

同濟大學

TONGJI UNIVERSITY

《编译原理课程设计》

实验报告

实验名称	LCC -类 C 编译器
姓 名	张伯阳 1551265
学院（系）	电子信息与工程学院
专 业	计算机科学与技术
指导教师	卫志华
日 期	2018 年 4 月 5 日

一、前言

1. 程序功能描述

LCC 编译程序分为两个版本，后台分别是面向 x86 和 MIPS 语言，其中 x86 的汇编代码是基于由 GNU 团队设计的 AT&T x86 汇编代码，而不是 intel 的 8086，在下文不会再作解释，文中所有的 x86 都是指 AT&T。LCC 编译程序达到了目标代码生成部分，运行方式为通过命令行的方式输入命令，然后 LCC 会输出汇编代码文件(.s)到目录下，输出的目标代码分别是 AT&T x86 和 MIPS 语言。本编译程序面向的高级程序设计语言为 C 语言，但其中将很多的 C 语言的语法精简了，故准确地应该是一种面向基于 C 语言的类 C 语言。

2. 词法规则

关键字: int | void | if | else | while | return 标识符: 字母(字母|数字)* (注: 不与关键字相同) 数值: 数字(数字)* 赋值号: = 算符: + | - | * | / | = | == | > | >= | < | <= | != 界符: ; 分隔符: , 注释号: /* * / // 左括号: (右括号:) 左大括号: { 右大括号: } 字母: a | ... | z | A | ... | Z 数字: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 结束符: #

3. 语法规则

Program ::= <声明串>

<声明串> ::= <声明> { <声明> }

<声明> ::= int <ID> <声明类型> | void <ID> <函数声明>

<声明类型> ::= <变量声明> | <函数声明>

<变量声明> ::= ;

<函数声明> ::= '(<形参>)' <语句块>

<形参> ::= <参数列表> | void

<参数列表> ::= <参数> { , <参数> }

<参数> ::= int <ID>

<语句块> ::= '{ <内部声明> <语句串> }'

<内部声明> ::= 空 | <内部变量声明> { ; <内部变量声明> }

<内部变量声明> ::= int <ID>

<语句串> ::= <语句> { <语句> }

<语句> ::= <if 语句> | <while 语句> | <return 语句> | <赋值语句>

<赋值语句> ::= <ID> = <表达式>;

<return 语句> ::= return [<表达式>] (注: [] 中的项表示可选)

<while 语句> ::= while '(<表达式>)' <语句块>

<if 语句> ::= if '(<表达式>)' <语句块> [else <语句块>] (注: [] 中的项表示可选)

<表达式> ::= <加法表达式> { relop <加法表达式> } (注: relop-><=>|>|==|!=)

<加法表达式> ::= <项> { + <项> | - <项> }

<项> ::= <因子> { * <因子> | / <因子> }

<因子> ::= num | '(<表达式>)' | <ID> FTYPE

FTYPE ::= <call> | 空

<call> ::= '(<实参列表>)'

<实参> ::= <实参列表> | 空

$\langle \text{实参列表} \rangle ::= \langle \text{表达式} \rangle \{, \langle \text{表达式} \rangle\}$

$\langle \text{ID} \rangle ::= \text{字母}(\text{字母} | \text{d 数字})^*$

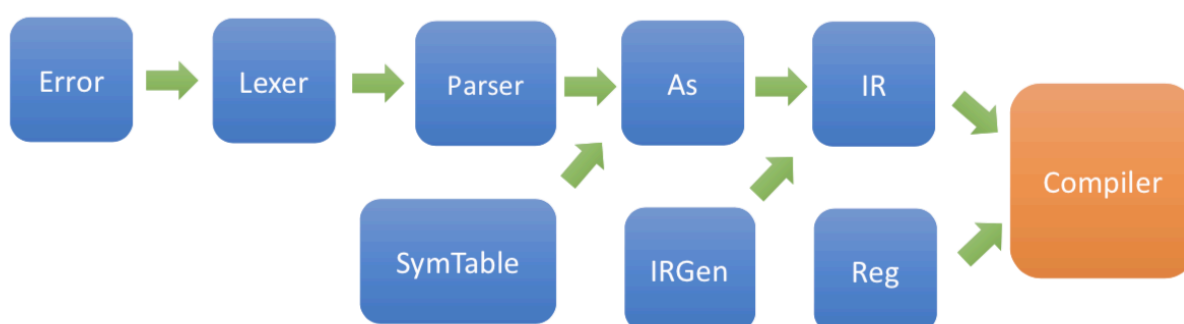
二、 LCC 编译程序的基本模块

1. LCC 编译程序框架

LCC 编译程序的设计方法为面向对象法设计，采用类的分块与类的继承进行设计。

LCC 编译程序每一模块都是一个类，采用类的方式设计的好处是方便单独的地调试该模块。当调试模块完成后，可以通过类的继承把每个类串联起来，从而构成 lcc 编译程序的总体架构。

根据类的继承关系，LCC 编译程序类的框架图如图所示



其中：

- class Error 处理出错时将信息输出到 stderr；
- class Lexer 对输入文件进行词行分析；
- class Parser 对单词进行语法分析构成语法树；
- class SymTable 处理符号表的相关操作；
- class As 对语法树进行语义分析；
- class IRGen 处理生成中间代码、以及基本块的操作
- class IR 为中间代码生成器
- Reg 为寄存器管理类
- Compiler 为编译程序，输出汇编代码

2. 内存管理

较复杂 C 或 C++ 程序，通常会根据自身特式，制定对应的内存管理，但由于 LCC 编译程序，面向的语法较为简单，运行时间短、且需要的内存空间容量不大，足够堆

空间的分配，不会造成什么系统问题，在其进程结束时，操作系统会回收相关的内存。因此，考虑到开发时间的问题，暂时不设计，在日后需要时可以设计一个内存分配管理的类，但暂时并不设计。

