设计文档

设计思路

这个实验的关键就是看懂原来 `tiny` 语言的文法代码，然后在其上面进行一些修改。这里我主要修改了下面的地方。

1. `not` 这个单元运算符。

这个运算符我是修改了 `factor` 函数中的实现。

```
TreeNode *factor() {
    TreeNode *t = nullptr;
    switch (currentTokenType) {
        case NUM :
            t = newExpNode(CONST_TYPE);
            if (currentTokenType == NUM)
                t->attr.val = atoi(currentTokenValue.c_str());
            match(NUM);
            break;
        case ID :
            t = newExpNode(ID_TYPE);
            if (currentTokenType == ID)
                t->attr.name = copyString(currentTokenValue);
            match(ID);
            break;
        case LPAREN :
            match(LPAREN);
            t = exp();
            match(RPAREN);
            break;
        case NOT:
            t = newExpNode(SINGLE_EXP_TYPE);
            if (currentTokenType == NOT)
                t->attr.name = copyString(currentTokenValue);
            match(NOT);
            t->child[0] = factor();
            break;
        default:
            cout << "ERROR: factor" << endl;
            break;
    }
    return t;
}
```

我们知道 `factor` 原来的文法规则如下：

```
factor -> (exp) | number | identifer
```

然后我为了实现单元运算符 `not` ，就将其修改为了：

```
factor -> (exp) | number | identifier | not factor
```

每当我们在 `factor` 中识别到了 `not` ，就递归地再次调用 `factor` 函数。

2. 新实现 `for` 循环文法规则。

这个实验规定的 `for` 循环文法规则如下：

```
for-stmt ->for identifier:=simple-exp  downto  simple-exp  do  stmt-sequence enddo
```

因此不难根据其它的代码模仿得到 `for` 循环的函数实现：

```
TreeNode *for_stmt() {
    TreeNode *t = newStmtNode(FOR_TYPE);
    match(FOR);
    t->child[0] = assign_stmt();
    if (currentTokenType == TO) {
        match(TO);
        t->attr.name = copyString("to");
    } else if (currentTokenType == DOWNTO) {
        match(DOWNTO);
        t->attr.name = copyString("downto");
    } else {
        cout << "ERROR in for stmt" << endl;
    }
    t->child[1] = exp();
    match(DO);
    t->child[2] = stmt_sequence();
    match(ENDDO);
    return t;
}
```

这里我将 `to/downto` 存在了 `t->attr.name` 这个属性中。

3. `if` 语句的修改。

```
/**
 * 规定if语句的代码段里面只能有一个语句（也就是statement而非statement_sequence）
 * @return
 */
TreeNode *if_stmt() {
    TreeNode *t = newStmtNode(IF_TYPE);
    match(IF);
    match(LPAREN);
    t->child[0] = exp();
    match(RPAREN);
    t->child[1] = statement();
    match(SEMI);
    if (currentTokenType == ELSE) {
        match(ELSE);
        t->child[2] = statement();
```

```
    }
    return t;
}
```

这个实验给 `if` 语句增加了括号，因此我们需要在 `if_stmt` 函数的实现中添加 `match(LPAREN)` 和 `match(RPAREN)` 。

4. 扩充的运算符号。

这个相对来说比较简单，只要在原来的符号表上增加符号的判断就可以了。

- 比较符的扩充

```
TreeNode *exp() {
    TreeNode *t = simple_exp();
    if ((currentTokenType == LT) || (currentTokenType == EQ) || (currentTokenType == GT) ||
        (currentTokenType == LE) || (currentTokenType == GE) || (currentTokenType == NE)) {
        TreeNode *p = newExpNode(OP_TYPE);
        p->child[0] = t;
        p->attr.op = currentTokenType;
        t = p;
        match(currentTokenType);
        t->child[1] = simple_exp();
    }
    return t;
}
```

