

《编译原理》 课程设计说明书



学 号_____1652262_____

姓 名_____涂远鹏_____

专 业_____计算机科学与技术_____

授课老师_____丁志军_____

目 录

一. 实验目的	3
二. 开发环境	3
三. 具体要求	3
四. 需求分析	3
4.1 程序任务输入及其范围	3
4.2 输出形式	4
4.3 程序功能	5
4.4 测试数据	6
五. 概要设计	7
六. 详细设计	8
6.1 词法分析	8
6.2 语法分析	9
6.3 中间代码和汇编代码	14
七. 调试分析	16
7.1 测试结果	16
7.2 时间复杂度分析	29
7.3 遇到的问题及解决	29
八. 用户使用说明	30
九. 问题和解决办法、心得体会	31

一、实验目的

- 1、掌握使用高级程序语言实现一个一遍完成的、简单语言的编译器的方法。
- 2、掌握简单的词法分析器、语法分析器、符号表管理、中间代码生成以及目标代码生成的实现方法。
- 3、掌握将生成代码写入文件的技术。

二、开发环境

编程语言:C++

使用工具: Qt Creator + mingw32 编译器

操作系统:Windows 10

三、具体要求

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

- (1) 使用高级程序语言作为实现语言，实现一个类 C 语言的编译器。编码实现编译器的组成部分。
- (2) 要求的类 C 编译器是个一遍的编译程序，词法分析程序作为子程序，需要的时候被语法分析程序调用。
- (3) 使用语法制导的翻译技术，在语法分析的同时生成中间代码，并保存到文件中。
- (4) 要求输入类 C 语言源程序，输出中间代码表示的程序；
- (5) 要求输入类 C 语言源程序，输出目标代码（可汇编执行）的程序。
- (6) 实现过程、函数调用的代码编译

其中 (1)、(2) (3) (4) 是必做内容，(5) (6) 是选作内容。

四、需求分析

4.1 程序任务输入及其范围

程序输入为类 C 语言程序，其要求为符合以下词法及文法规则：

【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： #

【文法1_包含过程调用】

```

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)*
  
```

4.2 输出形式

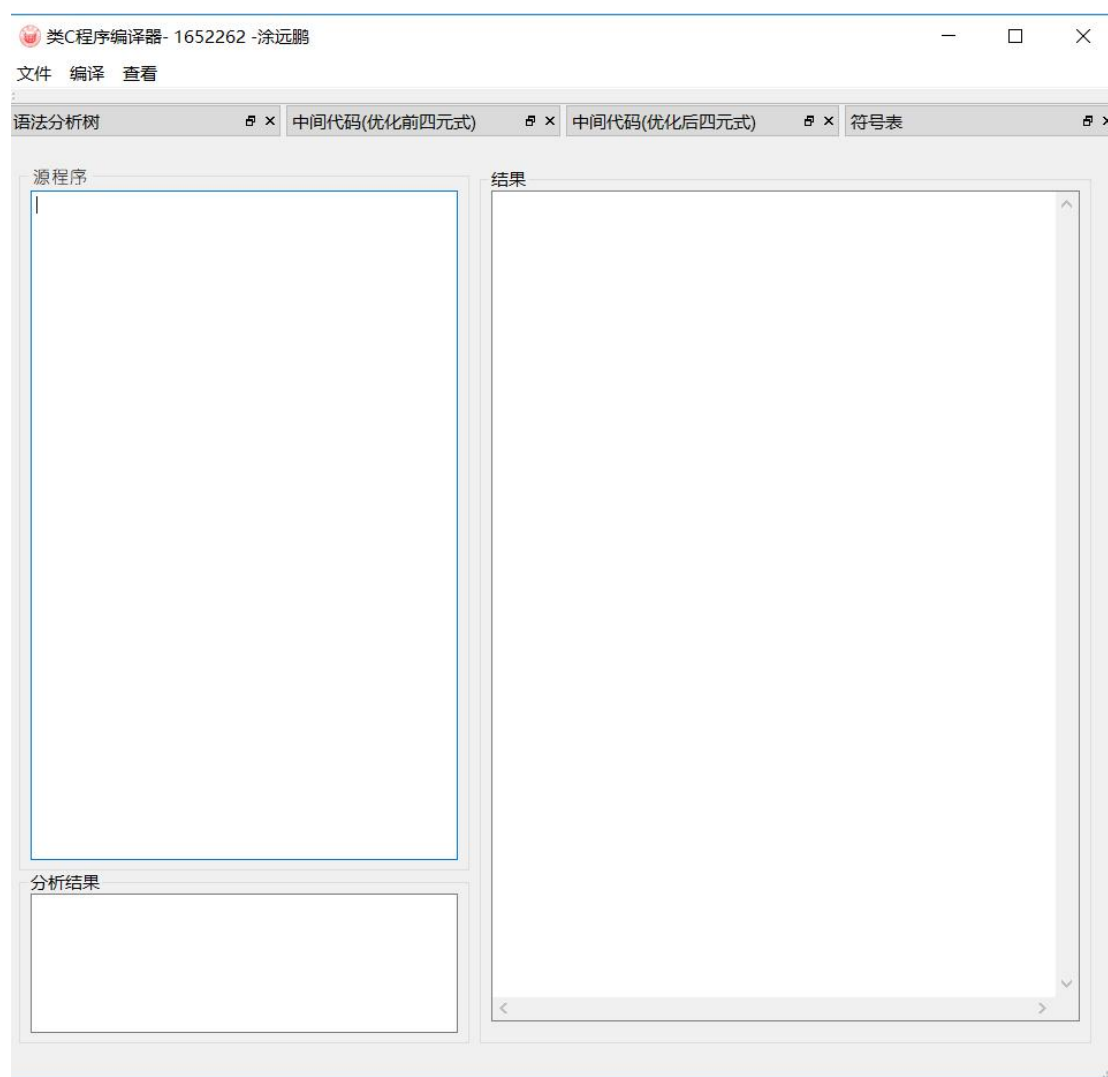
输出形式包含两种，屏幕输出和文件输出，屏幕输出包含词法分析结果（符号表由 tablewidget 展示显示在主窗体中），语法分析结果（语法分析树由 treewidget 展示完整结构，在 DockWidget 浮动窗体中显示），中间代码生成结

果（四元式表由 tablewidget 展示, 在 DockWidget 浮动窗体中显示），优化后的中间代码生成结果（优化后四元式表由 tablewidget 展示, , 在 DockWidget 浮动窗体中显示），目标代码生成结果（汇编代码由 textBrowser 文本框展示, 在 DockWidget 浮动窗体中显示）。

输出到文件的格式包含三种, 第一种为词法分析结果, 第二为中间代码结果, 第三个为目标代码结果, 由于语法生成树无法输出到文件, 所以只能在屏幕上以 treewidget 的形式展示。

4.3 程序功能

程序主界面如下图所示：



程序界面分为四个区域。图中第一片区域是代码编辑区，打开或新建的代码在该区域进行查看和编辑；第二片区域是编译结果输出文本框，该区域可以显示词法分析、目标代码的生成结果；第三片区域是编译提示区，该区域可以显示编译的结果，在编译错误的情况下显示相关的错误信息。第四片区域为四个悬浮窗口，分别显示语法分析结果（语法分析树）、优化前的四元式、优化后的四元式、

符号表。

通过按键对程序功能的描述如下表所示：

选项		功能
文件	打开	打开已有的 .c/.txt/.cpp 文件
	新建	新建源代码
	退出	退出程序
	保存	将源代码保存到当前路径
	关闭	关闭打开的源代码
	另存为	将源代码保存到另一指定路径
分析		对源代码进行编译并生成相关文件
查看	词法分析	查看词法分析的结果
	语法分析	查看语法分析的结果
	优化前中间代码	查看生成的中间代码
	目标代码	查看生成的目标代码
	优化后中间代码	查看优化后的四元式表
	符号表	查看 LR 生成的符号表

