编译原理课程设计

摘要

本报告有两部分。

第一部分是一个自动化的类 C 编译器，根据语法配置文件 grammar.in、语义规则配置文件 rules.in 来对源文件进行分析，使用 LR 自动机，一遍扫描产生语法树、四元式中间代码，然后再处理中间代码，产生 LLVM IR 目标汇编代码。生成的目标代码可以汇编执行。目前的语义规则支持 if 条件语句、whlie 循环语句、函数调用与取得返回值等功能。

第二部分是一个解释性的符号计算器，以 Mathematica 为学习目标，实现了一个符号计算器。

附件中包含本报告的 docx 文件、类 C 编译器的代码（compiler.zip）、符号计算器的代码与日志（mymma.zip）。

目录

摘要.....1
1　一个自动化的类 C 编译器.....3
1.1 要求.....3
1.2 设计.....3
1.3 词法分析.....4
1.3.1 词法规则.....4
1.3.2 词法分析器类 Lexer.....4
1.4 语法分析.....5
1.4.1 语法配置文件的规则.....5
1.4.2 本次使用的语法配置文件（即语法规则）.....5
1.4.3 自动机.....6
1.4.4 效果截图.....13
1.5 语义分析与中间代码生成.....14
1.5.1 语法分析树节点类 ASTnode.....14
1.5.2 语义配置文件以及其规则.....15
1.5.3 符号表以及相关类.....16
1.5.4 回填.....18
1.5.5 常量复用.....19
1.5.6 效果截图.....20
1.6 目标代码生成.....22
1.7 最终效果展示.....23
1.7.1 测试程序源代码.....23
1.7.2 中间代码.....24
1.7.3 目标代码.....25
1.7.4 可执行文件与执行结果.....26
2　一个解释性的符号计算器.....27
2.1 原理与日志.....27
2.2 使用说明.....28
2.3 效果展示.....28
参 考 文 献.....31

1　一个自动化的类 C 编译器

1.1 要求

1.　使用 C++实现一个类 C 语言的编译器，可以提供词法分析、语法分析、符号表管理、中间代码生成以及目标代码生成等功能。具体要求如下：

（1）　使用高级程序语言作为实现语言，实现一个类 C 语言的编译器。编码实现编译器的组成部分。

（2）　要求的类 C 编译器是个一遍的编译程序，词法分析程序作为子程序，需要的时候被语法分析程序调用。

（3）　使用语法制导的翻译技术，在语法分析的同时生成中间代码，并保存到文件中。

（4）　要求输入类 C 语言源程序，输出中间代码表示的程序;

（5）　要求输入类 C 语言源程序，输出目标代码（可汇编执行）的程序。

（6）　实现过程、函数调用的代码编译。

（7）　拓展类 C 语言文法，实现包含数组的中间代码以及目标代码生成。

其中（1)、(2)（3）（4）（5）是必做内容，（6）（7）是选作内容。

1.2 设计

这个编译器，接收的是我定的语法规则下语言的源程序，输出的目标代码是 LLVM IR 汇编，然后通过 LLVM llc 与 gcc 将其翻译成可执行文件。

其中，主体是语法分析，使用的是 LR 自动机，该自动机由文法配置文件生成，采用自底向上的分析方式。在其分析过程中，接受的是由词法分析器接口给出的 Token 元素，词法分析器与源文件输入流绑定，需要时从输入流中读取字符直到分析出一个完整的 Token，然后给出该 Token。语法分析的过程中规约构建出语法树，每次规约的过程中，调用语义分析程序同时生成中间代码（四元式），语义分析程序根据语义规则配置文件进行分析。LR 自动机成功跑完后，根据生成的四元式序列，处理并生成被 LLVM llc 接收的 LLVM IR 汇编代码。最后，用 LLVM 工具链将 LLVM IR 汇编代码转化为可执行文件（好处在于，LLVM IR 是跨平台的，可以转化为各平台下的可执行文件）。

其中，我的程序（从读取源程序直到最终生成 LLVM IR 汇编目标程序）是无依赖的，未用除 c++标准库外的其它库。而且我的程序根据配置文件进行分析，实现了自动化。

总的来说，这个编译过程，用到了这几个文件：

（1）　我的 LLVM IR 生成器的可执行文件。

（2）　语法规则配置文件 grammar.in

（3）　语义规则配置文件 rules.in

（4）　待编译的源代码文件

（5）　llc、gcc 等汇编工具

会生成这几个文件

（1）　我的可执行文件生成的 LLVM IR 汇编文件 code.ll

（2）　llc 生成的目标机器上的汇编文件 code.s

（3）　gcc 汇编生成的目标机器上的可执行文件 code.out

（4）　分析源程序过程中的日志文件 log.txt

1.3 词法分析

出于简洁性考虑，以及该语言本来就不太大，所以词法分析直接使用手写代码进行分析，而不是根据配置文件构建自动机。

1.3.1 词法规则

关键字 ： bool | char | int | true | false | and | or | not | if | then | while | break | return
ID ： (letter | _) (letter | digit | _) *，且不与关键字相同
NUMCONST ： digit (digit)*
CHARCONST ： ' (letter)* '，注意转义符\
STRINGCONST ： "(letter)*"，注意转义符\
运算符 ： = | *= | /= | += | -= | < | > | == | != | >= | <= | + | - | * | / | % | - | *
分隔符 ： 其余单个的特殊字符，具体处理在语义分析时进行

1.3.2 词法分析器类 Lexer

因为是以语法分析为主体，所以词法分析器返回的 Token 也不专门设成一个类了，而是直接返回一个 ASTnode*，语法分析树节点指针，里面包含了 Token 类型以及具体内容的信息。语法分析自动机接收到词法分析器返回的指针后，直接检查并将其作为终结符压栈。

```
class Lexer {
public:
    std::ifstream fin;
    std::pair<int, int> pos;  // 当前字符位于源文件的位置
    int curCh;

    // 构造函数
    Lexer(const char *fname): fin(fname), pos(1, 0), flag(0) {
        assert(fin.is_open());
        getNextChar();
    }

    // 从流中获取一个字符，返回该字符。
    // 返回 0 代表结束
    char getNextChar();

    ASTnode* getNextToken();
};
```

1.4 语法分析

这是这个编译器的一个重点。

我首先定了一个语法配置文件的规则，然后写了代码来读取并分析配置文件并生成自动机，以及语法分析跑自动机的过程。

1.4.1 语法配置文件的规则

（1）保留字：|、::=、<_>、EMPTY

（2）非终结符的表示：由尖括号括起来的一个字符串。例：<program>

（3）终结符的表示：除了保留字、非终结符，其它的字符串。例：ID、%

（4）语法规则：非终结符 ::= 规则列表。其中，规则列表是由 | 分割的非空字符串列表。例：<varDeclInit> ::= <varDeclId> | <varDeclId> = <simpleExp>