但考虑到日后改进问题，这里暂时给出针对 LCC 编译程序，对应内存分配管理类的概念图。内存分配管理类 Alloc 的概念图，如图 2.2.1 所示。

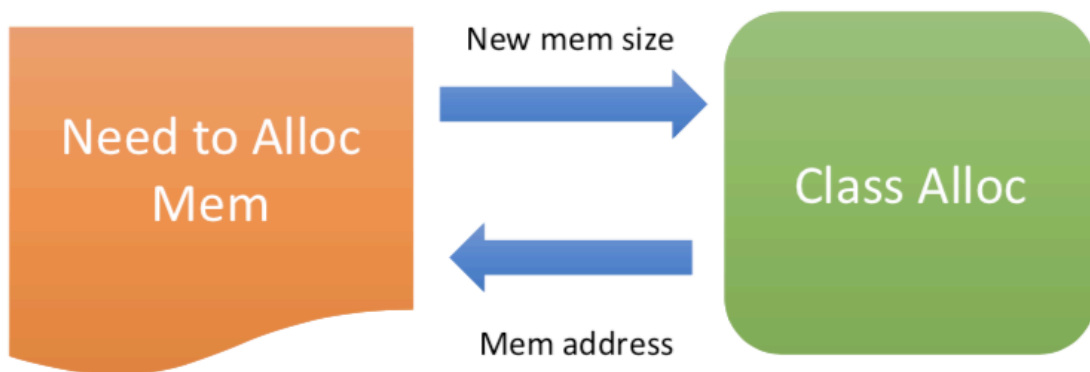


图 2.2.1 Alloc 的概念图

LCC 编译程序内存由若干个 `class mblock` 组成，通过链表的方式将若干个块组成一个单链表。其数据结构示意图如图 2.2.2 所示

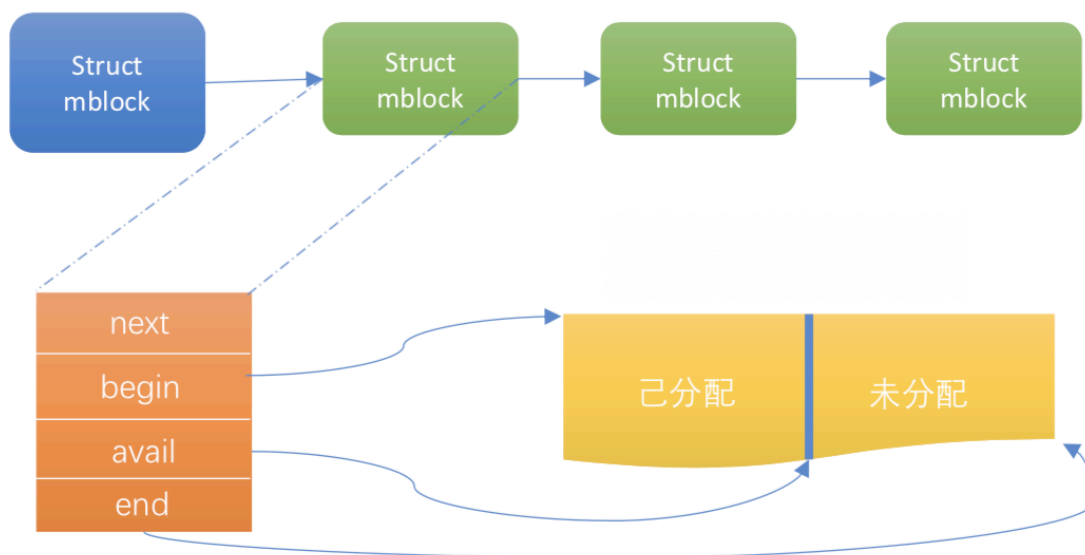


图 2.2.2 class mblock

3. LCC 编译程序的符号管理

在这一节，将初步地介绍 lcc 编译程序的符号表管理，与符号表管理相关的类为

`class SymTable`，代码主要在 `symbol.h` 和 `symbol.cpp` 中。LCC 编译程序内部需要对所有使用到的符号进行分类，并建立相应的数据结构来记录与符号的相关信息。图

2.3 的结构体 `struct symbol` 用于记录与符号的相关信息，这些信息如图 2.3 第 7 行至 17 行所示。第 8 行的 `kind` 用于区别不同类别的符号，其取值范围如图 2.3 第 1 行

至第 5 行所示；第 9 行的 `name` 用于记录符号的名称。其余变量将会在下文解释。

```
enum
{
    SK_Constant, SK_Variable, SK_Temp,
    SK_Label, SK_Function, SK_Register
};

#define SYMBOL_COMMON \
    int kind; \
    string name; \
    string aname; \
    int level; \
    int ref; \
    int val; \
    int line; \
    struct symbol *reg; \
    struct symbol *link; \
    struct symbol *next;

typedef struct bblock *BBlock;

typedef struct symbol
{
    SYMBOL_COMMON
} *Symbol;
```

图 2.3 struct symbol

下面将结合具体例子来说明编译程序内部对符号的分类，如图 2.4 所示

1	int a;	10	function main()
2	int b;	11	-----
3	int main(void)	12	BB0:
4	{	13	a = 3
5	int a;	14	b = 4
6	int b;	15	c = 2
7	int c;	16	t0 = b+c
8	a=3;	17	a = t0
9	b=4;	18	return 0
10	c=2;	19	ret
11	a = b + c;		
12	return 0;		
13	}		

图 2.4 符号示例

图 2.4 的第 1 至 13 行是一个简单的 C 程序，而第 10 至 19 行为对应的中间代码，结合图 2.4 和图 2.3，可以看到以下几种符号：

- 1) 常量 3, 4, 5，对应的类别为 `SK_Constant`;
- 2) 全局变量 `a, b`，和局部变量 `a, b, c` 对应的类别为 `SK_Variable`;
- 3) 临时变量 `t0` 对应的类别为 `SK_Temp`;

上文中所有的符号类型正是图 2.3 第 1 行至第 5 行所示。值得注意的是，LCC 编译程序会把寄

寄存器名也当作符号来管理，对于 x86 形如 “%eax”，而对于 MIPS 形如 “\$a0”，对应的分类类型为图 2.3 中的 SK_Register。这 7 种符号中，有一些符号的相关信息用 struct symbol 对象就可以记录，如 SK_Register，但有一些符号，则需要记录更多的信息，如 SK_Variable、SK_Function，就需要再扩展对象的结构了。

如图所示。

```
typedef struct variableSymbol
{
    SYMBOL_COMMON
    int idata;
    int offset;
} *VariableSymbol;
```

在图中可以看到 SYMBOL_COMMON，因此，其实可以把 struct variableSymbol 看作为 struct symbol 的子类。图 2.5 的 reg 指的是当前变量存放在哪一个寄存器中，因此变量可以通过 Symbol reg 指针知道当前哪一个寄存器存放了它的值。

现在，可能会有一个问题，到目前为止还不知道 struct symbol 的对象及其子类存放的在哪里。因此，这里引入了 struct functionSymbol 以及 struct table 两个结构体。如图所示。

```
typedef struct functionSymbol
{
    SYMBOL_COMMON
    vector<Symbol> params;
    vector<Symbol> locals;
    int nbblock;
    BBlock entryBB;
    BBlock exitBB;
} *FunctionSymbol;

typedef struct table
{
    vector<Symbol> buckets;
    int level;
    struct table *outer;
} *Table;
```

图 2.6 中第 1 行至 10 行的 struct functionSymbol 用来描述 SK_Function 类别的符号的相关信息。第 3 行的 param 记录了函数的形参，而第 4 行的 local 记录了函数的局部变量，第 6 行的 nbblock 记录了函数中基本块的个数，第 7 行 entryBB 记录了函数的第一个基本块入口，对应地第 8 行的 exitBB 对应函数基本块出口。图 2.7 表示了 struct functionSymbol 的逻辑结构。

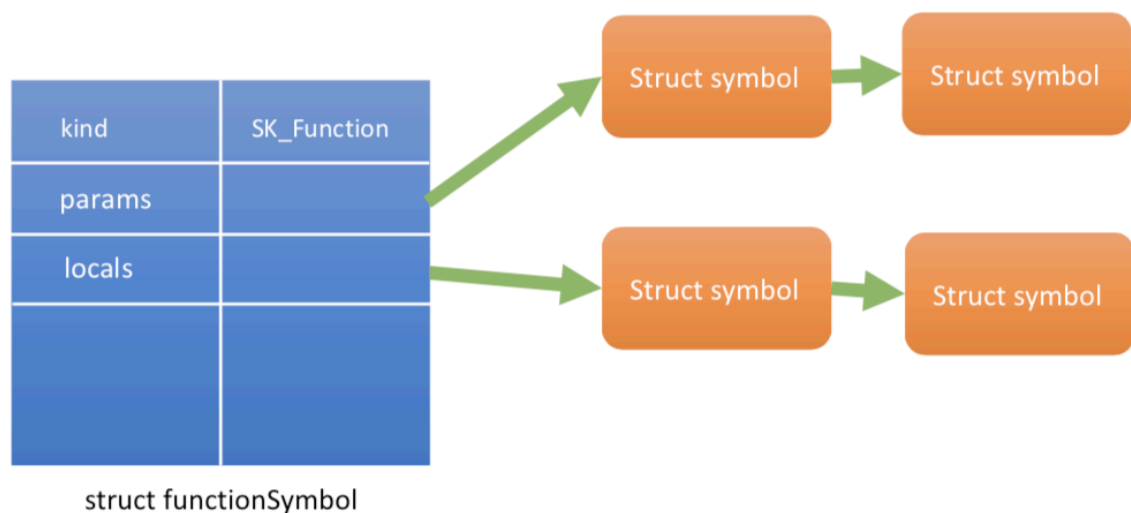


图 2.7 struct functionSymbol 逻辑示意图

但是，struct functionSymbol 没有解决大量的 struct symbol 对象存放的问题。因此，图 2.6 中的 struct table 正是用来解决这个问题。通过 AddSymbol 函数，可以将 struct Symbol 加入到当前的符号表 table 中。如图所示。

```
Symbol SymTable::AddSymbol(Table tbl, Symbol sym)
{
    sym->level = tbl->level;
    tbl->buckets.push_back((Symbol)sym);

    return sym;
}
```

由于同一个变量名可以在不同作用域中被多次声明，我们需要把同一个符号加入到不同的符号表中，因此 struct table 中的层数正是解决这个问题，看以看到图 2.6 中 struct table 的成员变量 outer。以下通过一个简单的 C 程序，说明一下 outer 的作用。

```
1 //深度 level is 0
2 int a = 0;
3 int main()      //level is 1
4 { // level is 2
5     int a;
6     { // level is 3
7         int a;
8         a = 40;
9     }
10    a = 50;
11    return 0;
12 }
13
```

