LAB ASSIGNMENT – 12

COMPILER DESIGN LAB – BCSE307P

NAME – ADITYARAJ SHRIVASTAVA

REG. NO. – 23BCE1968

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

typedef enum {
    TOKEN_INT, TOKEN_IF, TOKEN_THEN, TOKEN_ELSE, TOKEN_ENDIF,
    TOKEN_PRINT,
    TOKEN_ID, TOKEN_NUM,
    TOKEN_LBRACE, TOKEN_RBRACE,
    TOKEN_LPAREN, TOKEN_RPAREN,
    TOKEN_LBRACKET, TOKEN_RBRACKET,
    TOKEN_SEMI, TOKEN_COMMA,
    TOKEN_ASSIGN, // '='
    TOKEN_PLUS, TOKEN_MINUS, TOKEN_MUL, TOKEN_DIV,
    TOKEN_GT,
    TOKEN_EOF,
    TOKEN_UNKNOWN
} TokenType;

typedef struct {
    TokenType type;
    char lexeme[64];
    int num_val;
} Token;

typedef struct {
    const char *src;
    int pos;
    Token cur_token;
} Lexer;

void lexer_error(const char *msg) {
    fprintf(stderr, "Lexer error: %s\n", msg);
    exit(1);
}

void skip_whitespace(Lexer *lx) {
    while (isspace(lx->src[lx->pos])) lx->pos++;
}

void skip_comment(Lexer *lx) {
    if (lx->src[lx->pos] == '/' && lx->src[lx->pos+1] == '*') {
        lx->pos += 2;
        while (lx->src[lx->pos] && !(lx->src[lx->pos] == '*' && lx->src[lx-
>pos+1] == '/')) {
            lx->pos++;
        }
        if (lx->src[lx->pos]) lx->pos += 2;
        else lexer_error("Unterminated comment");
    }
}

int is_keyword(const char *str, TokenType *type) {
    if (strcmp(str, "int") == 0) {*type = TOKEN_INT; return 1;}
    if (strcmp(str, "if") == 0) {*type = TOKEN_IF; return 1;}
    if (strcmp(str, "then") == 0) {*type = TOKEN_THEN; return 1;}
    if (strcmp(str, "else") == 0) {*type = TOKEN_ELSE; return 1;}
    if (strcmp(str, "endif") == 0) {*type = TOKEN_ENDIF; return 1;}
    if (strcmp(str, "print") == 0) {*type = TOKEN_PRINT; return 1;}
    return 0;
}

void lexer_next_token(Lexer *lx) {
    while (1) {
        skip_whitespace(lx);
        if (lx->src[lx->pos] == '/' && lx->src[lx->pos+1] == '*') {
            skip_comment(lx);
            continue;
        }
        break;
    }
    if (lx->src[lx->pos] == 0) {
        lx->cur_token.type = TOKEN_EOF;
        return;
    }

    char c = lx->src[lx->pos];

    if (isalpha(c) || c == '_') {
        int start = lx->pos;
        while (isalnum(lx->src[lx->pos]) || lx->src[lx->pos] == '_') lx->pos++;
        int len = lx->pos - start;
        if (len >= (int)sizeof(lx->cur_token.lexeme)) lexer_error("Identifier too
long");
        strncpy(lx->cur_token.lexeme, lx->src + start, len);
        lx->cur_token.lexeme[len] = 0;
        TokenType type;
        if (is_keyword(lx->cur_token.lexeme, &type)) {
            lx->cur_token.type = type;
        } else {
            lx->cur_token.type = TOKEN_ID;
        }
        return;
    }

    if (isdigit(c)) {
        int val = 0;
        while (isdigit(lx->src[lx->pos])) {
            val = val*10 + (lx->src[lx->pos] - '0');
            lx->pos++;
        }
        lx->cur_token.type = TOKEN_NUM;
        lx->cur_token.num_val = val;
        return;
    }

