

课 程 设 计 报 告

设计题目：类C语言编译器

班 级：计算机1702班

组长学号：20174627

组长姓名：王惟

指导教师：肖桐

设计时间：2019 年 12 月

设计分工

组长学号及姓名：20174627 王惟

分工：

1. 任务分配需求分析，头文件接口设计
2. 语法分析控制器
3. 语义分析
4. 符号表
5. 四元式生成
6. 四元式优化
7. 目标代码的运行
8. 小组代码整合调试修改

组员 1 学号及姓名：20174442 周雨桓

分工：

1. 协调分工
2. 文法的制定
3. 基础文法的拓展（函数）
4. 文法的分词
5. 文法和符号的数值化、first 集合算法实现、项目集的 get_closure 算法实现、项目集的 goto 算法实现
6. LR（1）分析表的生成算法实现

组员 2 学号及姓名：20174710 秦立国

分工：

1. 扫描器
2. 常数处理机

3. token、以及语义分析前符号表

4. 目标代码生成的全部工作（包含 if、while、函数等）

组员 3 学号及姓名：20174473 马超

分工：

1. 词法分析中注释的跳过

2. 四元式优化基本块划分

3. 词法、语法、语义报错

摘 要

本次课设，我们组采用 LR(1) 分析法，经过完整的五大编译步骤，将类 C 语言翻译为 8086 汇编语言，并在 DOS 环境下运行，并调用 debug-t-g-d 查看程序运行后内存值，验证编译过程是否正确。

文法支持整数类型及判断语句中产生的布尔类型，函数的定义与调用，if、else 分支语句，while 循环语句等功能，并允许各模块之间相互嵌套使用。

本组课设最主要的特点是 LR(1) 分析以及采用机械构表方法，即给定文法后由函数自动生成转移表，既避免了人工推导分析表的复杂性，又保证了文法的可扩充性。

此外，本组课设在每个编译步骤中都有一定广度扩展：语法语义分析中共有 77 条文法产生式、112 个状态语法状态、26 个语义动作；词法语法语义分析报错系统精确到行的具体位置；包含 3 种情况的中间代码优化；目标代码运行验证等等。

关键词：LR(1)，机械构表，函数，优化，目标代码运行验证……

目 录

摘要	4
1 概述	6
2 课程设计任务及要求	7
2.1 设计目的	7
2.2 设计任务	7
2.3 设计要求	7
3 编译系统总体设计	8
3.1 编译器结构设计	8
3.2 文法设计	8
3.3 符号表设计	13
4 编译器前端设计	14
4.1 词法分析器与常数处理机	14
4.2 词法分析注释处理，词法分析、语法语义分析报错	18
4.3 文法的数值化及 LR（1）分析表的生成	22
4.4 语法分析，语义分析控制器，四元式与符号表生成	31
5 编译器后端设计	36
5.1 四元式的优化	36
5.2 目标代码具体实现	36
5.3 在 DOS 下运行汇编代码，并用 debug 验证正确性	38
6 结论	41
7 参考文献	41
8 收获体会和建议	42

1 概述

编译原理课程兼有很强的理论性和实践性，是计算机专业的一门非常重要的专业基础课程，在系统软件中占有十分重要的地位。编译原理课程设计是本课程重要的综合实践教学环节，是对平时实验的一个补充。通过编译器相关子系统的设计，使学生能够更好地掌握编译原理的基本理论和编译程序构造的基本方法和技巧，融会贯通本课程所学专业理论知识；培养学生独立分析问题、解决问题的能力，以及系统软件设计的能力；培养学生创新能力及团队协作精神。编译程序是一种翻译程序，特指把某种高级程序设计语言翻译成具体计算机上低级程序设计语言。

本次我们小组设计并实现了一种基于 LR（1）分析法用以将类 C 语言翻译为 8086 汇编语言的简单编译器，设计的主要方面如下：

1. 文法的设计，包括分支、循环、函数等语句
2. 符号表的设计，保存用户自定义变量、数组及临时变量，能够区分形参、实参并能通过偏移地址找出各变量，为目标代码生成打下基础。
3. 词法分析器，识别变量名、数字、关键字符、字符串并生成 TOKEN 序列
4. LR（1）转移表生成器，机械算法由文法自动生成 LR（1）转移表
5. 语法分析器控制器，识别源代码生成的 TOKEN 串是否符合文法
6. 语义分析器，设计翻译文法，将 token 序列转化为已设计的四元式中间代码，同时填写符号表
7. 中间代码优化，常值表达式、公共子表达式，删除无用赋值。
8. 目标代码生成，生成可执行的 8086 汇编代码。
9. 执行 8086 汇编代码，查看程序结果，验证编译器的正确性。

