同济大学计算机系

词法和语法分析工具设计与实现
实验报告

院　　系 电子与信息工程学院

专　　业 计算机科学与技术\信息安全

组　员 1　2152118　史君宝

组　员 2　2154062　赵书玥

组　员 3　2151638　林天野

授课老师 丁志军

一、实验内容及需求分析

1.1. 实验内容

本实验根据要求写出一个中间代码生成器，完成静态语义错误的诊断和处理，在此基础上，考虑更为通行的高级语言的语义检查和中间代码生成所需要主义的内容，并输出分析结果。

1.2. 程序功能

本程序以四元式的形式作为中间代码，实现中间代码生成。

向本程序输入源程序，本程序能够处理静态语义的诊断和处理。

1.3. 输入信息

输入无论正确与否的源程序

```
#include<cstdio>
int main()
{
}
```

1.4. 输出信息

详见之后的输出部分

1.5. 测试数据

在test 目录下给出了源程序，作为测试文件。程序输出以图形化界面展示。

具体的测试细节参考调试分析与结果展示部分。

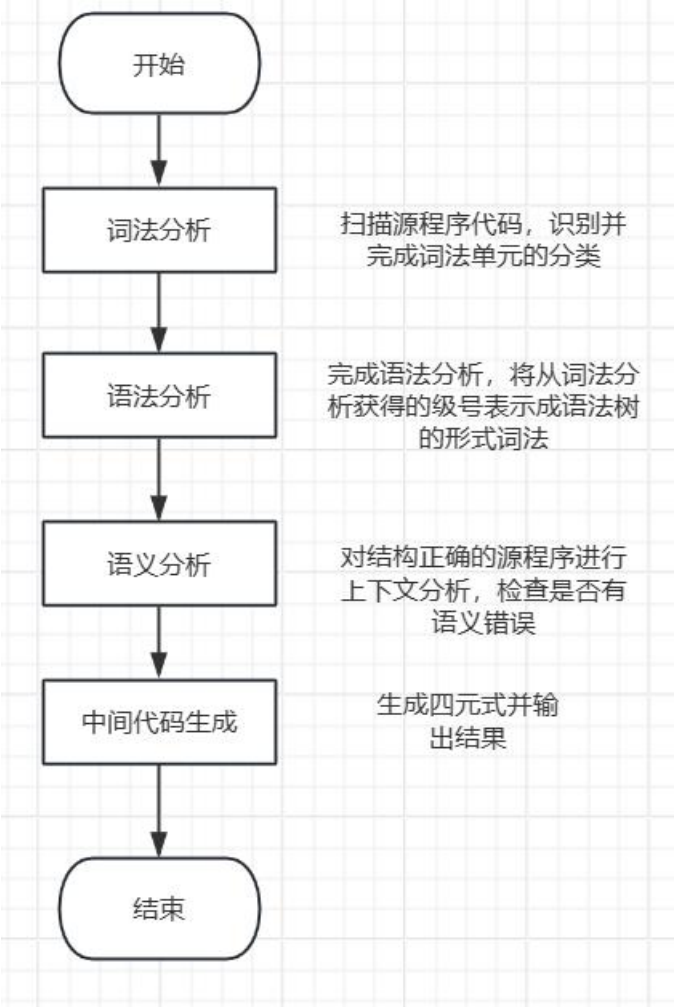
二、项目总体设计

2.1 中间代码生成器

我们已知编译器是将高级语言代码转换成计算机可以理解的低级代码的程序，而中间代码生成器是其中一个组成部分。中间代码生成器负责将高级语言代码转换为中间代码。中间代码通常是一种简化的语言，它可以用来表示程序的控制流、数据流和对象关系。可以让编译器更容易对代码进行优化和转换。

中间代码生成器主要包括以下两个部分：词法分析、语法分析和代码生成三个主要部分。在词法分析阶段，生成器读入字符并识别其类型，将输入的代码划分成单词，并将每个单词的类型（如关键字、标识符、数字等）保存在变量中；在语法分析阶段，根据对应的文法规则调用各个函数进行语法分析，包括赋值、条件和循环语句等，逐步构建出目标程序的中间代码。最后得到中间代码并输出。由于四元式是更接近目标代码的中间代码形式，便于优化处理，故此处采用四元式作为最后输出形式。

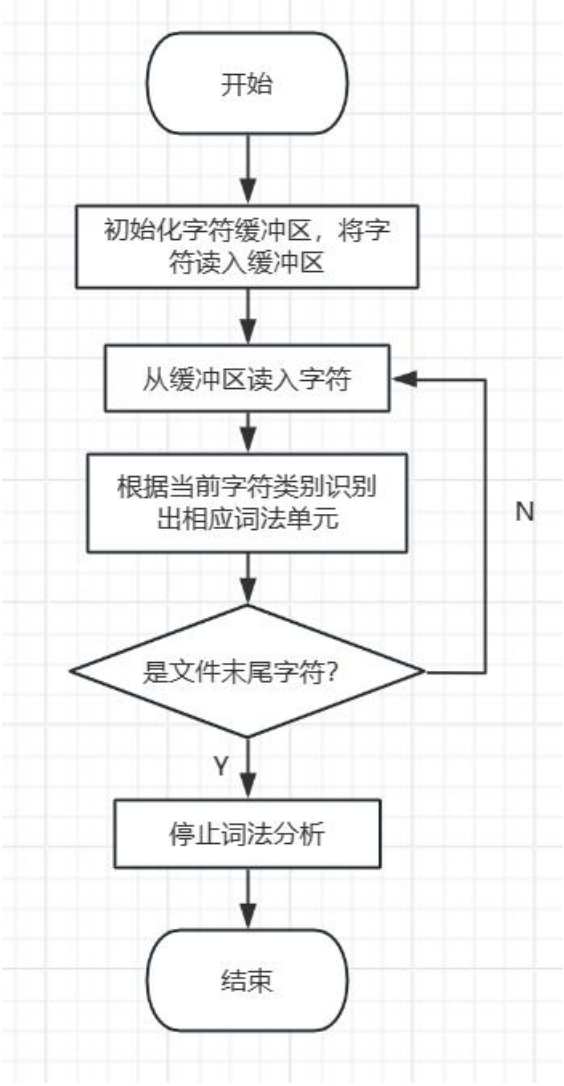
中间代码生成器具体逻辑如下：



2.2 词法分析

词法分析部分的主要由以下部分组成：输入缓冲区、词法单元识别和词法单元输出。其中词法单元识别需要识别源代码中的词法单元，可以通过将字符串划分为不同类型的词法单元实现；单元输出则是以便进行后续的语法分析和代码生成。

