myAtom.type_ = VARIABLE;
            myAtom.atom_ = (void *)var;
            sexpr->sexpr_.push_back(myAtom);
        }
        else{
            myAtom.type_ = TOKEN;
            Token *myToken = new Token(tok_stream_[i]);
            myAtom.atom_ = (void *)myToken;
            sexpr->sexpr_.push_back(myAtom);
        }
```

```
    }
    //cout<<"\n\n";
    return i;
}


pair<SExpr *,size_t> SExprParser::getSExpr(){
    pair<SExpr *, size_t> ret;
    paren_match = 0;
    ret.first = new SExpr;
    ret.second = parse_(0,ret.first);
    destruct();
    /*if(paren_match){
        I_error e("Parentheses mismatch");
        e.print();
    }*/
    return ret;
}


pair<Variable, size_t> VariableParser::getVariable(){
    pair<Variable, size_t> ret;
    Variable var;
    ret.first = var;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t VariableParser::parse_(size_t begin, Variable &var){
    size_t i;
    LExprParser lParser;
    pair<LExpr *,size_t> lres;

    i=begin;
    if(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op==LBRACKET){
            lParser.construct(tok_stream_,++i);
            lParser.setDelimiter(RBRACKET);
            lres = lParser.getLExpr();
            i += lres.second;

            var.context_ = lres.first;
            //i++;
        }
        else{
            I_error e("1␣Wrong␣usage␣of␣VARIABLE");
            e.print();
        }
    }
    else if(tok_stream_[i].type_==SYMBOL);
    else{
        I_error e("2␣Wrong␣usage␣of␣VARIABLE");
        e.print();
    }
    //tok_stream_[i].print();
    if(tok_stream_[i].type_ == SYMBOL){
        strcpy(var.var_name_, tok_stream_[i].token_.sym);
        return ++i;
    }
    else{
        I_error e("3␣Wrong␣usage␣of␣VARIABLE");
        e.print();
    }
}




pair<Logical_not, size_t> Logical_notParser::getLogical_not(){
    pair<Logical_not, size_t> ret;
    Logical_not expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t Logical_notParser::parse_(size_t begin, Logical_not &expr){
    size_t i = begin;
    SExprParser sParser;
    VariableParser vParser;
    LExprParser lParser;
    Variable *var;
    pair<Variable, size_t> vres;
    pair<SExpr *, size_t> sres;
    pair<LExpr *, size_t> lres;
    atom myAtom;

    if(tok_stream_[begin].type_==OPERATOR){
        if(tok_stream_[begin].token_.op==NOT){
            expr.not_ = true;
            i++;
        }
        else expr.not_ = false;
    }

    size_t count = 0,j;
    for(j=i; j<tok_stream_.size(); j++){
        //tok_stream_[j].print();
        if(tok_stream_[j].type_ == VAR_MARKER){
            vParser.construct(tok_stream_, ++j);
            vres = vParser.getVariable();
            j += vres.second;
        }

        if(tok_stream_[j].type_==OPERATOR){
            if(tok_stream_[j].token_.op == LPAREN){
                sParser.construct(tok_stream_, ++j);
                sres = sParser.getSExpr();
                j += sres.second;
                j--;
            }
            else if(tok_stream_[j].token_.op == LCURL){
                lParser.construct(tok_stream_, ++j);
                lres = lParser.getLExpr();
                j += lres.second;
                j--;
            }
            else if(tok_stream_[j].token_.op == delimiter_){
                //count--;
                break;
            }
            else{
                break;
            }
        }
        count++;
    }
    //cout<<"\n\n";
    if(count > 1){
        sParser.construct(tok_stream_, i);
        sParser.setBareDelimited();
        //cout<<endl<<"BARE DELIMITED SET\n"<<endl;
        sres = sParser.getSExpr();
        i += sres.second;
        myAtom.type_ = SEXPR;
        myAtom.atom_ = (void *)sres.first;
        expr.logical_not_.push_back(myAtom);
    }
    else{
        //tok_stream_[i].print();
        if(tok_stream_[i].type_ == VAR_MARKER){
            myAtom.type_ = VARIABLE;
            var = new Variable(vres.first);
            myAtom.atom_ = (void *)var;
            expr.logical_not_.push_back(myAtom);
        }
        else if(tok_stream_[i].type_ == OPERATOR){
            if(tok_stream_[i].token_.op == LPAREN){
                myAtom.type_ = SEXPR;
                myAtom.atom_ = (void *)(sres.first);
                expr.logical_not_.push_back(myAtom);
            }
            else if(tok_stream_[i].token_.op == LCURL){
                myAtom.type_ = LEXPR;
                myAtom.atom_ = (void *)(lres.first);
                expr.logical_not_.push_back(myAtom);
            }
            else if(tok_stream_[i].token_.op == delimiter_){
                return ++i;
            }
            else{
                I_error e("WRONG␣FORMAT␣OF␣ARITHMATIC␣EXPRESSION,␣AMBIGUOUS");
                e.print();
            }
        }
        else{
            myAtom.type_ = TOKEN;
            Token *myTok = new Token(tok_stream_[i]);
            myAtom.atom_ = (void *)myTok;
            expr.logical_not_.push_back(myAtom);
        }
        //cout<<"\n\n"<<j<<"\n\n";
    }
    return j;
}


pair<Logical_xor, size_t> Logical_xorParser::getLogical_xor(){
    pair<Logical_xor, size_t> ret;
    Logical_xor expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t Logical_xorParser::parse_(size_t begin, Logical_xor &expr){
    Logical_notParser lnParser;
    pair<Logical_not, size_t> lnres;

    size_t i=begin;
    lnParser.construct(tok_stream_, i);
    lnParser.setDelimiter(delimiter_);
    lnres = lnParser.getLogical_not();

    i += lnres.second;
    expr.logical_xor_.push_back(lnres.first);


    while(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op == XOR){
            lnParser.construct(tok_stream_, ++i);
            lnParser.setDelimiter(delimiter_);
            lnres = lnParser.getLogical_not();

            i+= lnres.second;
            expr.logical_xor_.push_back(lnres.first);
        }
        else break;
    }

    return i;
}


pair<Logical_and, size_t> Logical_andParser::getLogical_and(){
    pair<Logical_and, size_t> ret;
    Logical_and expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}
```