4.4 测试数据

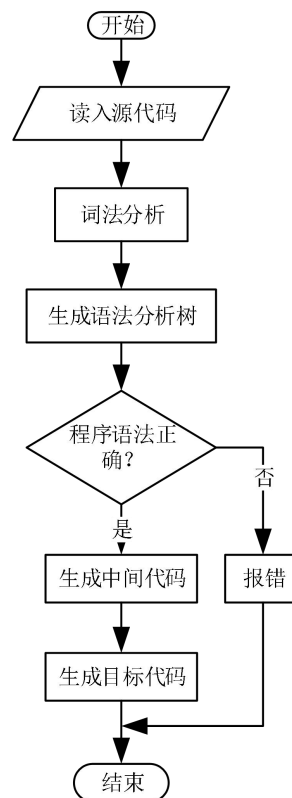
类 C 语言程序实例（包含过程调用）：

```
int a;
int b;
int program(int a,int b,int c)
{
    int i;
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    while(i<=100)
    {
        i=j*2;
        j=i;
    }
    return i;
}
```

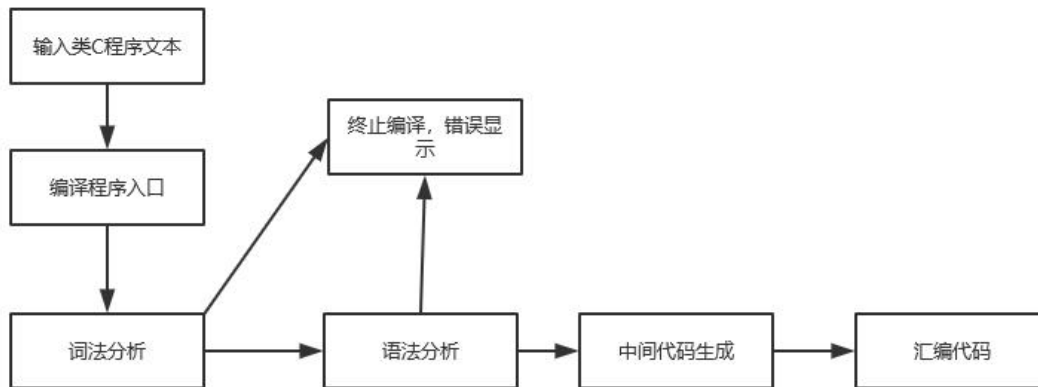
```
}  
int demo(int a)  
{  
    a=a+2;  
    return a*2;  
}  
void main(void)  
{  
    int a;  
    int b;  
    int c;  
    a=3;  
    b=4;  
    c=2;  
    a=program(a,b,demo(c))  
    return;  
}
```

五、概要设计

类 C 语言程序编译器的程序流程图大致如下所示



整体框架如下：



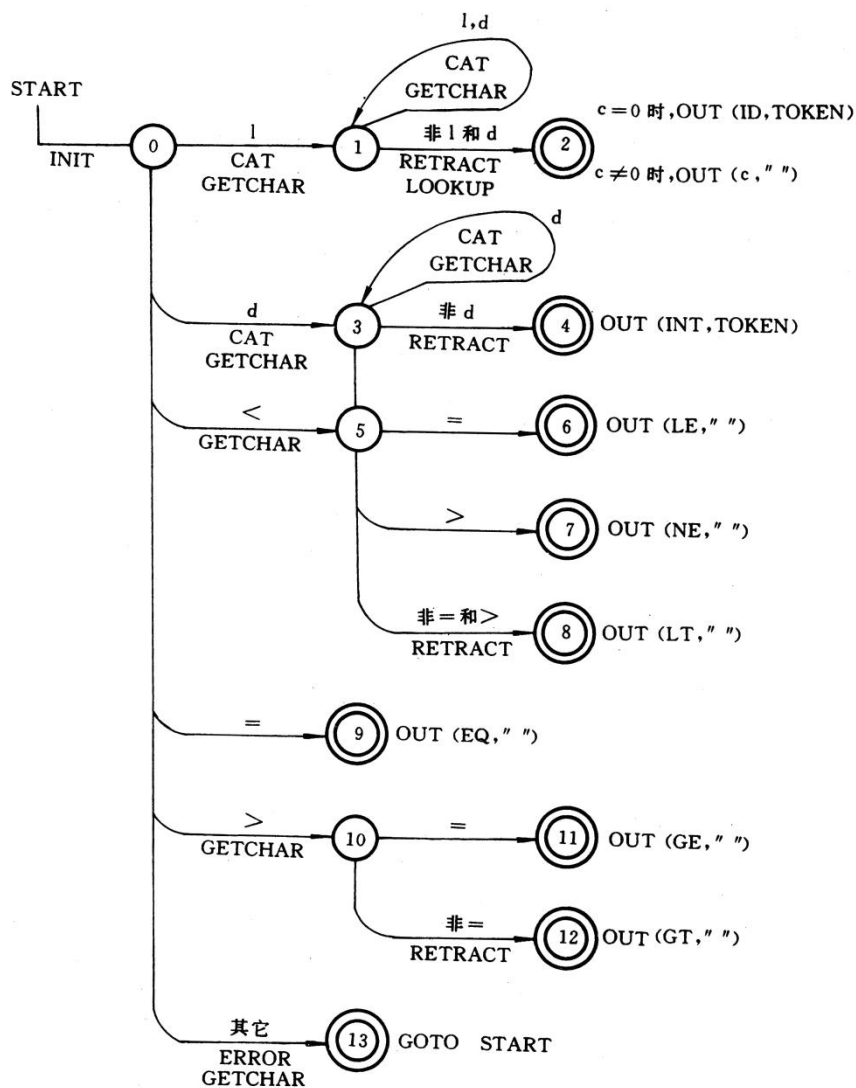
六、详细设计

6.1 词法分析部分

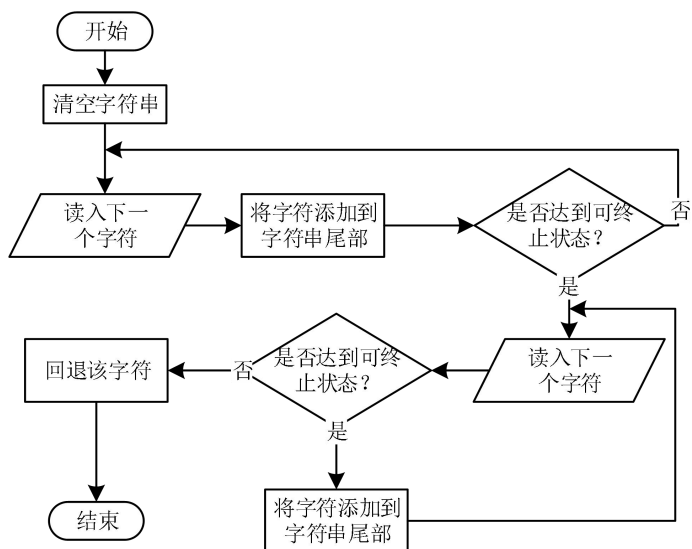
A. 词法规则

词项	编码
int	1
void	2
if	3
else	4
while	5
return	6
ID	7
NUM	8
=	9
+	10
-	11
*	12
/	13
repo	14
;	15
,	16
(17
)	18
{	19
}	20
#	-1
NONE	0

B. 状态转换图



C. 核心算法流程图



6.2 语法分析

6.2.1 所用的语法如上面所述，这里不再赘述

6.2.2 LR(1) 算法实现详述

6.2.2.1 求 First 集算法

连续使用下面的规则，直到每个集合 FIRST 不再增大为止：

- (1) 若 X 属于 VT ，则 $FIRST(X) = \{X\}$ 。
- (2) 若 X 属于 VN ，且有产生式 $X \rightarrow a\cdots$ ，则把 A 加入到 $FIRST(X)$ 中；若 $X \rightarrow \xi$ 也是一条产生式，则把 ξ 也加入到 $FIRST$ 中。

