单位代码	BUAA
学号	39061416
分类号	
密级	

北京航空航天大學
BEIHANG UNIVERSITY

《编译技术》课程设计申优文档

扩充 PIO 文法编译器的实现

院（系）名称	高等工程学院
专业名称	计算机科学与技术
学生姓名	黄建宇
学号	39061416

2012 年 1 月 1 日

摘 要

本系统实现了一个以扩充 p10 文法为输入语言的采用 C++及面向对象思想设计的编译器，可以生成符合 386 指令集规范的汇编代码，用 Masm32v10 汇编可生成 32 位应用程序。本文从编译实现的各个阶段详细描述该系统的实现方法。

关键字 扩充 p10 文法 C++ 汇编 MASM32

目录

1 准备工作

1.1 学习 32 位汇编

从我们这级开始，汇编放在编译之后学习，这给我们完成编译高难度作业带来了很大的难度。所以我们最好在编译课设的第一节课上之前就能到图书管理借些汇编书自学。就编译本身而言，需要的汇编知识并不是像实用的高级应用那么困难，只需了解一些汇编的最基本的知识就行了。关于这方面的书，我特别推荐王爽写的《汇编语言》，这本书尽管是 8086 汇编，但里面讲了许多关于汇编底层的知识，且讲解非常通俗易懂，让我在编译大作业遇到诸如 eax，edx 的值异常改变这样的情况时，能够从容自如地处理。

1.2 软件工程、项目管理

可能大家到目前为止写程序还是习惯于写成一个文件。但是到了编译课设这样庞大的工程，写成一个文件不仅维护困难，而且造成局部变量极容易重名，发生各种莫名其妙的错误。代码更新时，也常会出现新版代码覆盖旧版代码后发现新版的代码其实是错误的，这时想找回就是的代码便是费时费力。如何解决上面提到的这些问题呢？

首先，我们可以将程序按功能不同，将不同的功能写在不同的文件当中，这样便于管理，而且对于每一个功能都是一个不大的程序，写起来也方便。不可否认的是，多文件也是会带来一些麻烦。比如引用其他文件的变量、符号重定义之类的问题，所以建议在动手之前先解决这些问题。在设计时就要想好那些变量是全局通用的。我的代码中的全局变量就是放在一个 GlobalData.h 的头文件中。

2 文法修改

2.1 任务

在我们的编译课程设计中，要求使用递归下降子程序法来写整个编译程序。这样我们在开始就必须对文档进行部分改写，使其符合递归下降子程序法可以分析的要求，并尽量使其达到 LL(1)文法要求：

1. 消除左递归
2. 使有‘|’存在的规则中 first 集合不相交。

2.2 具体实现

对现有的文法，我在不改变语义的前提下做了下面的改写：

<程序> ::= <分程序>.

<分程序> ::= [<常量说明部分>][<变量说明部分>]{[<过程说明部分>]| [<函数说明部分>]}<复合语句>

<常量说明部分> ::= const<常量定义>{,<常量定义>};

班号： 392312 姓名： 黄建宇 学号：39061416

<常量定义> ::= <标识符>＝ <常量>

<常量> ::= [+| -]<无符号整数>|<字符>

<字符> ::= '(<字母> | <数字>)'

<字符串> ::= "{(<字母> | <数字>| ' ')}"

<无符号整数> ::= <数字>{<数字>}

<标识符> ::= <字母>{<字母>|<数字>}

<变量说明部分> ::= var <变量说明> ; {<变量说明>;}

<变量说明> ::= <标识符>{, <标识符>} : <类型>

<类型> ::= <基本类型> | array '['<无符号整数>']' of <基本类型>

<基本类型> ::= integer | char

<过程说明部分> ::= <过程首部><~~分程序~~>;{<~~过程~~说明部分>;}(与上面的<分程序>说明有重复)

<函数说明部分> ::= <函数首部><~~分程序~~>;{ <函数说明部分>;}(与上面的<分程序>说明有重复)

<过程首部> ::= procedure<标识符>'('[<形式参数表>]')';

<函数首部> ::= function <函数标识符>'('[<形式参数表>]')': <基本类型>;

