自定义语言编译器前端

词法分析

语法分析

语法制导翻译

中间代码生成

设计文法

文法规则代表着一个语言的标准，使用者遵循该规则编写程序代码，编译器根据规则解析用户程序，进而翻译成可执行文件

一些基本的文法有

变量的声明定义

控制流语句 if else for while swtch...

函数的声明定义

目前阶段仅仅对变量的声明与定义，if else，while 提供了支持。另外不需要显示指出变量类型，在解析的过程中会自动进行类型推导

文法的定义如下：

program -> stmts ;

var_decls -> var_decls var_decl | @ ; 变量的声明和定义 如 a = 10

var_decl -> ident | ident = expr ;

expr -> ident = expr | numeric | ident | expr operation expr ; 表达式，包括赋值，比较等

operation -> + | - | * | / ; 二元运算符

comparsion -> > | < | == | >= | <= | != ; 二元比较符
numeric -> INT | REAL | STRING | CHAR ; 类型，用于推导使用，如 10 会被规约成 INT, "hello"会被规约成 STRING
block -> { stmts } | { } ; 语句块，用在 if, else 和 while 后面
stmts -> stmt | stmts stmt ;
stmt -> var_decl | expr | if_stmt | logical_stmt | while_stmt ; 语句

M -> @ ;
N -> @ ;

logical_stmt -> logical_stmt lop M logical_stmt | ! logical_stmt | expr comparsion expr | (logical_stmt) ; 逻辑表达式
lop -> && | || ;

if_stmt -> if (logical_stmt) M block | if (logical_stmt) M block N else M block ; 条件表达式
while_stmt -> while M (logical_stmt) M block ; 循环表达式

词法分析
词法分析阶段主要用于将程序代码解析成一个个单词，为后续流程提供 token。
比如 a = 10 会被解析成 a, =, 10 这三个单词。在解析的同时也需要根据读到的单词判断它的类型，比如
对于关键字（如 if, while, else 等）保留其符号，token 中存储的就是关键字
对于变量（如上述的 a），需要将其转换成文法中表示变量的符号（ident），变量名。同时将该变量添加到符号表中
对于分界符（如, ;, (,), {, }等），同样保留符号
对于常量字面值（如 10， 2.5，“hello”等），将其转换成文法中表示常量的符号（INT, DOUBLE, STRING），变量名。

词法分析的程序采用一个字符一个字符读取的方式实现（状态机），每读到一个字符，根据文法中定义的规则判断是否继续读取以构成完成的符号，还是重新读取。比如

对于变量的定义，方法是字母数字下划线，所以解析时也是按照这种方式解析

对于关键字和变量的解析，首先判断读到的字符是否是字母，如果是，则执行
继续读取字母或数字或下划线，读完之后组装成一个 token

```
char ch = fin.get();
if(is_letter(ch)) {
    token.append(1, ch);
    ch = fin.get();
    while(is_letter(ch) || is_digit(ch) || ch == '_') {
        token.append(1, ch);
        ch = fin.get();
    }
    fin.unget();
    if(auto code = query_keyword(token); code != ID::UNKNOWN) {
        tokens_.emplace_back(0, token, token);
    }
    else {
        std::size_t i { 0 };
        for(; i != symbol_table_.size(); ++i) {
            if(symbol_table_[i].name == token) {
                break;
            }
        }
        if(i != symbol_table_.size()) {
            tokens_.emplace_back(i, "ident", token);
        }
        else {
            symbol_table_.emplace_back("ident", token);
            tokens_.emplace_back(symbol_table_.size() - 1, "ident", token);
        }
    }
    token.clear();
}
```

对于字面值数字常量，又分为整型数和浮点数，所以需要判断是否存在小数点

```
else if(is_digit(ch)) {
    token.append(1, ch);
    ch = fin.get();
    bool is_real = false;
    while(is_digit(ch) || ch == '.' || ch == 'e') {
        if(ch == '.') {
            is_real = true;
        }
        token.append(1, ch);
        ch = fin.get();
    }
    fin.unget();
    if(is_real) {
        tokens_.emplace_back(0, "REAL", token);
    }
    else {
        tokens_.emplace_back(0, "INT", token);
    }
    token.clear();
}
```

对于其它符号的解析方法都相同，不再一一列举

语法分析

语法分析的目标是判断程序代码是否符合语法规则，实现的方法有自顶向下和自底向上两种。

对于自顶向下常用的有 LL 分析，即递归下降分析法

对于自底向上常用的有 LR 分析，即移进和规约

实验中采用自底向上分析中的 LR(1)分析法实现语法分析。

FIRST 集

在 LR(1)分析法中，首先需要求出每个非终结符的 FIRST 集，实现方法如下（大写字母代表非终结符，小写字母代表终结符，特殊字符代码由若干终结符或非终结符组成的串）

如果 A -> Bα，那么需要将 FIRST(B)中的所有元素添加到 FIRST(A)中

如果 A -> aβ，那么需要将 a 添加到 FIRST(A)中

如果 A -> BCθ，且 FIRST(B)中包含空字符，那么同时也需要将 FIRST(C)中的所有元素添加到 FIRST(A)中，一直进行下去直到不再出现空字符