```

size_t Logical_andParser::parse_(size_t begin, Logical_and &expr){
    Logical_xorParser lxParser;
    pair<Logical_xor, size_t> lxres;

    size_t i=begin;
    lxParser.construct(tok_stream_, i);
    lxParser.setDelimiter(delimiter_);
    lxres = lxParser.getLogical_xor();

    i += lxres.second;
    expr.logical_and_.push_back(lxres.first);


    while(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op == AND){
            lxParser.construct(tok_stream_, ++i);
            lxParser.setDelimiter(delimiter_);
            lxres = lxParser.getLogical_xor();

            i+= lxres.second;
            expr.logical_and_.push_back(lxres.first);
        }
        else break;
    }

    return i;
}

pair<Logical_or, size_t> Logical_orParser::getLogical_or(){
    pair<Logical_or, size_t> ret;
    Logical_or expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t Logical_orParser::parse_(size_t begin, Logical_or &expr){
    Logical_andParser loParser;
    pair<Logical_and, size_t> lores;

    size_t i=begin;
    loParser.construct(tok_stream_, i);
    loParser.setDelimiter(delimiter_);
    lores = loParser.getLogical_and();

    i += lores.second;
    expr.logical_or_.push_back(lores.first);


    while(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op == OR){
            loParser.construct(tok_stream_, ++i);
            loParser.setDelimiter(delimiter_);
            lores = loParser.getLogical_and();

            i+= lores.second;
            expr.logical_or_.push_back(lores.first);
        }
        else break;
    }

    return i;
}

pair<Unary, size_t> UnaryParser::getUnary(){
    pair<Unary, size_t> ret;
    Unary expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t UnaryParser::parse_(size_t begin, Unary &expr){
    Logical_orParser loParser;
    pair<Logical_or, size_t> lores;
    size_t i = begin;
    bool unary_op_ = false;

    if(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op==PLUS || tok_stream_[i].token_.op==MINUS){
            unary_op_ = true;
            if(tok_stream_[i].token_.op == MINUS) expr.minus_ =true;
            loParser.construct(tok_stream_, ++i);
            loParser.setDelimiter(delimiter_);
            lores = loParser.getLogical_or();
            i += lores.second;
            expr.unary_.push_back(lores.first);
```

```
                return i;
            }
        }

        if(! unary_op_){
            loParser.construct(tok_stream_, i);
            loParser.setDelimiter(delimiter_);
            lores = loParser.getLogical_or();
            i += lores.second;
            expr.unary_.push_back(lores.first);
            return i;
        }
}


pair<Exponent, size_t> ExponentParser::getExponent(){
    pair<Exponent, size_t> ret;
    Exponent expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t ExponentParser::parse_(size_t begin, Exponent &expr){
    UnaryParser unParser;
    pair<Unary, size_t> unres;

    size_t i=begin;
    unParser.construct(tok_stream_, i);
    unParser.setDelimiter(delimiter_);
    unres = unParser.getUnary();

    i += unres.second;
    expr.exponent_.push_back(unres.first);


    while(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op == POW){
            unParser.construct(tok_stream_, ++i);
            unParser.setDelimiter(delimiter_);
            unres = unParser.getUnary();

            i+= unres.second;
            expr.exponent_.push_back(unres.first);
        }
        else break;
    }

    return i;
}



pair<Modulus, size_t> ModulusParser::getModulus(){
    pair<Modulus, size_t> ret;
    Modulus expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t ModulusParser::parse_(size_t begin, Modulus &expr){
    ExponentParser exParser;
    pair<Exponent, size_t> exres;

    size_t i=begin;
    exParser.construct(tok_stream_, i);
    exParser.setDelimiter(delimiter_);
    exres = exParser.getExponent();

    i += exres.second;
    expr.modulus_.push_back(exres.first);


    while(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op == MOD){
            exParser.construct(tok_stream_, ++i);
            exParser.setDelimiter(delimiter_);
            exres = exParser.getExponent();

            i+= exres.second;
            expr.modulus_.push_back(exres.first);
        }
        else break;
    }

    return i;
}
```

```

pair<FactorDiv, size_t> FactorDivParser::getFactorDiv(){
    pair<FactorDiv, size_t> ret;
    FactorDiv expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t FactorDivParser::parse_(size_t begin, FactorDiv &expr){
    ModulusParser moParser;
    pair<Modulus, size_t> mores;

    size_t i=begin;
    moParser.construct(tok_stream_, i);
    moParser.setDelimiter(delimiter_);
    mores = moParser.getModulus();

    i += mores.second;
    expr.factor_div_.push_back(mores.first);


    while(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op == DIV){
            moParser.construct(tok_stream_, ++i);
            moParser.setDelimiter(delimiter_);
            mores = moParser.getModulus();

            i+= mores.second;
            expr.factor_div_.push_back(mores.first);
        }
        else break;
    }