词法分析的具体步骤如下：



2.3 词法分析代码实现

2.3.1 读入字符和对是否读到文件末尾的判断

```
//获取下一个字符
wait_ch = sourceFile.get();

//未读到文件末尾
while (wait_ch != EOF) //
{
    while (isWhitespace(wait_ch)) //遇到空白字符
    {
        if (wait_ch == ' ') {
            col++;
        }
        else if (wait_ch == '\n') {
            row++;
            col = 1;
        }
        wait_ch = sourceFile.get();
    }
```

2.3.2 对字符类型的判断

```
//是否是空白符
bool isWhitespace(char ch) {
    return (ch == ' ' || ch == '\t' || ch == '\n' || ch == '\r');
}

//是否是数字
bool isDigit(char ch) {
    return (ch == '0' || ch == '1' || ch == '2' || ch == '3' || ch == '4'
||
        ch == '5' || ch == '6' || ch == '7' || ch == '8' || ch == '9');
}

//是否是英文字符
bool isABC(char ch) {
    return (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z');
}
```

2.3.3 词法单元输出

篇幅受限，此处仅以判断关键字并输出为例：

```
            map<string, string>::iterator it = key_words.find(tmp_ABC);

            //如果没有找到
            if (it == key_words.end()) {
                //不在关键字表中，是标识符
                //将这个内容添加到字符表中
                word_content = make_pair(tmp_ABC, "ID");
                word_pos = make_pair(row, col);
                word_indi = make_pair(word_content, word_pos);
                Word_table.push_back(word_indi);

                ID_table.insert(tmp_ABC);//将获得到的 ID 放在 ID 表里
            }
            else {
                //在关键字表中
                //将这个内容添加到字符表中
                word_content = make_pair(it->first, it->second);
                word_pos = make_pair(row, col);
                word_indi = make_pair(word_content, word_pos);
                Word_table.push_back(word_indi);
            }
```

其余继续进行以上操作即可实现词法单元的分类和输出。

2.3.4 错误判断

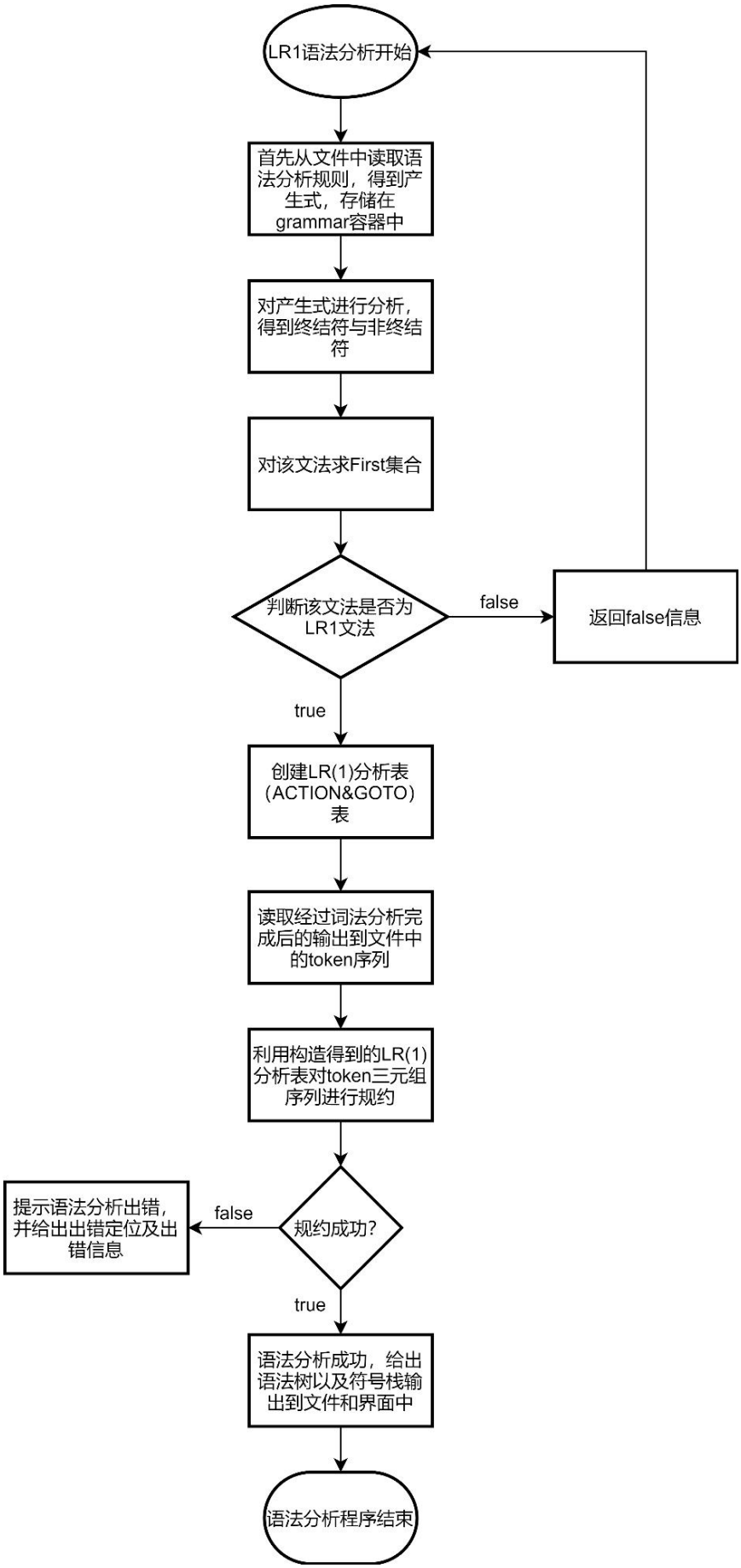
```
void WordAnalyze::Word_output(string choice)
{
    if(choice == "Error")
    {
        fstream fout(R"(file\Word_error.txt)");
        if (!ERROR_LIST.empty())
        {

            fout << "词法分析过程出错" << endl;
            fout << "错误信息如下所示" << endl;
```

```
            for (auto it = ERROR_LIST.begin(); it != ERROR_LIST.end();
it++)
            {
                fout << it->first.second << "    " << it->first.first <<
"    ";
                fout << "位置：" << it->second.first << "行" <<
it->second.second << "列" << endl;
            }
        }
        else{
            fout << "词法分析过程未出错" << endl;
            fout << "可以查看具体的符号表" << endl;
        }

        return;
    }

    else if(choice == "ID")
    {
        fstream fout(R"(file\Word_ID.txt)");
        if (!ERROR_LIST.empty())
        {
            fout << "词法分析过程出错" << endl;
            fout << "请进入错误报告查看具体内容" << endl;

        }
        else{
            fout <<  "标识符表;" << endl;
            for (auto it = ID_table.begin(); it != ID_table.end(); it++)
            {
                fout << "字符值:" << setiosflags(ios::left)
                    << setw(6) << *it << "符号类别:标识符" << endl;
            }
        }

        return;
    }

    else if(choice == "Sign")
    {
        fstream fout(R"(file\Word_Sign.txt)");
        if (!ERROR_LIST.empty())
        {
            fout << "词法分析过程出错" << endl;
            fout << "请进入错误报告查看具体内容" << endl;

        }
        else{
            fout << endl << "符号表;" << endl;
            for (auto it = Sign_table.begin(); it != Sign_table.end();
it++)
```

```
            {
                fout << "符号值:" << setiosflags(ios::left)
                    << setw(6) << *it << "符号类别:Sign" << endl;
            }
        }

        return;
    }

    else if(choice == "ALL")
    {
        fstream fout(R"(file\Word_all.txt)");
        if (!ERROR_LIST.empty())
        {
            fout << "词法分析过程出错" << endl;
            fout << "请进入错误报告查看具体内容" << endl;

        }
        else{
            fout << "词法分析结果" << endl << endl;
            for (auto it = Word_table.begin(); it != Word_table.end();
it++)
            {
                fout << "字符值:" << setiosflags(ios::left) << setw(6) <<
it->first.first
                    << "符号类别:" << setw(6) << it->first.second
                    << "  位 置 :" << it->second.first << "," <<
it->second.second << endl;
            }
        }

        return;
    }
}
```