    switch (c) {
        case '{': lx->cur_token.type = TOKEN_LBRACE; lx->pos++; return;
        case '}': lx->cur_token.type = TOKEN_RBRACE; lx->pos++; return;
        case '(': lx->cur_token.type = TOKEN_LPAREN; lx->pos++; return;
        case ')': lx->cur_token.type = TOKEN_RPAREN; lx->pos++; return;
        case '[': lx->cur_token.type = TOKEN_LBRACKET; lx->pos++; return;
        case ']': lx->cur_token.type = TOKEN_RBRACKET; lx->pos++; return;
        case ';': lx->cur_token.type = TOKEN_SEMI; lx->pos++; return;
        case ',': lx->cur_token.type = TOKEN_COMMA; lx->pos++; return;
        case '=': lx->cur_token.type = TOKEN_ASSIGN; lx->pos++; return;
        case '+': lx->cur_token.type = TOKEN_PLUS; lx->pos++; return;
        case '-': lx->cur_token.type = TOKEN_MINUS; lx->pos++; return;
        case '*': lx->cur_token.type = TOKEN_MUL; lx->pos++; return;
        case '/': lx->cur_token.type = TOKEN_DIV; lx->pos++; return;
        case '>': lx->cur_token.type = TOKEN_GT; lx->pos++; return;
        default:
            lexer_error("Unknown character");
    }
}

void lexer_expect(Lexer *lx, TokenType t) {
    if (lx->cur_token.type != t) {
        fprintf(stderr, "Expected token %d but got %d (%s)\n", t, lx-
>cur_token.type, lx->cur_token.lexeme);
        exit(1);
    }
    lexer_next_token(lx);
```

```c
}

typedef enum {
    NODE_DECL, NODE_ASSIGN, NODE_BINOP, NODE_NUM,
    NODE_ID, NODE_ARRAY_REF, NODE_IF, NODE_PRINT, NODE_BLOCK, NODE_UNARY
} NodeType;

typedef struct Node {
    NodeType type;
    union {
        struct { char name[64]; int array_size; } decl;
        struct { struct Node *lhs; struct Node *rhs; } assign;
        struct { int op; struct Node *left; struct Node *right; } binop;
        int num_val;
        struct { char name[64]; } id;
        struct { char name[64]; struct Node *index; } array_ref;
        struct { struct Node *cond; struct Node *then_block; struct Node
*else_block; } ifstmt;
        struct { struct Node *expr; } printstmt;
        struct { struct Node **stmts; int stmt_count; } block;
        struct { int op; struct Node *expr; } unary;
    };
} Node;

Node *make_node(NodeType type) {
    Node *n = malloc(sizeof(Node));
    if (!n) { perror("malloc"); exit(1); }
    memset(n, 0, sizeof(Node));
    n->type = type;
    return n;
}

// Forward declarations
Node *parse_block(Lexer *lx);

Node *parse_primary(Lexer *lx);

Node *parse_expression(Lexer *lx);

Node *parse_factor(Lexer *lx);

Node *parse_term(Lexer *lx);

Node *parse_unary(Lexer *lx);
Node *parse_statement(Lexer *lx);

Node *parse_declaration(Lexer *lx);

// Parsing functions

Node *parse_declaration(Lexer *lx) {
    lexer_expect(lx, TOKEN_INT);
    Node *block = make_node(NODE_BLOCK);
    Node **decls = NULL;
    int count = 0;

    while (1) {
        if (lx->cur_token.type != TOKEN_ID) break;