2 课程设计任务及要求

2.1 设计目的

编译原理课程兼有很强的理论性和实践性，是计算机专业的一门非常重要的专业基础课程，在系统软件中占有十分重要的地位。编译原理课程设计是本课程重要的综合实践教学环节，是对平时实验的一个补充。通过编译器相关子系统的设计，使学生能够更好地掌握编译原理的基本理论和编译程序构造的基本方法和技巧，融会贯通本课程所学专业理论知识；培养学生独立分析问题、解决问题的能力，以及系统软件设计的能力；培养学生的创新能力及团队协作精神。

2.2 设计任务

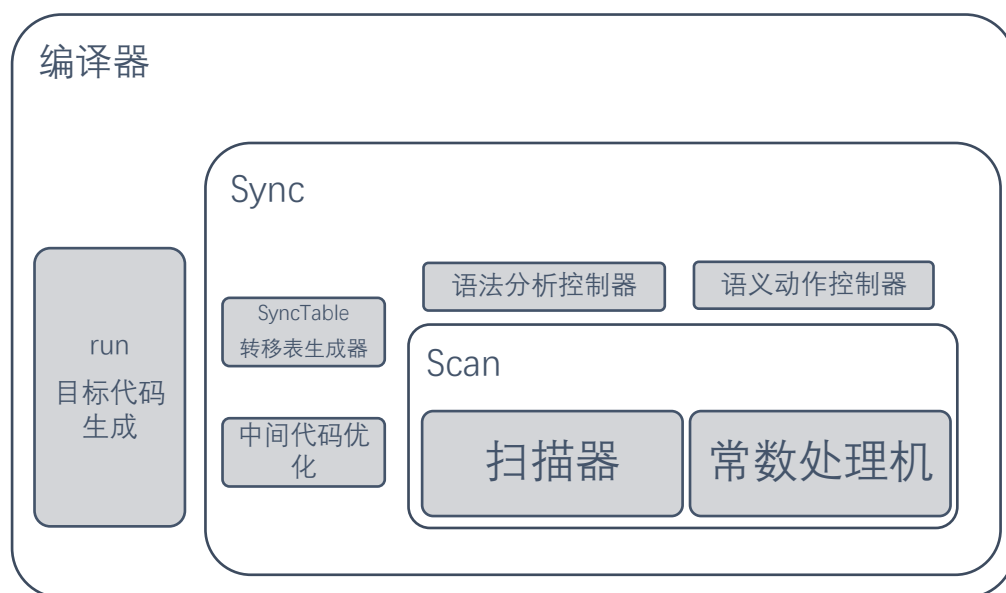
一个简单文法的编译器的设计与实现，包括前端和后端。

2.3 设计要求

- 1、在深入理解编译原理基本原理的基础上，对于选定的题目，以小组为单位，先确定设计方案；
- 2、设计系统的数据结构和程序结构，设计每个模块的处理流程。要求设计合理；
- 3、编程序实现系统，要求实现可视化的运行界面，界面应清楚地反映出系统的运行结果；
- 4、确定测试方案，选择测试用例，对系统进行测试；
- 5、运行系统并要通过验收，讲解运行结果，说明系统的特色和创新之处，并回答指导教师的提问；
- 6、提交课程设计报告。