2.4 语法分析和语义分析

语法分析的目的是判断程序的结构是否符合文法的要求，语义分析则是对结构正确的源程序进行上下文分析，检查源程序代码有无语义错误。


LR1语法分析开始
首先从文件中读取语法分析规则，得到产生式，存储在grammar容器中
对产生式进行分析，得到终结符与非终结符
对该文法求First集合
判断该文法是否为LR1文法
false
返回false信息
true
创建LR(1)分析表（ACTION&GOTO）表
读取经过词法分析完成后的输出到文件中的token序列
利用构造得到的LR(1)分析表对token三元组序列进行规约
规约成功?
false
提示语法分析出错，并给出出错定位及出错信息
true
语法分析成功，给出语法树以及符号栈输出到文件和界面中
语法分析程序结束

2.5 语法分析和语义分析代码实现

2.5.1 错误类型定义

```
/*错误类型
* 1.尝试打开错误的文件
* 2.使用了不合法的符号
* 3.头文件未声明或声明错误
* 4.未进行变量定义
* 5.错误定义变量
* 6.条件语句作为执行
* 7.赋值语句使用错误
* 8.While 语句没有配套的 do
* 9.While 语句之间定义了变量
* 10.While 语句之后缺少判断语句
* 11.While 语句组成不完整
* 12.for 语句缺少组成成分
* 13.在 for 语句之中定义变量
* 14.重定义相同的变量
* 15.在 While 语句之中定义了变量
* 16.if 语句组成不完整
* 17.if 语句没有配套的 then
* 18.then 没有后续语句
* 19.{没有配套的}作为结尾
* 20.赋值表达式没有;作为结尾
* 21.变量定义语句没有;作为结尾
* 22.输入语句没有正确书写
* 23.输出语句没有正确书写
* 24.输入语句输入了多个变量或表达式
* 25.输出语句输出了多个变量或表达式
* 26.（没有）作为匹配
* 27.文法没有执行语句
* 28.文法没有定义语句
* 29.使用了未定义的变量
* 30.除法表达式使用 0 作为分母
*/
```

2.5.1 语法表

```
/*语法表
<程序> ::= <头文件定义>{<分程序>}
<头文件定义>::= #include<iostream> int main()
<分程序> ::= <变量定义><执行语句>
 <变量定义>::= <变量类型><标识符>;{<变量定义 >}
<标识符> ::= <字母>{<字母>|<数字>}
<执行语句>::=<输入语句>|<输出语句>|<赋值语句>|<条件语句>|<While 语句>|<For
语句>|{<执行语句>}
<赋值语句>::=<标识符> <赋值运算符> <表达式 > ;
<While 语句>::= while(<条件语句>)do<执行语句>
<For 语句>::= for([<赋值语句>] <条件语句> ; <赋值语句> ) <执行语句>
<条件语句>::= if( <条件> ) then <执行语句> [ else <执行语句> ]
<条件语句>::= <逻辑或表达式> { || <逻辑或表达式> }
<逻辑或表达式>::= <逻辑与表达式> { && <逻辑与表达式> }
```

```
<逻辑与表达式>::= <表达式><关系运算符><表达式>
<表达式>::= <按位或表达式> { | < 按位或表达式>}
<按位或表达式>::= <按位与表达式> { & < 按位与表达式>}
<按位与表达式>::= <位移表达式> { <位移运算符> <位移表达式>}
<位移表达式>::= <乘除表达式>{<加减运算符> <乘除表达式>}
<乘除表达式>::= <运算式>{<乘除运算符> <运算式>}
<运算式>::=(<运算式>) |<标识符>|<整数>
<乘除运算符>::= * | /
<加减运算符>::= + | -
<位移运算符>::= >> | <<
<关系运算符> ::= == | != | < | <= | > | >=
<赋值运算符>::= = | += | -= | *= | /=
<输入语句> ::= scanf(<标识符>)
<输出语句> ::= printf(<标识符>)
<变量类型>::= int | char | string
<字母> ::= a|b|...|X|Y|Z
<数字> ::= 0|1|2|...|8|9
<整数> ::= [-] <数字>
*/
```