<函数首部> ::= function <函数标识符>'('[<形式参数表>]')': <基本类型>;（加函数两个字以示区别）

<形式参数表> ::= <形式参数段>{; <形式参数段>}

<形式参数段> ::= [var]<标识符>{, <标识符>}: <基本类型>

<语句> ::= <赋值语句>|<条件语句>|<情况语句>|<过程调用语句>|<复合语句>|<读语句>|<写语句>|<for 循环语句>|<空>

（First 集合一样，在程序处理中采用毋哲学长的做法，词法分析时进行跳读 1 个字符判断）

<赋值语句> ::= <标识符> := <表达式>| <函数标识符> := <表达式>|<标识符>'['<表达式>']':= <表达式>

<函数标识符> ::= <标识符>

<表达式> ::= [+|-]<项>{<加法运算符><项>}

<项> ::= <因子>{<乘法运算符><因子>}

<因子> ::= <标识符>|<无符号整数> |'('<表达式>')' | <函数调用语句>|<标识符>'['<表达式>']'

<函数调用语句> ::= <标识符>'('[<实在参数表>]')'

<实在参数表> ::= <实在参数> {, <实在参数>}

<实在参数> ::= <表达式>|<函数调用语句>（删去<标识符>，因为可由<表达式>推出）

<加法运算符> ::= +|-

<乘法运算符> ::= *|/

<条件> ::= <表达式><关系运算符><表达式>

<关系运算符> ::= <|<=|>|>= |=|<>

<条件语句> ::= if<条件>then<语句>[else<语句>](更符合前面的 BNF 规范)

<情况语句> ::= case <表达式> of <情况表元素>{; <情况表元素>}end

<情况表元素> ::= <情况常量表> : <语句>

<情况常量表> ::= <常量>{, <常量>}

<for 循环语句> ::= for <标识符> := <表达式> （downto | to） <表达式> do <语句> //步长为 1

~~<过程调用语句> ::= <标识符>'('[<实在参数表>]')'~~（删去此句，因为可用<函数调用语句> ::= <标识符>'('[<实在参数表>]')'同样处理。程序中将 procedure 和 function 可以一起处理，只是一个有返回值 PROCINT 或者 PROCCHAR，另一个是 PROCNONE）

<复合语句> ::= begin<语句>{; <语句>}end

<读语句> ::= read'('<标识符>{,<标识符>}')'

<写语句> ::= write'(' <字符串>,<表达式> ')'|write'(' <字符串>')'|write'('<表达式>')'

<字母> ::= a|b|c|d…x|y|z |A|B…|Z

<数字> ::= 0|1|2|3…8|9

在此基础上，我对文法新增了一些内容，以更具普适性（当然，防止老师的测试程序中会有这些东西，我报了错，但最终汇编码是能容错的）：

1.表达式中可以有字符(但出现后会报错，生成的汇编码无误)

2.字符串中可以有其他字符(但出现后会报错，生成的汇编码无误)。

3.对于一些常见的错误，比如:=写成=，能实现报错同时生成正确的汇编码。容错机制相对完善。

3 词法分析程序

3.1 任务

词法分析是编译课设的第一步，也是最简单的一步。它可以让我们增添对这门课程的信心与兴趣。词法分析说白了，就与高级语言程序设计（2）中的字符串处理程序是相类似的，甚至比其还要简单。

这里，我们可以从上面的文法中找出生成单词的规则：

<加法运算符> ::= +|-

<乘法运算符> ::= *|/

<关系运算符> ::= <|<=|>|>= |=|<>

<字母> ::= a|b|c|d…x|y|z |A|B…|Z

<数字> ::= 0|1|2|3…8|9

<字符> ::= '(<字母> | <数字>)'

<字符串> ::= "{(<字母> | <数字>| ' ')}"

<无符号整数> ::= <数字>{<数字>}

<标识符> ::= <字母>{<字母>|<数字>}

3.2 单词编号

我们可以根据汇编理论课上的 ppt 上所讲的，将保留字和分界符采用一符一类。

我设计的编号如下：

编号	符号	编号	符号	编号	符号
0		20	write	40	<>
1	const	21	identifier	41	:=
2	int	22	number	42	:
3	char	23	character	43	.
4	var	24	string	44	'
5	array	25	(	45	"
6	of	26	)		
7	integer	27	[		
8	procedure	28	]		
9	function	29	,		
10	if	30	;		
11	then	31	+		
12	case	32	-		
13	for	33	*		
14	to	34	/		
15	downto	35	<		
16	do	36	<=		
17	begin	37	>		
18	end	38	>=		
19	read	39	=		

在全局变量放置的 GlobalData.h 中，可以用一个字符串数组来存取编号信息：

```
static string KEYWORD[50]={"", "const", "int", "char", "var", "array", "of", "integer", "procedure", "function", "if", "then", "case", "for", "to", "downto", "do", "begin", "end", "read", "write", "identifier","number","character","string","(", ")", "[", "]", ",", ";", "+", "-", "*", "/", "<", "<=", ">", ">=", "=", "<>", ":=", ":", ".", "\'", "\""};
```