        Node *decl = make_node(NODE_DECL);
        strncpy(decl->decl.name, lx->cur_token.lexeme, sizeof(decl->decl.name));
        decl->decl.array_size = 0;

        lexer_next_token(lx);
        if (lx->cur_token.type == TOKEN_LBRACKET) {
            lexer_next_token(lx);
            if (lx->cur_token.type != TOKEN_NUM) {
                fprintf(stderr, "Expected number in array size\n");
                exit(1);
            }
            decl->decl.array_size = lx->cur_token.num_val;
            lexer_next_token(lx);
            lexer_expect(lx, TOKEN_RBRACKET);
        }

        decls = realloc(decls, sizeof(Node*)*(count+1));
        decls[count++] = decl;

        if (lx->cur_token.type == TOKEN_COMMA) {
            lexer_next_token(lx);
        } else {
            break;
        }
    }
    lexer_expect(lx, TOKEN_SEMI);

    block->block.stmts = decls;
    block->block.stmt_count = count;
    return block;
```

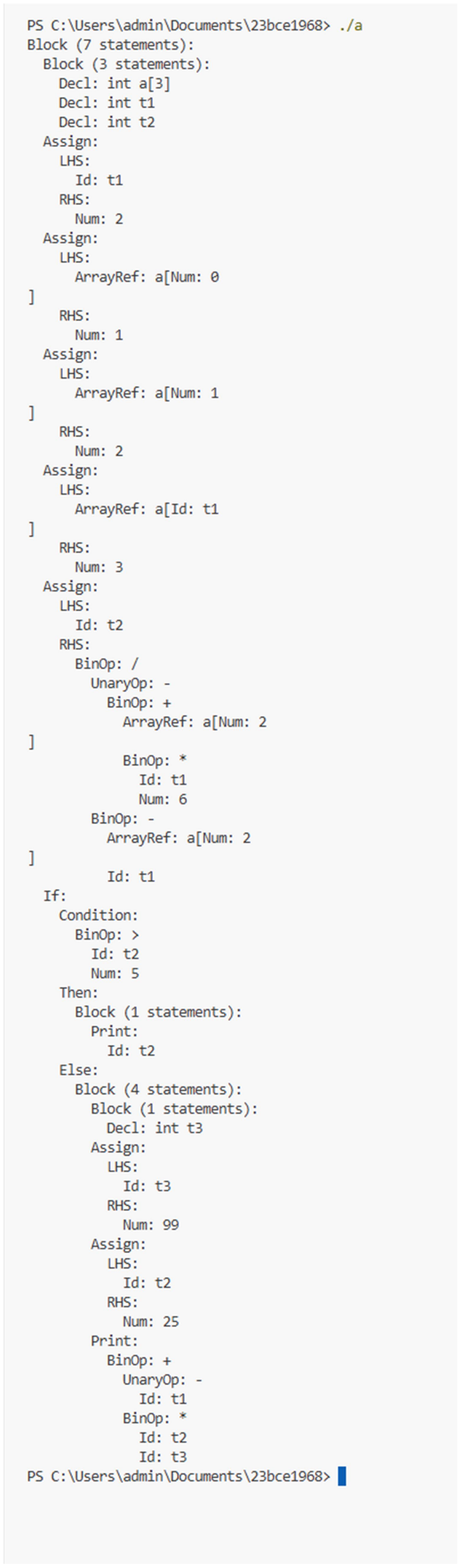
```c
}

Node *parse_primary(Lexer *lx) {
    if (lx->cur_token.type == TOKEN_NUM) {
        Node *n = make_node(NODE_NUM);
        n->num_val = lx->cur_token.num_val;
        lexer_next_token(lx);
        return n;
    }
    if (lx->cur_token.type == TOKEN_ID) {
        char name[64];
        strncpy(name, lx->cur_token.lexeme, sizeof(name));
        lexer_next_token(lx);
        if (lx->cur_token.type == TOKEN_LBRACKET) {
            lexer_next_token(lx);
            Node *index = parse_expression(lx);
            lexer_expect(lx, TOKEN_RBRACKET);
            Node *n = make_node(NODE_ARRAY_REF);
            strncpy(n->array_ref.name, name, sizeof(n->array_ref.name));
            n->array_ref.index = index;
            return n;
        } else {
            Node *n = make_node(NODE_ID);
            strncpy(n->id.name, name, sizeof(n->id.name));
            return n;
        }
    }
    if (lx->cur_token.type == TOKEN_LPAREN) {
        lexer_next_token(lx);
        Node *n = parse_expression(lx);
        lexer_expect(lx, TOKEN_RPAREN);
        return n;
    }
    fprintf(stderr, "Unexpected token in primary: %d (%s)\n", lx->cur_token.type,
lx->cur_token.lexeme);
    exit(1);
}