6.2.2.2 构造 CLOSURE(I) 算法

- (1) I 中的所有项目都属于 $CLOSURE(I)$ ；
- (2) 若项目 $[A \rightarrow a \cdot B \beta, a]$ 属于 $CLOSURE(I)$ ， $B \rightarrow \xi$ 是文法的一个产生式，则对于任何 $b \in FIRST(\beta a)$ ，如果 $[B \rightarrow \cdot \xi, b]$ 原来不在 $CLOSURE(I)$ 中，则把它加入 $CLOSURE(I)$ ；
- (3) 重复执行步骤 (2)，直到 $CLOSURE(I)$ 不再增大为止

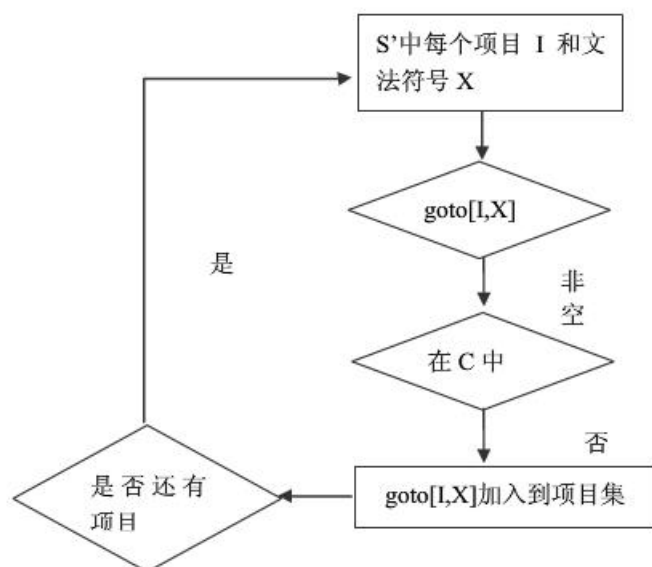
6.2.2.3 LR(1) 项目集族 C 构造步骤：

- 1、建立有限状态自动机 DFA、哈希表 H 、项目集队列 P ，放入初始项目集 I_0 ，存入哈希表中并作为 DFA 的初始状态。
- 2、取出队首元素 I ，对于 I 的每个项目 X ，求 $I' = GO(I, X)$ ，若 I' 不在哈希表中，则将其加入 P 和 H 中，并添加为 DFA 新的状态。为 DFA 添加一条边 (I, X, I') 。
- 3、循环此操作直到 P 为空为止，DFA 即代表了文法 G 的 LR(1) 项目集族。

伪代码：

```
BEGIN
  I0: C = {closure([S' → • S, #])}
  FOR C 中的每个项目集 I 和 G' 的每个符号 X DO
    IF GO(I, X) 非空且不属于 C, THEN 把 GO(I, X) 加入 C 中
  UNTIL C 不再增大
END
```

流程图：



6.2.2.4 $GOTO(I, X)$ 算法计算算法

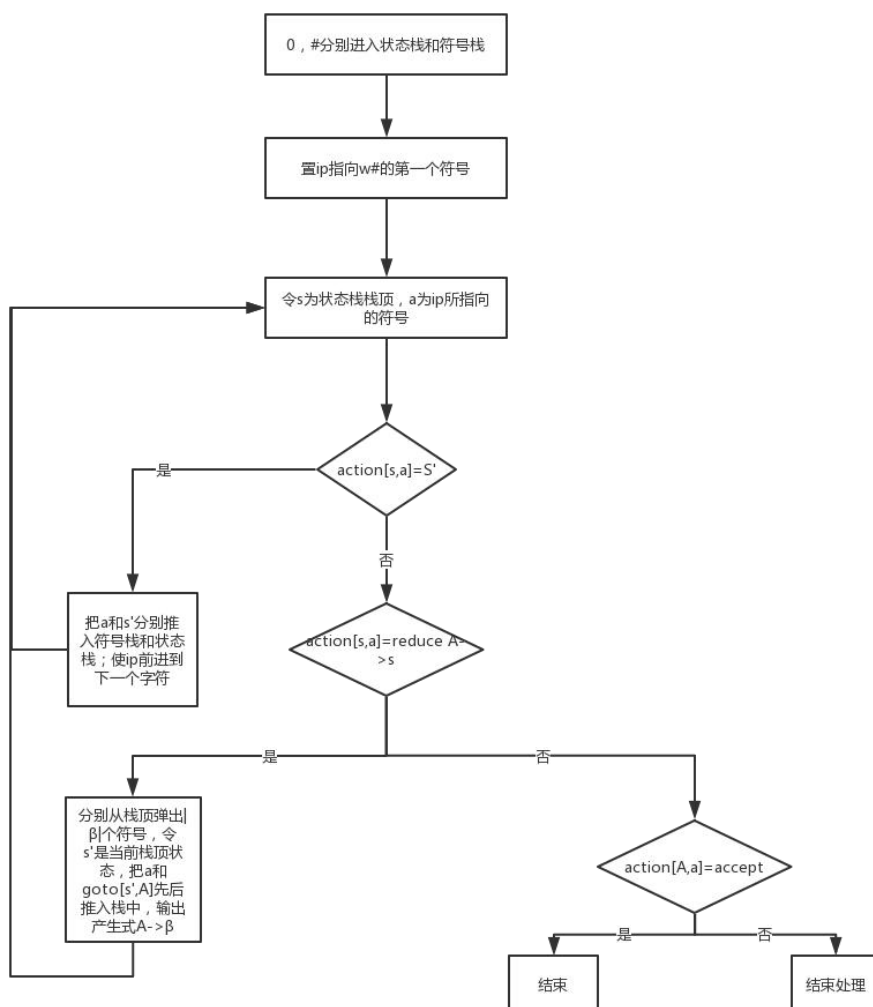
$GOTO(I, X) = CLOSURE(J)$ 其中 $J = \{ \text{任何形如 } [A \rightarrow aX. B, a] \text{ 的项目 } [A \rightarrow a. X. B, a] \text{ 属于 } I \}$

6.2.2.5 LR(1) 预测分析表生成算法

- 1、若项目 $[A \rightarrow \cdot a, b]$ 属于 I_k 且 $GOTO(I_k, a) = I_j$, a 为终结符, 则置 $ACTION[k, a]$ 为 “ s_j ”。
- 2、若项目 $[A \rightarrow \cdot a]$ 属于 I_k , 则置 $ACTION[k, a]$ 为 “ r_j ”; 其中假定 $A \rightarrow$ 为文法 G 的第 j 个产生式。
- 3、若项目 $[S \rightarrow S\cdot, \#]$ 属于 I_k , 则置 $ACTION[k, \#]$ 为 “acc”。
- 4、若 $GOTO(I_k, A) = I_j$, 则置 $GOTO[k, A] = j$ 。
- 5、分析表中凡不能用规则 1 至 4 填入信息的空白栏均填上 “出错标志” (即将表中空白格均置为 “报错标志”)。

在实现 $GOTO(I, X)$ 时, 记录下状态的转化。得到分析表中的移进部分。然后再扫描所有的项目集, 找到其中包含归约项目的哪些项目集, 根据其中项目, 得到分析表中那些规约的部分。

