实验报告：语法制导翻译与中间代码生成

一、实验目的

本实验旨在通过实现 `analyse_and_translate(string src)` 函数，达到设计语法制导翻译过程的目标，具体包括设计中间代码（四元式或三元式）的分析与生成过程。通过该实验，进一步理解语法分析、语义分析及中间代码生成的基本原理和实现方法。

二、实验原理

语法制导翻译是编译器技术中的一个重要概念，结合语法分析和语义分析，将源代码翻译成中间表示。中间代码通常采用三元式或四元式形式，这些中间表示独立于具体机器，便于后续的代码优化和目标代码生成。

三、代码实现

以下是主要函数 `analyse_and_translate(string src)` 函数的具体实现代码：

```
void analyse_and_translate(string src) {
    cout << "steps\t" << "op-stack\t" << "input\t" << "operation\t" << "state-
stack\t" << "ACTION\t" << "GOTO" << endl;
    vector<char> op_stack;
    vector<int> st_stack;
    vector<string> pl_stack; // 记录place属性
    src += "#";
    op_stack.push_back('#');
    st_stack.push_back(0);
    pl_stack.push_back("$");
    int steps = 1;

    for (int i = 0; i < src.length(); i++) {
        char u = src[i];
        int top = st_stack.back();
        Content &act = action[top][u]; // u隐式转化为int，故可以被索引

        if (act.type == 0) {
            cout << steps++ << "\t" << get_stk(op_stack) << "\t" << src.substr(i)
<< "\tshift\t" << get_stk(st_stack) << "\t" << act.out << "\t" << "" << endl;
            op_stack.push_back(u);
            pl_stack.push_back("$");
            st_stack.push_back(act.num);
        } else if (act.type == 1) {
            WF &tt = wf[act.num];
            int y = st_stack[st_stack.size() - tt.right.length() - 1];
            int x = Goto[y][tt.left[0]];
            cout << steps++ << "\t" << get_stk(op_stack) << "\t" << src.substr(i)
<< "\t" << get_shift(tt) << "\t" << get_stk(st_stack) << "\t" << act.out << "\t"
<< x << endl;

            // 生成翻译语句
            string var1, var2, var_tmp;
            switch (act.num) {
                case 0:
                    // S->E
```

```
                for (int j = 0; j < 1; j++) {
                    st_stack.pop_back();
                    op_stack.pop_back();
                }
                var_tmp = pl_stack.back();
                pl_stack.pop_back();

                op_stack.push_back(tt.left[0]);
                st_stack.push_back(x);
                pl_stack.push_back(var_tmp);
                break;
            case 1:
                // E->E+T
                for (int j = 0; j < 3; j++) {
                    st_stack.pop_back();
                    op_stack.pop_back();
                }
                var1 = pl_stack.back();
                pl_stack.pop_back();
                pl_stack.pop_back();
                var2 = pl_stack.back();
                pl_stack.pop_back();
                var_tmp = newTemp();

                quads.push_back({"+", var2, var1, var_tmp});
                varToQuad[var_tmp] = to_string(quads.size());

                op_stack.push_back(tt.left[0]);
                st_stack.push_back(x);
                pl_stack.push_back(var_tmp);
                break;
            case 2:
                // E->T
                for (int j = 0; j < 1; j++) {
                    st_stack.pop_back();
                    op_stack.pop_back();
                }
                var_tmp = pl_stack.back();
                pl_stack.pop_back();

                op_stack.push_back(tt.left[0]);
                st_stack.push_back(x);
                pl_stack.push_back(var_tmp);
                break;
            case 3:
                // T->T*F
                for (int j = 0; j < 3; j++) {
                    st_stack.pop_back();
                    op_stack.pop_back();
                }
                var1 = pl_stack.back();
                pl_stack.pop_back();
                pl_stack.pop_back();
                var2 = pl_stack.back();
                pl_stack.pop_back();
                var_tmp = newTemp();
```

```
            quads.push_back({"*", var2, var1, var_tmp});
            varToQuad[var_tmp] = to_string(quads.size());

            op_stack.push_back(tt.left[0]);
            st_stack.push_back(x);
            pl_stack.push_back(var_tmp);
            break;
        case 4:
            // T->F
            for (int j = 0; j < 1; j++) {
                st_stack.pop_back();
                op_stack.pop_back();
            }
            var_tmp = pl_stack.back();
            pl_stack.pop_back();

            op_stack.push_back(tt.left[0]);
            st_stack.push_back(x);
            pl_stack.push_back(var_tmp);
            break;
        case 5:
            // F->(E)
            for (int j = 0; j < 3; j++) {
                st_stack.pop_back();
                op_stack.pop_back();
            }
            pl_stack.pop_back();
            var_tmp = pl_stack.back();
            pl_stack.pop_back();
            pl_stack.pop_back();

            op_stack.push_back(tt.left[0]);
            st_stack.push_back(x);
            pl_stack.push_back(var_tmp);
            break;
        case 6:
            // F->a
            for (int j = 0; j < 1; j++) {
                st_stack.pop_back();
                op_stack.pop_back();
                pl_stack.pop_back();
            }
            var_tmp = "a";
            op_stack.push_back(tt.left[0]);
            st_stack.push_back(x);
            pl_stack.push_back(var_tmp);
            break;
        case 7:
            // F->b
            for (int j = 0; j < 1; j++) {
                st_stack.pop_back();
                op_stack.pop_back();
                pl_stack.pop_back();
            }
            var_tmp = "b";
```

```
                    op_stack.push_back(tt.left[0]);
                    st_stack.push_back(x);
                    pl_stack.push_back(var_tmp);
                    break;
                case 8:
                    // F->c
                    for (int j = 0; j < 1; j++) {
                        st_stack.pop_back();
                        op_stack.pop_back();
                        pl_stack.pop_back();
                    }
                    var_tmp = "c";
                    op_stack.push_back(tt.left[0]);
                    st_stack.push_back(x);
                    pl_stack.push_back(var_tmp);
                    break;
            }

            i--;
        } else if (act.type == 2) {
            cout << steps++ << "\t" << get_stk(op_stack) << "\t" << src.substr(i)
<< "\tAccept\t" << get_stk(st_stack) << "\t" << act.out << "\t" << "" << endl;
        } else
            continue;
    }

    // 打印四元式
    for (size_t i = 0; i < quads.size(); ++i) {
        cout << "(" << i + 1 << ") (" << quads[i].op << ", " << quads[i].arg1 <<
", " << quads[i].arg2 << ", " << quads[i].result << ")" << endl;
    }

    // 打印三地址码
    for (const auto& quad : quads) {
        cout << quad.result << " = " << quad.arg1 << " " << quad.op << " " <<
quad.arg2 << endl;
    }

    // 生成并打印逆波兰式
    stack<char> s;
    string rpn;
    for (char c : src) {
        if (isalnum(c)) {
            rpn += c;
        } else if (c == '(') {
            s.push(c);
        } else if (c == ')') {
            while (!s.empty() && s.top() != '(') {
                rpn += s.top();
                s.pop();
            }
            s.pop(); // pop '('
        } else {
            while (!s.empty() && precedence(s.top()) >= precedence(c) && s.top()
!= '(') {
                rpn += s.top();
```

```
                s.pop();
            }
            s.push(c);
        }
    }
    while (!s.empty()) {
        rpn += s.top();
        s.pop();
    }

    // 移除末尾的 '#'
    if (!rpn.empty() && rpn.back() == '#') {
        rpn.pop_back();
    }
    cout << rpn << endl;
}
```