Node *parse_unary(Lexer *lx) {
    if (lx->cur_token.type == TOKEN_MINUS) {
        lexer_next_token(lx);
        Node *expr = parse_unary(lx);
        Node *n = make_node(NODE_UNARY);
        n->unary.op = TOKEN_MINUS;
        n->unary.expr = expr;
        return n;
    }
    return parse_primary(lx);
}

Node *parse_term(Lexer *lx) {
    Node *left = parse_unary(lx);
    while (lx->cur_token.type == TOKEN_MUL || lx->cur_token.type == TOKEN_DIV) {
        int op = lx->cur_token.type;
        lexer_next_token(lx);
        Node *right = parse_unary(lx);
        Node *n = make_node(NODE_BINOP);
        n->binop.op = op;
        n->binop.left = left;
        n->binop.right = right;
        left = n;
    }
    return left;
}

Node *parse_expression(Lexer *lx) {
    Node *left = parse_term(lx);
    while (lx->cur_token.type == TOKEN_PLUS || lx->cur_token.type == TOKEN_MINUS)
{
        int op = lx->cur_token.type;
        lexer_next_token(lx);
        Node *right = parse_term(lx);
        Node *n = make_node(NODE_BINOP);
        n->binop.op = op;
        n->binop.left = left;
        n->binop.right = right;
        left = n;
    }
    return left;
}

Node *parse_condition(Lexer *lx) {
    Node *left = parse_expression(lx);
    if (lx->cur_token.type == TOKEN_GT) {
        int op = lx->cur_token.type;
        lexer_next_token(lx);
        Node *right = parse_expression(lx);
        Node *n = make_node(NODE_BINOP);
        n->binop.op = op;
        n->binop.left = left;
```

```c
        n->binop.right = right;
        return n;
    }
    return left;
}

Node *parse_assignment(Lexer *lx) {
    // lhs can be ID or array_ref
    Node *lhs;
    if (lx->cur_token.type == TOKEN_ID) {
        char name[64];
        strncpy(name, lx->cur_token.lexeme, sizeof(name));
        lexer_next_token(lx);
        if (lx->cur_token.type == TOKEN_LBRACKET) {
            lexer_next_token(lx);
            Node *index = parse_expression(lx);
            lexer_expect(lx, TOKEN_RBRACKET);
            lhs = make_node(NODE_ARRAY_REF);
            strncpy(lhs->array_ref.name, name, sizeof(lhs->array_ref.name));
            lhs->array_ref.index = index;
        } else {
            lhs = make_node(NODE_ID);
            strncpy(lhs->id.name, name, sizeof(lhs->id.name));
        }
    } else {
        fprintf(stderr, "Expected ID in assignment\n");
        exit(1);
    }

    lexer_expect(lx, TOKEN_ASSIGN);
    Node *rhs = parse_expression(lx);
    lexer_expect(lx, TOKEN_SEMI);

    Node *assign = make_node(NODE_ASSIGN);
    assign->assign.lhs = lhs;
    assign->assign.rhs = rhs;
    return assign;
}

Node *parse_print(Lexer *lx) {
    lexer_expect(lx, TOKEN_PRINT);
    lexer_expect(lx, TOKEN_LPAREN);
    Node *expr = parse_expression(lx);
    lexer_expect(lx, TOKEN_RPAREN);
    lexer_expect(lx, TOKEN_SEMI);
    Node *printn = make_node(NODE_PRINT);
    printn->printstmt.expr = expr;
    return printn;
}