3 编译系统总体设计

3.1 编译器结构设计



编译器设计结构图

3.2 文法设计

包含 if、else、while、函数等的文法设计如下：

```
1 S0 -> translation_unit
2 translation_unit -> external_declaration
3 external_declaration -> function_definition | function_definition
external_declaration
4 declaration -> type_specifier id ;
5 type_specifier -> void | int | double | string | char
6 function_definition -> type_specifier function_name parameter
compound_statement
7 function_name -> id
8 parameter -> ( ) | ( function_declaration_list )
9 compound_statement -> { statement_list }
10 decalration_list -> declaration | decalration_list declaration
```



```

11 statement_list -> statement | statement_list statement
12  statement      ->  compound_statement      |      expression_statement      |
selection_statement | iteration_statement | jump_statement | decalration_list
13 expression_statement -> ; | expression ;
14 expression -> assignment_expression
15 assignment_expression -> primary_expression = assignment_expression |
logical_or_expression
16      logical_or_expression      ->      logical_or_expression      ||
logical_and_expression | logical_and_expression
17 logical_and_expression -> logical_and_expression && equality_expression
| equality_expression
18 equality_expression -> equality_expression == relational_expression |
relational_expression
19 relational_expression -> relational_expression < additive_expression |
relational_expression > additive_expression | relational_expression <=
additive_expression | relational_expression >= additive_expression |
additive_expression
20 additive_expression -> additive_expression - multiplicative_expression
| additive_expression + multiplicative_expression | multiplicative_expression
21      multiplicative_expression      ->      multiplicative_expression      *
last_expression      |      multiplicative_expression      /      last_expression      |
multiplicative_expression % last_expression | last_expression
22 primary_expression -> id | double_num | int_num | ( expression )
23 selection_statement -> if ( expression ) statement | if ( expression )
statement selection_continue_statement
24 iteration_statement -> while ( expression ) statement
25 jump_statement -> continue ; | break ; | return ; | return expression ;
26      function_declaration_list      ->      function_declaration      |
function_declaration_list , function_declaration

```

```

27 function_declaration -> type_specifier id
28 selection_continue_statement -> else end_selection_statement | else
selection_statement
29 end_selection_statement -> compound_statement | expression_statement |
iteration_statement | jump_statement
30 s_declaration_list -> s_declaration | s_declaration_list , s_declaration
31 s_declaration -> id
32 last_expression -> function_name ( s_declaration_list ) | id | double_num
| int_num | ( expression ) | function_name ( )

```

整理后的单产生式文法如下共 77 条：

```

1 S0 -> translation_unit
2 translation_unit -> external_declaration
3 external_declaration -> function_definition
4 external_declaration -> function_definition external_declaration
5 declaration -> type_specifier id ;
6 type_specifier -> void
7 type_specifier -> int
8 type_specifier -> double
9 type_specifier -> string
10 type_specifier -> char
11 function_definition -> type_specifier function_name parameter
compound_statement
12 function_name -> id
13 parameter -> ( )
14 parameter -> ( function_declaration_list )
15 compound_statement -> { statement_list }

```

```

16 decalration_list -> declaration
17 decalration_list -> decalration_list declaration
18 statement_list -> statement
19 statement_list -> statement_list statement
20 statement -> compound_statement
21 statement -> expression_statement
22 statement -> selection_statement
23 statement -> iteration_statement
24 statement -> jump_statement
25 statement -> decalration_list
26 expression_statement -> ;
27 expression_statement -> expression ;
28 expression -> assignment_expression
29 assignment_expression -> primary_expression = assignment_expression
30 assignment_expression -> logical_or_expression
31     logical_or_expression     ->     logical_or_expression     ||
logical_and_expression
32 logical_or_expression -> logical_and_expression
33 logical_and_expression -> logical_and_expression && equality_expression
34 logical_and_expression -> equality_expression
35 equality_expression -> equality_expression == relational_expression
36 equality_expression -> relational_expression
37 relational_expression -> relational_expression < additive_expression
38 relational_expression -> relational_expression > additive_expression
39 relational_expression -> relational_expression <= additive_expression
40 relational_expression -> relational_expression >= additive_expression
41 relational_expression -> additive_expression
42 additive_expression -> additive_expression - multiplicative_expression
43 additive_expression -> additive_expression + multiplicative_expression