6.2.2.6 LR(1) 驱动程序流程图:



6.2.3 数据结构

yfq 类		
变量名	类型	作用
end_flag	int	程序结束标志符
read_main	int	是否已读过 main 函数
num_zdkh	int	左大括号数量, 用于判断是否为全局变量
return_flag	int	是否为返回语句
return_words[100]	char	返回字数组
c	char	临时读入变量
bl_name[100]	char	临时记录变量名
hs_name[100]	char	临时记录函数名
len_hs	int	从函数名到左括号的长度
avaliable_T	int	Ti, i=avaliable_T
avaliable_L	int	Li, i=avaliable_L

var_name[10][100]	char	记录参数名
kh[100]	int	记录读到的右括号
len_kh	int	从函数到右括号长度
headwords[100]	char	句首字符
is_hs_head	int	是否函数居于句首标志
is_hs_dy	int	是否为函数定义
len_hs	int	函数信息表长度
break_hs	int	在函数名数组中搜索给定函数名时断开的数
hs[500]	hs_info	函数信息表数组
len_global_bl	int	全局变量长度
global_bl[500]	bl_info	全局变量名数组
len_local_bl	int	局部变量长度
local_bl[500]	bl_info	局部变量名数组
fin	ifstream	类 C 语言程序文件句柄
fin1	ifstream	文法文件句柄
fout1	ofstream	中间代码文件句柄
fout2	ofstream	汇编代码文件句柄
函数名	类型	作用
zs_handle()	bool	判断是否为有效注释 && 去除注释
u_handle()	bool	因子处理函数
s_handle()	bool	加法表达式处理函数
exist_bl(char name[], int mode)	bool	判断变量名是否存在, mode=1: 局部变量 mode = 0: 全局变量
exist_hs(char name[], int var_num)	bool	判断函数名是否存在 && 参数数目是否正确
insert_bl(char name[], int mode)	void	将定义的变量插入 global_bl[], local_bl[], mode 0->全局变量 1->局部变量
insert_hs(char name[], int var_num)	void	将定义的函数插入 hs[], var_num: 参数
delete_bl()	void	读到'}'删除该区间读到的变量, '{'开始(并生成中间代码)
right_num(char c[], int len_c)	bool	判断是否为合法数字
fh_rank(char fh)	int	计算符号等级
r_handle(char c[], int len_c, int mode)	bool	表达式处理函数
bl_handle	bool	句首为变量的处理函数

void_handle	bool	句首为 void 的处理函数
int_handle	bool	句首为 int 的处理函数
get_kind(char c[], int len_c)	int	合法变量名，合法数字
error_rep()	void	报错及相关处理函数
readhead()	int	读句首
analyze()	void	整体分析函数

```
struct bi_info{//记录变量结构体
char bl_name[20];
int pos;
int num;
};
```

```
struct hs_info{ //记录函数的数据结构
char hs_name[20];
int f_return;
char w_return[20];
int var_num;
};
```

全局变量：

```
int available_reg[32]; //有效寄存器号数组
int max_num; //reg1 恒定值为 0，全局变量从$31 向前分配寄存器， max_num
为全局变量使用的最小的寄存器
```

6.2.4 设计思路

6.2.4.1 总体设计架构

本程序使用了 LR1 的设计思想，并做了较大的改动。我最初的想法是严格按照 LR1 算法的思想，以 program 为 S'，并根据状态转换图画出 action 表。但是，在实现的过程中，发现这样的问题很大。首先，类 c 语法有 26 个产生式，要准确画出全部的状态转换图是十分困难的。其次，这种方式也是十分不必要的。比如，当程序读到了 int，那么接下来的步骤我们可想而知是继续读，反映到 LR1 算法中，就是当前状态 i 与当前待输入字符，对应的为 S_j。而若读到了 ';'，则对应的一定为 r_j。所以当读到 ';'、' '、'}' 等就一定是指向规约操作，而读到其他的就会继续读入。所以，该程序以 LR1 算法思想为指导，并根据对 c 语言语法的理解。

6.2.4.2 变量和函数声明的处理

Void 关键字一定是函数声明。Void_handle() 继续读入 void 后序部分，直到 '{'，若 exist_hs() == yes || exist_hs() == error 说明已有该函数名，或函数名/参数名命名错误。会报错，因为该编译器不支持函数重载。不然，将函数名和参数数目写入数

组 Hs_info hs[]。无参函数形式为 函数名(), 不是 (void)。函数是否需要返回标志 f_return=0; (只能根据定义时是 int 还是 void 来判断, 意味着如果 void 函数有返回值或 int 函数无返回值这种错误是没办法发现的)。

Int 关键字可能是函数定义, 也可能是变量定义, 其中变量定义还分为局部变量定义和全局变量定义。先判断是函数定义还是变量定义, 从 int 后第一个非' '的字符开始读到) 或者;。然后判断方法及操作同 void。如果是变量, 根据 f_inside(初始化为 0, 每读到一个' { ', f_inside++; 读到一个' }', f_inside--) 是否为 0 判断此时是否为全局 还是局部变量。如果是全局变量, 并且 exist_bl(变量名, 0) == no 则插入全局变量数组; 如果是局部变量, 并且 exist_bl(变量名, 1) == no, 则插入局部变量数组。尤其是, 当读到了' { ', 也需要插入到局部变量数组。

变量声明的过程不会有寄存器的分配, 只有在使用变量的过程中才会有寄存器分配。

当给全局变量赋值时, 会将该值存储在内存中。而当使用该变量时(等号右边), 则会为该变量分配寄存器。从\$31 开始依次向前 分配寄存器。其 pos 置 2。当为局部变量赋值时, 会将该变量的值存储在内存中, 当使用该变量的时候, 会从内存中取出该值分配给寄存器。寄存器分配过程: 从 available_reg 中取最小的为 1 的值 j, 并把 \$j 分配给该变量, 同时, 该变量的 pos 置 2。

```
for(int j=1; j<max_reg; j++) {
    if(available_reg[j]==1)
        break;
}
```

当读到了' }', 则会在局部变量栈中持续退栈, 直至第一个' { '也退栈, 该过程即为局部变量的释放。

6.2.4.3 函数调用处理

函数调用若有参数, 则需要先把参数入栈(将实参的 pos、num 赋予形参), 类似于传地址的参数传递方式。如果函数有返回值, 则先传值, 再 return; 否则, 直接 return。

6.2.4.4 表达式 expression 处理

对有小括号的情况处理的不好, 经常会程序中断, 没有解决。将整个表达式 如: if(r), 则 r 为表达式 读入并传递给 r_handle(), r_handle() 根据<, >, =进行拆分, 如 s1<s2, 则将 s1, s2 分别传递给 s_handle()。r_handle() 扫描, 确定* / (rank = 2) 和 + - (rank = 1) 的数目 num1, num2, 将栈和 num1, num2 传递给 s_handle()。然后 s_handle() 利用栈和递归函数的方法最终得到一个空栈:

如有 s1 = a+b*c+d 则初始的符号栈为 a+b*c+d, 读到*, 则将 b*c 退栈, Ti 入栈, num2-- (表示 rank=2 的数目), 并将栈 a+Ti+b, num1, num2 传递给下一个调用的 s_handle()。

6.2.4.5 if, while 条件语句处理

if 和 while 翻译成中间代码的形式分别为:

if	L0: (jx, a, b, L2) L1: (j, -, -, \$)
----	---

	L2: (-, -, -, -) Li:
while	L0: (jx, a, b, L2) L1: (j, -, -, \$) L2: (-, -, -, -) Li-1: (j, -, -, L0) Li:

\$表示待定的地址，x 表示相应的操作符。因为无法在第一遍扫描即得知\$的值，所以，要先把字符串“L0”、“:”、“(“、“jx”、“,”、“a”...”\$”、“:”读到一个二维字符数组 s[500][20]。然后，当\$:确定后，再把 s[][] 写入中间代码文件，其中\$用最后的地址(Li)来替代。

6.2.4.6 结束语句处理

读到 main 函数后，read_main=1;f_kh 初始化为 0，每读到一个” { “，f_kh++，读到一个” } ” f_kh--。若 read_main = 1 && f_kh == 0，程序结束。大题程序框架如下：

```
while (end_flag == 0) {
    int headword = readhead();
    if (headword == _bl) {
        ...
    }
    else if (headword == _hs) {
        ...
    }
    else if (headword == _void) {
        ...
    }
    else if (headword == _int) {
        ...
    }
    else if (headword == _if) {
        ...
    }
    else if (headword == _while) {
        ...
    }
    else if (headword == _return) {
        ...
    }
    else if (headword == _ydkh) {
        ...
    }
    else {
        error();//出错
    }
}
```