实现方案就是按照上述三步不断进行，直到所有非终结符的 FIRST 集不再更新

```
/* 计算每个非终结符的First集，规则如下
 * 如果非终结符A可以推出终结符，那么将终结符添加到A的First集中
 *  如A -> (* | + | @, 则将(,*,@三个终结符添加到A的First集中
 * 如果非终结符A可以推出其它非终结符，那么将推出的非终结符的First集添加到A的First集中
 *  如A -> BC, 则将B的First集添加到A的First集中
 * 如果A推出的非终结符的First集中包含空字符@，则接着添加
 *  如A -> BCD, B的First集中包含空字符@，则将B的First集添加后还需要继续添加C的First集，持续向后直到不再出现@
 */
void anaysis_first_set() {
    auto grammer = grammer_;
    std::unordered_set<std::string> done_set;
    while(done_set.size() != grammer.size()) {
        for(auto& [code, code_lists] : grammer) {
            bool done = true;
            /* 对于每个code和它对应的文法链表，判断后面的内容是否可以添加到First集中
             * 规则是不存在一个目前还没有找到First集的code */
            for(auto& code_list : code_lists) {
                if(code_list.empty()) {
                    continue;
                }
                if(grammer.count(code_list.front()) && !done_set.count(code_list.front())) {
                    if(code_list.front() != code) {
                        done = false;
                        break;
                    }
                }
            }
```

```
            /* code下的每个链表的第一个字符
             * 或者可以添加到code的First集中
             * 或者它的First集可以添加到code的First集中 */
            if(done) {
                for(auto it = code_lists.begin(); it != code_lists.end();) {
                    if(it->empty()) {
                        code_lists.erase(it++);
                    }
                    /* 非终结符，直接添加到code的First集中 */
                    else if(!grammer.count(it->front())) {
                        first_set_[code].emplace_back(std::move(it->front()));
                        /* 删除该链表 */
                        code_lists.erase(it++);
                    }
                    /* 已经找到First集的字符，将它的First集添加到code的First集中 */
                    else if(it->front() != code && done_set.count(it->front())) {
                        bool has_empty = false;
                        /* 判断是否存在空字符，
                         * 如果存在，说明需要考虑它后面的字符，此时不能删除这个链表，而只删除第一个字符即可
                         * 如果不存在，直接删除链表 */
                        for(auto& s : first_set_[it->front()]) {
                            if(s == "@")
                                has_empty = true;
                            first_set_[code].emplace_back(s);
                        }
                        if(has_empty) {
                            it->pop_front();
                            ++it;
                        }
                        else {
                            code_lists.erase(it++);
                        }
                    }
                    else {
                        code_lists.erase(it++);
                    }
                }
```

```
                /* 如果code下已经没有链表存在，说明已经找到code的First集，记录在done_set中 */
                if(code_lists.empty()) {
                    /* 排序去重 */
                    first_set_[code].sort();
                    first_set_[code].unique();
                    done_set.emplace(code);
                }
            }
        }
    }
```

对于开篇给出的文法，每个非终结符的 FIRST 集如下（其中第一行是非终结符，第二行是它的 FIRST 集）

```
numeric
CHAR INT REAL STRING

expr
CHAR INT REAL STRING ident

program0
! ( CHAR INT REAL STRING ident if while

var_decls
@

var_decl
ident

comparsion
!= < <= == > >=

operation
* + - /

program
! ( CHAR INT REAL STRING ident if while

N
@

stmt
! ( CHAR INT REAL STRING ident if while

logical_stmt
! ( CHAR INT REAL STRING ident
```

```
if_stmt
if

lop
&& ||

M
@

block
{

while_stmt
while

stmts
! ( CHAR INT REAL STRING ident if while
```

状态转移图

LR(1)分析法的第二步是构造状态转移图，实际上 LR 将整个分析的过程看做一个个状态，当状态转移图构造完成后，只需要判断当前状态下遇到某个单词时应该转换到的下一个状态即可，这个过程称为移进

当移进到某些状态后无法继续移进时，会进行适当的返回操作，即将之前读到的某几个单词合并成一个新的单词，这个过程称为规约

另外为了解决 LR 中的移进规约冲突，LR(1)采用向前看一个单词的方法解决

构造状态转移图的步骤如下

构造拓广文法

构造一个个状态闭包 closure

根据闭包之间的生成情况构造 goto 表

根据状态转移图构造 action 表

```
SetOfItems CLOSURE(I) {
    repeat
        for ( I中的每个项 [A → α·Bβ, a] )
            for ( G'中的每个产生式 B → γ )
                for ( FIRST(βa)中的每个终结符号 b )
                    将 [B → ·γ, b] 加入到集合I中;
    until 不能向I中加入更多的项;
    return I;
}

SetOfItems GOTO(I, X) {
    将J初始化为空集;
    for ( I 中的每个项 [A → α·Xβ, a])
        将项 [A → αX·β, a]加入到集合J中;
    return CLOSURE(J);
}

void items(G') {
    将 C 初始化为{CLOSURE}({[S' → ·S, $]});
    repeat
        for ( C 中的每个项集 I )
            for (每个文法符号 X )
                if ( GOTO(I, X) 非空且不在C中)
                    将 GOTO(I, X) 加入 C中;
    until 不再有新的项集加入到C中;
}
```