```

```

44 additive_expression -> multiplicative_expression
45     multiplicative_expression    ->    multiplicative_expression    *
last_expression
46     multiplicative_expression    ->    multiplicative_expression    /
last_expression
47     multiplicative_expression    ->    multiplicative_expression    %
last_expression
48 multiplicative_expression -> last_expression
49 primary_expression -> id
50 primary_expression -> double_num
51 primary_expression -> int_num
52 primary_expression -> ( expression )
53 selection_statement -> if ( expression ) statement
54     selection_statement    ->    if    (    expression    )    statement
selection_continue_statement
55 iteration_statement -> while ( expression ) statement
56 jump_statement -> continue ;
57 jump_statement -> break ;
58 jump_statement -> return ;
59 jump_statement -> return expression ;
60 function_declaration_list -> function_declaration
61     function_declaration_list    ->    function_declaration_list    ,
function_declaration
62 function_declaration -> type_specifier id
63 selection_continue_statement -> else end_selection_statement
64 selection_continue_statement -> else selection_statement
65 end_selection_statement -> compound_statement
66 end_selection_statement -> expression_statement
67 end_selection_statement -> iteration_statement

```

```

68 end_selection_statement -> jump_statement
69 s_declaration_list -> s_declaration
70 s_declaration_list -> s_declaration_list , s_declaration
71 s_declaration -> id
72 last_expression -> function_name ( s_declaration_list )
73 last_expression -> id
74 last_expression -> double_num
75 last_expression -> int_num
76 last_expression -> ( expression )
77 last_expression -> function_name ( )

```

3.3 符号表设计

Parse

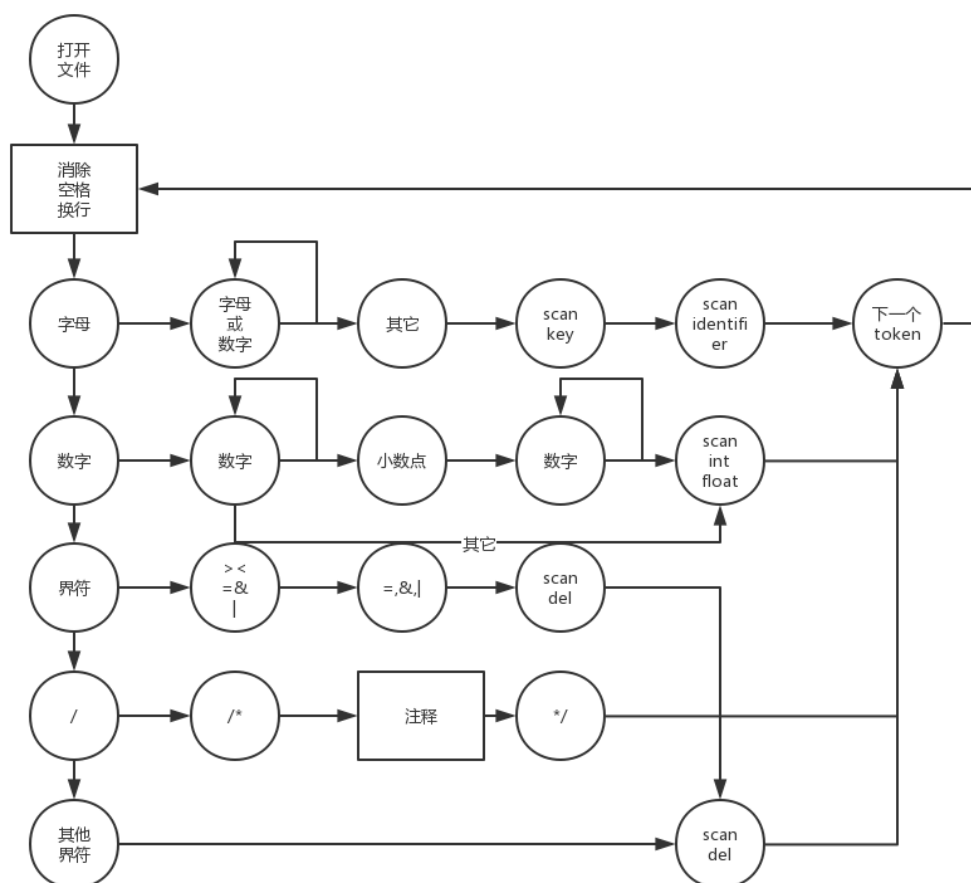
Synbl	<ul style="list-style-type: none"> • name , cat • type , addr/point to Pfinfl
Pfinfl	<ul style="list-style-type: none"> • name , level • para~ start addr , para~ num
Consl	<ul style="list-style-type: none"> • name • int/bool value
VALL	<ul style="list-style-type: none"> • name • addr start ,addr end