2.5.3 语法分析

```
//语法分析的具体程序
void GrammerAnalyze::Program()
{//头文件检测
    int i = 0;
    //检测 P 的 first 集合
    for (it = Word_result.begin(); it != Word_result.begin() + 10; it++, i++)
    {
        switch (i)
        {
            case 0:
                if (it->first.second != "BEGIN")
                {
                    Error_type = make_pair(it->first.second, "Should start
with a '#'");
                    cout << "Should start with a '#'";

wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
                    exit(0);
                }
                break;
            case 1:
                if (it->first.second != "INCLUDE")
                {
                    Error_type = make_pair(it->first.first, "Expect for
'include'");

wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
                    cout << "Expect for 'include'";
                    exit(0);
                }
```

```
                break;
            case 2:
                if (it->first.first != "<")
                {
                    Error_type  =  make_pair(it->first.first,  "please  ues
'<'");

wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
                    cout << "please ues '<'";
                    exit(0);
                }
                break;
            case 3:
                if (it->first.second != "IOSTREAM")
                {
                    Error_type  =  make_pair(it->first.first,  "Expect  for
'iostream' to start");

wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
                    cout << "Expect for 'iostream' to start";
                    exit(0);
                }
                break;
            case 4:
                if (it->first.first != ">")
                {
                    Error_type  =  make_pair(it->first.first,  "please  use
'>'");

wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
                    cout << "please use '>'";
                    exit(0);
                }
                break;
            case 5:
                if (it->first.first != "int")
                {
                    Error_type = make_pair(it->first.first, "need 'int'");

wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
                    cout << "need '{'";
                    exit(0);
                }
                break;
            case 6:
                if (it->first.first != "main")
                {
                    Error_type = make_pair(it->first.first, "need '{'");

wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
                    cout << "need '{'";
```

```
                    exit(0);
                }
                break;
            case 7:
                if (it->first.first != "(")
                {
                    Error_type = make_pair(it->first.first, "need '{'");

wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
                    cout << "need '{'";
                    exit(0);
                }
                break;
            case 8:
                if (it->first.first != ")")
                {
                    Error_type = make_pair(it->first.first, "need '{'");

wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
                    cout << "need '{'";
                    exit(0);
                }
                break;
            case 9:
                if (it->first.first != "{")
                {
                    Error_type = make_pair(it->first.first, "need '{'");

wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
                    cout << "need '{'";
                    exit(0);
                }
                break;
        }
    }

    //这里检查变量的声明
    Define();
    Start();

    if (it->first.first != "}")
    {
        cout << "缺少}作为结尾" << "位置 " << it->second.first << "行" <<
it->second.second << "列";
        exit(0);
    }
    else
    {
        it++;
    }
}
```

2.5.4 变量声明检查

```
//在这个函数中，会检查变量的声明过程中是否有问题
void GrammerAnalyze::Define()
{
    //会检查 int 型声明
    if (it->first.second == "INT")
    {
        it++;
        if (it->first.second == "ID")
        {

            if (IDList.find(it->first.first) != IDList.end())//检查重定义
            {
                cout << "重定义元素！" << "     错误位置" << it->second.first
<< "行" << it->second.second << "列";
                exit(0);
            }
            IDNEED = make_pair(it->first.first, "_");
            IDList.insert(IDNEED);
            string TempID = it->first.first;
            it++;
            if (it->first.first == ";")
            {
                it++;
                //生成 D 的四元式
                Four("INT", TempID, "_", "_");
            }
            else
            {
                cout << "缺少;结尾" << "     错误位置"
                    << it->second.first << "行" << it->second.second << "列
";
                exit(0);
            }
        }
        else
        {
            Error_type = make_pair(it->first.second, "D()fail");
            cout << "缺少声明的变量名 " << "     错误位置"
                << it->second.first << "行" << it->second.second << "列";
            wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
            exit(0);
        }
        if (it->first.second == "INT" || it->first.second == "CHAR" ||
it->first.second == "STRING")
            Define();
    }
    //会检查 char 型声明
    else if (it->first.second == "CHAR")
    {
```

```
        it++;
        if (it->first.second == "ID")
        {

            if (IDList.find(it->first.first) != IDList.end())//检查重定义
            {
                cout << "重定义元素！" << "     错误位置" << it->second.first
<< "行" << it->second.second << "列";
                exit(0);
            }
            IDNEED = make_pair(it->first.first, "_");
            IDList.insert(IDNEED);
            string TempID = it->first.first;
            it++;
            if (it->first.first == ";")
            {
                it++;
                //生成D的四元式
                Four("CHAR", TempID, "_", "_");
            }
            else
            {
                cout << "缺少;结尾" << "     错误位置"
                    << it->second.first << "行" << it->second.second << "列
";

                exit(0);
            }

        }
        else
        {
            Error_type = make_pair(it->first.second, "D()fail");
            cout << "缺少声明的变量名 " << "     错误位置"
                << it->second.first << "行" << it->second.second << "列";
            wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
            exit(0);
        }
        if (it->first.second == "INT" || it->first.second == "CHAR" ||
it->first.second == "STRING")
            Define();
    }
    //会检查string型声明
    else if (it->first.second == "STRING")
    {
        it++;
        if (it->first.second == "ID")
        {

            if (IDList.find(it->first.first) != IDList.end())//检查重定义
            {
                cout << "重定义元素！" << "     错误位置" << it->second.first
```

```
<< "行" << it->second.second << "列";
                exit(0);
            }
            IDNEED = make_pair(it->first.first, "_");
            IDList.insert(IDNEED);
            string TempID = it->first.first;
            it++;
            if (it->first.first == ";")
            {
                it++;
                //生成D的四元式
                Four("STR", TempID, "_", "_");
            }
            else
            {
                cout << "缺少;结尾" << "     错误位置"
                    << it->second.first << "行" << it->second.second << "列
";
                exit(0);
            }
        }
        else
        {
            Error_type = make_pair(it->first.second, "D()fail");
            cout << "缺少声明的变量名 " << "     错误位置"
                << it->second.first << "行" << it->second.second << "列";
            wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
            exit(0);
        }
        if (it->first.second == "INT" || it->first.second == "CHAR" ||
it->first.second == "STRING")
            Define();
    }
    //其他声明过程
    else
    {
        Error_type = make_pair(it->first.second, "定义变量错误");
        cout << "Please Check where ID was define" << "     错误位置"
            << it->second.first << "行" << it->second.second << "列";
        wordanalyze.ERROR_LIST.insert({ Error_type,it->second });
        exit(0);
    }
}
```