图 4-40　为文法 G'构造 LR(1)项集族的算法

分别对应下面三个函数

```cpp
void anaysis_closure(std::vector<Item>& item_set) {
    while(true) {
        bool done = true;
        for(std::size_t i = 0; i != item_set.size(); ++i) {
            if(item_set[i].dot >= item_set[i].prod_vec.size()) {
                continue;
            }
            std::string s;
            for(std::size_t j = item_set[i].dot + 1; j < item_set[i].prod_vec.size(); ++j) {
                s.append(item_set[i].prod_vec[j]);
                s.append(1, ' ');
            }
            s.append(item_set[i].suffix_code);
            auto fs = query_first_set(s);
            if(fs.empty()) {
                continue;
            }
            if(!grammer_.count(item_set[i].prod_vec[item_set[i].dot])) {
                continue;
            }
            for(auto& code_list : grammer_[item_set[i].prod_vec[item_set[i].dot]]) {
                for(auto& symbol : fs) {
                    if(grammer_.count(symbol)) {
                        continue;
                    }
                    if(symbol.empty()) {
                        continue;
                    }
                    Item new_item(0, item_set[i].prod_vec[item_set[i].dot], symbol);
                    new_item.prod_vec.reserve(code_list.size());
                    new_item.prod_vec.insert(new_item.prod_vec.end(), code_list.begin(), code_list.end());
                    if(new_item.prod_vec[0] == "@") {
                        ++new_item.dot;
                    }
                    if(std::find(item_set.begin(),
                                 item_set.end(),
                                 new_item) != item_set.end()) {
                        continue;
                    }
                    item_set.emplace_back(std::move(new_item));
                    done = false;
                }
            }
        }
        if(done) {
            break;
        }
```

```cpp
void anaysis_goto(std::size_t closure_id, const std::string& suffix) {
    std::vector<Item> item_set;
    for(auto& item : closure_[closure_id]) {
        auto& prod = item.prod_vec;
        if(item.dot != prod.size() && prod[item.dot] == suffix) {
            Item new_item { item.dot + 1, item.prefix_code, item.suffix_code };
            new_item.prod_vec = prod;
            if(std::find(item_set.begin(),
                         item_set.end(),
                         new_item) != item_set.end()) {
                continue;
            }
            item_set.emplace_back(std::move(new_item));
        }
    }

    anaysis_closure(item_set);
    if(!item_set.empty()) {
        if(auto it = std::find(closure_.begin(), closure_.end(), item_set);
                it == closure_.end()) {
            closure_.emplace_back(std::move(item_set));
            goto_[closure_id][suffix] = closure_.size() - 1;
        }
        else {
            goto_[closure_id][suffix] = it - closure_.begin();
        }
    }
}
```

```cpp
void anaysis_items() {
    grammer_[start_code_ + "0"].emplace_back(std::list{ start_code_ });
    anaysis_first_set();
    check_first_set();

    std::unordered_set<std::string> symbol_set;
    for(auto& [code, code_lists] : grammer_) {
        symbol_set.insert(code);
        for(auto& code_list : code_lists) {
            for(auto& s : code_list) {
                symbol_set.insert(s);
            }
        }
    }
    std::vector<std::string> symbols;
    symbols.reserve(symbol_set.size());
    for(auto& symbol : symbol_set) {
        symbols.emplace_back(symbol);
    }

    std::cout << "items start............." << std::endl;
    std::vector<Item> item_set;
    item_set.emplace_back( 0, start_code_ + "0", "$", std::vector{ start_code_ }) ;
    anaysis_closure(item_set);
    closure_.emplace_back(std::move(item_set));

    for(std::size_t i = 0; i != closure_.size(); ++i) {
        for(auto& symbol : symbols) {
            if(symbol == "@") {
                continue;
            }
            anaysis_goto(i, symbol);
        }
    }
```