（5）起始符号：只能有一个起始符号，假设起始符号是<P>，那么使用唯一一条语法规则 <_> ::= <P> 来指明它是起始符号。

（6）EMPTY 代表空串

1.4.2 本次使用的语法配置文件（即语法规则）

<_> ::= <program>

<assignop> ::= = | *= | /= | += | -=
<relop> ::= < | > | == | != | >= | <=
<sumop> ::= + | -
<mulop> ::= * | / | %
<unaryop> ::= - | *

<program> ::= <declList>

<declList> ::= <declList> <decl> | <decl>
<decl> ::= <varDecl> | <funDecl>

<typeSpec> ::= bool | char | int

<varDecl> ::= <typeSpec> <varDeclList> ;
<scopedVarDecl> ::= <typeSpec> <varDeclList> ;
<varDeclList> ::= <varDeclList> , <varDeclInit> | <varDeclInit>
<varDeclInit> ::= <varDeclId> | <varDeclId> = <simpleExp>
<varDeclId> ::= ID | ID [NUMCONST]

<funDecl> ::= <funHead> <compoundStmt>
<funHead> ::= <typeSpec> ID (<parms>)
<parms> ::= <parmList> | EMPTY
<parmList> ::= <parmList> ; <parmTypeList> | <parmTypeList>
<parmTypeList> ::= <typeSpec> <parmIdList>
<parmIdList> ::= <parmIdList> , <parmId> | <parmId>

<parmId> ::= ID | ID [ ]

<stmtList> ::= <stmtList> <stmt> | EMPTY
<stmt> ::= <expStmt> | <compoundStmt> | <selectStmt> | <iterStmt> | <returnStmt> | <breakStmt>
<expStmt> ::= <exp> ; | ;
<compoundStmt> ::= { <localDecls> <stmtList> }
<localDecls> ::= <localDecls> <scopedVarDecl> | EMPTY
<selectStmt> ::= if (<simpleExp>) <stmt> | if (<simpleExp>) <stmt> <else> <stmt>
<else> ::= else
<iterStmt> ::= while (<simpleExp>) <stmt>
<returnStmt> ::= return ; | return <exp> ;
<breakStmt> ::= break ;

<exp> ::= <mutable> = <exp> | <simpleExp>
<simpleExp> ::= <simpleExp> or <andExp> | <andExp>
<andExp> ::= <andExp> and <unaryRelExp> | <unaryRelExp>
<unaryRelExp> ::= not <unaryRelExp> | <relExp>
<relExp> ::= <sumExp> <relop> <sumExp> | <sumExp>
<sumExp> ::= <sumExp> <sumop> <mulExp> | <mulExp>
<mulExp> ::= <mulExp> <mulop> <unaryExp> | <unaryExp>
<unaryExp> ::= <unaryop> <unaryExp> | <factor>
<factor> ::= <mutable> | <immutable>
<mutable> ::= ID | ID [<exp>]
<immutable> ::= (<exp>) | <call> | <constant>
<call> ::= ID (<args>)
<args> ::= <argList> | EMPTY
<argList> ::= <argList> , <exp> | <exp>
<constant> ::= NUMCONST | CHARCONST | STRINGCONST | true | false

1.4.3 自动机

本来想着先搞个 LR(0)，然后再稍微改改变成 LR(1)，结果发现 LR(0)直接就能接受这个语言规则了，那就没再改成 LR(1)了。（起始 LR(1)还不用写 FOLLOW，反而更简单一点，不过就是状态更多了）

构建 LR(0)自动机，主要有这几部分：FIRST、FOLLOW、CLOSURE、GOTO，跳转表，以及最后跑自动机的过程。

而 LR(1)自动机，将状态里的项多了一个向前看分量，就不用再求 FOLLOW，再相应地修改一下 CLOSURE 即可。

下面是这几个部分的实现

（1）FIRST

FIRST(A) 的定义是，符号 A 能推导出的字符串的首终结符的集合。

注意，FIRST(A)包含 EMPTY，当且仅当 FIRST(A)可以推导出空串。

其实就是一个不断迭代，直到集合不再改变的过程。具体看代码吧。

注意处理好 EMPTY。

```
void get_first() {
    FIRST.clear();

    // 终结符号的 FIRST
    for (auto X: TERMINALS) {
        FIRST[X].insert(X);
    }

    // 非终结符号的 FIRST
    // FIRST[X] 中有 EMPTY，当且仅当 X =*=> EMPTY
    for (bool flag_done=false; !flag_done; ) {
        flag_done = true;
        for (auto [X, S]: statements) {
            bool flag_all_empty = true;
            for (auto Y: S) {

                for (auto t: FIRST[Y]) if (t != "EMPTY" && FIRST[X].count(t) == 0) {
                    flag_done = false;
                    FIRST[X].insert(t);
                }

                if (FIRST[Y].count("EMPTY") == 0) {flag_all_empty=false; break;}
            }

            // 别忘了这个
            if (flag_all_empty && !FIRST[X].count("EMPTY")) {
                flag_done = false;
                FIRST[X].insert("EMPTY");
            }
        }
    }
}
```

（2）FOLLOW

FOLLOW(A) 意思是，符号 A 匹配完之后，可能会遇到的第一个终结符的集合。

和 FIRST 类似，也是一个不断迭代，直到集合不变的过程。

同样注意 EMPTY 的处理。

```
void get_follow() {
    FOLLOW.clear();

    FOLLOW[STARTSYMBOL].insert("$");
```

```
    for (bool flag_done=false; !flag_done; ) {
        flag_done = true;
        for (auto [A, S]: statements) {

            int n = S.size();

            for (int i=0, nxt; i<n-1; i = nxt) {
                auto B = S[i];
                // 2023.5.20 加了 nxt, 用于 EMPTY 的判断, 处理类似
<stmtList>      ::=     <stmtList> EMPTY <stmt>
                nxt = i+1;
                while (nxt < n && S[nxt] == "EMPTY") nxt++;
                if (nxt >= n) break;
                for (auto x: FIRST[S[nxt]]) {
                    if (x != "EMPTY" && FOLLOW[B].count(x) == 0) {
                        flag_done = false;
                        FOLLOW[B].insert(x);
                    }
                }
            }

            for (int i=0; i<n; i++) if (i==n-1 || FIRST[S[i+1]].count("EMPTY")) {
                auto B = S[i];
                for (auto x: FOLLOW[A]) {
                    if (x != "EMPTY" && FOLLOW[B].count(x) == 0) {
                        flag_done = false;
                        FOLLOW[B].insert(x);
                    }
                }
            }
        }
    }
}
```