3.3 词法分析返回的数据结构设计

词法分析是被语法分析调用的，它必须提供给语法分析足够的信息，所以这里的数据结构设计也很重要。我是这样设计的：

```
class LexerItem//记录词法分析得到的每个单词信息
{
public:
    int kind;
    string symbol;
};
```

其中的 kind 就是上面提到的单词编号。

3.4 跳读

在前面的文法改写中曾提到：

<语句> ::= <赋值语句>|<条件语句>|<情况语句>|<过程调用语句>|<复合语句>|<读语句>|<写语句>|<for 循环语句>|<空>

（First 集合一样，在程序处理中采用毋哲学长的做法，词法分析时进行跳读 1 个字符判断）

如何进行跳读呢？可以在词法分析中加入一个队列，当需要跳读的时候向后读取并将其存在队列中，当普通读取的时候如果队列中有单词先取队列中的单词。

这里需要特别注意普通读取的优先级：先看跳读队列中是否为空：为空则读取下一个，不为空则先读取跳读队列中的字符。

4 语法分析

4.1 任务

语法分析，填符号表，对语法错误进行检查，以便于语义分析时正确生成汇编码。

语法分析可以直接采用编译理论课上讲的递归下降子程序法，这里要回顾前面将的文法修改与跳读，对于那些 First 集相同的语法成分，我们大可不必费半天劲去学乔姆斯基体系，然后化简出一个完全陌生的文法。

语法分析我基本是按照一个尖括号<>里的元素变成一个函数这样的方式来实现的。下面就是我的程序结构。可以看出，基本上每一个文法中的元素都有一个函数与之相对应。

```
class Parser//语法分析与语义分析
{
public:
    (内容部分省略)
```

```
        void program();//程序
        void block();//分程序
        void constDes();//常量说明部分
        void constDefine();//常量定义
        string constValue();//<常量>
        void varDes();//变量说明部分
        void varDefine();//变量说明
        void function(int lastProcNum);//函数说明部分
        void functionStart(int lastProcNum);//函数首部
        void procedure(int lastProcNum);//过程说明部分
        void procedureStart(int lastProcNum);//过程首部
        void argTable();//形式参数表
        void arg(bool tempflag);//形式参数段
        void complexSentense(int lastProcNum);//复合语句
        void statement(int lastProcNum);//语句
        void becomesState(int lastProcNum);//赋值语句
        string expression(int lastProcNum);//表达式
        string term(int lastProcNum);//项
        string factor(int lastProcNum);//因子
        string callFunction(int lastProcNum);//函数调用语句
        string callProcedure(int lastProcNum);//过程调用语句
        int valueTable(int lastProcNum,int ProcIndex,string procName);//实在参数
表
        void ifState(int lastProcNum);//条件语句
        string condition(int lastProcNum);//条件
        void caseState(int lastProcNum);//情况语句
        void Parser::situationTable(int lastProcNum,string target,string flag,string
smallflag);//情况表元素
        void forState(int lastProcNum);//for 循环语句
        void readState(int lastProcNum);//读语句
        void writeState(int lastProcNum);//写语句

    };
```

5 错误处理

5.1 任务

处理在语法分析中遇到的各种错误。在编译理论课中，我们学过：错误处理技术主要有两种：错误改正与错误局部化处理。

5.2 错误处理方法

错误改正与错误局部化处理两种方法我都用到了。下面各举一例。

错误局部化处理

```
If (message=="变量说明部分缺少;")//语法分析传给错误处理程序的信息
    {
        cout<<"("<<x<<","<<y<<"):语法分析发现错误："<<message<<endl;
    while(myParser.sym.kind!=21&&!isStartMark2(myParser.sym.symbol)&&myParser.sym.kind!=-1)//停止符号集，即跳过所在的语法成分（短语或者语句），一般是调到语句右界符，然后从新语句继续向下分析。
        {
            myParser.sym=myLexer.getSym();
        }
        if(myParser.sym.kind==-1) exit(0);
        return;
    }
```

5.3 错误改正

```
if(message=="赋值语句或 for 循环语句缺少:=")
    {
        cout<<"("<<x<<","<<y<<"):语法分析发现错误："<<message<<endl;
        if(myParser.sym.symbol=="=")//这里采取的是预测的方式。由于:=写成=是一种常见错误，这里就有将其特别提取出来，特殊处理
        {
            cout<<"\t\t\t:=错写成=,汇编中已做更正"<<endl;
            myParser.sym=myLexer.getSym();
            return;
        }
        return;
    }
```