将结果输出，可以看到每个状态下的产生式（仅显示部分状态）

```
165
var_decl -> ident = . expr , }
expr -> ident = . expr , }
expr -> ident = . expr , /
expr -> ident = . expr , -
expr -> ident = . expr , *
expr -> ident = . expr , +
expr -> ident = . expr , >=
expr -> ident = . expr , ==
expr -> ident = . expr , >
expr -> ident = . expr , <=
expr -> ident = . expr , !=
expr -> ident = . expr , <
var_decl -> ident = . expr , while
expr -> ident = . expr , while
var_decl -> ident = . expr , if
```

```
203
logical_stmt -> logical_stmt lop . M logical_stmt , }
logical_stmt -> logical_stmt lop . M logical_stmt , while
logical_stmt -> logical_stmt lop . M logical_stmt , if
logical_stmt -> logical_stmt lop . M logical_stmt , (
logical_stmt -> logical_stmt lop . M logical_stmt , INT
logical_stmt -> logical_stmt lop . M logical_stmt , !
logical_stmt -> logical_stmt lop . M logical_stmt , STRING
logical_stmt -> logical_stmt lop . M logical_stmt , CHAR
logical_stmt -> logical_stmt lop . M logical_stmt , ident
logical_stmt -> logical_stmt lop . M logical_stmt , REAL
logical_stmt -> logical_stmt lop . M logical_stmt , ||
logical_stmt -> logical_stmt lop . M logical_stmt , &&
M -> @ . , (
M -> @ . , INT
M -> @ . , !
M -> @ . , STRING
M -> @ . , CHAR
M -> @ . , ident
M -> @ . , REAL
```

```
245
while_stmt -> while M ( logical_stmt ) M block . , }
while_stmt -> while M ( logical_stmt ) M block . , while
while_stmt -> while M ( logical_stmt ) M block . , if
while_stmt -> while M ( logical_stmt ) M block . , (
while_stmt -> while M ( logical_stmt ) M block . , INT
while_stmt -> while M ( logical_stmt ) M block . , !
while_stmt -> while M ( logical_stmt ) M block . , STRING
while_stmt -> while M ( logical_stmt ) M block . , CHAR
while_stmt -> while M ( logical_stmt ) M block . , ident
while_stmt -> while M ( logical_stmt ) M block . , REAL
```

```
252
if_stmt -> if ( logical_stmt ) M block N else . M block , }
if_stmt -> if ( logical_stmt ) M block N else . M block , while
if_stmt -> if ( logical_stmt ) M block N else . M block , if
if_stmt -> if ( logical_stmt ) M block N else . M block , (
if_stmt -> if ( logical_stmt ) M block N else . M block , INT
if_stmt -> if ( logical_stmt ) M block N else . M block , !
if_stmt -> if ( logical_stmt ) M block N else . M block , STRING
if_stmt -> if ( logical_stmt ) M block N else . M block , CHAR
if_stmt -> if ( logical_stmt ) M block N else . M block , ident
if_stmt -> if ( logical_stmt ) M block N else . M block , REAL
M -> @ . , {
```

Action 表

最后根据这个状态表和代码中生成的转移表构造 action 表（将 goto 和 action 合并）

输入：一个增广文法 G'。

输出：G'的规范 LR 语法分析表的函数 ACTION 和 GOTO。

方法：

1）构造 G'的 LR(1)项集族 $C'=\{I_0, I_1, \cdots, I_n\}$。

2）语法分析器的状态 i 根据 I_i 构造得到。状态 i 的语法分析动作按照下面的规则确定：

① 如果$[A\to\alpha\cdot a\beta, b]$在 I_i 中，并且 $\mathrm{GOTO}(I_i, a)=I_j$，那么将 $\mathrm{ACTION}[i,a]$设置为“移入 j”。这里 a 必须是一个终结符号。

② 如果$[A\to\alpha\cdot, a]$在 I_i 中且 $A\neq S'$，那么将 $\mathrm{ACTION}[i, a]$设置为“规约 $A\to\alpha$”。

③ 如果$[S'\to S\cdot, \$]$在 I_i 中，那么将 $\mathrm{ACTION}[i, \$]$设置为“接受”。

如果根据上述规则会产生任何冲突动作，我们就说这个文法不是 LR(1)的。在这种情况下，这个算法无法为该文法生成一个语法分析器。

3）状态 i 相对于各个非终结符号 A 的 goto 转换按照下面的规则构造得到：如果 $\mathrm{GOTO}(I_i, A)=I_j$，那么 $\mathrm{GOTO}[i, A]=j$。

4）所有没有按照规则(2)和(3)定义的分析表条目都设为“报错”。

5）语法分析器的初始状态是由包含$[S'\to\cdot S, \$]$的项集构造得到的状态。 □

实际上只需要记录当前状态遇到某个单词后是移进，规约还是 acc 即可，如果是规约，再记录需要从栈中弹出多少个状态，代码如下

```cpp
void anaysis_action() {
    for(std::size_t i = 0; i != closure_.size(); ++i) {
        for(std::size_t j = 0; j != closure_[i].size(); ++j) {
        /* for(auto& proj : closure_[i]) { */
            auto& proj = closure_[i][j];
            auto dot = proj.dot;
            auto& pv = proj.prod_vec;
            auto& prefix = proj.prefix_code;
            auto& suffix = proj.suffix_code;
            if(pv.size() == 1 && pv.front() == start_code_ && dot == 1 && prefix == start_code_ + "0" && suffix == "$") {
                action_[i]["$"] = std::string{ "acc" };
                continue;
            }
            else if(dot != pv.size() && goto_[i].count(pv[dot])) {
                action_[i][pv[dot]] = std::size_t{ goto_[i][pv[dot]] };
            }
            else if(dot == pv.size() && prefix != start_code_ + "0") {
                if(pv.size() == 1 && pv.back() == "@") {
                    action_[i][suffix] = std::make_pair(prefix, std::make_pair(std::size_t{ 0 }, pv));
                    /* action_[i][suffix] = std::make_pair(prefix, std::size_t{ 0 }); */
                }
                else {
                    action_[i][suffix] = std::make_pair(prefix, std::make_pair(std::size_t{ dot }, pv));
                    /* action_[i][suffix] = std::make_pair(prefix, dot); */
                }
            }
        }
    }
}
```