    return i;
}


pair<FactorMult, size_t> FactorMultParser::getFactorMult(){
    pair<FactorMult, size_t> ret;
    FactorMult expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t FactorMultParser::parse_(size_t begin, FactorMult &expr){
    FactorDivParser fdParser;
    pair<FactorDiv, size_t> fdres;

    size_t i=begin;

    fdParser.construct(tok_stream_, i);
    fdParser.setDelimiter(delimiter_);
    fdres = fdParser.getFactorDiv();

    i += fdres.second;
    expr.factor_mult_.push_back(fdres.first);

    while(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op == MULT){
            fdParser.construct(tok_stream_, ++i);
            fdParser.setDelimiter(delimiter_);
            fdres = fdParser.getFactorDiv();

            i+= fdres.second;
            expr.factor_mult_.push_back(fdres.first);
        }
        else break;
    }

    return i;
}

pair<TermAdd, size_t> TermAddParser::getTermAdd(){
    pair<TermAdd, size_t> ret;
    TermAdd expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t TermAddParser::parse_(size_t begin, TermAdd &expr){
    FactorMultParser fmParser;
    pair<FactorMult, size_t> fmres;

    size_t i=begin;
    fmParser.construct(tok_stream_, i);
    fmParser.setDelimiter(delimiter_);
    fmres = fmParser.getFactorMult();

    i += fmres.second;
    expr.term_add_.push_back(fmres.first);

    while(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op == PLUS){

            fmParser.construct(tok_stream_, ++i);
            fmParser.setDelimiter(delimiter_);
            fmres = fmParser.getFactorMult();

            i+= fmres.second;
            expr.term_add_.push_back(fmres.first);
        }
        else break;
    }

    return i;
}

pair<TermSub, size_t> TermSubParser::getTermSub(){
    pair<TermSub, size_t> ret;
    TermSub expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t TermSubParser::parse_(size_t begin, TermSub &expr){
    TermAddParser taParser;
    pair<TermAdd, size_t> tares;

    size_t i=begin;
    taParser.construct(tok_stream_, i);
    taParser.setDelimiter(delimiter_);
    tares = taParser.getTermAdd();

    i += tares.second;
    expr.term_sub_.push_back(tares.first);


    while(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op == MINUS){
            taParser.construct(tok_stream_, ++i);
            taParser.setDelimiter(delimiter_);
            tares = taParser.getTermAdd();

            i+= tares.second;
            expr.term_sub_.push_back(tares.first);
        }
        else break;
    }

    return i;
}

pair<Component, size_t> ComponentParser::getComponent(){
    pair<Component, size_t> ret;
    Component expr;
    ret.first = expr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t ComponentParser::parse_(size_t begin, Component &expr){
    TermSubParser tsParser;
    pair<TermSub, size_t> tsres;
    size_t i = begin;
    tsParser.construct(tok_stream_, i);
    tsParser.setDelimiter(delimiter_);

    tsres = tsParser.getTermSub();
    i += tsres.second;
    expr.component_.push_back(tsres.first);

    if(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op == COND){
            tsParser.construct(tok_stream_, ++i);
            tsParser.setDelimiter(delimiter_);
            tsres = tsParser.getTermSub();
            i += tsres.second;
            expr.component_.push_back(tsres.first);

            if(tok_stream_[i].type_ == OPERATOR){
                if(tok_stream_[i].token_.op == ELSE){
                    tsParser.construct(tok_stream_, ++i);
                    tsParser.setDelimiter(delimiter_);
                    tsres = tsParser.getTermSub();
                    expr.component_.push_back(tsres.first);
                    //tsres.first.print();
                    //cout<<"\n\n";
                    i += tsres.second;

                }
            }
        }
    }
    return i;
}


pair<LExpr *, size_t> LExprParser::getLExpr(){
    pair<LExpr *, size_t> ret;
    ret.first = new LExpr;
    ret.second = parse_(0, ret.first);
    destruct();
    return ret;
}

size_t LExprParser::parse_(size_t begin, LExpr *lexpr){
    ComponentParser coParser;
    pair<Component, size_t> cores;
    size_t i = begin;

    coParser.construct(tok_stream_, i);
    coParser.setDelimiter(delimiter_);
    cores = coParser.getComponent();
    i += cores.second;
    lexpr->lexpr_.push_back(cores.first);

    while(tok_stream_[i].type_ == OPERATOR){
        if(tok_stream_[i].token_.op == COMMA){
            coParser.construct(tok_stream_, ++i);
            coParser.setDelimiter(delimiter_);
            cores = coParser.getComponent();

            i += cores.second;
            lexpr->lexpr_.push_back(cores.first);
        }
        else if(tok_stream_[i].token_.op == delimiter_)
```

```
            return ++i;
        else break;
    }
    return i;
}


void atom::print(){
    Token *tok;
    SExpr *sexpr;
    LExpr *lexpr;
    Variable *var;
    cout<<"<ATOM>"<<endl;
    switch(type_){
        case TOKEN:
            //cout<<"<ATOM>"<<endl;
            tok = (Token *)atom_;
            tok->print();
            //cout<<"</ATOM>"<<endl;
            break;
        case SEXPR:
            //cout<<"<S-EXPR>"<<endl;
            sexpr = (SExpr *)atom_;
            sexpr->print();
            //cout<<"</S-EXPR>"<<endl;
            break;
        case LEXPR:
            //cout<<"<L-EXPR>"<<endl;
            lexpr = (LExpr *)atom_;
            lexpr->print();
            //cout<<"</L-EXPR>"<<endl;
            break;
        case VARIABLE:
            //cout<<"<VARIABLE>"<<endl;
            var = (Variable *)atom_;
            var->print();
            //cout<<"</VARIABLE>"<<endl;
            break;
        default:
            cout<<"<ERROR/>␣"<<endl;
    }
    cout<<"</ATOM>"<<endl;
}