（3）CLOSURE

一个 CLOSURE 即为自动机中的一个状态。里面包含的是一些项。在 LR(0) 自动机中，一个项由一个二元组（statement, pos）组成，代表当前符号栈，从上往下，可以匹配到 statement 右部前 pos-1 个符号。一个项集 I 的 CLOSURE，闭包，定义为改符号栈状态下所有能符合的项的集合。

求 CLOSURE 的过程也是一个不断迭代直到集合不变的过程。

```
SetOfItems CLOSURE(SetOfItems I) {
    for (bool flag_done=false; !flag_done; ) {
        flag_done = true;
```

```
        auto oldI = I;
        for (auto [sid, pos]: oldI) if (pos < statements[sid].second.size()) {
            for (int i=0; i<statements.size(); i++)
                if (statements[i].first == statements[sid].second[pos])
                    if (I.count(Item(i, 0)) == 0) {
                        flag_done = false;
                        I.insert(Item(i, 0));
                    }

            // EMPTY
            if (statements[sid].second[pos]=="EMPTY" && I.count(Item(sid,
pos+1))==0) {
                flag_done = false;
                I.insert(Item(sid, pos+1));
            }

        }
    }
    return I;
}
```

（4）GOTO

GOTO 是个映射函数，由 (CLOSURE, Symbol) 映射到 (CLOSURE)。求的是某个 CLOSURE 在接受 Symbol 符号后，能到达的项闭包。也是自动机图中探索新状态的过程。

转移的过程，其实就是接受符号的过程，看 pos 下一个可能的符号是不是对应的 Symbol 即可。

```
SetOfItems GOTO(int cid, string X) {
    if (gotoMap.count({cid, X})) return closures[gotoMap[{cid, X}]];

    SetOfItems ret;
    for (auto [sid, pos]: closures[cid]) {
        const auto &[L, R] = statements[sid];
        while (pos < R.size() && R[pos] == "EMPTY") pos++;
        if (pos < R.size() && R[pos] == X) {
            ret.insert(Item(sid, pos+1));
        }
    }
    return CLOSURE(ret);
}
```

（5）跳转表

这是自动机最终的状态转移表，是利用 GOTO 不断迭代生成的。约定了在某个状态，

遇到某个符号时应做的操作（移入/规约）。

注意，由于我这个语言不可避免地会有二义性（在 if else 嵌套时），所以我人为规定，在移入规约有冲突时，优先使用移入。当然，遇到冲突时我进行了相应的提示输出。结果显示，我的语法还是性质挺好的，只有 if else 这一处存在二义性冲突。

```
void generate_DFA() {
    closures.clear();
    closures.push_back(CLOSURE(SetOfItems{Item(0, 0)}));

    for (bool flag_done=false; !flag_done; ) {
        flag_done = true;

        for (int i=0; i<closures.size(); i++) {
            for (auto &X: SYMBOLS) if (X != "EMPTY") {
                auto tmpI = GOTO(i, X);
                if (tmpI.empty()) continue;
                for (int j=0; j<closures.size(); j++) if (closures[j] == tmpI)
                    gotoMap[{i, X}] = j;
                if (gotoMap.count({i, X}) == 0) {
                    gotoMap[{i, X}] = closures.size();
                    closures.push_back(tmpI);
                    flag_done = false;
                }
            }
        }
    }
}

void generate_table() {
    table.clear();

    for (int i=0; i<closures.size(); i++) {
        for (auto [sid, pos]: closures[i]) {

            const auto &[L, R] = statements[sid];

            if (pos < R.size()) {   // 移入
                if (is_terminal(R[pos]) && R[pos]!="EMPTY" && gotoMap.count({i,
R[pos]})) {
                    check_table(i, R[pos], T_S, gotoMap[{i, R[pos]}]);
                    table[{i, R[pos]}] = {T_S, gotoMap[{i, R[pos]}]};
                }
```

```
            }
            else {      // 规约
                if (L != STARTSYMBOL) {
                    for (auto x: FOLLOW[L]) if (x!="EMPTY") {
                        if (!check_table(i, x, T_R, sid))
                            continue;   // 优先移入，处理 else
                        table[{i, x}] = {T_R, sid};
                    }
                }
                else {
                    table[{i, EOFTERMINAL}] = {T_ACC, 0};
                }
            }
        }
    }
}
```

（6）自动机的使用过程

自动机的运行过程，即扫描文本的分析过程。

这其实是个下推自动机，运行过程包含图状态以及一个状态栈、一个符号栈。

简单地说，就是根据当前状态以及下一个符号，来决定是移入（符号压栈，状态转移到对应的 GOTO，新状态入栈），还是规约（根据对应的产生式，从符号栈、状态栈顶弹出相应数量的符号、状态（也就是自动机图往回走的一个过程），再将产生式左部符号压栈，并状态转移到对应的 GOTO，新状态入栈）。其间规约的时候，就可以生成语法子树，调用语义分析代码进行语义检查并生成中间代码。

跑自动机的时候，若没有对应的移入、规约操作，则说明语法不对应，产生报错（我的报错输出了对应的源代码位置，以及出错时所在的状态）。

```
ASTnode* analyze(Lexer &lexer) {

    vector<int> stack_state;
    vector<ASTnode*> stack_node;

    stack_state.push_back(0);

    ASTnode *curNode = lexer.getNextToken();

    while (1) {
        if (curNode == NULL) curNode = new ASTnode("$", "$");

        int top_sid = stack_state.back();
        string X = curNode->type;
```

```
        if (table.count({top_sid, X})) {
            const auto &[t, v] = table[{top_sid, X}];
            if (t == T_ACC) {       // 完成
                break;
            }
            else if (t == T_S) {    // 移入
                stack_state.push_back(v);
                codegen(curNode, -1);   // 生成代码
                stack_node.push_back(curNode);

                curNode = lexer.getNextToken(); // 读取下一个输入
            }
            else if (t == T_R) {    // 规约

                auto [L, R] = statements[v];

                // 先规约
                vector<ASTnode*> tmpvec;
                for (int i=R.size()-1; i>=0; i--) {
                    if (R[i] == "EMPTY") {
                        ASTnode *emptyNode = new ASTnode("EMPTY", "EMPTY");
                        codegen(emptyNode, -1);     // 为了 if、while 的回填
                        tmpvec.push_back(emptyNode);
                    }
                    else {
                        assert(stack_node.size());
                        tmpvec.push_back(stack_node.back());
                        stack_node.pop_back();
                        stack_state.pop_back();
                    }
                }
                reverse(tmpvec.begin(), tmpvec.end());
                auto *newNode = new ASTnode(L, L, tmpvec);
                codegen(newNode, v);   // 生成代码
                stack_node.push_back(newNode);

                // 再 GOTO
                assert(gotoMap.count({stack_state.back(), L}) > 0);
                stack_state.push_back(gotoMap[{stack_state.back(), L}]);
            }
            else {                  // 未知
                logError("???1", curNode->info.pos, top_sid, X);
            }
        }
```

```
        else {
            logError("???2", curNode->info.pos, top_sid, X);
        }

        for (auto x: stack_state) cout <<x <<" ";
        cout <<'\t';
        for (auto x: stack_node) cout <<x->type <<" ";
        cout <<endl;
        cout <<endl;
}
```