至此整个 LR(1)分析器的构造已经完成，接下来的任务就是对源程序进行分析，因为上面已经构造出状态转移表，所以只需要一步步执行即可，如果顺利完成，说明程序没有语法问题，否则出错

语法制导翻译和中间代码生成

实验中采用三地址码的翻译方案将源程序翻译成中间代码，因为 LR(1)规约的过程就相当于是对抽象语法树的后序遍历，所以只需要在规约的时候进行适当的翻译

为了实现上的方便，对词法分析中提到的 Token 进行扩充，包含以下信息

- 符号表中的下标
- 单词类型（文法中的符号）
- 名字（如果是变量，则作为变量名存储）
- 值（保存变量的值，常量的值）
- 类型（类型推导产生的类型，如 int, double, string 等）
- Instr, True_list 与 False_list（用于对布尔表达式的翻译）

```cpp
struct Token
{
    Token() {}
    Token(std::size_t i, std::string tt, std::string n)
        : index(i), token_type(tt), name(n) {}
    std::size_t index;
    std::string token_type;
    std::string name;
    std::string value;
    std::string type;

    std::map<std::size_t, std::string> true_list, false_list, next_list;
    std::size_t instr;
};
```

此后在规约的过程中，完全可以使用每个 Token 记录当前规约的状态，便于后续的翻译

对 numeric -> INT | REAL | STRING | CHAR 的翻译

当检测到此时的规约是按照上述产生式执行时，进行简单的翻译操作，该操作不产生中间代码，仅仅用于类型推导，将推导出的类型保存在新生成的 numeric_token 中。此时就相当于属性栈的栈顶是一个常量，它的类型和值都是已知的（保存在 token 中），比如存在一个定义语句 a = 10，那么对于 10 这个常量就会进行 numeric -> INT 规约

```
else if(production == "numeric -> INT") {
    token_list.front().token_type = "numeric";
    token_list.front().type = "int";
    token_list.front().value = token_list.front().name;
}
else if(production == "numeric -> REAL") {
    token_list.front().token_type = "numeric";
    token_list.front().type = "double";
    token_list.front().value = token_list.front().name;
}
else if(production == "numeric -> STRING") {
    token_list.front().token_type = "numeric";
    token_list.front().type = "string";
    token_list.front().value = token_list.front().name;
}
else if(production == "numeric -> CHAR") {
    token_list.front().token_type = "numeric";
    token_list.front().type = "char";
    token_list.front().value = token_list.front().name;
}
```

对 expr -> numeric 的翻译

对于 a = 10 这个定义语句，需要定义一个临时变量 t，使其保存 numeric 的值（这里是 10），该操作会产生一个中间代码 t1 = 10

```
else if(production == "expr -> numeric") {
    Token num = token_list.front();
    token_list.pop_front();

    Token expr;
    expr.token_type = "expr";
    expr.name = new_tmp_name();
    expr.value = num.value;
    expr.type = num.type;

    std::cout << "....................................." << std::endl;
    std::cout << expr.name << " " << expr.value << " " << expr.type << std::endl;
    std::string s(expr.name + " = " + num.value);
    append_to_conv_result(new_list_no(), s);
    std::cout << "....................................." << std::endl;

    expr.index = symbol_table_.size();
    symbol_table_.emplace_back(expr.type, expr.name, expr.value);

    token_list.push_front(expr);
}
```

对 expr -> ident = expr 的翻译

仍然是 a = 10 这条语句，此时右侧的 expr 是上面翻译 expr -> numeric 产生的临时变量 t1。此时需要从 Token 栈中弹出三个 token 分别代表右侧的 ident，=以及 expr，然后创建新的 Token new_expr 代表产生式左侧的 expr 进行规约，将 expr 的值保存到 ident 的符号表中，然后将 new_expr 添加到 Token 栈中

此时产生中间代码 a = t1（由此推导出 a 的类型是 t1 的类型，也就是 int）

```
else if(production == "expr -> ident = expr") {
    Token expr = token_list.front();
    token_list.pop_front();
    Token assign = token_list.front();
    token_list.pop_front();
    Token idn = token_list.front();
    token_list.pop_front();

    idn.value = expr.value;
    idn.type = expr.type;

    std::cout << "....................................." << std::endl;
    Symbol& symbol = symbol_table_[idn.index];
    symbol.type = idn.type;
    symbol.name = idn.name;
    symbol.value = idn.value;
    std::string s(symbol.name + " = " + expr.name + "\t(" + symbol.name + " type: " + idn.type + ")");
    append_to_conv_result(new_list_no(), s);
    std::cout << "....................................." << std::endl;

    idn.token_type = "expr";
    token_list.push_front(idn);
}
```

经过上述三步规约，就可以将 a = 10 翻译完成，结果如下

```
100: t1 = 10
101: a = t1     (a type: int)

a int 10
t1 int 10
```