图 2.9 C 语言作用域

如图 2.9 所示，第 2 行定义了一个全局变量 a，第 5 行定义了一个同名的局部变量 a，而第 7 行同样定义了一个变量 a。由 C 的文法，复合语句以左大括号开始，之后跟着若干个声明，最后右大括号。在 C 语言中，函数实际上是一个复合语句。

每个复合语句对应一个新的作用域，每当进入一个新的作用域，就会创建一张新的符号表 **table**，用于记录在该作用域中声明的符号。图 2.10 给出了编译程序为图 2.9 所创建的符号表。

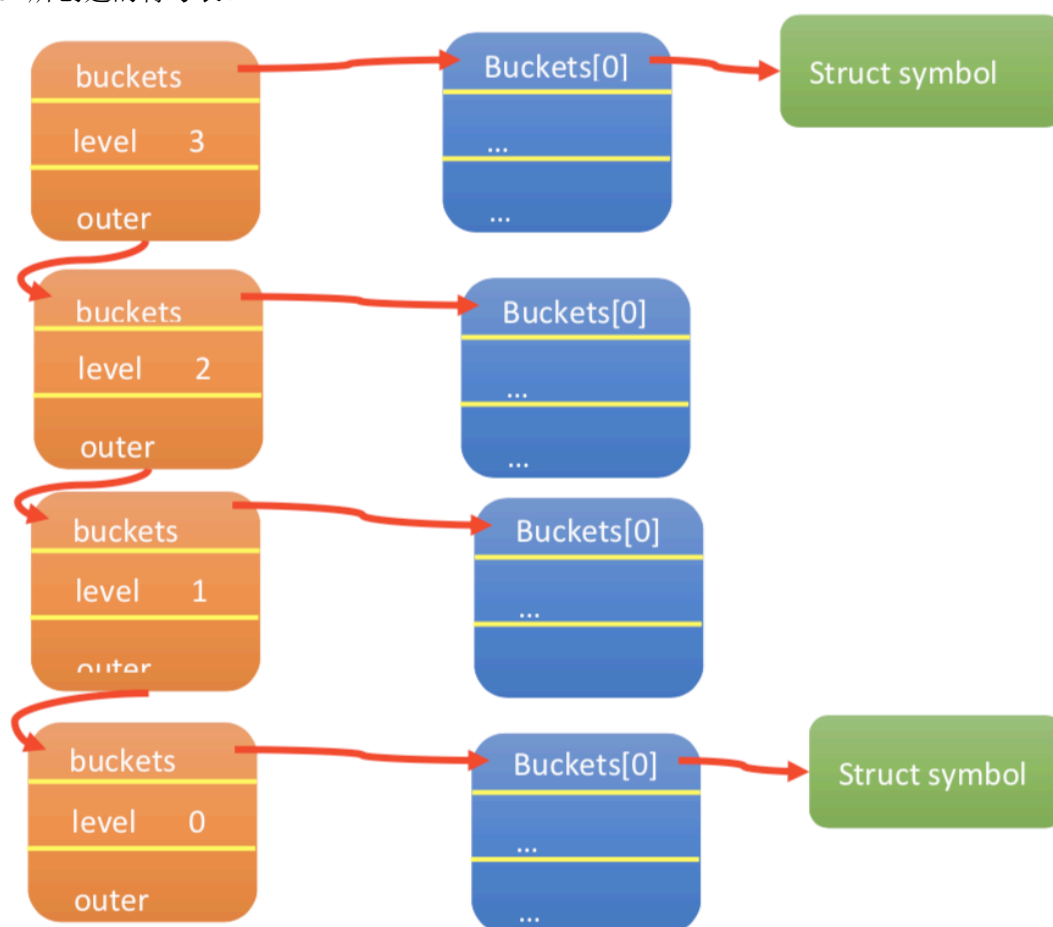


图 2.10 多个作用域的符号表

由图 2.10，可以看出，图 2.9 第 2 行的符号 **a** 存放在深度为 0 的符号表中，而深度为 1 的符号表中，我们没有定义局部变量。图 2.9 第 5 行的符号 **a** 存放在深度为 2 的符号表中，而第 7 行的符号 **a** 则存放在深度为 3 的符号表中。当程序执行到图 2.36 的第 8 行时，当前符号为 **level** 为 3。我们首先检查当前符号表中是否存在 **a**，若没有，则通过 **outer** 指针查找外层的符号表。通过多级符号表，实现了 C 语言“作用域”的概念。

三、词法分析

1. 词法规则

关键字: `int | void | if | else | while | return`

标识符: 字母 (字母|数字)* (注: 不与关键字相同)

数值: 数字 (数字)*

赋值号: `=`

算符: `+ | - | * | / | = | > | >= | < | <= | !=`

界符: `;`

分隔符: `,`

注释号: `/* */ | //`

左括号: `(`

右括号: `)`

左大括号: `{`

右大括号: `}`

字母: `|a|...|z|A|...|Z|`

数字: `0|1|2|3|4|5|6|7|8|9|`

结束符: `#`

词法规则在第一章有提及过, 这里作进一步的分析, 词法规则可以分为 8 类别

- 注释
- 数据类型
- 符号类别
- 表达式
- 括号
- 关键词
- 比较符
- 运算符

与词法分析相关的文件为 `lexer.h` 和 `lexer.cpp`。词法分析器的主要目的是读入文件中的内容, 然后分析每个词, 并为它们贴上标签。每一个词分析后都会产生一个对象 `struct Word`, 每一个 `struct Word` 代表一个词的分类记号, 记录对应的词的分类以及相关信息, 这些信息如图如图 3.1 所示, `struct Word` 的结构示意图。

```

enum TSymbol {
    $LCMT = 1, $RCMT, $CMT,
    $INT, $VOID,
    $ID, $NUM,
    $SEMICOLON, $COMMA, $ASSIGN,
    $LPAREN, $RPAREN, $LCURLY, $RCURLY,
    $RETURN, $WHILE, $IF, $ELSE,
    $LE, $LT, $GE, $GT, $EQUAL, $NEQUAL,
    $PLUS, $MIN, $STAR, $SLASH,
    $FINISH
};

typedef map<TSymbol, string> SymbolList;

typedef struct Word {
    TSymbol t_symbol; //symbol type
    string token;
    int wordline;
} Word;

```

由图知，每个 Word 的有对应的类别，对于图第 14 行，第 15 行的 token 代表单词的字符串，第 16 行的 wordline 为对应文件中的行数。第 14 行对应的类型为图的第 1 行至 11 行的内容，对应语法树的终结符号。

2. 遍历分析

在前文提及到，由于 LCC 编译程序是当过类的继写编写的，因此，词法分析器也示例外，在 lexer.h 和 lexer.cpp 中定义了 class Lexer 的声明。如图所示。

```

class Lexer : protected Error {
protected:
    vector<Word> wordList;
    SymbolList symbolList;
    void initializeSymbolList();

private:
    int _keyword(string);
    int _id(string);
    int _num(string);
    int _operator(string);
    int _delimiter(string);
    int _digit(char);
    int _letter(char);

    Word createWord(TSymbol, string);
    string::iterator iter;
    int line;

public:
    void analyze(string input);
    void printResult();
};

```

由图可知，class Lexer 是继承自 class Error，因为错误信息是由分析开始出现 的。所以通过继承 class Error 可以将错误的信息交给 class Error 来做。在图的第 3 行为一个 vector 容器，wordlist，存放的类型为图的 struct Word，通过分析后的 Word 都放到 class Lexer 的 wordlist 中，然后通过类的继承，继承的类可以

访问分析后的 wordlist; 第 4 行 SymbolList 和第 5 行 initializeSymbolList()为一个 map<TSymbol, string>, 目的是方便检测分析后的结果是否正确; 第 7 行至第 13 行是用来判断输入字符串是否为该类别, 这里给出_id()来说明, 如图所示。

```
int Lexer::_id(string input)
{
    string::iterator iter = input.begin();

    if (_letter(*iter)) {
        for (++iter; iter != input.end(); ++iter) {
            if (_digit(*iter) || _letter(*iter)) {
                ;
            }
            else {
                return 0;
            }
        }
    }
    else {
        return 0;
    }

    return 1;
}
```