2.5.5 对语句开始符号分类检查

```
//这里会对每一行语句的开始符号进行检查
//比如如果某一行的开始字符是if，它将转向if对应的检查函数
void GrammerAnalyze::Start()
{
    //如果语句以if开始
    if (it->first.second == "IF")
```

```
    {
        forstart++;
        //转向 If 语句对应的检查函数
        IFCHECK();
        forstart--;
        if ((it->first.second == "IF" || it->first.second == "WHILE" ||
it->first.second == "ID" ||
            it->first.second == "CHAR" || it->first.second == "FOR" ||
it->first.second == "PRINTF"
            || it->first.second == "{" || it->first.second == "SCANF"))
        {
            //继续进行检查语句的开始
            Start();
        }
    }
    //如果语句以 while 开始
    else if (it->first.second == "WHILE")
    {
        forstart++;
        WhileCheck();
        forstart--;
        if ((it->first.second == "IF" || it->first.second == "WHILE" ||
it->first.second == "ID" ||
            it->first.second == "CHAR" || it->first.second == "FOR" ||
it->first.second == "PRINTF"
            || it->first.second == "{" || it->first.second == "SCANF"))
        {
            Start();
        }
    }
    //如果语句以{开始
    else if (it->first.first == "{")
    {
        it++;
        jumpsymbol = false;
        Start();
        if (it->first.first == "}")
        {
            it++;
            if ((it->first.second == "IF" || it->first.second == "WHILE" ||
it->first.second == "ID" ||
                it->first.second == "CHAR" || it->first.second == "FOR" ||
it->first.second == "PRINTF"
                || it->first.second == "{" || it->first.second == "SCANF"))
            {
                if (forstart == Sdeep && (forstart != 0) && (Sdeep != 0))
                    return;
                Start();
            }
        }
        else
```

```
        {
            cout << "括号不匹配" << it->second.first << "," <<
it->second.second;
            exit(0);
        }

    }
    //如果语句以变量声明和变量开始
    else if (it->first.second == "ID" ||
        it->first.second == "CHAR" ||
        it->first.second == "STRING")
    {
        while (it->first.second == "ID" || it->first.second == "CHAR" ||
it->first.second == "STRING")
        {
            Numgiven();
            if (jumpsymbol == true)
            {
                jumpsymbol = false;
                break;
            }
        }
        if ((it->first.second == "IF" || it->first.second == "WHILE" ||
it->first.second == "ID" ||
            it->first.second == "CHAR" || it->first.second == "FOR" ||
it->first.second == "PRINTF"
            || it->first.second == "{" || it->first.second == "SCANF"))
        {
            Start();
        }
    }
    //如果语句以 for 开始
    else if (it->first.second == "FOR")
    {
        forstart++;
        FORCHECK();
        forstart--;
        if ((it->first.second == "IF" || it->first.second == "WHILE" ||
it->first.second == "ID" ||
            it->first.second == "CHAR" || it->first.second == "FOR" ||
it->first.second == "PRINTF"
            || it->first.second == "{" || it->first.second == "SCANF"))
        {
            // if (forstart == Sdeep && (forstart != 0) && (Sdeep != 0))
                 //return;
            Start();
        }
    }
    //如果语句以 printf 开始
    else if (it->first.second == "PRINTF")
    {
```

```
        Printout();
        if ((it->first.second == "IF" || it->first.second == "WHILE" ||
it->first.second == "ID"
            || it->first.second == "FOR" || it->first.second == "PRINTF" ||
it->first.second == "{"
            || it->first.second == "SCANF"))
        {
            Start();
        }
    }
    //如果语句以 scanf 开始
    else if (it->first.second == "SCANF")
    {
        Scanfin();
        if ((it->first.second == "IF" || it->first.second == "WHILE" ||
it->first.second == "ID" ||
            it->first.second == "CHAR" || it->first.second == "FOR" ||
it->first.second == "PRINTF"
            || it->first.second == "{" || it->first.second == "SCANF"))
        {
            Start();
        }
    }
    //如果是以其他符号开始，就会报错
    else
    {
        cout << "缺少执行语句！" << "    错误位置:" << it->second.first << "
行"
            << it->second.second << "列";;
        exit(0);
    }
}
```

2.6 生成四元式并输出到文件中

```
void GrammerAnalyze::output(string choice)
{


        fstream fout(R"(file\Grammar_fourline.txt)");

        vector <Foursentence>::iterator itout = FourLine.begin();
        fout << "生成四元式结果：" << endl << endl;
        for (; itout != FourLine.end(); itout++)
        {
            fout << itout->Posi << " ";
            fout <<"(" << setiosflags(ios::left) << setw(4) << itout->Type <<
" ," << setw(4) << itout->Number_1;
            fout << "," << setw(4) << itout->Number_2 << "," << setw(4) <<
itout->Name << ")" << endl;

        }
```

```
            return;
    }
```

2.7 静态语义错误诊断和处理

2.7.1 错误诊断

使用未经定义的变量指的是在变量定义之前使用此变量执行运算、赋值等指令。错误诊断思路为：在对表达式执行语义分析时，识别出表达式中的各个变量，查找这些变量是否经过了声明语句的定义，若未声明，则出错。