void Logical_not::print(){
    cout<<"<LOGICAL_NOT␣is_not=\""<<(not_?"true":"false")<<"\">"<<endl;
    for(size_t i=0; i<logical_not_.size(); i++)
        logical_not_[i].print();
    cout<<"</LOGICAL_NOT>"<<endl;
}

void Logical_xor::print(){
    cout<<"<LOGICAL_XOR>"<<endl;
    for(size_t i=0; i<logical_xor_.size(); i++)
        logical_xor_[i].print();
    cout<<"</LOGICAL_XOR>"<<endl;
}


void Logical_and::print(){
    cout<<"<LOGICAL_AND>"<<endl;
    for(size_t i=0; i<logical_and_.size(); i++)
        logical_and_[i].print();
    cout<<"</LOGICAL_AND>"<<endl;
}

void Logical_or::print(){
    cout<<"<LOGICAL_OR>"<<endl;
    for(size_t i=0; i<logical_or_.size(); i++)
        logical_or_[i].print();
    cout<<"</LOGICAL_OR>"<<endl;
}

void Unary::print(){
    cout<<"<UNARY-OPERATOR␣is_minus=\""<<(minus_?"true":"false")<<"\">"<<endl;
    for(size_t i=0; i<unary_.size(); i++)
        unary_[i].print();
    cout<<"</UNARY-OPERATOR>"<<endl;
}

void Exponent::print(){
    cout<<"<EXPONENT>"<<endl;
    for(size_t i=0; i<exponent_.size(); i++)
        exponent_[i].print();
    cout<<"</EXPONENT>"<<endl;
}

void Modulus::print(){
    cout<<"<MODULUS>"<<endl;
    for(size_t i=0; i<modulus_.size(); i++)
```

```
        modulus_[i].print();
    cout<<"</MODULUS>"<<endl;
}

void FactorDiv::print(){
    cout<<"<DIVISION>"<<endl;
    for(size_t i=0; i<factor_div_.size(); i++)
        factor_div_[i].print();
    cout<<"</DIVISION>"<<endl;
}

void FactorMult::print(){
    cout<<"<MULTIPLICATION>"<<endl;
    for(size_t i=0; i<factor_mult_.size(); i++)
        factor_mult_[i].print();
    cout<<"</MULTIPLICATION>"<<endl;
}

void TermAdd::print(){
    cout<<"<ADDITION>"<<endl;
    for(size_t i=0; i<term_add_.size(); i++)
        term_add_[i].print();
    cout<<"</ADDITION>"<<endl;
}

void TermSub::print(){
    cout<<"<SUBTRACTION>"<<endl;
    for(size_t i=0; i<term_sub_.size(); i++)
        term_sub_[i].print();
    cout<<"</SUBTRACTION>"<<endl;
}

void Component::print(){
    cout<<"<CONDITIONAL>"<<endl;
    if(component_.size()==0){
        cout<<"<ERROR/>"<<endl;
    }
    else if(component_.size()==1){
        cout<<"<ALWAYS>"<<endl;
        component_[0].print();
        cout<<"</ALWAYS>"<<endl;
    }
    else if(component_.size()==2){
        cout<<"<CONDITION>"<<endl;
        component_[0].print();
        cout<<"</CONDITION>"<<endl;

        cout<<"<SATISFIED>"<<endl;
        component_[1].print();
        cout<<"</SATISFIED>"<<endl;
    }
    else if(component_.size()==3){
        cout<<"<CONDITION>"<<endl;
        component_[0].print();
        cout<<"</CONDITION>"<<endl;

        cout<<"<SATISFIED>"<<endl;
        component_[1].print();
        cout<<"</SATISFIED>"<<endl;

        cout<<"<ELSE>"<<endl;
        component_[2].print();
        cout<<"</ELSE>"<<endl;
    }
    else{
        cout<<"<FAULTY-CONDITIONAL>"<<endl;
        for(size_t i=0; i<component_.size(); i++)
            component_[i].print();
        cout<<"</FAULTY-CONDITIONAL>"<<endl;
    }
    cout<<"</CONDITIONAL>"<<endl;
}


void LExpr::print(){
    cout<<"<L-EXPRESSION>"<<endl;
    for(size_t i=0; i<lexpr_.size(); i++)
        lexpr_[i].print();
    cout<<"</L-EXPRESSION>"<<endl;
}

void Variable::print(){
    cout<<"<VARIABLE>"<<endl;
    if(context_ != NULL){
        cout<<"<CONTEXT>"<<endl;
        context_->print();
        cout<<"</CONTEXT>"<<endl;
    }
    cout<<"<VAR-NAME>"<<var_name_<<"</VAR-NAME>"<<endl;
    cout<<"</VARIABLE>"<<endl;
```

```
}

void SExpr::print(){
    cout<<"<S-EXPRESSION>"<<endl;
    for(size_t i=0; i<sexpr_.size(); i++)
        sexpr_[i].print();
    cout<<"</S-EXPRESSION>"<<endl;
}



/**************************************************
                        MAIN.CPP
**************************************************/

#include "parser.hpp"
using namespace std;

bit64 line_no = 1;
bit64 col_no = 0;
int indentLevel = 0;

int main(){
    StatementParser parse("$s+{$[]a+$[~(-7),8,true?x:{y+z}]b}=5");
    Statement *expr = parse.getStmt();
    expr->print();
    cout<<endl;
    return 0;
}


/**************************************************
                    PARSE_TREE.XML
**************************************************/