由图可知, 函数遍历通过函数的参数 input, 判断该输入是否为正确的变量形式。第 5 行通过调用_letter()判断第一个是否为字母, 若是则继续分析; 若不是则返回 0。通过继续的遍历变量 input, 判断变量 input 的每一个字母是否都为字母或数字, 若有一个不符合直接返回 0, 直至遍历完成, 返回 1。

通过以上的分析可知图第 7 行至第 13 行的各个函数返回大于 0 时, 为判断正确可以生成 struct Word。通过调用 Lexer::createWord()来生成 struct Word, 然后将它放入 wordList 中。

```
Word Lexer::createWord(TSymbol symbol, string token)
{
    Word word;
    word.t_symbol = (TSymbol)symbol;
    word.token = token;
    word.wordline = line;
    return word;
}
```

在图的第 4 行通过分析后的类型，生成对应的终结符，并将该字符串放到 `token` 中，第 6 行为保存该字符在文件中的行数，其中 `line` 为在 `Lexer` 的私有变量。

以上都是介绍了一些分析过程必要的工具，下面介绍与外界接口的 `analyze()`，`analyze()`是整个 `class Lexer` 的核心，编译程序通过调用 `Lexer::analyzer()`来对输入字符进行分析。

```
void Lexer::analyze(string input)
{
    initializeSymbolList();

    string token;
    iter = input.begin();
    int annotation = 0;

    //the first line
    line = 1;

    for (; iter != input.end(); ++iter) {

        if (*iter == ' ' || *iter == '\t' || *iter == '\n' || *iter == '\r')
        {
            if (*iter == '\n')
                line = line + 1;

            if (*iter == '\n' && annotation == 1)
                annotation = 0;
            continue;
        }

        token += *iter;

        if (token == "#")
        {
            Word word = createWord($FINISH, token);
            wordList.push_back(word);
            token.clear();
        }
        else if (token == "//")
        {
            Word word = createWord($CMT, token);
            wordList.push_back(word);
            token.clear();
            annotation = 1;
        }
        else if (token == "/*")
        {
            Word word = createWord($LCMT, token);
            wordList.push_back(word);
            token.clear();
            annotation = 2;
        }
        else if (token == "*/")
        {
            Word word = createWord($CMT, token);
            wordList.push_back(word);
            token.clear();
            annotation = 0;
        }
    }
}
```

```

else if (int sym = _delimiter(token)) {
    Word word = createWord((TSymbol)sym, token);
    if (!annotation)
        wordList.push_back(word);
    token.clear();
}
.....
}

```

图的第 6 行是标记着当前的字符是否需要加入 wordlist, 若 annotation 不为 0, 则不加入, 如第 31 行至 32 行所示。若遇到符号“//”时, 则将 annotation 置为 1, 直到遇到换行号才把 annotation 置 0; 若遇到“/*”, 则将 annotation 置为 2, 直到遇到“*/”才将 annotation 置为 0。通过调用判断函数, 即图中的 7 行到 13 行函数, 若判断正确且 annotation 为 0, 才把新增一个 Word 并把它放入 wordList 中; analyze 一直进行遍历, 直到遇到“#”结束符, 整个分析才结束。

图中第 3 行的初始化函数在前文中有提及, 只是初始化 symbolList[], 方便 printResult() 的打印, 便于调试。

```

void Lexer::initializeSymbolList()
{
    symbolList[$LCMT] = "$LCMT";
    symbolList[$RCMT] = "$RCMT";
    symbolList[$CMT] = "$CMT";
    symbolList[$INT] = "$INT";
    symbolList[$VOID] = "$VOID";
    symbolList[$ID] = "$ID";
    symbolList[$NUM] = "$NUM";
    symbolList[$SEMICOLON] = "$SEMICOLON";
    symbolList[$COMMA] = "$COMMA";
    symbolList[$ASSIGN] = "$ASSIGN";
    symbolList[$LPAREN] = "$LPAREN";
    symbolList[$RPAREN] = "$RPAREN";
    symbolList[$LCURLY] = "$LCURLY";
    symbolList[$RCURLY] = "$RCURLY";
    symbolList[$RETURN] = "$RETURN";
    symbolList[$WHILE] = "$WHILE";
    symbolList[$IF] = "$IF";
    symbolList[$ELSE] = "$ELSE";
    symbolList[$LE] = "$LE";
    symbolList[$LT] = "$LT";
    symbolList[$GE] = "$GE";
    symbolList[$EQUAL] = "$EQUAL";
    symbolList[$NEQUAL] = "$NEQUAL";
    symbolList[$PLUS] = "$PLUS";
    symbolList[$MIN] = "$MIN";
    symbolList[$STAR] = "$STAR";
    symbolList[$SLASH] = "$SLASH";
}

```

四、 语法分析

1. 语法规则

```
Program ::= <声明串>
<声明串> ::= <声明> { <声明> }
<声明> ::= int <ID> <声明类型> | void <ID> <函数声明>
<声明类型> ::= <变量声明> | <函数声明>
<变量声明> ::= ;
<函数声明> ::= ' ( ' <形参> ' ) ' <语句块>
<形参> ::= <参数列表> | void
<参数列表> ::= <参数> { , <参数> }
<参数> ::= int <ID>
<语句块> ::= ' { ' <内部声明> <语句串> ' } '
<内部声明> ::= 空 | <内部变量声明> { ; <内部变量声明> }
<内部变量声明> ::= int <ID>
<语句串> ::= <语句> { <语句> }
<语句> ::= <if 语句> | <while 语句> | <return 语句> | <赋值语句>
<赋值语句> ::= <ID> = <表达式>;
<return 语句> ::= return [ <表达式> ] (注:[ ]中的项表示可选)
<while 语句> ::= while ' ( ' <表达式> ' ) ' <语句块>
<if 语句> ::= if ' ( ' <表达式> ' ) ' <语句块> [ else <语句块> ] (注:[ ]中的项表示可选)
<表达式> ::= <加法表达式> { relop <加法表达式> } (注:relop-> <|<=>|>|=|!=|)
<加法表达式> ::= <项> { + <项> | - <项> }
<项> ::= <因子> { * <因子> | / <因子> }
<因子> ::= num | ' ( ' <表达式> ' ) ' | <ID> FTYPE
FTYPE ::= <call> | 空
<call> ::= ' ( ' <实参列表> ' ) '
<实参> ::= <实参列表> | 空
<实参列表> ::= <表达式> { , <表达式> }
<ID> ::= 字母(字母|d 数字)*
```

从第 4 章开始，本文进入了语法分析阶段，相关的代码主要在在 `parser.h` 和 `parser.cpp`。在本章，通过结合给出的语法规则来进行语法分析，并生成并构建语法树。本章的目的就是给出一棵语法树，供之后的语义分析和中间代码生成使用。因此，需要先熟悉语法树上最基本的结点，语法树由终结点和非终结点构成，终结点的类型为上一章所列出的终结符号，而非终点符号结点的类型，如图所示。

```
enum NTSymbol {
    $Program=1, $DeclBlock, $Declaration, $DeclType, $DeclVar, $DeclFunc, $FparaBlock,
    $FparaList, $Fparameter, $StatBlock, $InnerDecl, $InnerDeclVar, $StatString,
    $Statement, $StatIf, $StatWhile,
    $StatReturn, $StatAssign, $Expression, $ExprArith, $Item, $Factor,
    $Ftype, $Call, $Aparameter, $AparaList, $TerminalSymbol
};

typedef map<NTSymbol, string> NTSymbolMap;

typedef struct TreeNode {
    vector<TreeNode> children;
    NTSymbol nt_symbol; //Non-terminal symbol type
};
```

```

    TSymbol t_symbol; //Terminal symbol type
    string token;
    int line;
    Symbol val;
} TreeNode;

```

第 11 行至第 18 行的结构体 `struct TreeNode` 是语法树中最基本的结点，第 13 行和第 14 行记录结点的类，第 13 行的取值范围由第 1 行至第 9 行的枚举常量来确定；第 12 行为该结点的所有孩子结点，终结点是没有孩子的；第 17 行的 `Symbol val` 为 `Variable` 的符号 `Symbol` 存放地址，在语义分析后加入。