其中前两行是翻译结果，后两行是符号表中的内容，因为程序在一边翻译一边计算，所以符号表中可以直接填写变量 a 的值

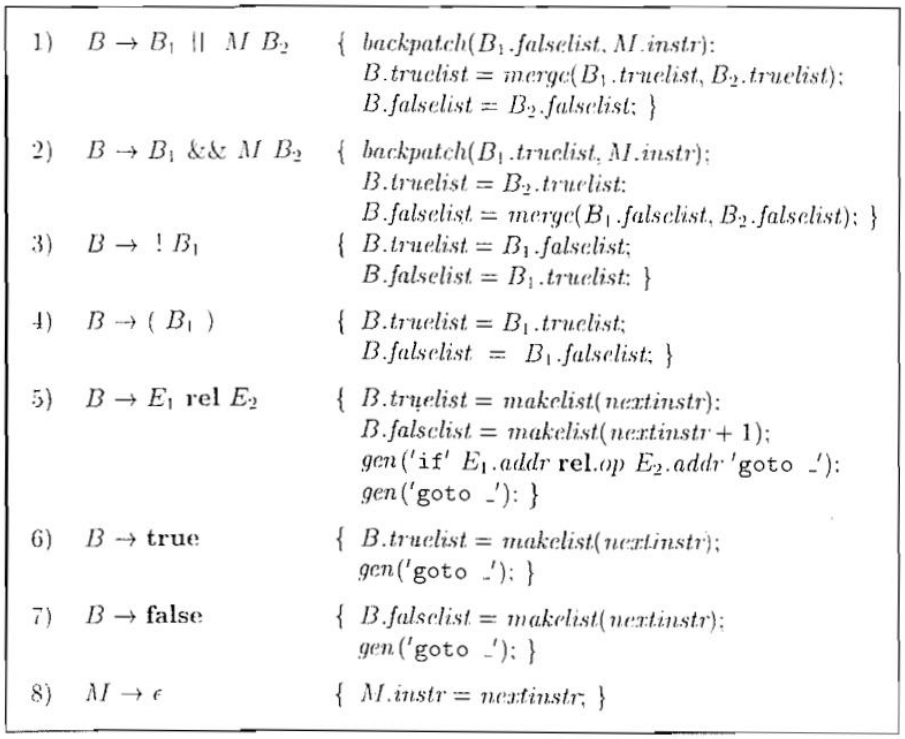
对于 expr -> expr operation expr 的翻译
Operation 是二元运算符，目前只支持加减乘除并带有类型检测，翻译时，只需要从右侧两个 expr 中取值，然后进行运算（由于 Token 中保存有类型，所以可以进行类型检查），最后将结果记录在左侧 expr 中并添加到 Token 栈中

```
else if(production == "expr -> expr operation expr") {
    Token expr2 = token_list.front();
    token_list.pop_front();
    Token op = token_list.front();
    token_list.pop_front();
    Token expr1 = token_list.front();
    token_list.pop_front();

    std::cout << expr1.type << " " << expr2.type << std::endl;
    Token expr;
    expr.token_type = "expr";
    expr.name = new_tmp_name();
    if(is_valid_operation(expr1.type, expr2.type)) {
        if(expr1.type == "double" || expr2.type == "double") {
            expr.value = std::to_string(binary_operation<double>(op.name, expr1.value, expr2.value));
            expr.type = "double";
        }
        else {
            expr.value = std::to_string(binary_operation<int>(op.name, expr1.value, expr2.value));
            expr.type = "int";
        }
    }
    else {
        throw std::runtime_error("binary operation type error: " + expr1.type + op.name + expr2.type);
    }

    std::cout << "....................................." << std::endl;
    std::cout << expr.name << " " << expr.value << " " << expr.type << std::endl;
    std::string s(expr.name + " = " + expr1.name + " " + op.name + " " + expr2.name);
    append_to_conv_result(new_list_no(), s);
    std::cout << "....................................." << std::endl;

    expr.index = symbol_table_.size();
    symbol_table_.emplace_back(expr.type, expr.name, expr.value);

    token_list.push_front(expr);
}
```

比如 a = 10 + 2.5，在进行 numeric -> INT | REAL，expr -> numeric，operation -> +的规约操作时，此时的状态就是 a = expr + expr，翻译结果为

```
100: t1 = 10
101: t2 = 2.5
102: t3 = t1 + t2
103: a = t3     (a type: double)

a double 12.500000
t1 int 10
t2 double 2.5
t3 double 12.500000
```

对 logical_stmt -> logical_stmt lop M logical_stmt 的翻译
该语句是布尔表达式（逻辑语句），主要是将其翻译成 if goto ... goto 的形式，由于在规约时无法确定后面语句的个数，所以 goto 的目标位置无法确定，实验中采用回填的方法（也就是上面 Token 中 true_list，false_list 以及 instr 的用处），翻译方案如下



1)	$B \rightarrow B_1 \ \|\| \ M \ B_2$	{ $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }
2)	$B \rightarrow B_1 \ \&\& \ M \ B_2$	{ $backpatch(B_1.truelist, M.instr)$; $B.truelist = B_2.truelist$; $B.falselist = merge(B_1.falselist, B_2.falselist)$; }
3)	$B \rightarrow \ ! \ B_1$	{ $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
4)	$B \rightarrow (\ B_1 \ )$	{ $B.truelist = B_1.truelist$; $B.falselist = B_1.falselist$; }
5)	$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $gen('\texttt{if}' \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ '\texttt{goto}\ \_')$; $gen('\texttt{goto}\ \_')$; }
6)	$B \rightarrow \mathbf{true}$	{ $B.truelist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
7)	$B \rightarrow \mathbf{false}$	{ $B.falselist = makelist(nextinstr)$; $gen('\texttt{goto}\ \_')$; }
8)	$M \rightarrow \epsilon$	{ $M.instr = nextinstr$; }