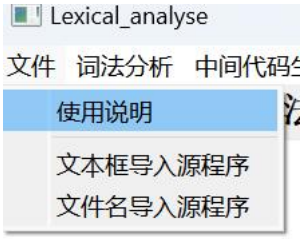
2.7.2 错误处理

本程序中若出现使用未经定义变量的情况，则直接报错，程序运行结束，同时在中端和 UI 文本框中输出错误及定位信息。

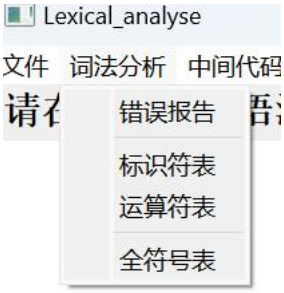
三、图形界面设计

1.左侧可输入语法或程序，右侧提示栏输出信息

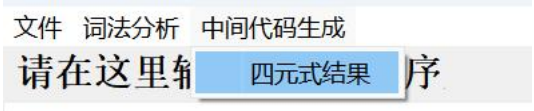
2.支持多种源程序导入方式且附有使用说明



3.含有有错误报告分析以及标识符、运算符、全符号表



4.可生成四元式



5.代码部分

```
StartWindow::StartWindow()
{
    this->setFixedSize(1280,800);

    this->setWindowTitle("Lexical_analyse 开始界面");
```

```
QLabel *label1=new QLabel;
label1->setParent(this);
QFont font1;
font1.setFamily("宋体");
font1.setPointSize(30);
QString str1=QString("编译原理作业——程序展示");
label1->setFont(font1);
label1->setText(str1);

label1->setGeometry((this->width()-30*str1.size())/3,100,this->width(),50);

QString str2=QString("2152118 史君宝");
QLabel *label2=new QLabel;
label2->setParent(this);
QFont font2;
font2.setFamily("宋体");
font2.setPointSize(26);
label2->setFont(font2);
label2->setText(str2);
label2->setGeometry(800,300,this->width()-800,50);

QString str3=QString("2151638 林天野");
QLabel *label3=new QLabel;
label3->setParent(this);
QFont font3;
font3.setFamily("宋体");
font3.setPointSize(26);
label3->setFont(font3);
label3->setText(str3);
label3->setGeometry(800,375,this->width()-800,50);


QString str4=QString("2154062 赵书玥");
QLabel *label4=new QLabel;
label4->setParent(this);
QFont font4;
font4.setFamily("宋体");
font4.setPointSize(26);
label4->setFont(font4);
label4->setText(str4);
label4->setGeometry(800,450,this->width()-800,50);


QPushButton * startBtn=new QPushButton ("");
startBtn->setParent(this);
startBtn->setFixedSize(200,200);

QString str5=QString("START");
QFont font5;
font5.setFamily("宋体");
font5.setPointSize(30);
```

```
    startBtn->move(this->width()*0.5-100,this->height()-200);
    startBtn->setText(str5);
    startBtn->setFont(font5);


    connect(startBtn,&QPushButton::clicked,[=](){
        mainwindow = new MainWindow();
        mainwindow->setGeometry(this->geometry());
        this->hide();
        mainwindow->show();
    });
}
```

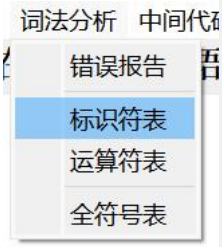
四、结果输出

以下均用 6.txt 输入作为测试样例

```
#include<iostream>
int main()
{
    int a;
   a = 3 ;
}
```

4.1 文档导入

4.2 词法分析



提示栏

```
标识符表;
字符值:a        符号类别:标识符
字符值:return   符号类别:标识符
字符值:b2b      符号类别:标识符
字符值:c        符号类别:标识符
字符值:i        符号类别:标识符
字符值:j        符号类别:标识符
字符值:str      符号类别:标识符
```

提示栏

```
符号表;
符号值:#        符号类别:Sign
符号值:(        符号类别:Sign
符号值:)        符号类别:Sign
符号值:;        符号类别:Sign
符号值:<        符号类别:Sign
符号值:=        符号类别:Sign
符号值:>        符号类别:Sign
符号值:{        符号类别:Sign
符号值:}        符号类别:Sign
符号值:+=       符号类别:Sign
符号值:-        符号类别:Sign
符号值:-=       符号类别:Sign
符号值:/        符号类别:Sign
符号值:/=       符号类别:Sign
符号值:;        符号类别:Sign
符号值:<        符号类别:Sign
符号值:<<       符号类别:Sign
符号值:<=       符号类别:Sign
符号值:=        符号类别:Sign
符号值:==       符号类别:Sign
符号值:>        符号类别:Sign
符号值:>=       符号类别:Sign
符号值:>>       符号类别:Sign
符号值:{        符号类别:Sign
符号值:|        符号类别:Sign
符号值:||       符号类别:Sign
符号值:}        符号类别:Sign
```

4.3 中间代码生成

提示栏

```
生成四元式结果:

100 (INT   ,a    ,_    ,_    )
101 (=     ,a    ,3    ,_    )
102 (FINISH ,_    ,_    ,_    )
```

4.4 错误输出

测试样例 1：

```
#include <iostream>
int main()
{
%
}
```

错误报告 1：

提示栏

```
词法分析过程出错
错误信息如下所示
不合法字符      %      位置：4行1列
```

测试样例 2:

```
#include<iostream>
int main()
{

}
```

错误报告 2:

五、更为通行的高级语言的语义检查

5.1 MSVL 程序设计语言

MSVL 语言，是一种时序逻辑程序设计语言，可用于软硬件系统的建模、仿真，验证。MSVL 语言与 C 语言在描述上基本相似，在结构上相较于 C 语言，多了投影语句、并行语句、随机选择语句等复杂结构语句。MSVL 以投影时序逻辑（PTL）为基础， 将其与常规程序设计语言结合在一起，使其不仅可以解决传 统时序逻辑编程中出现的并发程序设计问题、同步通信问题和框架化问题， 于其他的时序逻辑程序设计语言相比，MSVL 具有更强的表达能力。

5.2 LLVM

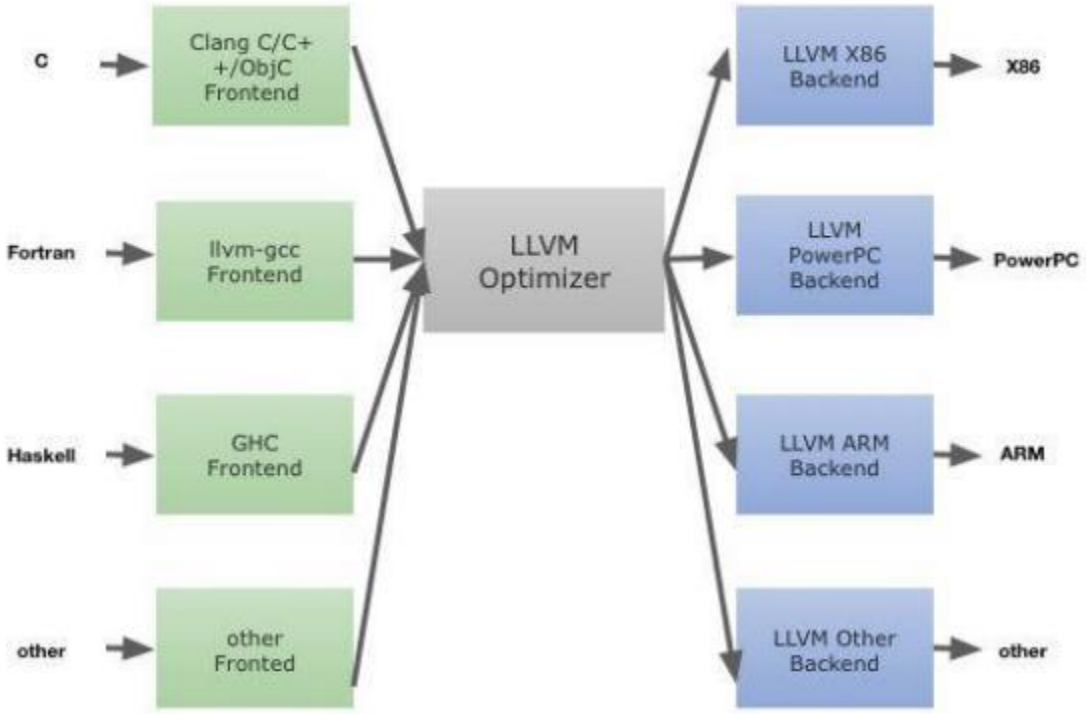
5.2.1 LLVM 简述

LLVM 是 Low Level Virtual Machine 底层虚拟机)的缩写。LLVM 通过在 编译时、链接时、运行时和运行之间的空闲时间为编译器转换提供高级信息，从 而支持对任意程序进行透明的、终身的程序分析和转换。 LLVM 以静态单赋值（SSA）的形式定义了一种常见的低级代码表示形式，具有如下特性：一个简单的、与语言无关的类型系统，它公开了通常用于实现高级语言特性的原语；用于输入地址运算的指令； 一个简单的机制， 可以用来实现高级语言的异常处理特性。