5.4 错误处理的分类

错误处理主要分为 3 类：词法错误、语法错误和语义错误。

编号	错误类型	错误字符
1	词法分析错误	字符中含有非法字符
2	词法分析错误	字符最后缺少匹配’
3	词法分析错误	字符串最后缺少匹配\″
4	词法分析错误	不能识别的字符
5	词法分析错误	字符中含有非法字符
6	语法分析错误	函数没有返回值
7	语法分析错误	分程序中没有复合语句

8	语法分析错误	常量说明部分开始没有 const
9	语法分析错误	常量说明部分最后缺少;
10	语法分析错误	常量定义开始不是标识符
11	语法分析错误	常量定义没有中间的=
12	语法分析错误	常量出现不符合定义的符号
13	语法分析错误	变量说明部分开始没有 var
14	语法分析错误	变量说明部分缺少;
15	语法分析错误	变量说明开始不是标识符
16	语法分析错误	变量说明缺少:
17	语法分析错误	变量说明没有类型
18	语法分析错误	变量说明定义数组缺少[
19	语法分析错误	变量说明定义数组缺少]
20	语法分析错误	变量说明定义数组缺少 of
21	语法分析错误	变量说明数组定义没有基本类型
22	语法分析错误	函数说明部分缺少;
23	语法分析错误	函数首部缺少 function
24	语法分析错误	函数首部开始缺少标识符
25	语法分析错误	函数首部缺少(
26	语法分析错误	函数首部缺少)
27	语法分析错误	函数首部缺少:
28	语法分析错误	函数首部没有设置基本类型
29	语法分析错误	函数首部缺少;
30	语法分析错误	过程说明部分缺少;
31	语法分析错误	过程首部缺少 function
32	语法分析错误	过程首部开始缺少标识符
33	语法分析错误	过程首部缺少(
34	语法分析错误	过程首部缺少)
35	语法分析错误	过程首部缺少;
36	语法分析错误	形式参数段缺少标志符
37	语法分析错误	形式参数段开始不是标识符
38	语法分析错误	形式参数段缺少:
39	语法分析错误	形式参数段缺少基本类型定义
40	语法分析错误	复合语句开始没有 begin
41	语法分析错误	复合语句结尾没有 end
42	语法分析错误	语句以标识符开头，却不是赋值语句与过程调用语句
43	语法分析错误	赋值语句开始不是标识符
44	语法分析错误	数组变量后面缺少]
45	语法分析错误	赋值语句或 for 循环语句缺少:=
46	语法分析错误	赋值语句缺少:=或者数组没有[]标号
47	语法分析错误	因子（<表达式>)缺少）
48	语法分析错误	不能识别的因子
49	语法分析错误	因子中不应有字符
50	语法分析错误	函数调用缺少标识符
51	语法分析错误	函数调用缺少（
52	语法分析错误	过程调用缺少标识符
53	语法分析错误	过程调用缺少（

54	语法分析错误	过程调用缺少)
55	语法分析错误	条件语句条件多了(
56	语法分析错误	条件语句条件多了)
57	语法分析错误	条件语句缺少 then
58	语法分析错误	条件中没有关系运算符
59	语法分析错误	情况语句开始缺少 case
60	语法分析错误	情况语句开始缺少 of
61	语法分析错误	情况语句缺少 end
62	语法分析错误	情况子语句缺少:
63	语法分析错误	for 循环语句开始缺少 for
64	语法分析错误	for 循环语句开始缺少标识符
65	语法分析错误	for 循环语句缺少 to/downto
66	语法分析错误	for 循环语句缺少 do
67	语法分析错误	读语句开始缺少 read
68	语法分析错误	读语句缺少(
69	语法分析错误	未声明或是常量或是数组但不能赋值
70	语法分析错误	读语句缺少标识符
71	语法分析错误	读语句缺少)
72	语法分析错误	写语句开始缺少 write
73	语法分析错误	写语句缺少(
74	语法分析错误	写语句缺少)
75	语法分析错误	程序中没有结尾.
76	语义分析错误	赋值语句中标识符后面有]但不是数组元素
77	语义分析错误	赋值语句中等式左边变量未定义|不是变量|是过程变量|是数组元素但后面没有[
78	语义分析错误	因子使用没有定义、不能读取的标识符
79	语义分析错误	函数使用错误
80	语义分析错误	因子使用没有定义的数组
81	语义分析错误	函数名不正确|应该是函数，而这里却是过程
82	语义分析错误	name+"函数名/过程不正确"
83	语义分析错误	参数数目不正确
84	语义分析错误	函数调用缺少)
85	语义分析错误	过程名不正确|应该是过程，而这里却是函数
86	语义分析错误	for 循环语句中标识符未定义或者不是变量或者是数组元素
87	语义分析错误	name+"重复定义"
88	语义分析错误	传地址形参不能是复杂表达式
89	语义分析错误	除数不能是 0