4 编译器前端设计

4.1 词法分析器与常数处理机

该部分完成人：秦立国

4.1.1 词法分析流程图



4.1.2 程序说明

Scan.show(): 输出全部 token 序列

run.ten_to_sixteen(): 将四元式中的十进制数转化为四位十六进制

run.bool_DB_or_DW(): 将四元式中逻辑表达式的字节型数据储存

Scan.next(): 返回下一个 token 的全部信息

4.1.3 功能

将存放在 hello.txt 文件中的 c 语言源程序识别为 token 序列。

4.1.4 数据结构:

Scan.h 中有各 token 序列 vector 以及用于文件打开, 扫描与行位置统计有关变量。

```
class Scan {
public:
    vector<string>scan_key;           //关键词
    vector<string>scan_del;           //界符
    vector<string>scan_identifier;    //标识符
    vector<string>scan_int;           //整型
    vector<string>scan_float;         //实型
    vector<string>scan_char;          //字符型
    vector<string>scan_str;           //字符串型
    vector<string>token_type;         //token 序列类型
    vector<int>token_id;              //token 位置
    string file;                      //文件
    ifstream infile;                  //文件
    string s;                         //存放 name
    int head;                         //代表 line_pos
    char c;                           //存放字符
    int line;                         //代表 line_number
    int order;                        //代表 pos
    void init_keyAndDel();            //初始化关键词与界符
    Scan();                           //
};
```

4.1.5 算法

按字符扫描, 每次读取文件中的一个字符并将此字符添加到字符串尾部直到符合情况, 分为以下情况:

1. \t \n 以及空格: 遇\n 行数+1, 位置数重置为 0, 遇\t 与空格位置数+1;

2. 大小写字母：遇字母或数字继续读取，扫描结束后判断关键词或标识符。
3. 数字：遇数字继续读取，扫描中途可判断整型还是实型。
4. 界符/：三种情况，//型注释，/**/型注释以及单个/。
5. 其他界符：遇>, <, =, &, |等还需确认下一个字符。

4.1.6 实验结果

源程序：

```

1  int gcd(int a, int b) {
2      //c=a+b;测试注释
3      int c;
4      c=a%b;
5      if(b==0) return a;
6      else return gcd(b, c);
7  }
8  int main() {
9      int e;
10     int f;
11     e=28;
12     f=24;
13     /*
14     测试注释
15     */
16     while(e==f)
17     {
18         e=e+f;
19         break;
20     }
21     int a;
22     int b;
23     a=1; b=10;
24     while(a<b)
25     {
26         a=a+1;
27         if(a==5)
28         {
29             continue;
30         }
31     }
32     gcd(e, f);
33     int k;
34     int r;
35     k=28;

```

生成 token:

关键词 token (提前构造)

scan_key:				
int	if	while	else	break
continue	return			

标识符 token

```
scan_identify:
gcd          a          b          c          main
e           f          k          r          h
```

整数 token:

```
scan_int:
0          28          24          1          10
5
```

符号 token（提前构造）

```
scan_del:
(          =          >          <          >=
==         <=         {          }          )
:          ;          +          -          *
/          ,          !=         &&         ||
%
```

词法分析全部属性

pos	name	line_number	line_pos	identifier_id	token_id	type	length
0	int	1	0	0	1	关键词	3
1	gcd	1	4	0	2	标识符	3
2	(1	7	0	3	界符	1
3	int	1	8	0	1	关键词	3
4	a	1	12	1	2	标识符	1
5	,	1	13	16	3	界符	1
6	int	1	14	0	1	关键词	3
7	b	1	18	2	2	标识符	1
8)	1	19	9	3	界符	1
9	{	1	20	7	3	界符	1
10	int	3	0	0	1	关键词	3
11	c	3	5	3	2	标识符	1
12	;	3	6	11	3	界符	1
13	c	4	1	3	2	标识符	1

121	int	34	1	0	1	关键词	3
122	r	34	5	8	2	标识符	1
123	;	34	6	11	3	界符	1
124	k	35	1	7	2	标识符	1
125	=	35	2	1	3	界符	1
126	28	35	3	1	4	整型	2
127	;	35	5	11	3	界符	1
128	r	36	1	8	2	标识符	1
129	=	36	2	1	3	界符	1
130	24	36	3	2	4	整型	2
131	;	36	5	11	3	界符	1
132	int	37	1	0	1	关键词	3
133	h	37	5	9	2	标识符	1
134	;	37	6	11	3	界符	1
135	}	38	0	8	3	界符	1