5.2.2 LLVM 体系结构

在 LLVM 项目之前，普通的编译器结构通常分为前端、优化和后端。前端 部分首先对源码进行分析， 确定其没有错误， 最后将分析过的源码转换成抽象语 法树； 优化部分主要是通过对语法树进行大量的等价转换，从而提高效率；后端 部分则根据不同的机器生成与之相对应的机器代码。由于语言的不同，需要为各种语言开发不同的前端、优化与后端，重用性较低，有时甚至无法重用。

LLVM 为了简化编译器开发流程， 将这三个阶段分离开来，用统一的中间语 言 IR 将这三个阶段串联起来。 这样就可以无视输入语言的不同带来的问题，只需要设计一个通用的优化器，对于后端模块来说，把 IR 作为它的输入，就能为特定的硬件平台开发位移的后端代码生成器。 因此要使用 LLVM 框架开发编译器，需要开发的仅仅是一个输入为 IR 的前端，其他的模块可以不用考虑。LLVM 编译器的结构如下图:



5.2.3 中间代码 IR

LLVM 中间表示代码 IR 是一种静态单赋值语句，IR 代码是一种类似于汇编代码的语言，但它又像其他程序设计语言一样具有函数和语句块。IR 代码不仅能够详细地展示出计算机底层，而且有着良好的可读性。IR 不仅被用于优化器的中间层分析和转换，还被设计用于过程间优化、运行时优化、程序全局分析等。

IR 代码是 LLVM 代码的一种表现形式，LLVM 代码主要有三种表现形式，通常根据它们的存储形式以及存储位置来区分这三种形式。第一种是以二进制的形式存储在内存上的 IR 代码，被称为中间表示 IR，由于其存储在内存上，编译器可以直接、迅速地对它进行分析和处理。第二种通常是以文件的形式来表示，文件存储在磁盘上，代码同样以二进制的形式写入，被称为字节码，在编译期运行时，可以将文件直接读入内存，非常方便。第三种同样为文件的形式，但文件内容是文本，被称为可读的 IR 代码。

5.3 64 位MSVL 编译器整体架构

5.3.1 语义分析

语义分析要对语法分析产生的语法树进行处理，通过语法分析得到的语法树能够完成对上下文无关语言部分的处理。但是对于那些语法规则涉及不到的语言部分无法处理，而语义分析就是要得到这些语句所代表的含义，根据它们的含义来对它们进行转换。语语法分析相比，语义分析更多的是关注语句自身所代表的含义而不是程序的合法性。语义分析主要实现两个功能，静态语义检查和语法树的转换。静态语义检查包含以下情况:

① 类型检查: 类型检查通过将变量、常量以及其他类型存储到一个符号表中来实现。当发现一个操作符跟与它对应的操作数之间的类型无法匹配时，编译器会直接输出错误信息。

② 一致性检查: 一致性检查首先要在每个变量定义或声明时通过查询符号表判断其是否已经被定义，还要检查在同一个作用域中是否包含着同名变量，检查选择分支控制语句的分支是否完全相同，若出现相同，编译器发出警告。

③ 控制检查: 控制检查要检查苏偶偶的控制流语句，保证它们可以正确地进行程序的跳转。如果无法正确的跳转，发出警告或报错。

④ 名字检查: 名字检查要对那些具有相同变量名或代码结构的程序进行一致性检查，若发生冲突则报错。

⑤ 区间长度检查: 对与规定长度的 MSVL 语句进行区间长度检查。有一些时序操作符会规定该语句的区间长度，不满足这些操作符的规定无法通过区间长度检查，编译器提示错误。若要进行区间长度检查，首先需要得到该语句的区间长度，对于赋值语句、顺序语句、skip 语句来说，能够直接得到其区间长度，但对于循环语句、分支语句、函数调用语句来说，只有在运行程序后才能够得到其区间长度。

5.3.2 IR 代码生成

编译器要完成目标代码的生成通常会先生成中间代码，也可以直接将源程序翻译为目标程序。中间代码，顾名思义，是位于目标代码与源代码之间的一种全新格式的代码，具有简单的结构、明确的含义，相对于源程序来说更容易被翻译成目标代码。

中间代码的形式有很多，比较常用的有三地址码、后缀表示法等。LLVM 框架所使用的中间代码为三地址码的形式，被称为 IR 代码。IR 代码是一个低级 RISC 类的虚拟指令集，是 LLVM 框架定义的一种有着明确语义的语言，支持轻量级运行时优化以及过程间优化。

在 IR 代码生成模块中，要用 IR 代码构建应用程序接口（API）。因为 LLVM 是通过 C++开发完成的，是一种面向对象的设计，因此 API 都是通过类和类内成员函数的形式提供。IR 代码有着与高级编程语言相似的结构与可读性。IR 代码生成模块将输入的语法树转换为相应的 IR 代码，随后要对语句进行类型检查，提供使用 LLVM 提供的类型系统可以方便快捷地实现类型检查功能。生成 IR 代码后将其存储在内存中，直到将所有的 IR 代码生成完成，将其以一种可读的 IR 格式写入文件并存入硬盘模块类是一种存储 IR 代码的容器，通常需要将 module 的元数据和 IR 代码存入容器中。IR 代码主要以全局变量的定义和声明、自定义类型的定义、函数的定义和生命为主。