6 符号表设计

6.1 任务

符号表在编译时主要用于存储符号信息以助于正确生成中间代码，并检查语义正确性。符号表的操作主要有两个：插入与查找。这与我们同一时间做的数据库大作业很像。

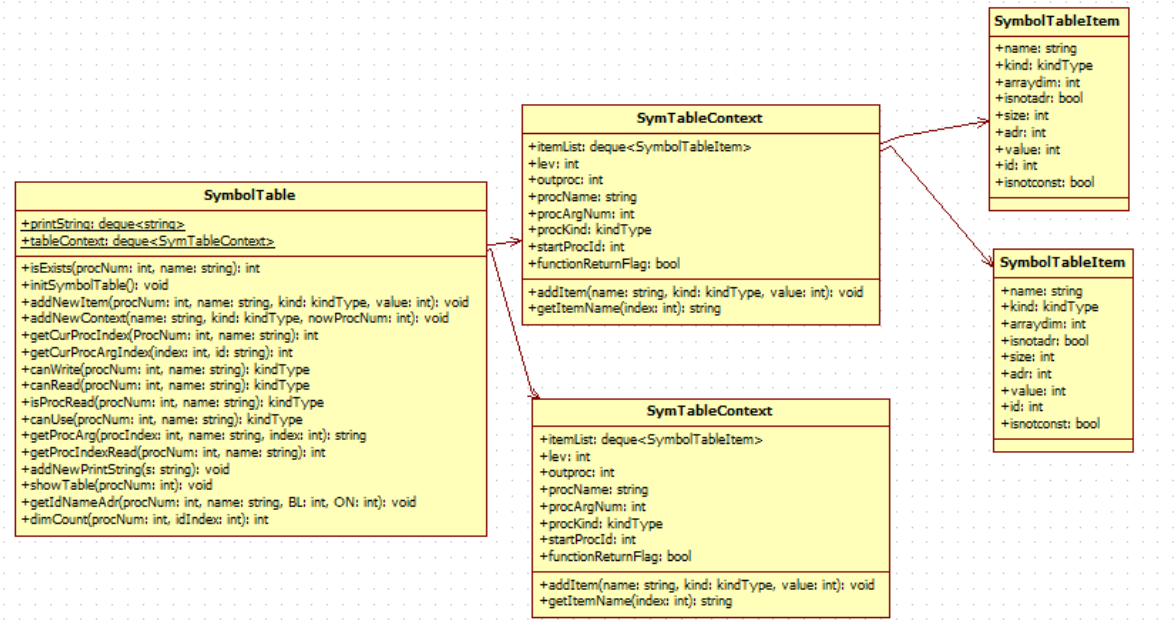
6.2 符号表生命周期

符号表设计对扩充 PL0 这样带嵌套结构的文法设计显得尤为重要，如何设计这种嵌套的符号表呢？我的许多同学最后按照编译理论课上所讲的，生成栈式符号表，认为前端编译部分完成，符号表在汇编代码生成等环节就没有作用。但这样我们在汇编代码生成时想找到变量类型与函数类型都不可能了。这样到最后会弄得很麻烦，使得四元式异常复杂或者代码无法优化。