1.4.4 效果截图

日志文件中包含了 FIRST 、FOLLOW、CLOSURE、跳转表的信息，具体见附件。这是日志文件输出的分析某个源程序的部分过程中，状态栈以及符号栈的信息。

log.txt

```
[ 138 ]              not      r    <selectStmt>  ->  | if ( <simpleExp> ) <stmt> <else> <stmt>
[ 138 ]           return      r    <selectStmt>  ->  | if ( <simpleExp> ) <stmt> <else> <stmt>
[ 138 ]             true      r    <selectStmt>  ->  | if ( <simpleExp> ) <stmt> <else> <stmt>
[ 138 ]            while      r    <selectStmt>  ->  | if ( <simpleExp> ) <stmt> <else> <stmt>
[ 138 ]                {      r    <selectStmt>  ->  | if ( <simpleExp> ) <stmt> <else> <stmt>
[ 138 ]                }      r    <selectStmt>  ->  | if ( <simpleExp> ) <stmt> <else> <stmt>

================== analyze ==================
0 10  int

0 6   <typeSpec>

0 6 17  <typeSpec> ID

0 6 17 22   <typeSpec> ID (

0 6 17 22 10  <typeSpec> ID ( int

0 6 17 22 55  <typeSpec> ID ( <typeSpec>

0 6 17 22 55 99   <typeSpec> ID ( <typeSpec> ID

0 6 17 22 55 97   <typeSpec> ID ( <typeSpec> <parmId>

0 6 17 22 55 98   <typeSpec> ID ( <typeSpec> <parmIdList>

0 6 17 22 53  <typeSpec> ID ( <parmTypeList>

0 6 17 22 52  <typeSpec> ID ( <parmList>

0 6 17 22 54  <typeSpec> ID ( <parms>

0 6 17 22 54 96   <typeSpec> ID ( <parms> )

0 4   <funHead>

0 4 13  <funHead> {

0 4 13 18   <funHead> { <localDecls>
```

1.5 语义分析与中间代码生成

这是另一个重点。
语义分析做的就是检查符号声明、变量类型等。
中间代码生成做的是根据语法树，生成类似汇编逻辑的中间代码，我用的是四元式。

1.5.1 语法分析树节点类 ASTnode

这个数据结构是语义分析与中间代码生成的主要数据结构。一个 ASTnode 代表一个语法分析树节点，其中包含了该节点的树结构信息（子节点、父节点等）、文法符号、以及一些辅助语义分析的详细信息（如 entry、addr、回填时用到的 trueList、falseList 等）。

```
struct ASTinfo {
    std::pair<int, int> pos;

    VarType     type;
    int         addr;
    int         entry;

    std::vector<int>    trueList;
    std::vector<int>    falseList;
    std::vector<int>    nextList;

    int                         tmpInt;
    std::vector<int>            tmpIntList;
    std::string                 tmpStr;
    std::vector<std::string>    tmpStrList;
};


struct ASTnode {
    std::string type;
    std::string hint;

    ASTinfo info;

    ASTnode *father;
    std::vector<ASTnode*> sons;

    ASTnode(): father(NULL) {}
    ASTnode(std::string t, std::string h): type(t), hint(h), father(NULL) {}
    ASTnode(std::string t, std::string h, const std::vector<ASTnode*> &s)
      : type(t), hint(h), sons(s), father(NULL) {
        for (auto p: sons) p->father = this;
        if (sons.size()) info.pos = sons[0]->info.pos;
    }
```

```
    void destroy() {
        for (auto &p: sons) {
            p->destroy();
            p = NULL;
        }
    }

    friend void append_son(ASTnode *f, ASTnode *s) {
        f->sons.push_back(s);
        s->father = f;
    }
};
```

1.5.2 语义配置文件以及其规则

我的语义分析与中间代码生成是自动化的，相当于弄了一个小的解释器，从配置文件中读取语义规则，进行语义分析与代码生成。

语义规则配置文件是这个形式：

rules.in

```
126  <selectStmt>    ::=     if ( <simpleExp> ) <stmt>
127  |   .   backpatch       @ 3 trueList        @ 5 entry
128  |   @ 0 nextList    =       @ 3 falseList
129  |   @ 0 nextList    push    @ 5 nextList
130  $
131
132  <selectStmt>    ::=     if ( <simpleExp> ) <stmt> <else> <stmt>
133  |   .   backpatch       @ 3 trueList        @ 5 entry
134  |   .   backpatch       @ 3 falseList       @ 7 entry
135  |   @ 0 nextList    =       @ 5 nextList
136  |   @ 0 nextList    push    @ 6 nextList
137  |   @ 0 nextList    push    @ 7 nextList
138  $
139
140  <else>          ::=     else
141  |   @ 0 tmpInt      =       get_nxtcid
142  |   @ 0 nextList    push    @ 0 tmpInt
143  |   .   gen     @ 0 null    goto        @ 0 null        @ 0 null
144  $
145
146  <iterStmt>      ::=     while ( <simpleExp> ) <stmt>
147  |   .   backpatch       @ 5 nextList        @ 3 entry
148  |   .   backpatch       @ 3 trueList        @ 5 entry
149  |   @ 0 nextList    push    @ 3 falseList
150  |   .   gen     @ 3 entry   goto        @ 3 entry       @ 3 entry
151  $
```