6.3 生成中间代码和汇编代码

由于使用的是语法制导，所以在上面语法检测分析的同时，中间代码和目标代码已生成：

扩充分析栈：增设语义栈

1. 在语法分析的基础上，增加一个语义栈，用于存放分析栈中文法符号所对应的属性值，栈内元素为语义结点。
2. 结点类是 S 属性文法的表示，判别每次语法分析所使用的产生式，实现不同的语义动作，

每当规约到特定非终结符时，即可产生中间代码。

扩充 LR 分析器功能：当执行规约产生式动作时，也执行产生式对应的语义动作。并且由于是归约时执行语义动作，限制语义动作仅能放在产生式右部的最右边。

符号表和函数表：每次规约识别出一个新的标识符，都会将其加入符号表中，符号的信息包括标识符、中间变量名、类型、占用空间、内存偏移量、作用的函数等。而当规约到函数定义的时候，则将函数名、形参列表、代号加入函数表，如下图所示：

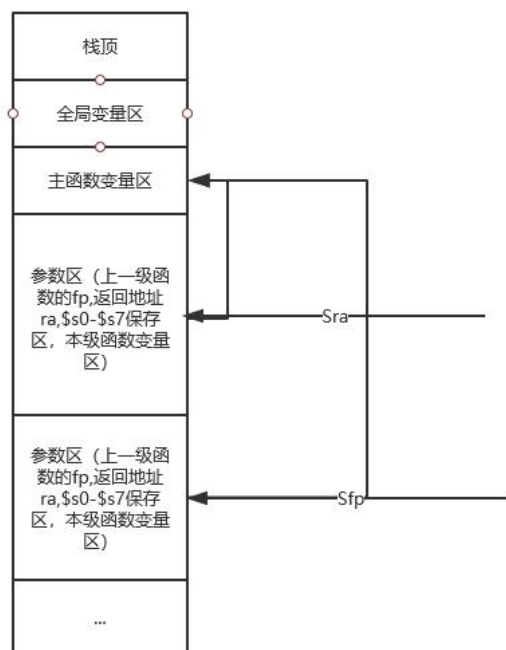
index	name	kind	value	address	paranum	isV
0	a	13	0	0	0	0
1	b	13	0	0	0	0
2	e	13	0	0	0	5
3	hj	4	0	0	0	0
4	program	2	1	0	0	0
5	a	3	0	0	0	0
6	b	3	0	1	0	0
7	c	3	0	2	0	0
8	i	13	0	0	0	0
9	j	13	0	0	0	0
10	m	13	0	0	0	0
5	demo	2	1	0	0	0
6	a	3	0	0	0	0
7	i	13	0	0	0	0
6	main	2	0	0	0	0
7	a	13	0	0	0	0
8	b	13	0	0	0	0
9	c	13	0	0	0	0

中间代码生成：将语句分为赋值语句、算术运算语句、函数调用语句、循环语句、选择语句、跳出语句、函数定义语句分别处理。

6.3.1 四元式设计

四元式	含义
(=, 2, , temp)	temp=2;
(call, f, , a)	a=f();
(call, f, ,)	f();
(<=, ., , a, b,)	a<=b;
(jne, ., , label)	if not satisfy(==false) then jump
(jmp, ., , label)	jump to label
ret	return
Par a	f(int a, ---)

6.3.2 存储分配方案



6.3.3 Mips 指令语义

含义	mips 指令	表达式
取立即数	li \$s1, 100	\$s1=100
加	add \$s1, \$s2, \$s3	\$s3=\$s1+\$s2
立即数加	addi \$s1, \$s2, 100	\$s3=\$s1+100
立即数减	subi \$s1, \$s2, 100	\$s3=\$s1-100
减	sub \$s1, \$s2, \$s3	\$s3=\$s1-\$s2
乘	mult \$s2, \$s3	Hi, Lo=\$s2*\$s3
除	div \$s2, \$s3	Lo=\$s2/\$s3 Hi=\$s2 Mod \$s3
取字	lw \$s1, 100(\$s1, 100(\$s2))	\$s1=memory[\$s1=memory[\$s2+100]]
存字	sw \$s1, 100(\$s1, 100(\$s2))	memory[\$s2+100]=\$s2+100]=\$s1
Beq	beq \$s1, \$s2, 100	if (\$s1==\$s2) goto PC+4+400
Bne	bne \$s1, \$s2, 100	if (\$s1!=\$s2) goto PC+4+400
Slt	slt \$s1, \$s2, \$s3	if (\$s2<\$s3) \$s1=1; else \$s1=0;
j	j 2000	Goto 8000;
jr	ja \$ra	Goto \$ra;
jal	jal 2500	\$ra=PC+4; Goto 10000;

6.3.4 存储器动态管理

```

struct bl_info
{
    char bl_name[20];
    int pos;    //0: 寄存器 1: 内存 2: 都在
    int num;    //寄存器号
};

```

当给全局变量赋值时，会将该值存储在内存中。而当使用该变量时（等号右边），则会为该变量分配寄存器。从\$31开始依次向前分配寄存器。其pos置2。当为局部变量赋值时，会将该变量的值存储在内存中，当使用该变量的时候，会从内存中取出该值分配给寄存器。寄存器分配过程：从available_reg中取最小的为1的值j，并把\$j分配给该变量，同时，该变量的pos置2。

当读到了'}'，则会在局部变量栈中持续退栈，直至第一个'{ '也退栈，该过程即为局部变量的释放。

6.3.5 四元式转汇编语言

对于四元式(op, num1, num2, result):

当 op==:

若 num1 为数，则操作为 li \$t0 2 sw \$t0 -24(\$fp) 若 num2 为变量，则先把 num2 移动到寄存器，再进行赋值操作： sw \$t0 -8(\$fp) li \$t0 2 sw \$t0 -24(\$fp)

当 op == + - * / :

1、若 num1、num2 为数，则只需要再分配一个寄存器，将 num1 的值存储进寄存器，op \$1 num2

2、若 num1、num2 一个为数，一个为变量，则为变量分配寄存器，Op \$1 num2，并置 bl.pos = 0，表示该变量的值只存在于内存中。记录新变量的属性。