我的符号表甚至在目标码生成后的代码优化都用到了。也就是说，符号表信息一直要保留到最后。

6.3 嵌套符号表

关于嵌套符号表，我得益于上学期 C/C++的课程的学习，用了类封装的方法，设计了 3 层嵌套符号表。先给出符号表的类图。



可以发现，这种逐层嵌套的嵌套符号表非常符合我们的思维习惯，便于我们查表、改表。

下面给出我的具体代码层次上的符号表设计：

```
class SymbolTableItem//记录声明后的每个标识符的信息
{
public:
```

```
        string name;//名字
        kindType kind;//类型（定义为
enum{CONSTINT,CONSTCHAR,INT,CHAR,ARRAYINT,ARRAYCHAR,PROCINT,PROCCHAR
,PROCNONE,ERROR}类型）
        //int arrayref;//数组时指向对应 ArrayTable 中的序号，否则指向 0;
        int arraydim;//有数组时表示数组维数;否则不定义          //最后试试用
构造函数初始化为 1;
        bool isnotadr;//true 表示不是地址，false 表示是地址
        int regnum;//最后全局寄存器分配时分配给变量的寄存器编号，默认为-1
        //bool isinreg;
        //bool isinmem;
        int size;//表示所占的大小，一般变量为 1，数组为相应元素个数。//这样
冗余甚至能去掉。
        int adr;//变量（形参）：相对起始的位移量            //对于函数或过程，
填写入口 Procnum
        int value;//对于常量存储其值
        bool isnotconst;//true 表示不是常数，false 表示是常数。
    };


    class SymTableContext//记录每个函数/过程体的信息，其中的 itemList 包含
SymbolTableItem 类
    {
    public:
        deque<SymbolTableItem>itemList;//记录函数/过程里的变量、常量信息
        int lev;//表示 proc/func 所在层数
        int outproc;//记录外层的函数编号
        string procName;//记录函数名
        int procArgNum;//记录函数的参数个数
        kindType procKind;//记录函数的种类
        int startProcId;//记录一维表开始的 id，用作索引
        bool functionReturnFlag;//记录函数有无返回值
    };

    class SymbolTable//符号表，其中的 tableContext 包含 SymTableContext 类
    {
    public:
        int isExists(int procNum,string name);//判断声明的标识符在符号表中是否
曾经出现

        static deque<string> printString;//记录字符串
        static deque<SymTableContext>tableContext;//记录函数/过程，每个元素
是上面的 SymTableContext 类
        //static deque<int>arrayList;
        //static int arrayNum;
```

```
        //void addArrayItem(string name,kindType kind,int value);
        //int getArrayItemDim(int index);
        //static int number;
        void initSymbolTable();//初始化符号表
        void addNewItem(int procNum,string name,kindType kind,int value);//在符号表 procNum 的函数/过程中增加新的符号（标识符）
        void SymbolTable::addNewContext(string name,kindType kind,int nowProcNum);//增加新的函数/过程，即新建 SymTableContext 类
        //int SymbolTable::getProcIndex(string name);
        //int SymbolTable::getProcArgIndex(int index,string id);
        int SymbolTable::getCurProcIndex(int ProcNum,string name);//获取当前函数/过程的索引（即位置）
        //int SymbolTable::getCurProcArgIndex(int index,string id);//获取当前函数的参数
        kindType SymbolTable::canWrite(int procNum,string name);//获取标识符是否可以赋值以及类型的信息
        kindType SymbolTable::canRead(int procNum,string name);//获取标识符是否可以读取值以及类型的信息
        kindType SymbolTable::isProcRead(int procNum,string name);//获取函数/过程是否可以调用
        kindType SymbolTable::canUse(int procNum,string name);//获取标识符是否可以使用以及类型的信息
        string SymbolTable::getProcArg(int procIndex,string name,int index);//获取函数的形式参数名
        int SymbolTable::getProcIndexRead(int procNum,string name);//获取可调用函数的位置
        void SymbolTable::addNewPrintString(string s);//增加打印的字符串
        void SymbolTable::showTable(int procNum);//打印符号表
        void SymbolTable::getIdNameAdr(int procNum,string name,int &BL,int &ON);//从当前函数/过程获取可以使用的符号（本层或者外层）的位置信息(二维)
        int SymbolTable::dimCount(int procNum,int idIndex);//获取符号表中某位置之前已使用空间(汇编中计算[ebp+?]使用)
    };
```

可以看出，3 个类之间环环相扣，可以存储符号的所有信息。还有一个好处在于，到最后突然要加一个别的元素给符号时，非常方便，不用担心造成过多的冗余。比如上面 SymbolTableItem 的 regnum 就是我在代码优化时才加上去的，用于最后全局寄存器分配时分配给变量的寄存器编号，默认为-1。