具体地，针对每一个产生式进行描述，每一段描述由一个产生式开头（这里右部只能

有一条，不能有 | 分隔的多条）。接下来若干行（可以是零行）执行规则，每行由 | 开头，接下来是一条命令。最后一行一个符号 $，代表该段的结束。

每条命令有几种形式：

（1）<arg1> 双目运算 <arg2>

双目运算，有 = 、 push 两种。

arg1、arg2 可以是指定某个节点的某个属性，也可以是某个命令（比如 get_nxtcid）。

（2）. 带参过程 <args>

例如 gen <arg1> <arg2> <arg3> <arg4>，argx 可以是指定某个节点的某个属性，也可以是某个字符串（针对 gen 的 goto 这种）。

（3）无参过程

例如 update_type 等。

命令中，指定某个节点的某个属性，格式是 @ <num> <attribute>，其中 num 是一个数字表示目标节点在产生式右部从左往右的编号（从 1 开始），attribute 是一个字符串表示目标节点的属性。

这是目前支持的命令列表：

get_nxtcid ：下一条中间代码的编号

new_temp ：新建一个中间变量

function_pack ：打包目前的函数信息到 curFunction 对象中
function_push ：将当前的 curFunction 加入到函数表 glb_context 中

env_push ：符号表压栈
env_pop ：符号表弹出栈

update_type ：检查运算的类型合法性

alloc_vars <name> ：在符号表中声明一个变量，名字是 <name>
get_var <name> ：在当前符号环境中取名字是 <name> 的变量编号

gen <res> <op> <a1> <a2> ：加入一条中间代码
gen_call ：加入一组函数调用的中间代码

backpatch <list> <cid> ：回填，用 <cid> 填入 <list> 中的中间代码条目

1.5.3 符号表以及相关类

各个函数间的代码应该是相互独立的，变量符号也是相互独立的（除了全局变量）。我使用了一个 Function 类来封装一个函数，以及全局有个 glb_context 来存储全局信息（比如函数列表等）。Function 中有相应的对象与方法维护生成的中间代码以及符号表。

```
typedef std::string                    VarType;
typedef std::pair<int, VarType>        VarInfo;
typedef std::map<std::string, VarInfo>  NameMap;
typedef std::pair<VarType, std::string> ConstInfo;

class Function {
public:
    std::string            name;
    VarType                type;
    int cnt;
    int cnt_const;
    std::vector<VarInfo>    variables;
    std::vector<VarInfo>    temps;
    std::vector<NameMap>    envs;

    std::vector<Code>       codes;

    std::vector<VarInfo>    args;

    std::map<ConstInfo, int>        constMap;
    std::vector<ConstInfo>          consts;

public:
    Function(): cnt(0), cnt_const(0), envs(1), codes(1), consts(1) {}

    //=====================================================
    // 符号处理相关 （变量、临时变量的 id > 0）
    //=====================================================
    void env_push();
    void env_pop();

    int new_var(std::string varName, VarType varType);
    int new_temp();

    VarInfo* get_var(std::string varName);
    VarInfo* get_var(int index);
    VarType  get_type(int index);   // >0 var, <0 const
    bool is_var(int index) {for (auto &[id, t]: variables) if (id==index) return
true; return false;}

    //=====================================================
    // 常量处理相关 （常量的 id < 0）
    //=====================================================
    int get_const(const ConstInfo &x);
```

```
    //==================================================
    // 代码处理相关
    //==================================================
    int next_cid() {return codes.size();}
    int new_code(int res, std::string op, int v1, int v2);

    Code& get_code(int index);

};
```

1.5.4 回填

由于是从下往上一遍扫描，所以跳转逻辑中需要使用回填。

书上的回填，利用了空串符号，在扫描到空串符号时存下起始地址。但是我这个语法分析用到 LR0，加上空串符号以后会出现冲突，于是我为每个节点增添了一个 entry 属性，让终结符的 entry 属性为分析到它时的 next_cid，非终结符的 entry 属性为其第一个子节点的 next_cid，这样就不用添加空串符号也能获取代码片段的首地址了。这是因为，每个节点对应的代码，都是连续的一段代码。

下面是相关语义规则（if else、while 以及布尔表达式的 trueList、falseList、nextList）

```
126  <selectStmt>    ::=     if ( <simpleExp> ) <stmt>
127  |   .   backpatch       @ 3 trueList        @ 5 entry
128  |   @ 0 nextList    =       @ 3 falseList
129  |   @ 0 nextList    push    @ 5 nextList
130  $
131
132  <selectStmt>    ::=     if ( <simpleExp> ) <stmt> <else> <stmt>
133  |   .   backpatch       @ 3 trueList        @ 5 entry
134  |   .   backpatch       @ 3 falseList       @ 7 entry
135  |   @ 0 nextList    =       @ 5 nextList
136  |   @ 0 nextList    push    @ 6 nextList
137  |   @ 0 nextList    push    @ 7 nextList
138  $
139
140  <else>          ::=     else
141  |   @ 0 tmpInt      =       get_nxtcid
142  |   @ 0 nextList    push    @ 0 tmpInt
143  |   .   gen     @ 0 null    goto        @ 0 null        @ 0 null
144  $
145
146  <iterStmt>      ::=     while ( <simpleExp> ) <stmt>
147  |   .   backpatch       @ 5 nextList        @ 3 entry
148  |   .   backpatch       @ 3 trueList        @ 5 entry
149  |   @ 0 nextList    push    @ 3 falseList
150  |   .   gen     @ 3 entry   goto        @ 3 entry       @ 3 entry
151  $
```

```
<simpleExp>     ::=     <simpleExp> or <andExp>
|   update_type
|   .   backpatch       @ 1 falseList       @ 3 entry
|   @ 0 trueList    =       @ 1 trueList
|   @ 0 trueList    push    @ 3 trueList
|   @ 0 falseList   =       @ 3 falseList
$

<simpleExp>     ::=     <andExp>
|   @ 0 trueList    =       @ 1 trueList
|   @ 0 falseList   =       @ 1 falseList
|   @ 0 type        =       @ 1 type
|   @ 0 addr        =       @ 1 addr
$

<andExp>        ::=     <andExp> and <unaryRelExp>
|   update_type
|   .   backpatch       @ 1 trueList        @ 3 entry
|   @ 0 trueList    =       @ 3 trueList
|   @ 0 falseList   =       @ 1 falseList
|   @ 0 falseList   push    @ 3 falseList
$

<andExp>        ::=     <unaryRelExp>
|   @ 0 type        =       @ 1 type
|   @ 0 addr        =       @ 1 addr
|   @ 0 trueList    =       @ 1 trueList
|   @ 0 falseList   =       @ 1 falseList
$

<unaryRelExp>   ::=     not <unaryRelExp>
|   update_type
|   @ 0 trueList    =       @ 2 falseList
|   @ 0 falseList   =       @ 2 trueList
$
```