主函数:

```
int main()
{
    int n;
    char s[MAX];
    string str;
    cout << "请输入文法:" << endl;
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        scanf("%s", s);
        int len = strlen(s), j;
        for (j = 0; j < len; j++)
            if (s[j] == '-')
                break;
        s[j] = 0;
        wf.push_back(WF(s, s + j + 2, -1, -1));
#ifdef DEBUG
        // wf[wf.size() - 1].print();
#endif
    }
    cout << "请输入输入串:" << endl;
    cin >> str;
    // cout << "YES" << endl;

    if(checkExpression(str) == false){
        return 0;
    }

    make_item();
    make_first();
    make_follow();
    make_set();
    make_V();
    make_go();
    make_table();
```

```
    //analyse(str);
    analyse_and_translate(str);
    puts("-------------翻译结果------------");
    for(int i = 0; i < result.size(); i++){
        cout << result[i] << endl;
    }

    return 0;
}
```

四、代码分析

`analyse_and_translate` 函数在完成语法分析和翻译时，分别生成了四元式、三地址码和逆波兰式。以下是它们各自的生成过程的简要分析：

1. 生成四元式

在规约操作部分，函数根据不同的规约规则（由 `act.num` 表示）生成对应的四元式。四元式的生成步骤如下：

- **规约规则1：E->E+T** 和 **规约规则3：T->T*F**：
  - 从 `pl_stack` 栈中弹出右部的变量，生成一个新临时变量 `var_tmp`。
  - 创建四元式，并将其添加到 `quads` 向量中。
  - 将新临时变量 `var_tmp` 压回 `pl_stack`。

```
quads.push_back({"+", var2, var1, var_tmp});
varToQuad[var_tmp] = to_string(quads.size());
```

2. 生成三地址码

在函数的最后部分，遍历 `quads` 向量，按照四元式的格式输出三地址码。三地址码的生成步骤如下：

- 遍历 `quads` 向量中的每一个四元式，并以 `result = arg1 op arg2` 的格式输出。

```
for (const auto& quad : quads) {
    cout << quad.result << " = " << quad.arg1 << " " << quad.op << " " <<
quad.arg2 << endl;
}
```