<STATEMENT>
<L-VALUE>
<S-EXPRESSION>
<ATOM>
<VARIABLE>
<VAR-NAME>s</VAR-NAME>
</VARIABLE>
</ATOM>
<ATOM>
<L-EXPRESSION>
```

```
<CONDITIONAL>
<ALWAYS>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<VARIABLE>
<CONTEXT>
<L-EXPRESSION>
<CONDITIONAL>
<ALWAYS>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</ALWAYS>
</CONDITIONAL>
</L-EXPRESSION>
</CONTEXT>
```

```
<VAR-NAME>a</VAR-NAME>
</VARIABLE>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<VARIABLE>
<CONTEXT>
<L-EXPRESSION>
<CONDITIONAL>
<ALWAYS>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="true">
<ATOM>
<S-EXPRESSION>
<ATOM>
<OPERATOR>MINUS</OPERATOR>
</ATOM>
<ATOM>
<INTEGER>7</INTEGER>
```

```
</ATOM>
</S-EXPRESSION>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</ALWAYS>
</CONDITIONAL>
<CONDITIONAL>
<ALWAYS>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<INTEGER>8</INTEGER>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</ALWAYS>
</CONDITIONAL>
```

```
<CONDITIONAL>
<CONDITION>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<SYMBOL>true</SYMBOL>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</CONDITION>
<SATISFIED>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<SYMBOL>x</SYMBOL>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
```

```
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</SATISFIED>
<ELSE>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<L-EXPRESSION>
<CONDITIONAL>
<ALWAYS>
<SUBTRACTION>
<ADDITION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<SYMBOL>y</SYMBOL>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
```

```
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
<MULTIPLICATION>
<DIVISION>
<MODULUS>
<EXPONENT>
<UNARY-OPERATOR is_minus="false">
<LOGICAL_OR>
<LOGICAL_AND>
<LOGICAL_XOR>
<LOGICAL_NOT is_not="false">
<ATOM>
<SYMBOL>z</SYMBOL>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</ALWAYS>
</CONDITIONAL>
</L-EXPRESSION>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</ELSE>
</CONDITIONAL>
</L-EXPRESSION>
```

```
</CONTEXT>
<VAR-NAME>b</VAR-NAME>
</VARIABLE>
</ATOM>
</LOGICAL_NOT>
</LOGICAL_XOR>
</LOGICAL_AND>
</LOGICAL_OR>
</UNARY-OPERATOR>
</EXPONENT>
</MODULUS>
</DIVISION>
</MULTIPLICATION>
</ADDITION>
</SUBTRACTION>
</ALWAYS>
</CONDITIONAL>
</L-EXPRESSION>
</ATOM>
</S-EXPRESSION>
</L-VALUE>
<R-VALUE>
<S-EXPRESSION>
<ATOM>
<INTEGER>5</INTEGER>
</ATOM>
</S-EXPRESSION>
</R-VALUE>
</STATEMENT>


/**************************************************
                    TESTER.CPP
**************************************************/

/****************************************************************************

     PROTOTYPE OF I PROGRAMMING LANGUAGE INTERPRETATION SYSTEM VERSION-3

****************************************************************************/

#ifndef LEXER_H_
#define LEXER_H_


```

```
#include "includes.hpp"
using namespace std;

bit64 line_no;
bit64 col_no;

class I_error{
    public:
        I_error(const char *msg):
            error_message_(msg), line_no_(line_no), col_no_(col_no) {}

        void print(){
            cout<<"ERROR␣::␣LINE="<<line_no_<<"␣CHARACTER="<<col_no_
                <<"␣::␣"<<error_message_<<endl;
            exit(0);
        }

    private:
        const char *error_message_;
        int line_no_;
        int col_no_;
};


enum TokenType{ SYMBOL = 0, INTEGER, REAL, OPERATOR, STRING , _NONE };
const char *typeString(TokenType t){
    switch(t){
        case SYMBOL:
            return "SYMBOL";
        case INTEGER:
            return "INTEGER";
        case REAL:
            return "REAL";
        case OPERATOR:
            return "OPERATOR";
        case STRING:
            return "STRING";
        case _NONE:
            return "_NONE";
        default:
            return "ERROR␣...";
    }
}

enum Operator{  PLUS, MINUS, MULT, DIV, MOD, POW,
                AND, OR, NOT, XOR,
                GT, LT, GTE, LTE, EQ, NEQ ,
                LPAREN, RPAREN, ERROR,
                ESCAPE, ASSIGN, LCURL, RCURL, COMMA };

const char *opString(Operator op){
    switch(op){
        case PLUS:  return "PLUS";
        case MINUS: return "MINUS";
        case MULT:  return "MULT";
        case DIV:   return "DIV";
        case MOD:   return "MOD";
        case POW:   return "POW";
        case AND:   return "AND";
        case OR:    return "OR";
        case NOT:   return "NOT";
        case XOR:   return "XOR";
        case GT:    return "GT";
        case LT:    return "LT";
        case GTE:   return "GTE";
        case LTE:   return "LTE";
        case EQ:    return "EQ";
        case NEQ:   return "NEQ";
        case LPAREN:return "LPAREN";
        case RPAREN:return "RPAREN";
        case ERROR: return "ERROR";
        case ESCAPE:return "ESCAPE";
        case ASSIGN:return "ASSIGN";
        case LCURL: return "LCURL";
        case RCURL: return "RCURL";
        case COMMA: return "COMMA";
        default:    return "\0";
    }
}

union TokenHolder{
    int integer;
    double real;
    char str[1024];
    char sym[1024];
    Operator op;
};