1.5.5 常量复用

Function 类中，我还设计了个常量索引表，将用到的常量统一处理，可以一定程度加大效率。

1.5.6 效果截图

log.txt 里面会有生成的语法树信息输出（不过有点狭长）。
还有生成的中间代码。
如下三图，分别是语法树结构、一段源代码以及其对应的中间代码。
源代码是一个求斐波那契数列某一项的递归函数。

testcode4.in

```
1   int fib(int x) {
2       int ret;
3       ret;
4
5       if (x < 3)
6           ret = 1;
7       else
8           ret = fib(x-2) + fib(x-1);
9
10      return ret;
11  }
```

中间代码中，%负数 代表常量。可以看到代码中用到了多次 1，四元式中对所有的 1 用的是同一个常量标号。

```
5202  ================== code ====================
5203  define int @fib(int, )
5204  [    ]  alloc %1 arg1
5205  [    ]  alloc %2
5206  [   0]
5207  [   1]  %3 = load %2
5208  [   2]  %4 = load %1
5209  [   3]  %5 = < %4 %-1
5210  [   4]  br %5 6
5211  [   5]  goto 8
5212  [   6]  store %-2 %2
5213  [   7]  goto 18
5214  [   8]  %6 = load %1
5215  [   9]  %7 = - %6 %-3
5216  [  10]  8 = call fib 1
5217  [  11]    arg0 %7
5218  [  12]  %9 = load %1
5219  [  13]  %10 = - %9 %-2
5220  [  14]  11 = call fib 1
5221  [  15]    arg0 %10
5222  [  16]  %12 = + %8 %11
5223  [  17]  store %12 %2
5224  [  18]  %13 = load %2
5225  [  19]  ret %13
5226
```

1.6 目标代码生成

这次的目标代码，我选择的是 LLVM 的 IR (Intermediate Representation)。它的好处是跨平台，而且不用抠太多细节（）。

LLVM IR 的规则比较简单，是个强类型的指令式语言（和普通的汇编指令基本上能一一对应，不过就是不用自己去选择寄存器）。它由若干指令块构成，每个指令块由 br（跳转） 或 ret（返回） 命令结束。

由前面的中间代码生成 LLVM IR 目标代码，若不考虑优化，要做的就是根据跳转语句，处理好出一个个指令块，同时可以得到一个程序流图，然后设置好标签，再逐条语句翻译成 LLVM IR 指令即可。

注意，函数调用的指令，在前面生成的四元式中，我用的是类似运行栈的方式将参数分多条语句加入，而在 LLVM IR 中是独立的一条语句 call type @funcName(args)，所以函数调用语句还得额外处理一下。

下图是前面斐波那契梳理递归函数生成的目标代码。

```
define i32 @fib(i32 %arg0) {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %arg0, i32* %1, align 4
  %3 = load i32, i32* %2, align 4
  %4 = load i32, i32* %1, align 4
  %5 = icmp slt i32 %4, 3
  br i1 %5, label %l0, label %l1

l0:
  store i32 1, i32* %2, align 4
  br label %l2

l1:
  %6 = load i32, i32* %1, align 4
  %7 = sub nsw i32 %6, 2
  %8 = call i32 @fib(i32 %7)
  %9 = load i32, i32* %1, align 4
  %10 = sub nsw i32 %9, 1
  %11 = call i32 @fib(i32 %10)
  %12 = add nsw i32 %8, %11
  store i32 %12, i32* %2, align 4
  br label %l2

l2:
  %13 = load i32, i32* %2, align 4
  ret i32 %13
  ret i32 0
}

```

1.7 最终效果展示

1.7.1 测试程序源代码

设计了一个根据 main 函数参数个数计算对应斐波那契数项的程序。

用到了 if else 语句，while 语句，还有函数调用（自我递归与调用其它函数）。

最后可以根据运行时间来看正确性。（因为没有做字符串类型的语义分析，所以没有办法输出到屏幕，但是可以用特定数量的循环，根据程序运行时间来获取变量信息）

```
int fib(int x) {
    int ret;
    ret;
    if (x < 3)
        ret = 1;
    else
        ret = fib(x-2) + fib(x-1);
    return ret;
}

int sleep_one_second() {
    int num_per_sec, i;
    num_per_sec = 430000000;
    i = 0;
    while (i<num_per_sec) {
        i = i+1;
    }
    return 0;
}

int main(int argc)
{
    int t_sec, cur_cnt;
    int tmp;

    t_sec = fib(argc);

    while (t_sec > 0) {
        sleep_one_second();
        t_sec = t_sec - 1;
    }

    return 0;
}
```

1.7.2 中间代码

执行命令脚本 make run，在 log.txt 中查看中间代码
如图，在系统的 stderr 中，我还输出了回填信息。

```
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ make clean
rm -rf ./a.out code.ll code.s
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ make
g++ src/auto.cpp ./common/AST.cpp ./common/UTILS.cpp ./common/LEXER.cpp ./common/CODEGEN.cpp -I ./include -std=gnu++17
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ make run
./a.out testcode4.in rules.in code.ll <grammar.in >log.txt
!!! 134 else  r    <selectStmt>  ->  | if ( <simpleExp> ) <stmt>  ( 1, 137)
fill 6 in [4 ]
fill 8 in [5 ]
fill 18 in [7 ]
fill 8 in [6 ]
fill 12 in [7 ]
fill 9 in [7 ]
fill 14 in [8 ]
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$
```

```
define int @fib(int, )
[      ]  alloc %1 arg1
[      ]  alloc %2
[    0]
[    1]  %3 = load %2
[    2]  %4 = load %1
[    3]  %5 = < %4 %-1
[    4]  br %5 6
[    5]  goto 8
[    6]  store %-2 %2
[    7]  goto 18
[    8]  %6 = load %1
[    9]  %7 = - %6 %-3
[   10]  8 = call fib 1
[   11]     arg0 %7
[   12]  %9 = load %1
[   13]  %10 = - %9 %-2
[   14]  11 = call fib 1
[   15]     arg0 %10
[   16]  %12 = + %8 %11
[   17]  store %12 %2
[   18]  %13 = load %2
[   19]  ret %13

define int @sleep_one_second()
[      ]  alloc %1
[      ]  alloc %2
[    0]
[    1]  store %-1 %1
[    2]  store %-2 %2
[    3]  %3 = load %2
[    4]  %4 = load %1
[    5]  %5 = < %3 %4
[    6]  br %5 8
[    7]  goto 12
[    8]  %6 = load %2
[    9]  %7 = + %6 %-3
[   10]  store %7 %2
[   11]  goto 3
[   12]  ret %-2

define int @main(int, )
[      ]  alloc %1 arg1
[      ]  alloc %2
[      ]  alloc %3
[      ]  alloc %4
[    0]
[    1]  %5 = load %1
[    2]  6 = call fib 1
[    3]     arg0 %5
[    4]  store %6 %2
[    5]  %7 = load %2
[    6]  %8 = > %7 %-1
[    7]  br %8 9
[    8]  goto 14
[    9]  9 = call sleep_one_second 0
[   10]  %10 = load %2
[   11]  %11 = - %10 %-2
[   12]  store %11 %2
[   13]  goto 5
[   14]  ret %-1
```