3、若 num1、num2 都为变量，则为这两个变量都分配寄存器，然后，Op \$1 \$2 同时置 bl1.pos = 0，记录新变量的属性。

当 op 为 call, ret, if, while, jne: 对应 bne, j, jnz, jne, jg, jge, jl, jle 等跳转汇编指令。

当 op 为 int, para 等变量定义语句: 对应 li, sw 等存储相关汇编指令。

当 op 为 func 等函数定义语句: 对应生成 label 标签汇编指令。

七、调试分析

1. 测试结果

正确数据测试：

类C语言源程序如下：

```

int a;
int b;
int program(int a, int b, int c)
{
    int i;
    int j;
    i=0;

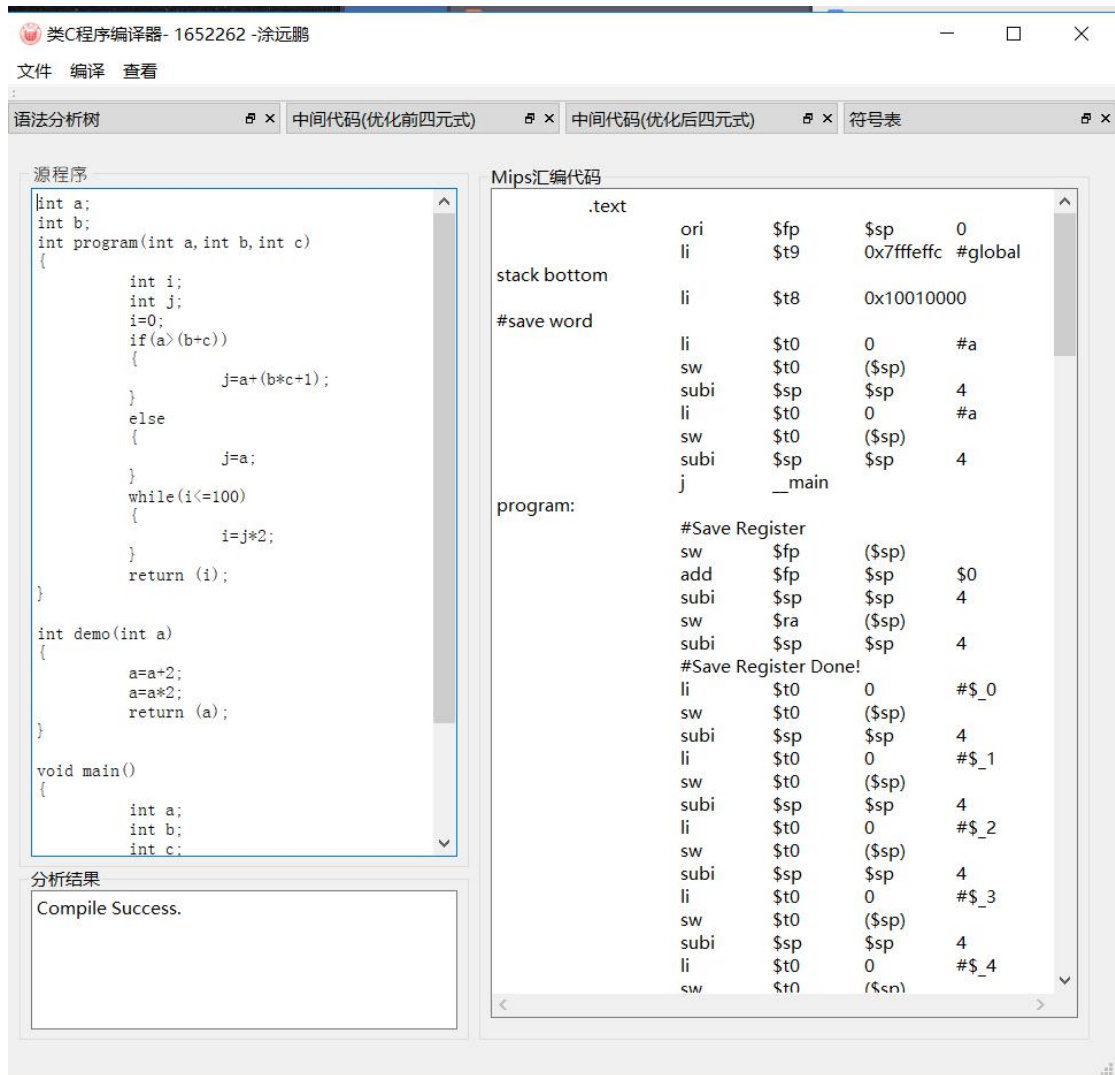
```

```
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    while(i<=100)
    {
        i=j*2;
    }
    return (i);
}

int demo(int a)
{
    a=a+2;
    a=a*2;
    return (a);
}

void main()
{
    int a;
    int b;
    int c;
    a=3;
    b=4;
    c=2;
    a=program(a, b, demo(c));
    return;
}
```

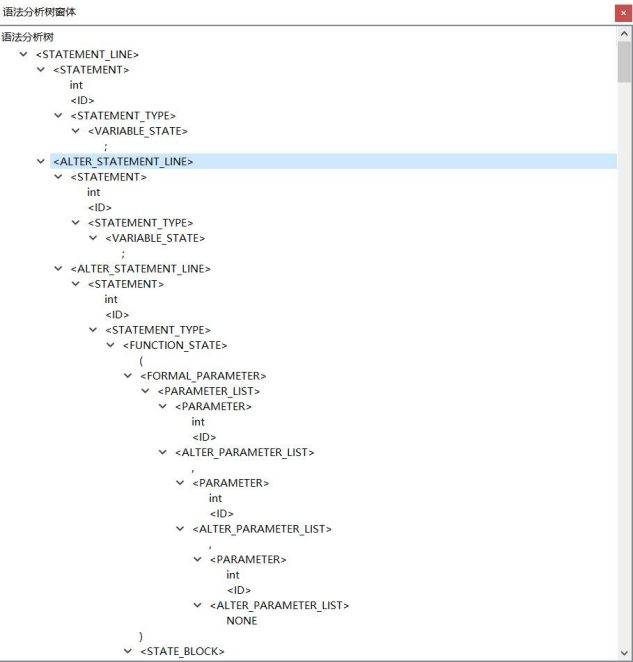
输入正确的类 C 源文件，输出显示“分析成功”，可以显示四种分析结果，并且在 Windows 底下 Mars4_5.jar 运行，成功运行：



生成的词法分析表：

词法分析表		
	单词值	单词类型
1	int	关键字
2	a	标识符
3	;	界符
4	int	关键字
5	b	标识符
6	;	界符
7	int	关键字
8	program	标识符
9	(界符
10	int	关键字
11	a	标识符
12	,	界符
13	int	关键字
14	b	标识符
15	,	界符
16	int	关键字
17	c	标识符
18)	界符
19	{	界符

生成的语法分析树：



生成的优化前的中间代码（四元式表）：

类C程序编译器 - 1652262 - 涂远鹏

文件 编译 查看

中间代码(优化后四元式) 符号表

优化后中间代码窗体

	Operation	Var1	Var2	Var3
1	int			a
2	int			b
3	func	int		program
4	para	int		a
5	para	int		b
6	para	int		c
7	int			i
8	int			j
9	=	0		i
10	+	b	c	\$_0
11	>	a	\$_0	
12	jne			_LABEL_0
13	*	b	c	\$_1
14	+	\$_1	1	\$_2
15	+	a	\$_2	\$_3
16	=	\$_3		j
17	jmp			_LABEL_1
18	lab:			_LABEL_0
19	=	a		j
20	lab:			_LABEL_1

生成的优化后的中间代码（四元式表）：

优化后中间代码窗体

	Opeartion	Var1	Var2	Var3
1	int			a
2	int			b
3	func	int		program
4	para	int		a
5	para	int		b
6	para	int		c
7	int			i
8	int			j
9	=	0		i
10	+	b	c	\$_0
11	>	a	\$_0	
12	jne			_LABEL_0
13	*	b	c	\$_1
14	+	\$_1	1	\$_2
15	+	a	\$_2	\$_3
16	=	\$_3		j
17	jmp			_LABEL_1
18	lab:			_LABEL_0
19	=	a		j
20	lab:			_LABEL_1
21	<=	i	100	
22	jne			_LABEL_3

LR(1)的符号表:

符号表窗体

	index	name	kind	value	address	paranum	isVec
1	-----	-----	-----	-----	-----	-----	-----
2	1	a	13	0	0	0	0
3	2	b	13	0	0	0	0
4	3	program	2	1	0	0	0
5	4	a	3	0	0	0	0
6	5	b	3	0	1	0	0
7	6	c	3	0	2	0	0
8	7	i	13	0	0	0	0
9	8	j	13	0	0	0	0
10	-----	-----	-----	-----	-----	-----	-----
11	4	demo	2	1	0	0	0
12	5	a	3	0	0	0	0
13	-----	-----	-----	-----	-----	-----	-----
14	5	main	2	0	0	0	0
15	6	a	13	0	0	0	0
16	7	b	13	0	0	0	0
17	8	c	13	0	0	0	0
18							