3. 生成逆波兰式

在函数的最后部分，通过对输入字符串 `src` 的遍历生成逆波兰式。生成逆波兰式的步骤如下：

- 使用一个栈 `s` 来暂存运算符。
- 对输入字符串 `src` 进行遍历：
  - 如果是操作数，直接添加到 `rpn` 字符串中。
  - 如果是左括号，压入栈 `s`。
  - 如果是右括号，弹出栈顶直到遇到左括号。
  - 如果是运算符，根据优先级将栈顶运算符弹出并添加到 `rpn` 字符串中，然后将当前运算符压入栈中。

- 最后，将栈中剩余的运算符全部弹出并添加到 `rpn` 字符串中。
- 移除末尾的 `#`，然后输出逆波兰式。

```
stack<char> s;
string rpn;
for (char c : src) {
    if (isalnum(c)) {
        rpn += c;
    } else if (c == '(') {
        s.push(c);
    } else if (c == ')') {
        while (!s.empty() && s.top() != '(') {
            rpn += s.top();
            s.pop();
        }
        s.pop(); // pop '('
    } else {
        while (!s.empty() && precedence(s.top()) >= precedence(c) && s.top() !=
'(') {
            rpn += s.top();
            s.pop();
        }
        s.push(c);
    }
}
while (!s.empty()) {
    rpn += s.top();
    s.pop();
}

// 移除末尾的 '#'
if (!rpn.empty() && rpn.back() == '#') {
    rpn.pop_back();
}
cout << rpn << endl;
```

五、实验结果

输入：

```
PS D:\code\C++\bianyi\5>  & 'c:\U
t-MIEngine-Out-qz3dxqfj.c0b' '--s
请输入文法:
9
S->E
E->E+T
E->T
T->T*F
T->F
F->(E)
F->a
F->b
F->c
请输入输入串:
a+b+c+(a*a)
```

输出:

```
steps   op-stack        input   operation       state-stack     ACTION  GOTO
1       #       a+b+c+(a*a)#    shift   0       S5
2       #a      +b+c+(a*a)#     reduce(F->a)    05      R6      3
3       #F      +b+c+(a*a)#     reduce(T->F)    03      R4      4
4       #T      +b+c+(a*a)#     reduce(E->T)    04      R2      2
5       #E      +b+c+(a*a)#     shift   02      S9
6       #E+     b+c+(a*a)#      shift   029     S6
7       #E+b    +c+(a*a)#       reduce(F->b)    0296    R7      3
8       #E+F    +c+(a*a)#       reduce(T->F)    0293    R4      12
9       #E+T    +c+(a*a)#       reduce(E->E+T)  02912   R1      2
10      #E      +c+(a*a)#       shift   02      S9
11      #E+     c+(a*a)#        shift   029     S7
12      #E+c    +(a*a)# reduce(F->c)    0297    R8      3
13      #E+F    +(a*a)# reduce(T->F)    0293    R4      12
14      #E+T    +(a*a)# reduce(E->E+T)  02912   R1      2
15      #E      +(a*a)# shift   02      S9
16      #E+     (a*a)#  shift   029     S1
17      #E+(    a*a)#   shift   0291    S5
18      #E+(a   *a)#    reduce(F->a)    02915   R6      3
19      #E+(F   *a)#    reduce(T->F)    02913   R4      4
20      #E+(T   *a)#    shift   02914   S10
21      #E+(T*  a)#     shift   0291410 S5
22      #E+(T*a )#      reduce(F->a)    02914105        R6      13
23      #E+(T*F )#      reduce(T->T*F)  029141013       R3      4
24      #E+(T   )#      reduce(E->T)    02914   R2      8
25      #E+(E   )#      shift   02918   S11
26      #E+(E)  #       reduce(F->(E))  0291811 R5      3
27      #E+F    #       reduce(T->F)    0293    R4      12
28      #E+T    #       reduce(E->E+T)  02912   R1      2
29      #E      #       Accept  02      acc
(1) (+, a, b, T0)
(2) (+, T0, c, T1)
(3) (*, a, a, T2)
(4) (+, T1, T2, T3)
(1) (+ a b)
(2) (+ (1) c)
(3) (* a a)
(4) (+ (2) (3))
ab+c+aa*+
```

六、实验总结

本实验通过实现`analyse_and_translate`函数，深入理解了语法分析、语义分析和中间代码生成的基本原理。在实现过程中，掌握了状态转换、移进和规约操作的具体实现方法，以及如何在规约过程中生成中间代码。实验结果验证了设计的正确性和有效性，为后续编译器优化和目标代码生成打下了基础。

通过本实验，进一步加深了对编译原理的理解和掌握，为将来的编译器设计和实现提供了宝贵的经验和借鉴。