1.7.3 目标代码

在上面 make run 的命令中，同时还生成了目标代码 code.ll。

```
define i32 @fib(i32 %arg0) {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %arg0, i32* %1, align 4
  %3 = load i32, i32* %2, align 4
  %4 = load i32, i32* %1, align 4
  %5 = icmp slt i32 %4, 3
  br i1 %5, label %l0, label %l1

l0:
  store i32 1, i32* %2, align 4
  br label %l2

l1:
  %6 = load i32, i32* %1, align 4
  %7 = sub nsw i32 %6, 2
  %8 = call i32 @fib(i32 %7)
  %9 = load i32, i32* %1, align 4
  %10 = sub nsw i32 %9, 1
  %11 = call i32 @fib(i32 %10)
  %12 = add nsw i32 %8, %11
  store i32 %12, i32* %2, align 4
  br label %l2

l2:
  %13 = load i32, i32* %2, align 4
  ret i32 %13
  ret i32 0
}

define i32 @sleep_one_second() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 430000000, i32* %1, align 4
  store i32 0, i32* %2, align 4
  br label %l0

l0:
  %3 = load i32, i32* %2, align 4
  %4 = load i32, i32* %1, align 4
  %5 = icmp slt i32 %3, %4
  br i1 %5, label %l1, label %l2

l1:
  %6 = load i32, i32* %2, align 4
  %7 = add nsw i32 %6, 1
  store i32 %7, i32* %2, align 4
  br label %l0

l2:
  ret i32 0
  ret i32 0
}

define i32 @main(i32 %arg0) {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %arg0, i32* %1, align 4
  %5 = load i32, i32* %1, align 4
  %6 = call i32 @fib(i32 %5)
  store i32 %6, i32* %2, align 4
  br label %l0

l0:
  %7 = load i32, i32* %2, align 4
  %8 = icmp sgt i32 %7, 0
  br i1 %8, label %l1, label %l2

l1:
  %9 = call i32 @sleep_one_second()
  %10 = load i32, i32* %2, align 4
  %11 = sub nsw i32 %10, 1
  store i32 %11, i32* %2, align 4
  br label %l0

l2:
  ret i32 0
  ret i32 0
}
```

1.7.4 可执行文件与执行结果

可以直接用 clang code.ll 来生成可执行文件，也可以先用 llc 生成 gcc 接受的汇编代码 code.s，再用 gcc 汇编生成可执行文件。

```
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ llc code.ll
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ gcc code.s -o code
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$
```

执行程序。这个程序做的事，是输出 fib[argc]，其中 fib[1] = fib[2] = 1，argc 是命令中的参数个数。

如下图，空转 cpu 还是挺稳定的（）。

```
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ time ./code

real    0m1.024s
user    0m1.020s
sys     0m0.000s
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ time ./code 2

real    0m1.026s
user    0m1.024s
sys     0m0.000s
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ time ./code 2 3

real    0m2.050s
user    0m2.047s
sys     0m0.000s
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ time ./code 2 3 4

real    0m3.074s
user    0m3.071s
sys     0m0.000s
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ time ./code 2 3 4 5

real    0m5.104s
user    0m5.101s
sys     0m0.000s
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ time ./code 2 3 4 5 6

real    0m8.178s
user    0m8.175s
sys     0m0.000s
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ time ./code 2 3 4 5 6 7

real    0m13.300s
user    0m13.297s
sys     0m0.000s
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$ time ./code 2 3 4 5 6 7 8

real    0m21.320s
user    0m21.314s
sys     0m0.001s
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/compiler$
```