5.4 64 位MSVL 编译器的关键问题

5.4.1 中间代码的生成

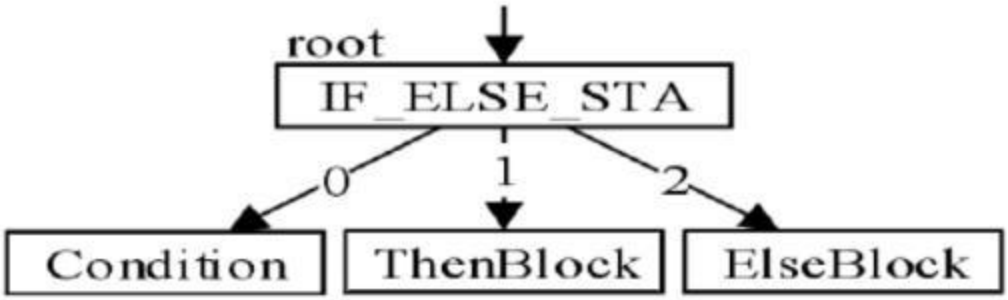
中间代码生成是编译器开发中很重要的一个模块，它是编译器前端的最后一部分，要将之前所有分析结束生成的语法树转换为中间代码。在对 MSVL 编译器的开发中，中间代码生成模块要完成 IR 代码的生成，实际上是要将 MSVL 程序转为 IR 代码。为此，需要将程序中包含的大量的数据类型、语句类型以及运算类型全部转换。例如，结构体、数组、循环语句、分支语句、AND 操作、OR 操作等。

5.4.2 构建IR 代码

要将 MSVL 程序转换为 IR 代码，主要是对通过词法分析、语法分析、语义分析处理后生成的 MSVL 语法树进行转换。在前期的语法分析语义分析模块将根据给定的规则将 MSVL 程序生成四类语法树：分别是自定义类型树、全局变量树、函数树和主语法树。处理顺序依次为自定义类型树、全局变量树、函数树和主语法树。对于主语法树，对其进行深度优先遍历，从最左的叶子节点开始遍历，直到整个转换完成。对简单的语句，例如算术运算、位运算与逻辑运算等，都可以将其直接翻译成 IR 代码。主要难点在于对函数调用语句、分支语句、循环语句的翻译。

① 分支语句的转换

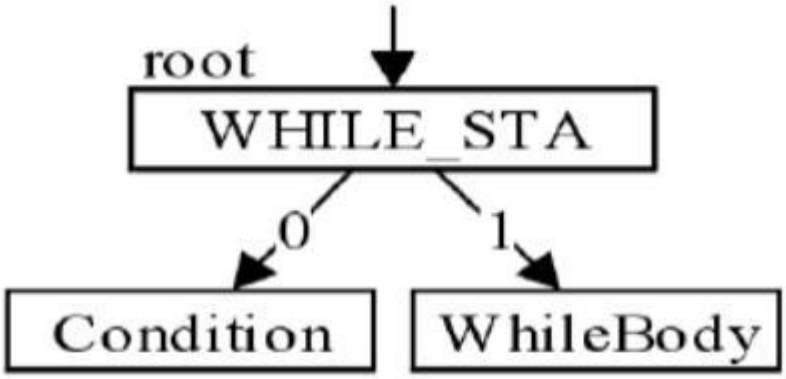
MSVL 的分支语句格式为 if(b) then{p} else{q}。对于分支语句语法树的转换，根据其语法树结构，根节点产生三个后继节点，从左到右分别是条件表达式产生的语法树，条件语句 then 产生的语法树以及条件语句 else 产生的语法树，结构如下图所示:



此部分通过 IR 类中的成员函数 If2IR 来实现， 基本过程为：创建一个基本块 ThenBB，并通过 Cond2IR 函数将条件表达式转换为 IR 指令及得到它的返回值 v；随后创建另外两个基本块 ElseBB 和 IfEnd，并通过判断 v 的值的真和假来跳 转到相应的基本块。 在每个模块完成后会将其翻译完成的 IR 语句插入到相应的 模块中。并在每个模块中添加了无条件跳转指令使其可以跳转到结束快 IfEnd， 从而完成分支语句的转换。

② 循环语句的转换

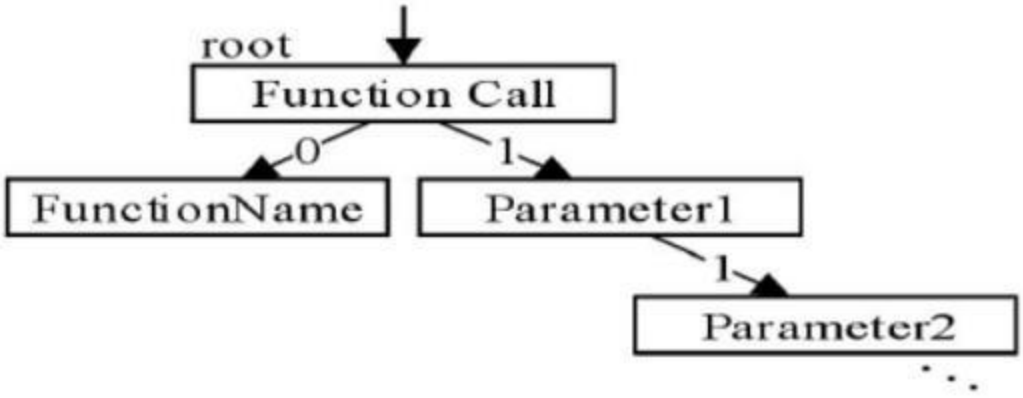
MSVL 的循环语句格式为 while(b) {p}。对于循环语句的语法树转换，根据 其语法树的结构， 其根节点会产生两个后继节点， 分别是循环条件表达式产生的 语法树以及循环体产生的语法树，具体结构如下图所示：



该转换会通过 IR 类中的 While2IR 成员函数来实现。与分支语句的转换代码 类似， 首先得到当前需要转换的函数， 随后创建一个基本块 WhileConBB，并且 为其创建转换指令，将转换后的 IR 代码添加到 WhileConBB 块中，然后通过之 前介绍过的 Cond2IR 函数得到循环条件表达式的值 v，继续创建另外两个基本 块。

③ 函数调用语句的转换

MSVL 的内部函数调用语句格式为 $\mathrm{fun}(x_1, x_2, \ldots x_n)$。对于函数调用语句的 语法树转换， 根据其语法树结构， 根节点有两个后继节点， 左边的节点代表的是 函数名， 右边的节点代表的是函数参数树， 当函数含有多个参数时， 函数参数语 法树将类似于二叉树，即其第二个孩子节点是其兄弟节点。特别的是， 函数名有 可能是函数指针或返回值为函数指针的表达式， 函数调用语句语法树结构如下图 所示：


root
Function Call
0
1
FunctionName
Parameter1
1
Parameter2
. . .