生成的汇编代码:


```

Mips汇编代码
.text
    ori    $fp    $sp    0
    li     $t9    0x7ffeffc #global
stack bottom
    li     $t8    0x10010000
#save word
    li     $t0    0        #a
    sw     $t0    ($sp)
    subi   $sp    $sp    4
    li     $t0    0        #a
    sw     $t0    ($sp)
    subi   $sp    $sp    4
    j      __main
program:
    #Save Register
    sw     $fp    ($sp)
    add    $fp    $sp    $0
    subi   $sp    $sp    4
    sw     $ra    ($sp)
    subi   $sp    $sp    4
    #Save Register Done!
    li     $t0    0        #$_0
    sw     $t0    ($sp)
    subi   $sp    $sp    4
    li     $t0    0        #$_1
    sw     $t0    ($sp)
    subi   $sp    $sp    4
    li     $t0    0        #$_2
    sw     $t0    ($sp)
    subi   $sp    $sp    4
    li     $t0    0        #$_3
    sw     $t0    ($sp)
    subi   $sp    $sp    4
    li     $t0    0        #$_4
    sw     $t0    ($sp)

```

```

.text
ori $fp $sp 0
li $t9 0x7ffeffc #global stack bottom
li $t8 0x10010000 #save word
li $t0 0 #a
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0 #a
sw $t0 ($sp)
subi $sp $sp 4
j __main
program:
#Save Register
sw $fp ($sp)
add $fp $sp $0
subi $sp $sp 4
sw $ra ($sp)
subi $sp $sp 4
#Save Register Done!
li $t0 0 #$_0
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0 #$_1
sw $t0 ($sp)
subi $sp $sp 4

```

```
li $t0 0 #$_2
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0 #$_3
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0 #$_4
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0 #program
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0 #program
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0
sw $t0 -28($fp)
lw $t0 8($fp)
lw $t1 4($fp)
add $t0 $t0 $t1
sw $t0 -8($fp)
lw $t0 12($fp)
lw $t1 -8($fp)
slt $t0 $t1 $t0
bne $t0 1 _LABEL_0
lw $t0 8($fp)
lw $t1 4($fp)
mul $t0 $t0 $t1
sw $t0 -12($fp)
lw $t0 -12($fp)
li $t1 1
add $t0 $t0 $t1
sw $t0 -16($fp)
lw $t0 12($fp)
lw $t1 -16($fp)
add $t0 $t0 $t1
sw $t0 -20($fp)
lw $t0 -20($fp)
sw $t0 -32($fp)
j _LABEL_1
_LABEL_0:
lw $t0 12($fp)
sw $t0 -32($fp)
_LABEL_1:
```

```
_LABEL_2:
lw $t0 -28($fp)
li $t1 100
slt $t0 $t1 $t0
li $t1 1
sub $t0 $t1 $t0
bne $t0 1 _LABEL_3
lw $t0 -32($fp)
li $t1 2
mul $t0 $t0 $t1
sw $t0 -24($fp)
lw $t0 -24($fp)
sw $t0 -28($fp)
j _LABEL_2
_LABEL_3:
lw $v0 -28($fp)
j __FEND_LAB_1
__FEND_LAB_1:
lw $ra -4($fp)
add $sp $fp $0
lw $fp ($fp)
jr $ra
demo:
#Save Register
sw $fp ($sp)
add $fp $sp $0
subi $sp $sp 4
sw $ra ($sp)
subi $sp $sp 4
#Save Register Done!
li $t0 0 #$_0
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0 #$_1
sw $t0 ($sp)
subi $sp $sp 4
lw $t0 4($fp)
li $t1 2
add $t0 $t0 $t1
sw $t0 -8($fp)
lw $t0 -8($fp)
sw $t0 4($fp)
lw $t0 4($fp)
li $t1 2
```

```
mul $t0 $t0 $t1
sw $t0 -12($fp)
lw $t0 -12($fp)
sw $t0 4($fp)
lw $v0 4($fp)
j __FEND_LAB_2
__FEND_LAB_2:
lw $ra -4($fp)
add $sp $fp $0
lw $fp ($fp)
jr $ra
__main:
#Save Register
sw $fp ($sp)
add $fp $sp $0
subi $sp $sp 4
sw $ra ($sp)
subi $sp $sp 4
#Save Register Done!
li $t0 0 #$_0
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0 #$_1
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0 #main
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0 #main
sw $t0 ($sp)
subi $sp $sp 4
li $t0 0 #main
sw $t0 ($sp)
subi $sp $sp 4
li $t0 3
sw $t0 -16($fp)
li $t0 4
sw $t0 -20($fp)
li $t0 2
sw $t0 -24($fp)
lw $t0 -24($fp)
sw $t0 ($sp)
subi $sp $sp 4
jal demo
```

```

nop
sw $v0 -8($fp)
lw $t0 -16($fp)
sw $t0 ($sp)
subi $sp $sp 4
lw $t0 -20($fp)
sw $t0 ($sp)
subi $sp $sp 4
lw $t0 -8($fp)
sw $t0 ($sp)
subi $sp $sp 4
jal program
nop
sw $v0 -12($fp)
lw $t0 -12($fp)
sw $t0 -16($fp)
j __FEND_LAB_3
__FEND_LAB_3:
lw $ra -4($fp)
add $sp $fp $0
lw $fp ($fp)
li $v0 10
syscall

```














测试生成的汇编代码是否正确, 显示 Assemble operation completed successfully:

The screenshot shows the MARS 4.5 IDE interface. The main window displays the assembly code for a MIPS program. The code includes instructions for setting up the stack frame, saving registers, and performing a system call. The right panel shows the registers, and the bottom panel shows the 'Messages' window with the output 'Assemble: assembling C:\Users\admin\Desktop\mips1.asm' and 'Assemble: operation completed successfully'.

结果也与原类 C 程序实际执行结果一致。

同时词法分析、优化前后的中间代码、汇编代码、符号表都输出到文件, 文件名分别为:

laxrst.txt, midcode.txt, optMidCode.txt, asmrst.asm, symbolTable.txt:

	asmrst.asm	2019-05-17 15:53	ASM 文件
	Compiler.exe	2019-05-16 22:15	应用程序
	grammar-en.txt	2018-12-14 19:53	文本文档
	laxrst.txt	2019-05-17 15:53	文本文档
	libgcc_s_dw2-1.dll	2015-12-29 6:25	应用程序扩展
	libstdc++-6.dll	2015-12-29 6:25	应用程序扩展
	libwinpthread-1.dll	2015-12-29 6:25	应用程序扩展
	midcode.txt	2019-05-17 15:53	文本文档
	optMidCode.txt	2019-05-16 22:30	文本文档
	Qt5Core.dll	2018-10-31 19:04	应用程序扩展
	Qt5Gui.dll	2017-01-19 4:50	应用程序扩展
	Qt5Widgets.dll	2017-01-19 4:56	应用程序扩展
	symbolTable.txt	2019-05-17 15:53	文本文档

错误数据测试:

错误的类 C 源程序如下:

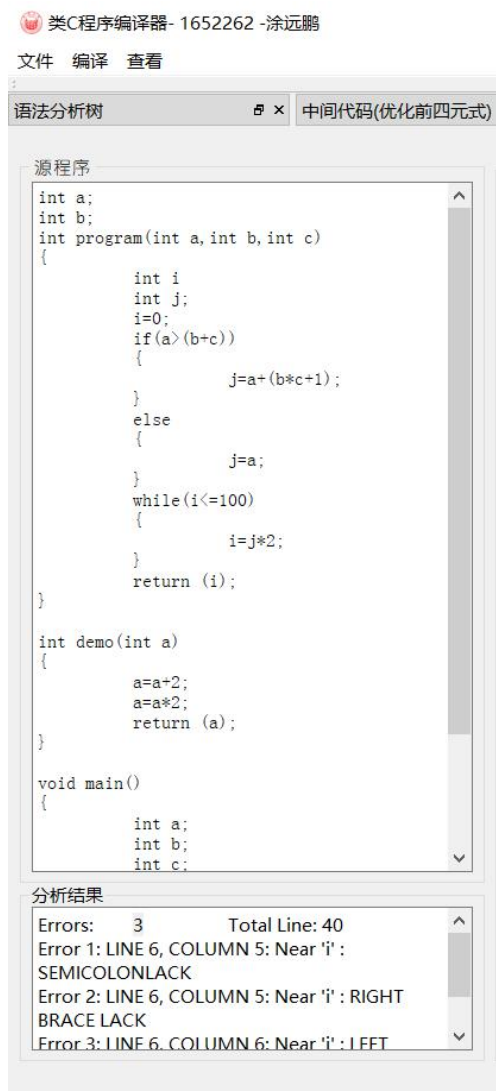
```
int a;
int b;
int program(int a, int b, int c)
{
    int i
    int j;
    i=0;
    if(a>(b+c))
    {
        j=a+(b*c+1);
    }
    else
    {
        j=a;
    }
    while(i<=100)
    {
        i=j*2;
    }
    return (i);
}

int demo(int a)
{
```

```
    a=a+2;
    a=a*2;
    return (a);
}

void main()
{
    int a;
    int b;
    int c;
    a=3;
    b=4;
    c=2;
    a=program(a, b, demo(c));
    return;
}
```