2. 生成语法树

类似地，语法分析器，也有对应的 `class Parser`，在 `parser.h` 和 `parser.cpp` 中定义。

```

class Parser : protected Lexer {
protected:
    TreeNode synTree; // Syntactical Tree
    NTSymbolMap ntSymbolList;
    void initializeNTSymbolList();

private:
    /**
     * Automation Recursive Functions
     * @return 1 - Success; 0 - Error;
     */
    int _program(TreeNode* parent);
    int _declBlock(TreeNode* parent);
    int _declaration(TreeNode* parent);
    int _declType(TreeNode* parent);
    int _declVar(TreeNode* parent);
    int _declFunc(TreeNode* parent);
    int _fparaBlock(TreeNode* parent);
    int _fparaList(TreeNode* parent);
    int _fparameter(TreeNode* parent);
    int _statBlock(TreeNode* parent);
    int _innerDeclar(TreeNode* parent);
    int _innerDeclVar(TreeNode* parent);
    int _statString(TreeNode* parent);
    int _statement(TreeNode* parent);
    int _statIf(TreeNode* parent);
    int _statWhile(TreeNode* parent);
    int _statReturn(TreeNode* parent);
    int _statAssign(TreeNode* parent);
    int _expression(TreeNode* parent);
    int _exprArith(TreeNode* parent);
    int _item(TreeNode* parent);
    int _factor(TreeNode* parent);
    int _ftype(TreeNode* parent);
    int _call(TreeNode* parent);
    int _aparameter(TreeNode* parent);
    int _aparaList(TreeNode* parent);

    void advance();
    void retrack(vector<Word>::iterator it);
    Word word; // Current definite symbol
    vector<Word>::iterator word_it;
    TreeNode createNode(NTSymbol nt_symbol, TSymbol t_symbol = (TSymbol)0, string token="",
int line = 0);
    void insertNode(TreeNode* parent, TreeNode child);

```



```

// void dfsResult(boost::property_tree::ptree* pt, TreeNode n);
public:
    void analyze(string input);
    // boost::property_tree::ptree generateResult();
    // string printResult(); // modify void to string
    void printResult(TreeNode n);
};

```

class Parser 继承自 class Lexer, 通过调用 Lexer 对输入进行词法分析后, 利用分析后的产物 wordlist 进行语法分析, 生成语法树。在图第 3 行为语法树的根结

点, 对应的结点分类为\$Program, 在第 7 行开始的私有函数为语法分析中每一层对应的处理函数, 针对每一层的结构会有不同的处理函数。

LCC 通过回溯法来对词法分析后的产物 wordlist 来进行分析, 生成语法树。分析过程为每次从 wordlist 中拿出一个字符, 这里通过 advance()函数来进行前进。因为采用的方法为回溯法, 故除了前进之外, 还要回溯的, 回溯是通过调用 retrack()来回溯上一个字符。函数的定义如图所示

```

void Parser::advance() {
    while (true) {
        ++word_it;
        if (word_it == wordlist.end()) {
            throw runtime_error("End of word list in parsing.");
        }
        if (word_it->t_symbol == $LCMT || word_it->t_symbol == $RCMT || word_it->t_symbol == $CMT) {
            continue;
        }
        else {
            break;
        }
    }
    word = *word_it;
}

void Parser::retrack(vector<Word>::iterator it) {
    word_it = it;
    word = *word_it;
}

```

图的第 4 行, 设置了异常处理, 若指针移动到 wordlist 的尾部(即界外), 则 throw error, 对于第 7 行, 若遇到的为"//", "/*"和"*/", 则继续从 wordlist 中拿出下一个字符来, 否则拿出当前的字符。在第 17 行与 19 行为 retrack()函数, 由于在分树语法树的过程中都会保留上一次的字符指针, 因此, retrack()函数的主要作用就是改变当前指向 wordlist 的指针。

```

void Parser::analyze(string input) {
    Lexer::analyze(input);

    initializeNTSymbolList();
    TreeNode newTree;
    newTree.nt_symbol = $Program;
    synTree = newTree;

    word_it = Lexer::wordList.begin();
    word = *word_it;
    while (word.t_symbol == $LCMT || word.t_symbol == $RCMT || word.t_symbol == $CMT) {
        word_it++;
        word = *word_it;
    }

    if (!Parser::_program(&synTree)) {
        cout << word.token << endl;
        //throw runtime_error("Syntax error detected.");
    };
}

```

Parser 为类的继承的函数接口，通过从 wordList 中拿出第一个非注释符号来进行生成语法树的操作，第 11 行至 13 行为拿出第一个非注释符号；然后进入下一层遍历的处理。

```

TreeNode Parser::createNode(NTSymbol nt_symbol, TSymbol t_symbol, string token, int line)
{
    TreeNode new_node;
    new_node.nt_symbol = nt_symbol;
    new_node.t_symbol = t_symbol;
    new_node.token = token;
    new_node.line = line;
    return new_node;
}

```

当遍历 wordlist 时判断成功后，通过调用 Parser::createNode() 来进行结点值的设置，包括类型、字符和行数。

五、 语义分析与中间代码生成

1. 功能

这个部分的主要功能就是生成中间代码, 同时检测语义错误, 进行类型检查等等。

我选取四元式作为中间代码表示形式。

2. 算法:

通过在语义分析的过程中插入翻译动作, 得到四元式输出结果。

在开发过程中, 中间代码生成分成了两个部分, 第一个部分是声明语句的翻译, 这个部分要进行符号表的添加和查询, 访问数组元素虽然不是声明语句, 但是也属于这个部分, 原因是动态访问数组元素如 $a[i][j]$, 是无法在编译时得到地址的, 只有运行时才能计算得出地址, 所以计算地址的过程也会成为中间代码输出。

第二个部分是操作语句的翻译, 这个部分相对前一部分比较简单, 需要维护符号栈 SYN 和操作符栈 SEM。要注意的地方是, 考虑中间代码生成的时候还要考虑目标代码生成的地址回填问题, 才能避免出现逻辑错误。

3. 算法流程:

- 1) 开始
- 2) 遇到标识符 PUSH (SEM, arr)
- 3) 遇到左括号, PUSH (SYN, +)
- 4) 遇到操作数 i, PUSH (SEM, i)
- 5) 如果不是第一个左括号则执行 Quat (), 否则跳过
- 6) 遇到右括号, PUSH (SYN,*)
- 7) 遇到操作数 j, PUSH (SEM,j)
- 8) Quat ()
- 9) 如果不是最后一个右括号则跳转到 2, 否则 Quat_a ()
- 10) 结束。

算法中的 Quat 操作即从 SYN 中取出运算符, 从 SEM 中取出两个操作数, 并计算结构, 步骤 9 中的 Quat_a()与这个操作基本类似, 区别在于存放运算结果的方式, 正常情况下运算结果都是临时变量, 然而考虑到目标代码生成, 为了区别于变量, 在计算出数组元素地址后, 并不能把它当做临时变量, 而应该是变量的地址, 也就是说访问这个元素就要对运算结果进行两次取值操作。

六、目标代码生成

1. 汇编代码生成简介

历经词法分析、语法分析、语义分析和中间代码生成阶段，终于来到了“目标代码生成阶段”，由于 LCC 编译程序的目标代码为 32 位的 x86 汇编代码和 MIPS 汇编代码，因此本章也称为“汇编代码生成”。LCC 编译程序分为两个版本，x86 和 MIPS，在此报告中只对 x86 进行介绍，MIPS 的生成本质上与 x86 一样，故这里不作解释。到了这一章，LCC 编译程序面对的输入早已不再是类 C 源代码，而主要是由各个基本块构成的链表，本阶段的目的是要把基本块里的中间代码翻译成 x86 汇编代码。

通过一个简单的例子来了解一下 LCC 编译程序的汇编代码生成，如图 6.1 所示。

```
# Code auto-generated by blank-black
```

```
.section .data
.section .bss
.lcomm a 4
.lcomm b 4
.section .text
.globl _program
_program:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $0, -4(%ebp)
movl -4(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
.globl _demo
_demo:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    movl $4, %ecx
    imull %ecx, %eax
    addl $2, %eax
    movl %eax, 8(%ebp)
movl 8(%ebp), %eax
    movl $2, %ecx
    imull %ecx, %eax
    movl %ebp, %esp
    popl %ebp
    ret
.globl _main
_main:
    pushl %ebp
    movl %esp, %ebp
    subl $12, %esp
    movl $3, -4(%ebp)
movl $4, -8(%ebp)
movl $2, -12(%ebp)
    pushl -12(%ebp)
    call demo
    addl $4, %esp
    pushl %eax
    pushl -8(%ebp)
    pushl -4(%ebp)
```

```

call program
addl $12, %esp
    movl %eax, -4(%ebp)
movl $0, %eax
    movl %ebp, %esp
    popl %ebp
    ret