Operator str2op(const char *str){
    switch(str[0]){
        case '+': return PLUS;
        case '-': return MINUS;
        case '*':
            if( str[1]=='*') return POW;
            return MULT;
        case '/': return DIV;
        case '%': return MOD;
        case '&': return AND;
        case '|': return OR;
        case '~':
            if( str[1]=='=') return NEQ;
            return NOT;
        case '^': return XOR;
        case '>':
            if( str[1]=='=') return GTE;
            return GT;
        case '<':
            if( str[1]=='=') return LTE;
            return LT;
        case '=':
            if(str[1]=='=') return EQ;
            return ASSIGN;
        case '(': return LPAREN;
        case ')': return RPAREN;
        case '{': return LCURL;
        case '}': return RCURL;
        case '\\':return ESCAPE;
        case ',': return COMMA;
        default:
            return ERROR;
    }
}

class Token{
    public:

        Token():type_(_NONE){}
        Token(TokenType type, const char *tok_str){
            makeToken(type, tok_str);
        }

        Token (const Token &oldToken){
            type_ = oldToken.type_;
            token_ = oldToken.token_;
        }

        void print(){
            cout<<"<␣"<<typeString(type_)<<",␣";
            switch(type_){
                case INTEGER:   cout<<token_.integer<<"␣>";           break;
                case REAL:      cout<<token_.real<<"␣>";              break;
                case SYMBOL:    cout<<token_.sym<<"␣>";               break;
                case STRING:    cout<<token_.str<<"␣>";               break;
                case OPERATOR:  cout<<opString(token_.op)<<"␣>";      break;
                case _NONE:     cout<<"␣>";                           break;
                default:        cout<<"␣>";                           break;
            }
        }

        void makeToken(TokenType type, const char *tok_str){
            type_ = type;
            switch(type_){
                case SYMBOL:
                    strcpy( token_.sym, tok_str);
                    break;
                case INTEGER:
                    token_.integer = atoi(tok_str);
                    break;
                case REAL:
                    token_.real = atof(tok_str);
                    break;
                case STRING:
                    strcpy(token_.str, tok_str);
                    break;
                case OPERATOR:
                    token_.op = str2op(tok_str);
                    break;
                case _NONE:
                    break;
                default:
                    cout<<"ERROR␣OCCURED␣IN␣LEXICAL␣ANALYSIS"<<endl;
                    break;
            }
        }

        TokenType getType(){ return type_; }

        TokenType type_;
        TokenHolder token_;
};
```

```


class Tokenizer{
    public:
        Tokenizer():expr_(NULL),type_(_NONE){}
        Tokenizer(const char *expr):expr_(expr){}

        bool next(Token &t){
            bool ret = getNextToken();
            t.makeToken(type_, token_);
            return ret;
        }

        const char* getExpr(){ return expr_; }

    private:
        const char *expr_;
        char token_[1024];
        TokenType type_;

        bool isOperator(char ch){
            if(strchr("+-*/%^=(){}[]&|~><?:$@,\\", ch)) return true;
            return false;
        }

        bool getNextToken(){
            char *temp;
            type_ = _NONE;
            temp = token_;
            *temp = '\0';

            if(!*expr_) return false;
            while(isspace(*expr_)) {
                expr_++ ;
                col_no++;
            }

            if( isOperator(*expr_)){
                while(isOperator(*expr_)){
                    *temp++ = *expr_++;
                    col_no++;
                }
                type_ = OPERATOR;
            }
            else if(isalpha(*expr_)){
                while(isalpha(*expr_) || *expr_ == '_'){
                    *temp++ = *expr_++;
                    col_no++;
                }
                type_ = SYMBOL;
            }
            else if(isdigit(*expr_)){
                while(isdigit(*expr_)){
                    *temp++ = *expr_++;
                    col_no++;
                }
                if(*expr_ == '.'){
                    *temp++ = *expr_++;
                    col_no++;
                    while(isdigit(*expr_)){
                        *temp++ = *expr_++;
                        col_no++;
                    }
                    type_ = REAL;
                }
                else type_ = INTEGER;
            }
            else if(*expr_ == '\"'){
                *expr_++;
                col_no++;
                while(*expr_ != '\"'){
                    *temp++ = *expr_++;
                    col_no++;
                }
                *expr_++;
                col_no++;
                type_ = STRING;
            }
            else{
                while(!isspace(*expr_)){
                    *temp++ = *expr_++;
                    col_no++;
                }
                type_ = SYMBOL;
            }

            *temp = '\0';
            return true;
        }

```

```
};

enum atomType{ TOKEN, SEXPR, LEXPR };
struct atom{
    atomType type_;
    void *atom_;
};


//-----------------------------------------------------------------------------
//       FORWARD DECLARATION OF L-EXPRESSION AND ITS PARSER FOR USE IN
//                  CLASSES USED FOR PARSING S-EXPRESSIONS
//-----------------------------------------------------------------------------

int indentLevel;

struct Logical_not{
    vector<atom> logical_not_;        // Unary operator ~
    void print();
};

struct Logical_xor{
    vector<Logical_not> logical_xor_;    // values separated by ^
    void print();
};

struct Logical_and{
    vector<Logical_xor> logical_and_;    // values separated by &
    void print();
};

struct Logical_or{
    vector<Logical_and> logical_or_; // Values separated by |
    void print();
};

struct Unary{
    vector<Logical_or>  unary_; // Unary operators +, -
    void print();
};

struct Exponent{
    vector<Unary> exponent_;     // Values separated by exponent op **
    void print();
};
```