输入错误的类 C 源文件，输出显示错误信息包含错误行数，以及分析所得的错误原因：



Errors: 3 Total Line: 40

Error 1: LINE 6, COLUMN 5: Near 'i' : SEMICOLONLACK//i 的分号确实

Error 2: LINE 6, COLUMN 5: Near 'i' : RIGHT BRACE LACK//右侧大括号缺失

Error 3: LINE 6, COLUMN 6: Near 'j' : LEFT PARENT LACK//j 左侧没有父函数

2. 时间复杂度分析

由于在生成汇编码时涉及到 while 的两重循环，所以程序的时间复杂度较高为 $O(n*n)$ 级别。

3. 遇到的问题及解决

本程序使用了 LR1 的设计思想,并做了较大的改动。我最初的想法是严格按照 LR1 算法的思想,以 program 为 S' , 并根据状态转换图画出 action 表。

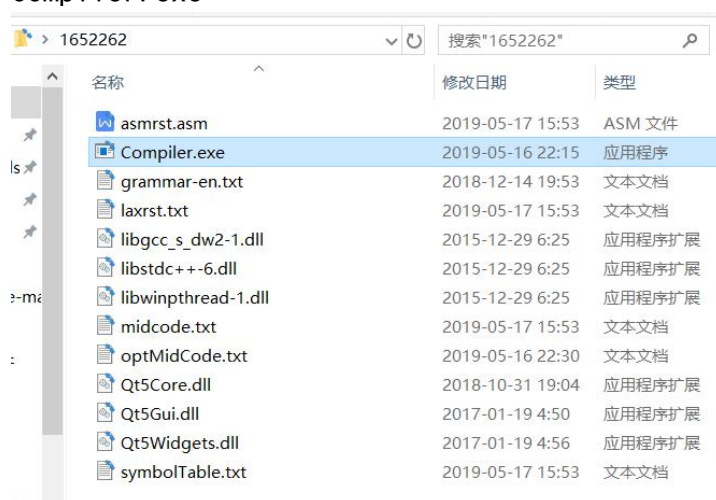
但是在实现的过程中,发现这样的问题很大。首先,类 c 语法有 26 个产生式,要准

确画出全部的状态转换图是十分困难的。其次，这种方式也是十分不必要的。比如，当程序读到了 `int`，那么接下来的步骤我们可想而知是继续读，反映到 LR1 算法中，就是当前状态 `i` 与当前待输入字符，对应的为 `Sj`。而若读到了 `' ; ' ; ' , ' }` 等就一定是指向规约操作，而读到其他的就会继续读入。所以，该程序以 LR1 算法思想为指导，并根据对 C 语言语法的理解。

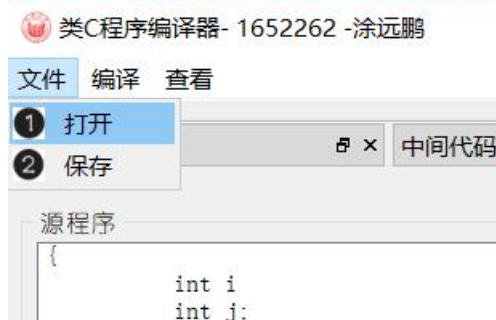
八、用户使用说明

使用步骤如下所示：

1. 首先将压缩包中的 1652262 文件夹复制到桌面，然后点击 1652262 文件夹中的 `Compiler.exe`



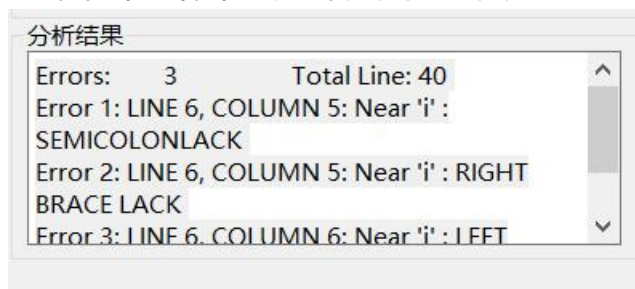
2. 选择运行的 `Compiler.exe` 中导航栏的“文件”选项，选择“打开”，选择要编译的类 C 源程序，选择的类 C 程序会显示在左侧的 `textBrowser` 中：



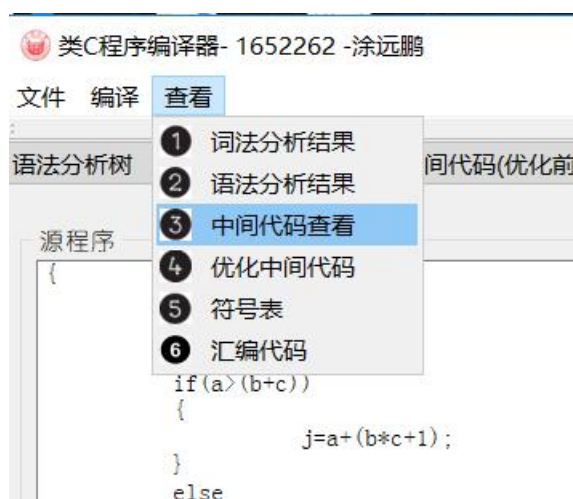
3. 点击导航栏中编译中的“运行”，即可对类 C 程序进行编译：



4. 如果编译成功将在下方的文本框中显示“Compile Success.”，如果失败则会显示编译错误信息，显示在第几行第几列出现了什么错误



5. 点击导航栏中的“查看”，选择其中的词法分析结果、语法分析结果、中间代码查看、优化中间代码、汇编代码、符号表，即可查看词法分析表、语法分析树、四元式表、优化后四元式表、汇编代码、符号表，并且会同时输出到 exe 所在文件夹下的 txt 文件中：



6. 上述六个选项中，其中语法分析结果、中间代码查看、优化中间代码、符号表会显示在四个悬浮窗口中：



7. 由于程序在开发时使用了比较多的全局变量，所以每次编译过后需要重新启动 exe 才能再次进行编译.....

九、问题和解决的方法、心得体会

在课程设计的过程中，我遇到了极大的困难。总共历时一个月，才完成了一个基本的编译器。通过这次课程设计，让我对编译器的工作原理有了较深的认识。同时也发现了自己程序功力的不足。该编译器还是有挺多不是很满意的地方。比如：

1、对测试程序格式要求较高，在编写的过程中，不是所有地方都记得加上空格是' \t' , ' \r' , ' \n' 的处理，导致测试程序格式要求较高。

2、while 和 if (或反之) 连用，即 while{if{...}}，可能会出现地址 L1:L2: 这样的情况。

主要因为基础功能的实现耗费了大量的时间，没有空余来完成原定的特性，可以说一个支持众多语法特性的编译器不能一蹴而就。

在语义分析的过程中，最大的难点是如何实现一遍扫描。MIPS 汇编代码最大的难题是函数调用的翻译，在前面已经说过解决思路了，就不再多言。通过此次编译原理课程设计我非常深入地理解了从形式语言到汇编代码的编译过程，获得了很多知识。