```

如图，第 1 行是人为的注释，用于说明是由谁生成的。第 2 行至第 5 行的汇编代码。是由 EmitGlobals()函数用于处理 C 语言中定义的全局变量，由于语法规则中，声明变量时，不能赋初值，所以所有的全局变量只是存放在 BSS 区域；第 7 至 37 行的汇编代码是通过 EmitFunction()产生的形如函数_main 的汇编代码。

图第 9 行到第 10 行用于保存寄存器的值，第 11 行用于在栈空间中预留内存空间，用来存放局部变量，这部分工作被称为”序言(Prologue)”，即在函数开始执行时要处理的工作。而图第 25 至 26 行被补为”尾声(Epilogue)”，用于恢复原先保存的寄存器值，第 27 行的汇编指令 ret 用于从栈中取出返回地址并返回。而函数的返回值在第 34 行将值放入寄存器 eax。函数中的每一基本块由 EmitBlock()函数用来为某一基本块生成汇编代码。

下面，分析一下用于生成图汇编代码的函数 TranslationUnit()，如图所示，

第 1 至 11 行的函数 EmitTranslationUnit()会为整个翻译单元产生汇编代码，第 6 行的 EmitGlobals()用于产生全局变量，第 12 行至 21 行为对应的代码，通过第 16 行的 for 循环应用在全局符号表上，找出 SK_Variable 类型，并通过 DefineGlobal()来输出到文件中。第 22 行至 31 行为 EmitFunctions()的代码，算法思想与 EmitGlobals()一样，并调用 EmitFunction()来为各个函数产生汇编代码。

```

void Compiler::TranslationUnit()
{
    // 始初始化寄存器
    SetupRegisters();

    //开始
    BeginProgram();

    //全局变量部分
    Data_Segment();
    EmitGlobals();

    //代码段部分
    Text_Segment();
    EmitFunctions();

    //结束，刷新缓冲区
    EndProgram();

    fclose(ASMFile);
}

```

```

void Compiler::EmitFunctions(void)
{
    vector<Symbol>::iterator p = GlobalIDs.buckets.begin();

    for (; p != GlobalIDs.buckets.end(); p++) {
        if ((*p)->kind == SK_Function) {
            EmitFunction((FunctionSymbol)*p);
        }
    }
}

```

EmitFunction()用于生成函数的汇编代码，根据函数里的基本块产生，其代码如图所示。

```

void Compiler::EmitFunction(FunctionSymbol p)
{
    BBlock bb;
    int varsize;

    Export((Symbol)p);
    DefineLabel((Symbol)p);

    LayoutFrame(p, 2);
    /*
        pushl %ebp
        movl %esp, %ebp
    */
    varsize = p->locals.size();
    EmitPrologue(varsize);

    bb = p->entryBB;
    while (true)
    {
        if (bb->ref != 0) DefineLabel(bb->sym);

        EmitBBlock(bb);

        if (bb != p->exitBB)
            bb = bb->next;
        else
            break;
    }
    /*
        movl %sebp, %esp
        popl %ebp
    */
    EmitEpilogue(varsize);
}

```

图中第 5 行调用的 Export()函数产生形如“.globl f”的函数声明，第 6 行的 DefineLabel()函数用于产生形如“.f:”的标号。第 6 行调用的 LayoutFrame()函数用来计算“形式参数、局部变量”在活动记录中的偏移。第 8 行调用 EmitPrologue()来产生“序言”，如图第 9 行至 10 行所显，图第 11 行的常数 12，就是“函数中局部变量所占栈内存的总和”，通过图第 14 至 25 行的 while 循环，我们可以为各基本块产生汇编代码，即调用 EmitBlock()函数来完成。第 26 行调用 EmitEpilogue()函数来产生“尾声”部分。

接下来，来介绍函数 `GetAccessName()`，其用作为返回该符号在文件中应出现的名称。包括变量名、寄存器名和函数名等。如图所示，第 7 至 9 行处理常数，形如“\$1”；对于局部变量、形式参数，在汇编代码中，用形如“20(%ebp)”这样这符号

来表示，图的 11 到 18 行是通过 `LayoutFram()`函数中计算出来的偏移，来设置相应的符号名。函数名和卷标都是直接使用符号中的 `name` 来返回。

`GetAccessName()`的返回值正是通过 `p->aname` 来返回给上一层调用。

```
string Compiler::GetAccessName(Symbol p)
{
    if (p->aname != "")
        return p->aname;

    switch (p->kind)
    {
        case SK_Constant:
            p->aname = "$" + p->name;
            break;
        case SK_Variable:
        case SK_Temp:
            if (p->level == 0)
                p->aname = p->name;
            else {
                char tmp[100];
                sprintf(tmp, "%d(%%ebp)", ((VariableSymbol)p)->offset);
                p->aname = tmp;
            }
            break;
        case SK_Label:
            p->aname = p->name;
            break;
        case SK_Function:
            p->aname = p->name;
            break;
    }

    return p->aname;
}
```

2. 寄存器的管理

在这一章，通过介绍寄存器来说明 LCC 如何管理寄存器。在前文提及过，寄存器的管理也是通过一个类来负责。Class Reg 是一个管理寄存器的类，如图所示。

```
enum {EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI};
```

```
class Reg {
protected:
    Symbol Regs[EDI+1];
    int UsedRegs;

public:
    Symbol CreateReg(string name, int no);

    void ClearRegs();
    void SpillReg(Symbol reg);
    int SelectSpillReg();
    int FindEmptyReg();

    Symbol GetRegInternal();
};
```

为了简单起见，避免复杂的数据流分析，编译程序只为临时变量分配寄存器。如图所示，第 4 行为管理的寄存器堆，其范围如图的枚量型所示，有 8 个寄存器，其对应 x86 的 4 字节寄存器，由于 LCC 只有 int 型，所以 4 字节即可；

```
Symbol Reg::GetRegInternal()
{
    int i;

    // 尝试找到一个还没有使用的寄存器，若返回不等于-1，代表存在
    i = FindEmptyReg();
    // 若没有空余的寄存器
    if (i == NO_REG)
    {
        i = SelectSpillReg();
        SpillReg(Regs[i]);
    }

    UsedRegs |= 1 << i;    //设置标志位

    return Regs[i];
}
int Reg::FindEmptyReg()
{
    int i;
    for (i = EAX; i <= EDI; i++)
    {
        if (Regs[i] != NULL && Regs[i]->link == NULL && !(1 << i & UsedRegs))
            return i;
    }

    return NO_REG;
}
```

接下来，来看一下用于分配寄存器的函数 GetRegInternal()，如图 6.5 所示。第 6 行调用 FindEmptyReg()函数来获取还未被分配的寄存器，如果不存在空寄存器，就过通第 10 行 SelectSpillReg()函数选择一个要回写的寄存器，再通过 SpillReg()函数将寄存器清空，然后在第 14

行设置 UsedRegs 相应的标志位，表示第 i 个寄存器已经被使用。第 18 行至 27 行的 FindEmptyReg()用于查找未分配的寄存器，第 223 行的条件”Regs[i]!=NULL”会排除 esp 和 ebp 两个栈寄存器，第 24 行返回对应寄存器的编号。若找不到，则在 26 行返回 NO_REG，表示所有寄存器都被分配了，接着通过第 10 行的 SelectSpillReg()来选择一个要淘汰的寄存器。

```
int Reg::SelectSpillReg()
{
    Symbol p;
    int i;
    int reg = NO_REG;
    int mref = INT_MAX;

    // 找出寄存器中引用数最小的，这里最少的通常为 0
    for (i = EAX; i <= EDI; i++)
    {
        if (Regs[i] == NULL || (1 << i & UsedRegs))
            continue;

        // 找到它指向的符号
        p = Regs[i]->link;
        if (p->ref < mref)
        {
            mref = p->ref;
            reg = i;
        }
    }

    return reg;
}
```

图是函数 SelectSpillReg()，LCC 会通过类似于 FIFO 和 LRU 算法选择，根据寄存器对临时变量的引用次数总和”来做选择，选择引用次数最少(通常为 0)的寄存器返回。