Node *parse_if(Lexer *lx) {
    lexer_expect(lx, TOKEN_IF);
    Node *cond = parse_condition(lx);
    lexer_expect(lx, TOKEN_THEN);

    Node *then_block = NULL;
    if (lx->cur_token.type == TOKEN_LBRACE) {
        then_block = parse_block(lx);
    } else {
        then_block = make_node(NODE_BLOCK);
        then_block->block.stmts = malloc(sizeof(Node*));
        then_block->block.stmts[0] = parse_statement(lx);
        then_block->block.stmt_count = 1;
    }

    Node *else_block = NULL;
    if (lx->cur_token.type == TOKEN_ELSE) {
        lexer_next_token(lx);
        if (lx->cur_token.type == TOKEN_LBRACE) {
            else_block = parse_block(lx);
        } else {
            else_block = make_node(NODE_BLOCK);
            else_block->block.stmts = malloc(sizeof(Node*));
            else_block->block.stmts[0] = parse_statement(lx);
            else_block->block.stmt_count = 1;
        }
    }

    lexer_expect(lx, TOKEN_ENDIF);

    Node *ifnode = make_node(NODE_IF);
    ifnode->ifstmt.cond = cond;
    ifnode->ifstmt.then_block = then_block;
    ifnode->ifstmt.else_block = else_block;

    return ifnode;
}

Node *parse_statement(Lexer *lx) {
    if (lx->cur_token.type == TOKEN_INT) {
        return parse_declaration(lx);
    }
    if (lx->cur_token.type == TOKEN_ID) {
        return parse_assignment(lx);
    }
    if (lx->cur_token.type == TOKEN_PRINT) {
        return parse_print(lx);
    }
    if (lx->cur_token.type == TOKEN_IF) {
        return parse_if(lx);
    }
    fprintf(stderr, "Unexpected token in statement: %d (%s)\n", lx-
>cur_token.type, lx->cur_token.lexeme);
    exit(1);
}

Node *parse_block(Lexer *lx) {
    lexer_expect(lx, TOKEN_LBRACE);
    Node **stmts = NULL;
    int count = 0;
    while (lx->cur_token.type != TOKEN_RBRACE && lx->cur_token.type != TOKEN_EOF)
{
        Node *stmt = parse_statement(lx);
        stmts = realloc(stmts, sizeof(Node*)*(count+1));
        stmts[count++] = stmt;
    }
    lexer_expect(lx, TOKEN_RBRACE);
    Node *block = make_node(NODE_BLOCK);
    block->block.stmts = stmts;
    block->block.stmt_count = count;
    return block;
}

// Print AST nicely

void print_indent(int indent) {
    for (int i=0; i<indent; i++) putchar(' ');
}

void print_node(Node *node, int indent) {
    if (!node) return;
    print_indent(indent);
    switch(node->type) {
        case NODE_DECL:
            if (node->decl.array_size > 0)
```