图 6-43　布尔表达式的翻译方案

由于文法的限制，只实现了 1, 2, 5 的翻译，主要方法就是在对对应表达式进行规约时对 true_list 和 false_list 进行修改，最终输出到中间代码结果中

上述的 1,2,5 分别对应文法中的两条文法（&&和||合并成 comparsion）

Logical_stmt -> logical_stmt lop M logical_stmt

```
else if(production == "logical_stmt -> logical_stmt lop M logical_stmt") {
    Token ls2 = front_and_pop(token_list);
    Token m = front_and_pop(token_list);
    Token lop = front_and_pop(token_list);
    Token ls1 = front_and_pop(token_list);

    Token ls;
    if(lop.name == "||") {
        back_patch(ls1.false_list, m.instr);
        ls.true_list = merge(ls1.true_list, ls2.true_list);
        ls.false_list = ls2.false_list;
    }
    else {
        back_patch(ls1.true_list, m.instr);
        ls.false_list = merge(ls1.false_list, ls2.false_list);
        ls.true_list = ls2.true_list;
    }

    ls.name = "logical_stmt";
    token_list.push_front(ls);
}
```

Logical_stmt -> expr comparsion expr

```
else if(production == "logical_stmt -> expr comparsion expr") {
    Token expr2 = front_and_pop(token_list);
    Token comp = front_and_pop(token_list);
    Token expr1 = front_and_pop(token_list);

    Token stmt;
    std::size_t n = new_list_no();
    stmt.true_list[n] = std::to_string(n) + std::string(":if ") + expr1.name + " " + comp.name + " " + expr2.n
    append_to_conv_result(n, stmt.true_list[n]);
    n = new_list_no();
    stmt.false_list[n] = std::to_string(n) + std::string(":goto ");
    append_to_conv_result(n, stmt.false_list[n]);

    stmt.name = "logical_stmt";
    token_list.push_front(stmt);
}
```

以 a=10 b=a a>10&&b<5 为例，翻译结果为

```
100: t1 = 5
101: a = t1     (a type: int)
102: t2 = a     (t2 type: int)
103: b = t2     (b type: int)
104: t3 = a     (t3 type: int)
105: t4 = 10
106: if t3 > t4 goto  108
107: 107:goto
108: t5 = b     (t5 type: int)
109: t6 = 5
110: 110:if t5 < t6 goto
111: 111:goto
```

因为逻辑表达式的后面没有任何语句，所以仅仅回填了 if t3 > t4 中的 goto，事实上，如果将该表达式与 if 语句结合使用，就会将所有的 goto 都回填完成

对 if_stmt -> if (logical_stmt) M block N else M block 的翻译
当通过该语句进行规约时，说明正在翻译一个完整的 if else 语句，这里的 M 和 N 是用来获取下一个语句标号，方便在规约时进行回填，翻译方案如下（但仅仅需要考虑 1, 2, 4, 6, 7）

1) $S \rightarrow \textbf{if}\,(\,B\,)\,M\,S_1$ { $backpatch(B.truelist,\ M.instr)$;
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist)$; }

2) $S \rightarrow \textbf{if}\,(\,B\,)\,M_1\,S_1\,N\,\textbf{else}\,M_2\,S_2$
{ $backpatch(B.truelist,\ M_1.instr)$;
$backpatch(B.falselist,\ M_2.instr)$;
$temp\ =\ merge(S_1.nextlist,\ N.nextlist)$;
$S.nextlist\ =\ merge(temp,\ S_2.nextlist)$; }

3) $S \rightarrow \textbf{while}\,M_1\,(\,B\,)\,M_2\,S_1$
{ $backpatch(S_1.nextlist,\ M_1.instr)$;
$backpatch(B.truelist,\ M_2.instr)$;
$S.nextlist\ =\ B.falselist$;
$gen('\texttt{goto}'\ M_1.instr)$; }

4) $S \rightarrow \{\ L\ \}$ { $S.nextlist\ =\ L.nextlist$; }

5) $S \rightarrow A\ ;$ { $S.nextlist\ =\ \textbf{null}$; }

6) $M \rightarrow \epsilon$ { $M.instr\ =\ nextinstr$; }

7) $N \rightarrow \epsilon$ { $N.nextlist\ =\ makelist(nextinstr)$;
$gen('\texttt{goto}\ \_')$; }

8) $L \rightarrow L_1\ M\ S$ { $backpatch(L_1.nextlist,\ M.instr)$;
$L.nextlist\ =\ S.nextlist$; }