```

struct Factor{                  // Values separated by *, / and %
    vector<Exponent> factor_;
    void print();
};

struct Term{                    //Values separated by + and -
    vector<Factor> term_;
    void print();
};

struct Component{               //The if-else operator _?_:_
    vector<Term> component_;
    void print();
};

struct LExpr{                   //Comma separated values
    vector<Component> lexpr_;
    void print();
};


class Logical_notParser{
    public:
        Logical_notParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        Logical_notParser(const vector<Token> &tok_stream)
            :tok_stream_(tok_stream){}
        Logical_notParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }

        pair<Logical_not, size_t> getLogical_not();

    private:
        vector<Token> tok_stream_;

        size_t parse_(size_t begin);
```

```

};

class Logical_xorParser{
    public:
        Logical_xorParser(const vector<Logical_not> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        Logical_xorParser(const vector<Logical_not> &tok_stream)
            :tok_stream_(tok_stream){}
        Logical_xorParser(){}

        void construct(const vector<Logical_not> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }

        pair<Logical_xor, size_t> getLogical_xor();

    private:
        vector<Logical_not> tok_stream_;

        size_t parse_(size_t begin);

};

class LExprParser{
    public:
        LExprParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        LExprParser(const vector<Token> &tok_stream):tok_stream_(tok_stream){}
        LExprParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }

        pair<LExpr *,size_t> getLExpr();

    private:
        const char *expr_;
        vector<Token> tok_stream_;
        Token tok_;
        LExpr *lexpr_;
        bit32 curl_match_;
        bool comma_delimited_;

        size_t parse_(size_t begin, LExpr *lexpr);
};

//--------------------------------------------------------------------------

struct SExpr{
    vector<atom> sexpr_;

    void print(){
        for(size_t i=0; i<sexpr_.size(); i++){
            if(sexpr_[i].type_ == TOKEN){
                Token *t = (Token *)sexpr_[i].atom_;
                t->print();
                cout<<"␣";
            }
            else if(sexpr_[i].type_ == SEXPR){
                indentLevel++;
                cout<<"\n";
                for(int j=0;j<indentLevel;j++)cout<<"\t";cout<<"(␣";
                SExpr *t = (SExpr *)sexpr_[i].atom_;
                t->print();
                indentLevel--;
                cout<<"␣)"<<endl;
                for(int j=0;j<indentLevel;j++)cout<<"\t";
            }
            else if(sexpr_[i].type_ == LEXPR){
                LExpr *l = (LExpr *)sexpr_[i].atom_;
                l->print();
            }
            else;
        }
    }
};

class SExprParser{
    public:
        SExprParser(const char *expr):expr_(expr),lex_(expr){
            while(lex_.next(tok_)) tok_stream_.push_back(tok_);
        }
        SExprParser(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }
        SExprParser(const vector<Token> &tok_stream):tok_stream_(tok_stream){}
        SExprParser(){}

        void construct(const vector<Token> &tok_stream, size_t begin){
            for(size_t i=begin; i<tok_stream.size(); i++){
                tok_stream_.push_back(tok_stream[i]);
            }
        }

        pair<SExpr *,size_t> getSExpr(){
            pair<SExpr *, size_t> ret;
            paren_match = 0;
            ret.first = new SExpr;
            ret.second = parse_(0,ret.first);
            if(paren_match){
                I_error e("Parentheses␣mismatch");
                e.print();
            }
            return ret;
        }

    private:
        const char *expr_;
        vector<Token> tok_stream_;
        Token tok_;
        SExpr *sexpr_;
        Tokenizer lex_;
        bit32 paren_match;

        size_t parse_(size_t begin, SExpr *sexpr){
            size_t i;
            atom myAtom;
            SExpr *newSexpr;
            LExprParser lParser;
            pair<LExpr *,size_t> lres;
            for(i=begin; i<tok_stream_.size(); i++){
                if(tok_stream_[i].type_ == OPERATOR){
                    switch(tok_stream_[i].token_.op){
                        case LPAREN:
                            paren_match++;
                            newSexpr = new SExpr;
                            i = parse_(i+1, newSexpr);
                            //atom myAtom;
                            myAtom.type_=SEXPR;
                            myAtom.atom_ = (void *)newSexpr;
                            sexpr->sexpr_.push_back(myAtom);
                            break;
                        case RPAREN:
                            paren_match--;
                            return i;
                        case LCURL:
                            lParser.construct(tok_stream_, i+1);
                            //i += lParser.parse_(0,);
                            lres = lParser.getLExpr();
                            i += lres.second;
                            //atom myAtom;
                            myAtom.type_ = LEXPR;
                            myAtom.atom_ = (void *)lres.first;
                            sexpr->sexpr_.push_back(myAtom);
                            break;
                        default:
                            //atom myAtom;
                            myAtom.type_=TOKEN;
                            Token *myToken = new Token(tok_stream_[i]);
                            myAtom.atom_ = (void *)myToken;
                            sexpr->sexpr_.push_back(myAtom);
                            break;
                    }
                }
                else{
                    //atom myAtom;
                    myAtom.type_ = TOKEN;
                    Token *myToken = new Token(tok_stream_[i]);
                    myAtom.atom_ = (void *)myToken;
                    sexpr->sexpr_.push_back(myAtom);
                }
            }
            return i;
        }
};

```