7 语义分析和生成中间代码

7.1 中间代码

中间代码代码是可以完全自己定义的代码，只要语义清晰，操作方便，便于后面的优化，和转成目标代码就可以。

下面给出我设计的原始中间代码：

Begin	function***			函数/过程开始
End	function***			函数/过程结束
Read	del1			读
WriteString	string1			写字符串
WriteExpression	t1（值）			写表达式
value=	vr1	a		传值或传地址
CallValue	function***	#t4		调用函数
call	function***			调用过程
=	#t4		del1	赋值语句
+	var1	var2	var3	代数运算符
-	var1	var2	var3	代数运算符
*	var1	var2	var3	代数运算符
/	var1	var2	var3	代数运算符
<>	choice	1	#t1	比较运算符
>	choice	2	#t1	比较运算符
>=	choice	3	#t1	比较运算符
<	choice	4	#t1	比较运算符
<=	choice	5	#t1	比较运算符
=	choice	6	#t1	比较运算符
IfFalse	#t1	flag		如果 false 则跳转
JmpTo	flag_flag3			跳转
SetFlag	flag_flag4			设置跳转点
[]=	B	k	#t1	取数组地址

此外，在窥孔优化过程中，我化简了部分四元式，具体是将 IfFalse 与比较运算符合并，构成新的四元式如下表所示。

Begin	function***			函数/过程开始
End	function***			函数/过程结束
Read	del1			读
WriteString	string1			写字符串
WriteExpression	t1（值）			写表达式
value=	vr1	a		传值或传地址
CallValue	function***	#t4		调用函数
call	function***			调用过程
=	#t4		del1	赋值语句
+	var1	var2	var3	代数运算符
-	var1	var2	var3	代数运算符
*	var1	var2	var3	代数运算符
/	var1	var2	var3	代数运算符
<>false	choice	1	#flag1	比较跳转运算符
>false	choice	2	#flag1	比较跳转运算符
>=false	choice	3	#flag1	比较跳转运算符
<false	choice	4	#flag1	比较跳转运算符
<=false	choice	5	#flag1	比较跳转运算符
=false	choice	6	#flag1	比较跳转运算符

JmpTo	flag_flag3			跳转
SetFlag	flag_flag4			设置跳转点
[]=	B	k	#t1	取数组地址

8 运行栈设计

8.1 任务

pl0 文法嵌套结构最关键的地方就在于运行栈的设计，便于内外层函数的变量调用。

8.2 具体实现

下面是我的运行栈的设计：

操作区
局部数据区
形参数据区
prev abp
ret addr
传入参数区
Display 区

其中，传入参数区是调用函数（caller）压入运行栈的参数，形参数据区是直接将传入参数区的参数复制到被调用函数（callee）的参数。

这里要特别注意设置 Display 区的算法：

从 i 层模块进入(调用)j 层模块，则：

1.若 j＝i＋1：复制 i 层的 display，然后增加一个指向 i 层模块记录基地址的指针

2.若 j≤i 即调用外层模块或同层模块：将 i 层模块的 display 区中的前面 j－1 个入口复制到第 j 层模块的 display 区

9 代码优化

9.1 窥孔优化

窥孔优化主要是针对一些非常简短的局部代码的优化。我对 SetFlag 重复的地方进行了合并，并且设计了新的四元式，将比较运算符与 iffalse 合并成新的四元式，代码中称为 CompareFalse。具体请见上文的优化后四元式。

9.2 DAG 图(消除公共子表达式)

DAG 图与节点表生成的方法相对比较简单，和书上的算法一样，但特别要注意 DAG 图数据结构的设计。

我设计的数据结构如下：

```
class DAGItem
{
public:
    string Name;
    int Left;
    int Right;
    bool isleaf;
    deque <int> FatherNode;
};
```

从 DAG 图重新生成优化后的四元式，是一个 NP 问题。书上给出了一种启发式算法如下：

1、初始化一个放置 DAG 图中间节点的队列

2、如果 DAG 图中还有中间节点未进入队列，则执行步骤 3，否则执行步骤 5。

3、选取一个尚未进入队列，但其所有父节点均已进入队列的中间节点 n，将其加入队列；或选取没有父节点的中间节点，将其加入队列

4、如果 n 的最左子节点符合步骤 3 的条件，将其加入队列；并沿着当前节点的最左边，循环访问其最左子节点，最左子节点的最左子节点等，将符合步骤 3 条件的中间节点依次加入队列；如果出现不符合步骤 3 条件的最左子节点，执行步骤 2。