3. 中间代码翻译

主要流程

LCC 编译程序的中间代码是如下四元式，包括运算符和 3 个操作数。

<运算符 opcode, 目的操作数 DST, 源操作数 SRC1, 源操作数 SRC2>

当然有一些指令只需要用到 2 个就够。LCC 为便于汇编代码的生成，在 template.h 中定义了许多汇编指令的模板。如

通过宏定义 X86_JMP 找到对应的模块。模板中的”%0”充当占位符的作用，代表第 0 个操作数，即目的操作数 DST。在汇编代码生成时，”%0”会被 DST 替代，”%1”被 SRC1，”%2”被 SRC2 替代。通过 PutASMCode 函数对相应的模块应行翻译。如图所示

```
void Compiler::PutASMCode(const char *str, Symbol opds[])
{
    int i;
    const char *fmt = str;
    PutChar('\t');
    while (*fmt)
    {
        switch (*fmt)
        {
            case ';':
                PutString("\n\t");
                break;

            case '%':
                // Linux:
```

```

// TEMPLATE(X86_MOVI4, "movl %1, %0")
fmt++;
if (*fmt == '%')
{
    PutChar('%');
}
else
{
    i = *fmt - '0';
    if (opds[i]->reg != NULL)
    {
        PutString((opds[i]->reg->name).c_str());
    }
    else
    {
        PutString(GetAccessName(opds[i]).c_str());
    }
}
break;

default:
    PutChar(*fmt);
    break;
}
fmt++;
}
PutChar('\n');
}

```

利用图的 PutASMCode()输出汇编代码。第 4 行至 26 行处理形如” addl %2, %0”的模版。在第 19 至 21 行会将占位符” %0”，“%1”，” %2”替换为相应的操作数名称。如果操作数的值已经被加载到寄存器中，则在第 14 行输出寄存器名称，否则通过第 1 节分析过的 GetAccessName()，输出操作数的名称。第 10 行处理形如” %%eax”的字符串，在 AT&R 的汇编代码中，寄存器前要加一个” %”，由于” %%”在字符串中会被当作” %”本身。故在 PutASMCode()函数处理后，会输出” %eax”的代码。

为算术运算产生汇编代码

现在来分析如何产生形如” t1:a+b”的一元算术运算。由于一条 x86 指令，最多出现 2 个操作数，而中间指令” DST:SRC1+SRC2”有 3 个操作数，因此，必须多产生一条 x86 指令。对于算术运算，按以下步骤处理：

- (1) 调用 AllocateReg()函数依次为 SRC1、SRC2 和 DST 分配寄存器。DST 是用于保存运算结果的临时变量，必然分配寄存器，若 SRC1 和 SRC2 不是临时变量，则没有分配
- (2) 若 DST 和 SRC1 对应的寄存器不一样，可产生一条 movl 指令，把 SRC1 的值传入 DST 的寄存器中。
- (3) 产生加法指令，进行 SRC2 和 DST->reg 的加法，并把结果寄存在 DST->reg 中。

```

void Compiler::EmitAssign(IRInst inst)
{
    switch (OP)
    {
        case $ADD:
        case $SUB:

            AllocateReg(inst, 1);
            AllocateReg(inst, 2);
            AllocateReg(inst, 0);
            //char name[40];
            //sprintf(name, "%s", SRC1->reg->name.c_str());
            //PutString(name);

            if (DST->reg != SRC1->reg)
            {
                Move(DST, SRC1);
            }
            if (OP == $ADD) PutASMCode(X86_ADDI4, inst->opds);
            if (OP == $SUB) PutASMCode(X86_SUBI4, inst->opds);
            break;

        case $MUL:
        case $DIV:
            if (SRC1->reg == Regs[EAX])
            {
                SpillReg(Regs[EAX]);
            }
            else
            {
                Symbol sym = Regs[EAX]->link;

                if (sym != NULL && sym->ref > 0) {
                    Symbol reg = GetRegInternal();
                    SpillReg(Regs[EAX]);

                    Move(reg, Regs[EAX]);
                    sym->reg = reg;
                    reg->link = sym;
                }
                else {
                    SpillReg(Regs[EAX]);
                }

                Move(Regs[EAX], SRC1);
            }
            // SpillReg(Regs[EDX]);
            // 将 SRC1 扩展后放在 EAX 与 EDX
            UsedRegs = 1 << EAX ;
            if (SRC2->kind == SK_Constant)
            {
                Symbol reg = GetRegInternal();

                Move(reg, SRC2);
                SRC2 = reg;
            }
            else
            {
                AllocateReg(inst, 2);
            }
        }
    }
}

```

```

    }

    if (OP == $MUL) PutASMCode(X86_MULI4, inst->opds);
    if (OP == $DIV) PutASMCode(X86_DIVI4, inst->opds);

    DST->link = Regs[EAX]->link;
    Regs[EAX]->link = DST;
    DST->reg = Regs[EAX];
    break;

default:
    break;
}

if (DST) DST->ref--;
if (SRC1) SRC1->ref--;
if (SRC2) SRC2->ref--;
}

```

图第 5 至第 13 行产生加法/减法运算的汇编代码; 第 7 行至第 9 行用于分配寄存器, LCC 只分配寄存器给临时变量; 第 10 行用作将 SRC1 的值放入 DST 的寄存器中; 然后根据运算符类型产生汇编代码。

第 14 行至 42 行为乘法/除法的汇编代码, 由于 x86 的乘法指令, 是将源操作数与 EAX 里的值相乘, 因此, 在第 30 行目的是为了清空 EAX 寄存器, 并把被乘数放入 EAX 中。然后, 根据, 在第 32 到 38 的目的是将源操作数放入寄存器中, 之后, 再根据 39 至 40 行的判断产生相应的汇编代码, 最后, 由于 x86 的乘法结果是存放在 EAX 中, 所以要将 DST 的寄存器指向 EAX, 代码 DST 的寄存器。

为转移指令产生汇编代码

转跳指令包括, 有”条件跳转”和”无条件跳转”。

- (1) 有条件转跳, 形如”if(a<b) goto BB2;”
- (2) 无条件转跳, 形如”goto BB3;”

```

void Compiler::EmitBranch(IRInst inst)
{
    BBlock p = (BBlock)DST;
    DST = p->sym;

    if (SRC2) {
        if (SRC2->kind != SK_Constant) {
            SRC1 = PutInReg(SRC1);
        }
    }

    if (SRC1->reg != NULL) {
        SRC1 = SRC1->reg;
    }

    SRC1->ref--;

    if (SRC2) {
        SRC2->ref--;
        if (SRC2->reg != NULL) {
            SRC2 = SRC2->reg;
        }
    }
}

```

```

ClearRegs();

if (OP == $JE) PutASMCode(X86_JEI4, inst->opds);
if (OP == $JGE) PutASMCode(X86_JGEI4, inst->opds);
if (OP == $JG) PutASMCode(X86_JGI4, inst->opds);
if (OP == $JL) PutASMCode(X86_JLI4, inst->opds);
if (OP == $JGE) PutASMCode(X86_JGEI4, inst->opds);
if (OP == $JLE) PutASMCode(X86_JLEI4, inst->opds);
}

```

在图中的第 1 至 24 行为 `EmitBranch()` 产生有条件的转跳指令。由于常会以”立即数的形式存放在代码区中，当程序运行时，CPU 会从代码区读出，因此当操作数 SRC2 为常数时，可以不必把 SRC1 的值加载到寄存器中，因为”同一条 X86 指令的两个操作数不可以都在内存中。第 9 行判断 SRC1 的值是否存寄存器中，如果已加载，则将 SRC1 改为 SRC1->reg，产生形如”`cmpl b, %eax`”的比较指令，然后调在第 18 至 23 行调用 `PutASMCode()` 产生跳跳指令。第 25 行至 29 行为无条件转件，只是将 DST 改为 DST->sym，然后产生汇编代码。

为函数调用产生汇编代码

在这里，来分析 LCC 如何产生函数调用和函数返回的汇编代码。根据 C 函数的约定，需要把参数从右向左入栈，当函数返回时，主调函数要负责把这些参数出栈，这可通过如”`all size, %esp`”的指令实现。