```

//--------------------------------------------------------------------------------
//        FUNCTION DEFINITIONS FOR THE L-EXPRESSION STRUCT AND ITS PARSER
//--------------------------------------------------------------------------------

size_t LExprParser::parse_(size_t begin, LExpr* lexpr){
    size_t i;
    atom myAtom;
    pair<SExpr *,size_t> sres;
    SExprParser sParser;
    for(i=begin; i<tok_stream_.size(); i++){
        if(tok_stream_[i].type_ == OPERATOR){
            switch(tok_stream_[i].token_.op){
                case LPAREN:
                    sParser.construct(tok_stream_, i+1);
                    sres = sParser.getSExpr();
                    i += sres.second;
                    //atom myAtom;
                    myAtom.type_ = SEXPR;
                    myAtom.atom_ = (void *)sres.first;
                    //lexpr->lexpr_.push_back(myAtom);
                    break;

                case COMMA:
                    comma_delimited_ = true;
                    break;
            }
        }
    }
}




inline void LExpr::print(){
    /*cout<<"\n";
    indentLevel++;
    for(int i=0;i<indentLevel;i++) cout<<"\t";cout<<"{ ";
    for(size_t i=0; i<lexpr_.size(); i++){
        if(lexpr_[i].type_ == TOKEN){
            Token *t=(Token *) lexpr_[i].atom_;
            t->print();
            cout<<" ";
        }
        else if(lexpr_[i].type_ == SEXPR){
            indentLevel++;
            cout<<endl;
            for(int j=0; j<indentLevel; j++) cout<<"\t";cout<<"( ";
            SExpr *s=(SExpr *)lexpr_[i].atom_;
            s->print();
            indentLevel--;
            cout<<" )"<<endl;
            for(int j=0; j<indentLevel; j++) cout<<"\t";
        }
        else if(lexpr_[i].type_ == LEXPR){
            LExpr *l=(LExpr *)lexpr_[i].atom_;
            l->print();
        }
        else;

        if(!arithmatic_){
            cout<<" , ";
        }
    }
    indentLevel--;
    cout<<" } "<<endl;
    for(int j=0; j<indentLevel; j++) cout<<"\t";*/
}

pair<LExpr *,size_t> LExprParser::getLExpr(){}


//------------------------------------------------------------------------------

struct Statement{
    bool store_;
    SExpr *lval_;
    SExpr *rval_;

    Statement():store_(true),lval_(NULL),rval_(NULL){}
    void print(){
        cout<<endl<<"␣L␣-␣V␣A␣L␣U␣E"<<endl<<"=================="<<endl;
        lval_->print();
        if(store_){
            cout<<endl<<"␣R␣-␣V␣A␣L␣U␣E"<<endl<<"=================="<<endl;
            rval_->print();
        }
    }

    ~Statement(){
        delete lval_;
        delete rval_;
    }
};

class StatementParser{
    public:
        StatementParser(const char *expr):expr_(expr){
            Tokenizer lex_(expr);
            Token t;
            vector<Token> tok_stream_l;
            vector<Token> tok_stream_r;

            stmt_ = new Statement;

            while(lex_.next(t)){
                if(t.type_ == OPERATOR){
                    if(t.token_.op == ASSIGN){
                        break;
                    }
                    else tok_stream_l.push_back(t);
                }
                else tok_stream_l.push_back(t);
            }
            SExprParser lval_parser_(tok_stream_l);
            stmt_->lval_ = lval_parser_.getSExpr().first;
            int count = 0;
            while(lex_.next(t)){
                count++;
                tok_stream_r.push_back(t);
            }
            if(!count)stmt_->store_=false;
            else{
                SExprParser rval_parser_(tok_stream_r);
                stmt_->rval_ = rval_parser_.getSExpr().first;
            }
        }

        Statement *getStmt(){
            return stmt_;
        }


    private:
        const char *expr_;
        vector <Token> tok_stream_;
        Statement *stmt_;
};



class SymTbl{
    public:
        typedef SymTbl type;
        typedef string key_type;
        typedef bit64 value_type;
        typedef bit64 count_type;

        SymTbl():offset(0),sym_count(0){}
        bool insert(const key_type key){
            if(table.count(key)) return false;
            table[key]=++sym_count;
            return true;
        }

        value_type operator [](const key_type &key){
            return offset+table[key];
        }

        const bool exists(const key_type &key) const{
            if(table.count(key)) return true;
            return false;
        }

        const bool empty() const{
            return sym_count?false:true;
        }

    private:
        count_type offset;
        count_type sym_count;
        map<key_type, value_type> table;
};


class machine{
    public:
        typedef bit64 word_type;
        typedef bit8 byte_type;
//---------------------------------------------------------------------------
```

```
//      TO WORK FROM HERE.  IMPLEMENT THE VIRTUAL MACHINE FOR INTERPRETATION
//      ALSO IMPLEMENT AN INTERMEDIATE REPRESENTATION IN BINARY IF POSSIBLE
//----------------------------------------------------------------------------------
        machine():current_state(0){}
        bool storeStatement(Statement *s){}
        bool eval(SExpr *expr){}


    private:
        word_type current_state;
        SymTbl sym_table;
        map<pair<word_type,word_type>,word_type> transition_table;
        set<word_type> final_states;
        vector<vector<SExpr *> > eval_rules;
        map<word_type, size_t> binder_table;
};



int main(){
    line_no = 1;
    col_no = 0;
    cout<<"This␣is␣a␣(testing␣sentence␣(a+b)␣>␣7.5␣)␣=␣(5!␣is␣120␣)␣and"
        <<"␣I␣am␣\"Ganesh␣Prasad␣1993\""<<endl;
    StatementParser parse("This␣is␣a␣(testing␣sentence␣(a+b)␣>␣7.5␣)"
                          "␣=␣(5!␣is␣120)␣and␣I␣am␣\"Ganesh␣Prasad␣1993\"");
    Statement *expr = parse.getStmt();
    expr->print();
    cout<<endl;
    return 0;
}

#endif
```