2　一个解释性的符号计算器

我对 Mathematica 挺感兴趣，以前还写过一个类似的符号计算器。

我做的这个符号计算器，是 Mathematica 的一个很小的子集，可以对加减乘除幂进行化简，还可以编写函数。

这个也是只使用了 c++ 标准库，代码与可执行文件在附件 mymma.zip 中。

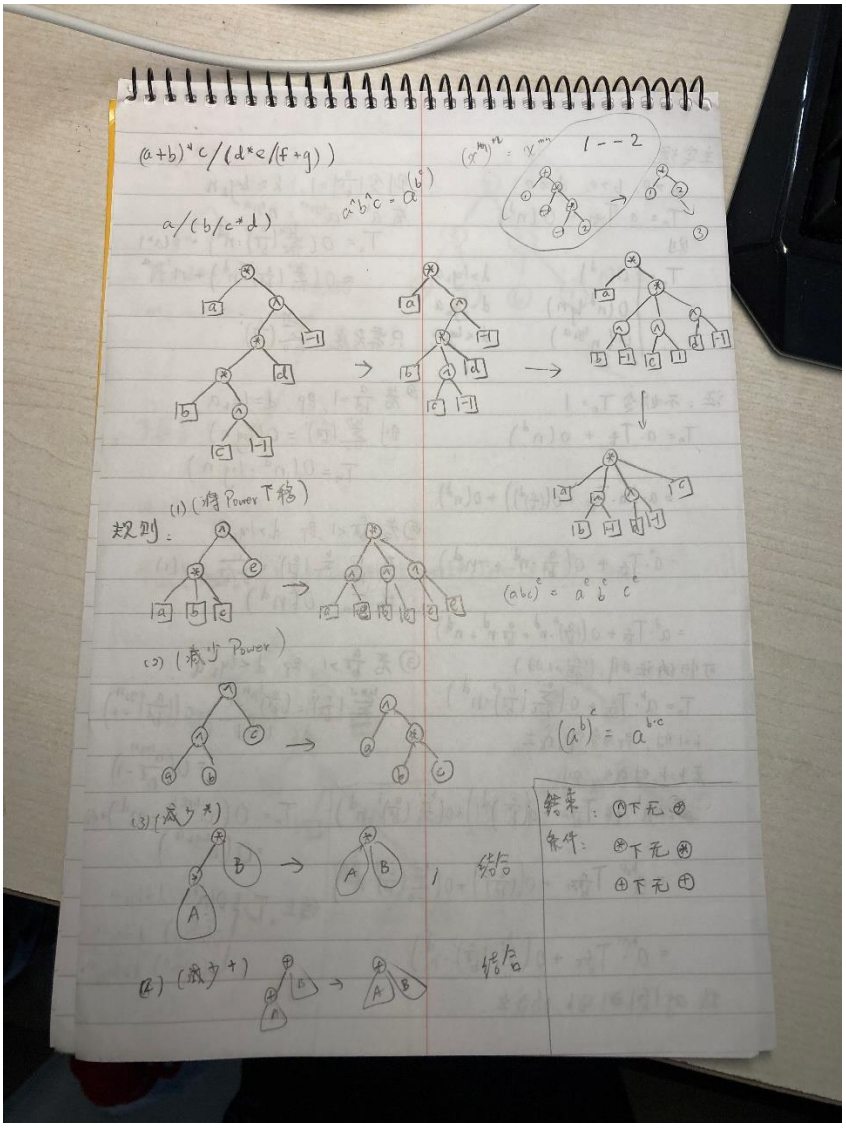
2.1 原理与日志

它的特点是符号计算，而不是数值计算。也就是，它是模仿人的行为根据某些规则对表达式进行恒等变换。也就是，很大一部分工作做的是匹配。

除此之外，它的类型思想也很有意思，所有表达式都是“符号”，并没有具体的类型，所以写起来会十分灵活，对数学推理十分友好。具体的实现上，我是将每个表达式用语法树来表示，计算全都是对语法树的操作。Mathematica 内核是如何实现的我就无从知道了。

当时写的过程中我有记录日志，具体的可以去看看附件中的日志。

这里贴一下化简加减乘除幂的思想笔记：



2.2 使用说明

这是一个解释性的程序，运行程序后，直接输入指令即可。

输入 Help[] 查看可使用的指令，具体含义可以看我写的日志，也可以直接去看 Mathematica 的语法文档。

2.3 效果展示

这个则展示了化简的功能。可以看到，他将相同底数 a 合并了。

```
> a^(a+1)*(1/b*2a)

Out = Times[2, Power[a, Plus[a, 2]], Power[b, -1]]
```

这展示了内部的排序函数，还定义了一个函数 fib，计算斐波那契数列。

```
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/mymma_demo$ ./demo.exe

> 魔法 = {1,10817, 1926, 114, 514, fun[{1,2}], 3}

Out = List[1, 10817, 1926, 114, 514, fun[List[1, 2]], 3]


> Sort[魔法]

Out = List[1, 3, 114, 514, 1926, 10817, fun[List[1, 2]]]


> 斐波那契数列 = fib := Function[{n},
            If[n<3, 1, fib[n-1]+fib[n-2]]
        ]

Out = Function[List[n], If[Less[n, 3], 1, Plus[fib[Subtract[n, 1]],
fib[Subtract[n, 2]]]]]


> fib[10]

Out = 55


> 斐波那契数列[9]

Out = 34


> fib[8]

Out = 21


> Exit

Out = Exit
```

这是展示了定义的因式分解函数

最下面三条，则展示了符号计算的特性：定义了 fx = x^2+2x+1，而没指明 x，之后再计算 {x=1, fx}，这是顺序列表，单独存在一个符号表，先定义了 x=1，然后计算 fx，用 1 替代 x，计算得到结果{1, 4}。

```
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/mymma_demo$ ./demo.exe

> getm := Function[ {n, k},
           If[Mod[n, k] == 0,
               1+getm[Quotient[n, k], k],
           0]
       ]

Out = Function[List[n, k], If[Equal[Mod[n, k], 0], Plus[1, getm[Quotient[n, k], k]], 0]
]


> fac := Function[ {n, k},
           If[ n<2, {},
               If[ n==k, g[k, 1],
                   If[ (tmp=getm[n, k]) == 0,
                       fac[Quotient[n, k^tmp], k+1],
                       Flatten[{g[k, tmp], fac[Quotient[n, k^tmp], k+1]}]
                   ]
               ]
           ]
       ]

Out = Function[List[n, k], If[Less[n, 2], List[], If[Equal[n, k], g[k, 1], If[Equal[Set
[tmp, getm[n, k]], 0], fac[Quotient[n, Power[k, tmp]], Plus[k, 1]], Flatten[List[g[k, t
mp], fac[Quotient[n, Power[k, tmp]], Plus[k, 1]]]]]]]]


> 因式分解 = Function[{n}, fac[n, 2]]

Out = Function[List[n], fac[n, 2]]


> 因式分解[10086]

Out = List[g[2, 1], g[3, 1], g[41, 2]]


> fx = x^2 + 2x + 1

Out = Plus[1, Power[x, 2], Times[x, 2]]


> {x=1, fx}

Out = List[1, 4]


> {x=-1, fx}

Out = List[-1, 0]


>
```

这则也是符号计算的特性展示。

powerN 定义为了一个函数，powerN[n] 代表一个函数 x^n。

powerN[3][5]，就是 (#^3&)[5]，即 5^3，125。

powerN[a][b]，也是合法的，(#^a&)[b]也就是 b^a。

```
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/mymma_demo$ make
g++ -Wall -fsanitize=address -static-libasan -std=c++17 -o demo.exe ./tests/test_
Driver.cpp ./common/Integer.cpp ./common/ASTnode.cpp ./common/Functions.cpp ./com
mon/Variable.cpp ./common/Tools.cpp
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/mymma_demo$
hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/mymma_demo$ ./demo.exe

> powerN = Function[{n}, Function[{x}, x^n]]

Out = Function[List[n], Function[List[x], Power[x, n]]]


> powerN[3][5]

Out = 125


> powerN[a][b]

Out = Power[b, a]


> Exit

Out = Exit

hehepig@DESKTOP-LD69MD6:/mnt/d/workspace/mymma_demo$
```

参 考 文 献

[1] Compilers: Principles, Techniques, and Tools, Alfred V.Aho, Ravi Sethi, Jeffrey D.Ullman
（这本书太全了，基本上这本就够了）
[2] https://llvm.org/
（llvm 官网，里面有许多文档，和项目实验）
[3] https://reference.wolfram.com/language/guide/Syntax.html
（Wolfram Mathematica 的一些文档，是报告第二部分参考的资料）