9) $L \rightarrow S$ { $L.nextlist\ =\ S.nextlist$; }

```
else if(production == "logical_stmt -> logical_stmt lop M logical_stmt") {
    Token ls2 = front_and_pop(token_list);
    Token m = front_and_pop(token_list);
    Token lop = front_and_pop(token_list);
    Token ls1 = front_and_pop(token_list);

    Token ls;
    if(lop.name == "||") {
        back_patch(ls1.false_list, m.instr);
        ls.true_list = merge(ls1.true_list, ls2.true_list);
        ls.false_list = ls2.false_list;
    }
    else {
        back_patch(ls1.true_list, m.instr);
        ls.false_list = merge(ls1.false_list, ls2.false_list);
        ls.true_list = ls2.true_list;
    }

    ls.name = "logical_stmt";
    token_list.push_front(ls);
}
```

```
else if(production == "M -> @") {
    Token m;
    m.token_type = "M";
    m.instr = new_list_no(false);
    std::cout << m.instr << std::endl;

    token_list.push_front(m);
}
else if(production == "N -> @") {
    Token n;
    n.token_type = "N";
    std::size_t num = new_list_no();
    n.next_list[num] = std::to_string(num) + ":goto ";
    append_to_conv_result(num, n.next_list[num]);
    token_list.push_front(n);
}
```

对 while_stmt -> while M (logical_stmt) M block 的翻译
翻译方案和上面相同

```
else if(production == "while_stmt -> while M ( logical_stmt ) M block") {
    std::vector<Token> tokens = front_and_pop_n(7, token_list);
    back_patch(tokens.back().next_list, tokens[1].instr);
    back_patch(tokens[3].true_list, tokens[5].instr);

    Token ws;
    ws.next_list = tokens[3].false_list;

    std::size_t n = new_list_no();
    std::string s("goto " + std::to_string(tokens[1].instr));
    append_to_conv_result(n, s);

    back_patch(ws.next_list, new_list_no(false));

    ws.name = "while_stmt";
    token_list.push_front(ws);
}
std::printf("%s\n%s\n\n", production.data(), "reduction");
```

测试程序
测试程序中包含了变量的定义，赋值，算数表达式，if，while
同时还实现了部分语义错误如类型操作不匹配，不能是用未定义的变量等

```
/* 变量 */
a = 10 + 2.2
b = a
c = b + a + 3
d = 5
str = "hello world"

/* if */
if(a < 10) {
    b = 1
}
else {
    b = 2
}

/* while */
while(a < 5 && a > 1) {
    a = a + 1
    b = a
    c = b + a
}

/* 嵌套if */
if(a < 20) {
    if(a > 10) {
        a = 5
    }
    else {
        a = 10
    }
}
else {
    a = 20
}

/* 嵌套while */
while(a < 5) {
    while(b > 1) {
        b = b + 1
    }
}
```

输出结果

变量定义赋值

```
100: t1 = 10
101: t2 = 2.2
102: t3 = t1 + t2
103: a = t3     (a type: double)
104: t4 = a     (t4 type: double)
105: b = t4     (b type: double)
106: t5 = b     (t5 type: double)
107: t6 = a     (t6 type: double)
108: t7 = 3
109: t8 = t6 + t7
110: t9 = t5 + t8
111: c = t9     (c type: double)
112: t10 = 5
113: d = t10    (d type: int)
114: t11 = hello world
115: str = t11  (str type: string)
```

if

```
116: t12 = a    (t12 type: double)
117: t13 = 10
118: if t12 < t13 goto  120
119: goto  123
120: t14 = 1
121: b = t14    (b type: int)
122: goto  125
123: t15 = 2
124: b = t15    (b type: int)
```

while

```
125: t16 = a    (t16 type: double)
126: t17 = 5
127: if t16 < t17 goto  129
128: goto  144
129: t18 = a    (t18 type: double)
130: t19 = 1
131: if t18 > t19 goto  133
132: goto  144
133: t20 = a    (t20 type: double)
134: t21 = 1
135: t22 = t20 + t21
136: a = t22    (a type: double)
137: t23 = a    (t23 type: double)
138: b = t23    (b type: double)
139: t24 = b    (t24 type: double)
140: t25 = a    (t25 type: double)
141: t26 = t24 + t25
142: c = t26    (c type: double)
143: goto 125
```

嵌套 if

```
144: t27 = a    (t27 type: double)
145: t28 = 20
146: if t27 < t28 goto  148
147: goto  158
148: t29 = a    (t29 type: double)
149: t30 = 10
150: if t29 > t30 goto  152
151: goto  155
152: t31 = 5
153: a = t31    (a type: int)
154: goto  157
155: t32 = 10
156: a = t32    (a type: int)
157: goto  160
158: t33 = 20
159: a = t33    (a type: int)
```

嵌套 while

```
160: t34 = a    (t34 type: int)
161: t35 = 5
162: if t34 < t35 goto  164
163: goto  174
164: t36 = b    (t36 type: double)
165: t37 = 1
166: if t36 > t37 goto  168
167: goto  173
168: t38 = b    (t38 type: double)
169: t39 = 1
170: t40 = t38 + t39
171: b = t40    (b type: double)
172: goto 164
173: goto 160
```

实验总结

通过本次实验，实现了词法分析，语法分析，语法制导翻译以及中间代码生成，初步完成了一个简单的解释器，理解了编译器的工作原理