- 位运算符的扩充

```
TreeNode *simple_exp() {
    TreeNode *t = term();
    while ((currentTokenType == PLUS) || (currentTokenType == MINUS) ||
           (currentTokenType == AND) || (currentTokenType == OR)) {
        TreeNode *p = newExpNode(OP_TYPE);
        p->child[0] = t;
        p->attr.op = currentTokenType;
        t = p;
        match(currentTokenType);
        t->child[1] = term();
    }
    return t;
}
```

- 新运算符的扩充

```
TreeNode *term() {
    TreeNode *t = factor();
    while ((currentTokenType == TIMES) || (currentTokenType == OVER) ||
           (currentTokenType == MOD) || currentTokenType == POW) {
        TreeNode *p = newExpNode(OP_TYPE);
        p->child[0] = t;
        p->attr.op = currentTokenType;
        t = p;
```

```
        match(currentTokenType);
        p->child[1] = factor();
    }
    return t;
}
```

5. 最后的结果树状展示。

这里我用了 `github` 上开源的 `cpp` 的 `json` 库，用来将存储的结构体链表转为 `json` 字符串输出。

```
json toJson(TreeNode *t) {
    vector<json> ret;
    while (t != nullptr) {
        json tmp;
        switch (t->nodeType) {
            case STMT_TYPE:
                switch (t->type.stmt) {
                    case IF_TYPE:
                        tmp["IfStatement"]["TestExpression"] = toJson(t->child[0]);
                        tmp["IfStatement"]["MainBody"] = toJson(t->child[1]);
                        if (t->child[2] != nullptr) {
                            tmp["IfStatement"]["ElseBody"] = toJson(t->child[2]);
                        }
                        ret.push_back(tmp);
                        break;
                    case REPEAT_TYPE:
                        tmp["RepeatStatement"]["body"] = toJson(t->child[0]);
                        tmp["RepeatStatement"]["end"] = toJson(t->child[1]);
                        ret.push_back(tmp);
                        break;
                    case ASSIGN_TYPE:
                        tmp["assign"]["identifier"] = t->attr.name;
                        tmp["assign"]["assignExpression"] = toJson(t->child[0]);
                        tmp["assign"]["type"] = t->op;
                        ret.push_back(tmp);
                        break;
                    case READ_TYPE:
                        tmp["read"] = t->attr.name;
                        ret.push_back(tmp);
                        break;
                    case WRITE_TYPE:
                        tmp["write"] = toJson(t->child[0]);
                        ret.push_back(tmp);
                        break;
                    case FOR_TYPE:
                        tmp["for"]["initExpression"] = toJson(t->child[0]);
                        tmp["for"]["endExpression"] = toJson(t->child[1]);
                        tmp["for"]["body"] = toJson(t->child[2]);
                        tmp["for"]["order"] = t->attr.name;
                        ret.push_back(tmp);
                        break;
                    default:
                        cout << "error node" << endl;
                        break;
                }
```

```
                break;
            case EXP_TYPE:
                switch (t->type.exp) {
                    case OP_TYPE:
                        tmp["op"]["value"] = tokenNameMap.find(t->attr.op)->second;
                        tmp["op"]["leftExpression"] = toJson(t->child[0]);
                        tmp["op"]["rightExpression"] = toJson(t->child[1]);
                        ret.push_back(tmp);
                        break;
                    case CONST_TYPE:
                        tmp["const"] = t->attr.val;
                        ret.push_back(tmp);
                        break;
                    case ID_TYPE:
                        tmp["identifier"] = t->attr.name;
                        ret.push_back(tmp);
                        break;
                    case SINGLE_EXP_TYPE:
                        tmp["single-expression"]["op"] = t->attr.name;
                        tmp["single-expression"]["expression"] = toJson(t->child[0]);
                        ret.push_back(tmp);
                        break;
                    default:
                        cout << "error node" << endl;
                        break;
                }
                break;
            default:
                cout << "error node" << endl;
                break;
        }
        t = t->sibling;
    }
    return ret;
}
```