```
                printf("Decl: int %s[%d]\n", node->decl.name, node-
>decl.array_size);
            else
                printf("Decl: int %s\n", node->decl.name);
            break;
        case NODE_ASSIGN:
            printf("Assign:\n");
            print_indent(indent+2);
            printf("LHS:\n");
            print_node(node->assign.lhs, indent+4);
            print_indent(indent+2);
            printf("RHS:\n");
            print_node(node->assign.rhs, indent+4);
            break;
        case NODE_BINOP:
            printf("BinOp: %c\n", (node->binop.op == TOKEN_PLUS) ? '+' :
                                   (node->binop.op == TOKEN_MINUS) ? '-' :
                                   (node->binop.op == TOKEN_MUL) ? '*' :
                                   (node->binop.op == TOKEN_DIV) ? '/' :
                                   (node->binop.op == TOKEN_GT) ? '>' : '?');
            print_node(node->binop.left, indent+2);
            print_node(node->binop.right, indent+2);
            break;
        case NODE_NUM:
            printf("Num: %d\n", node->num_val);
            break;
        case NODE_ID:
            printf("Id: %s\n", node->id.name);
            break;
        case NODE_ARRAY_REF:
            printf("ArrayRef: %s[", node->array_ref.name);
            print_node(node->array_ref.index, 0);
            printf("]\n");
            break;
        case NODE_IF:
            printf("If:\n");
            print_indent(indent+2);
            printf("Condition:\n");
            print_node(node->ifstmt.cond, indent+4);
            print_indent(indent+2);
            printf("Then:\n");
            print_node(node->ifstmt.then_block, indent+4);
            if (node->ifstmt.else_block) {
                print_indent(indent+2);
                printf("Else:\n");
                print_node(node->ifstmt.else_block, indent+4);
            }
            break;
        case NODE_PRINT:
            printf("Print:\n");
            print_node(node->printstmt.expr, indent+2);
            break;
        case NODE_BLOCK:
            printf("Block (%d statements):\n", node->block.stmt_count);
            for (int i=0; i<node->block.stmt_count; i++) {
                print_node(node->block.stmts[i], indent+2);
            }
            break;
        case NODE_UNARY:
            printf("UnaryOp: %c\n", (node->unary.op == TOKEN_MINUS) ? '-' : '?');
            print_node(node->unary.expr, indent+2);
            break;
        default:
            printf("Unknown node type\n");
    }
}

int main() {
    const char *code =
    "{\n"
    "int a[3],t1,t2;\n"
    "t1=2;\n"
    "a[0]=1;\n"
    "a[1]=2;\n"
    "a[t1]=3;\n"
    "t2=-( a[2]+t1*6)/(a[2]-t1);\n"
    "if t2 > 5 then\n"
    "print(t2);\n"
    "else\n"
    "{ int t3;\n"
    "t3=99;\n"
    "t2=25;\n"
    "print(-t1+t2*t3);\n"
    "}\n"
    "endif\n"
    "}\n";

    Lexer lx = {code, 0};
    lexer_next_token(&lx);
```

```c
    Node *ast = parse_block(&lx);

    print_node(ast, 0);

    return 0;
}
```

OUTPUT:



```
PS C:\Users\admin\Documents\23bce1968> ./a
Block (7 statements):
  Block (3 statements):
    Decl: int a[3]
    Decl: int t1
    Decl: int t2
  Assign:
    LHS:
      Id: t1
    RHS:
      Num: 2
  Assign:
    LHS:
      ArrayRef: a[Num: 0
]
    RHS:
      Num: 1
  Assign:
    LHS:
      ArrayRef: a[Num: 1
]
    RHS:
      Num: 2
  Assign:
    LHS:
      ArrayRef: a[Id: t1
]
    RHS:
      Num: 3
  Assign:
    LHS:
      Id: t2
    RHS:
      BinOp: /
        UnaryOp: -
          BinOp: +
            ArrayRef: a[Num: 2
]
            BinOp: *
              Id: t1
              Num: 6
        BinOp: -
          ArrayRef: a[Num: 2
]
          Id: t1
  If:
    Condition:
      BinOp: >
        Id: t2
        Num: 5
    Then:
      Block (1 statements):
        Print:
          Id: t2
    Else:
      Block (4 statements):
        Block (1 statements):
          Decl: int t3
        Assign:
          LHS:
            Id: t3
          RHS:
            Num: 99
        Assign:
          LHS:
            Id: t2
          RHS:
            Num: 25
        Print:
          BinOp: +
            UnaryOp: -
              Id: t1
            BinOp: *
              Id: t2
              Id: t3
PS C:\Users\admin\Documents\23bce1968>
```