因此，按以下步骤来翻译调用指令

- (1) 参数从右到左，依次入栈;
- (2) 若需保护寄存器，需回写
- (3) 若函数名为 `demo`，则产生形如”`call demo`”的指令
- (4) 根据入栈参数的总和，调整寄存器 `esp`;

```

void Compiler::EmitCall(IRInst inst)
{
    ParameterList argslist = (ParameterList)SRC2;
    Symbol arg;
    int i, stksize = 0;

    for (i = argslist->npara - 1; i >= 0; i--)
    {
        arg = argslist->args[i];
        PushArgument(arg);
        stksize += sizeof(int) * 1;
    }
    // call func
    PutASMCode(X86_CALL, inst->opds);

    // 如果参数不为 0，则返回时需要复原栈顶
    if (stksize != 0) {
        Symbol p;
        p = IntConstant(stksize);
        PutASMCode(X86_REDUCEP, &p);
    }

    if (DST) DST->ref--;
    // 这里 DST 不会 DST == NULL，因为根据要求为 ID = call ID.
}

```

```

//函数返回值放在 EAX
//AllocateReg(inst, 0);
//if (DST->reg != Regs[EAX])
//{
// Move(DST, Regs[EAX]);
//}
DST->reg = Regs[EAX];
}

```

如图 6.9 所示，第 6 行至第 10 从最后一个参数到第一个依次入栈，调用 `PushArgument()` 函数产生 "pushl %0" 的指令。然后第 11 行产生 "call func"，最后，若调用函数参数大于 0，则在 12 行至 16 调整栈的大小。

```

void Compiler::EmitReturn(IRInst inst)
{
    if (DST->reg != Regs[EAX])
    {
        Move(Regs[EAX], DST);
    }
}

```

图为函数返回时的指令，函数返回约定将返回值放入 EAX 中。

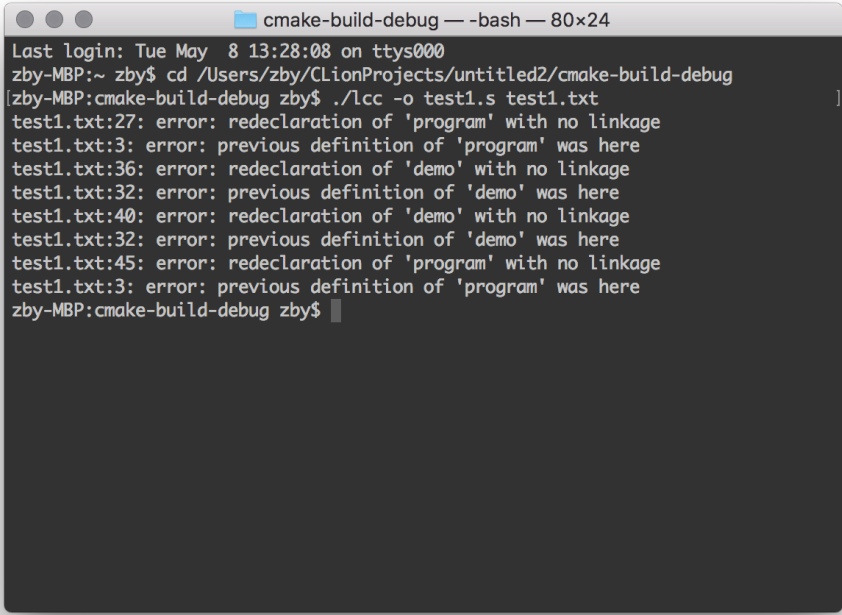
七、 执行结果

LCC 的执行界面为命令行方式，由于命令行的输入方式可以更快地执行输出，并且更贴合编译程序的形式，故这里使用了命令行界面。

输入方式为

```
lcc -o test1.s test1.txt
```

若有出错，LCC 会给出提示，告诉使用者哪里出错了。

A terminal window titled 'cmake-build-debug -- -bash -- 80x24'. The window shows the output of a command to compile test1.txt with LCC. The output displays several error messages related to redeclarations of 'program' and 'demo' with no linkage. The errors are as follows:

```
Last login: Tue May  8 13:28:08 on ttys000
zby-MBP:~ zby$ cd /Users/zby/CLionProjects/untitled2/cmake-build-debug
[zby-MBP:cmake-build-debug zby$ ./lcc -o test1.s test1.txt
test1.txt:27: error: redeclaration of 'program' with no linkage
test1.txt:3: error: previous definition of 'program' was here
test1.txt:36: error: redeclaration of 'demo' with no linkage
test1.txt:32: error: previous definition of 'demo' was here
test1.txt:40: error: redeclaration of 'demo' with no linkage
test1.txt:32: error: previous definition of 'demo' was here
test1.txt:45: error: redeclaration of 'program' with no linkage
test1.txt:3: error: previous definition of 'program' was here
zby-MBP:cmake-build-debug zby$
```

若没有错误，则结果会是这样。汇编成功，到目录下查找。

```
cmake-build-debug — -bash — 80x24
Last login: Tue May  8 13:28:08 on ttys000
zby-MBP:~ zby$ cd /Users/zby/ClionProjects/untitled2/cmake-build-debug
zby-MBP:cmake-build-debug zby$ ./lcc -o test1.s test1.txt
test1.txt:27: error: redeclaration of 'program' with no linkage
test1.txt:3: error: previous definition of 'program' was here
test1.txt:36: error: redeclaration of 'demo' with no linkage
test1.txt:32: error: previous definition of 'demo' was here
test1.txt:40: error: redeclaration of 'demo' with no linkage
test1.txt:32: error: previous definition of 'demo' was here
test1.txt:45: error: redeclaration of 'program' with no linkage
test1.txt:3: error: previous definition of 'program' was here
zby-MBP:cmake-build-debug zby$ ./lcc -o test2.s test2.txt
zby-MBP:cmake-build-debug zby$
```

```
test2.s
1 # Code auto-generated by blank-black
2 .section .data
3 .section .bss
4     .lcomm a 4
5     .lcomm b 4
6 .section .text
7 .globl _program
8 _program:
9     pushl %ebp
10    movl %esp, %ebp
11    subl $8, %esp
12    movl $0, -4(%ebp)
13    movl -4(%ebp), %eax
14    movl %ebp, %esp
15    popl %ebp
16    ret
17 .globl _demo
18 _demo:
19     pushl %ebp
20     movl %esp, %ebp
21     movl 8(%ebp), %eax
22     movl $4, %ecx
23     imull %ecx, %eax
24     addl $2, %eax
25     movl %eax, 8(%ebp)
26     movl 8(%ebp), %eax
27     movl $2, %ecx
28     imull %ecx, %eax
29     movl %ebp, %esp
30     popl %ebp
31     ret
32 .globl main
```


八、总结

终于完成了编译原理课程设计，感谢卫老师一直以来的教导，谢谢老师！

不知不觉已经将编译程序的具体部分设计并编写出来。一直以来，在编写程序语言的时候，都并不太了解程序之后是如何执行的，变量的存放位置在哪里，程序是如何达到我想要的结果的。通过自己亲手编写的编译程序，从而对编译程序有了更进一步的了解，明白了变量是如何放进符号表，函数的作用域是如何定义的，语法、语义是如何检查的，这些都通过自己写的 LCC 编译程序加深了印象，很有成就感。

还记得第一次上编译原理课时，对所有的这些知识只是停留在书本知识层面上，但到第一次亲手编写词法、语法分析器时，总摸不着头脑，总是存在一些迷惑，明白了“纸上得来终觉浅，绝知此事要躬行”的道理，计算机科学与技术除了书本上的知识，更深一步的还是要通过自己动手才能把一门理论搞透彻，把理论的知识提升到更高的层次，深深地刻印在脑海中。

从词法、语法、语义和中间代码，到最后的目標代码生成，一步步走来，真不容易。词法、语法、语义和中间代码生成的代码都是采用了上一学期的代码，不过在编写目标代码生成时，发现还是有很多的缺憾、不足，需要重新考虑、设计，在原有代码的基础上进行修改，而适合目标代码生成，但是，通过修改代码，还而加深了对编译程序的认识，如何优化每一部分是这里的关键，必须反复的阅读数据才能把这一部分的优化做好。其中，每一部分，LCC 都是采用了类的设计来继承，正是采用了软件开发的思想和，但 LCC 只是很小很小的程序。通过类的设计把每一个类继承起来，从而组成了 LCC 编译程序。

最后，现在已经有 GCC 等的编译程序，功能都是十分强大，通过自己动手写编译程序，明白了现今的工业界的编译程序真是十分强大。要达到这样的境界，真的要通过很深的功力。