5、将中间节点队列逆序输出，便得到中间节点的计算顺序，将其整理成中间代码序列

其实上面的父节点、子节点的算法，就是**拓扑图的消减算法**，所以大家可以从网上搜一些 ACM 这方面的代码学习。可以看出，编译带给我们的是一笔财富。

9.3 全局数据流分析

9.3.1 划分基本块

对于我的中间代码，我划分基本的算法是：

1) 所有中间代码的 Flag 设为 0
2) 读取一条中间代码
3) 如果这条中间代码是 Begin 或者 End 或者 SetFlag，则将这一句的 Flag 设置为 1
4) 如果这条中间代码是 compareFalse 或者 JmpTo，则将这句的下一句的 Flag 设置为 1
5) 没有下一条中间代码，进入(6)，否则进入(2)
6) 两个 1 之间的就是一个基本块（含前面的一个 1，不含后面的 1）
7) 算法结束

9.3.2 确定基本块流向

1) 读取一条指令直到结束或者是一个基本块的最后一句，读完则算法结束
2) 如果这条指令是 compareFalse,则将下一个基本块和 compareFalse 的基本块加入 Out 集合，同时把进入的基本块的 In 集合加入这个基本块编号
3) 如果这条语句是 JmpTo,则将 JmpTo 跳转的基本块加入 Out 集合，同时把进入的基本块的 In 集合加入基本块编号
4) 如果不是以上几种，并且不是 End，则将下一个基本块加入 Out 集合，同时把进入的基本块加入基本块编号
5) 返回（1）

9.3.3 活跃数据流分析之 DefineUse

四元式定义如下(op,target1,target2,result)

定义两个动作：

addUse(id):如果 id 不出现于 Define 中则加入 Use

addDefine(id):如果 id 不出现于 Use 中则加入 Define

1) 读取一条指令，读完则算法结束
2) 如果指令是 WriteExpression,则 addUse(target1)
3) 如果指令是 Read,则 addDefine(target1)
4) 如果指令是运算操作符，则 addUse(target1),addUse(target2),addDefine(result)
5) 如果指令是=，则 addUse(target1),addDefine(result)
6) 如果指令是 compareFalse,value= ,则 addUse(target1)
7) 返回（1）

9.3.4 活跃数据流分析之 InOut

1) 将包括代表流出口基本块的所有基本块的 in 集合，初始化为空集
2) 根据方程 out[B]= $\cup_{B\text{ 的后驱基本块}}$ in[P],in[B]=use[B]∪(out[B]-def[b])，为每个基本块 B 依次计算集合 out[B]和 in[B]。如果计算得到某个基本块 in[B]与此前计算得出的该基本块不同，则循环执行步骤 2，直到所有基本块 in[B]集合不再产生变化为止

9.4 构建冲突图,图着色算法

图着色算法可以完全按照书本上的实现：

如果可供分配 k 个全局寄存器，那么我们就尝试用 k 种颜色给该冲突图着色

1.找到第一个连接边数目小于 K 的节点，将它从图 G 中移走，形成图 G’

2.重复步骤 1，直到无法再从 G’中移走节点

3.在图中选取适当的节点，将它记录为“不分配全局寄存器”的节点，并从图中移走

4.重复步骤 1~步骤 3，直到图中仅剩余 1 个节点

5.给剩余的最后一个节点选取一种颜色，然后按照节点被移走的顺序，反向将节点和边添加进去，并依次给新加入的节点选取颜色

9.5 全局寄存器与局部寄存器池优化

由于我设置了 esi 用于传地址以及数组的寻址，全局寄存器我只分配了 edi 和 ebx，而局部寄存器池我设置了 eax、ecx 与 edi。全局寄存器可由上面的图着色算法得出的结果直接分配，要特别注意要预先留好优化开关，方便全局寄存器的分配。

这里列举局部寄存器池的实现算法：

进入基本块时，清空临时寄存器池

为当前中间代码生成目标代码时，无论临时变量还是局部变量（抑或全局变量和静态变量），如需使用临时寄存器，都可以向临时寄存器池申请

临时寄存器池接到申请后，

如寄存器池中有空闲寄存器，则可将该寄存器标识为被该申请变量占用，并返回该空闲寄存器

如寄存器池中没有空闲寄存器，则将在即将生成代码中不会被使用的寄存器写回相应的内存空间，标识该寄存器被新的变量占用，返回该寄存器

在基本块结尾，或者函数调用发生前，将寄存器池中所有被占用的临时寄存器写回相应的内存空间，清空临时寄存器池

以+ a,b,c（a,b,c 可能有重复）这样的目标代码为例，开始我没有用局部寄存器池优化时，我基本上采用的是

```
Load a,eax;
Load b,ebx;
Add eax,ebx;
Store eax,c
```

这样的复杂结构，但用了全局寄存器以及临时寄存器池，也许就只是这么一句话：

```
Add eax,ebx;
```