4.2 词法分析注释处理，词法分析、语法语义分析报错

该部分完成人：马超

4.2.1 词法分析中注释的处理

功能：词法分析中会遇到注释，需要跳过注释，扫描注释后的内容

算法：词法分析加入判断，当读入的字符为’ /’ 时，进行注释的判断，读入下一个字符，如果为’ /’ ，直接跳到下一行，同时将记录行数 的变量 line 加一，记录位置的变量置零，若字符为’ *’ ，将一直读入知道该字符为*且下一个为/。如果既非’ /’ 也不是’ *’ ，则为除号，返回为 Token。

运行结果

```

hello.txt  x SyncTable.h  run.h  LibFunc.h  Parse.h  Scan.h  syn2.txt  Syntax.cpp
1  int gcd(int a, int b) {
2      if(b==0) return a; //zhushi1
3      int c; /*zhushi2
4      */
5      c=a*b;
6      return gcd(b, c);
7  }
8  int main() {
9      int a;
10     int b;
11     a=70;
12     b=25;
13     if(a==b)
14     {
15         a=a+b;
16     }
17 }

```

```

E:\x64\Debug\Project50.exe
expression      -> assignment_expression
expression_statement -> expression ;
statement        -> expression_statement
statement_list   -> statement
compound_statement -> { statement_list }
statement        -> compound_statement
selection_statement -> if ( expression ) statement
statement        -> selection_statement
statement_list   -> statement_list statement
compound_statement -> { statement_list }
function_definition -> type_specifier function_name parameter compound_statement
external_declaration -> function_definition
external_declaration -> function_definition external_declaration
translation_unit -> external_declaration
LR(1)分析成功

token
(key, 0) (del, 0) (key, 0) (identify, 1) (del, 16)
(key, 0) (identify, 2) (del, 9) (del, 7) (key, 1)
(del, 0) (identify, 2) (del, 5) (del, 9) (key, 11)
(identify, 1) (del, 11) (key, 0) (identify, 3) (del, 11)
(identify, 3) (del, 1) (identify, 1) (del, 20) (identify, 2)
(del, 11) (key, 11) (identify, 0) (del, 0) (identify, 2)
(del, 16) (identify, 3) (del, 9) (del, 11) (del, 8)
(key, 0) (identify, 4) (del, 0) (del, 9) (del, 7)
(key, 0) (identify, 1) (del, 11) (key, 0) (identify, 2)
(del, 11) (identify, 1) (del, 1) (int, 1) (del, 11)
(identify, 2) (del, 1) (int, 2) (del, 11) (key, 1)
(del, 0) (identify, 1) (del, 5) (identify, 2) (del, 9)

```

词法分析报错

功能：在词法分析阶段扫描到非法字符时进行报错

算法：通过 Token 类中的成员变量 line_pos 和 line_number 记录扫描时的行数和行里的位置，扫描到非法字符时，将错误信息输出并停止扫描。

示例

```

hello.txt  Token.cpp  Syntax.cpp  Scan.cpp  run.cpp  汇编头文件.txt  SyncTable.cpp  Pa
1  int gcd(int a, int b) {
2  if (b==0) return a; //zhushi
3  int c;
4  c=a%b;
5  return gcd(b, c);
6  }
7  int main() {
8  int a;
9  int b;
10 a=70;
11 b=25;
12 if (a==b)
13 {
14     a=a+b;
15 }
16 }

```

```

E:\x64\Debug\Project50.exe
equality_expression == relational_expression
logical_and_expression -> equality_expression
logical_or_expression -> logical_and_expression
assignment_expression -> logical_or_expression
expression -> assignment_expression
last_expression -> id
multiplicative_expression -> last_expression
additive_expression -> multiplicative_expression
relational_expression -> additive_expression
equality_expression -> relational_expression
logical_and_expression -> equality_expression
logical_or_expression -> logical_and_expression
assignment_expression -> logical_or_expression
expression -> assignment_expression
error!!! 词法分析报错 line:2 pos:18

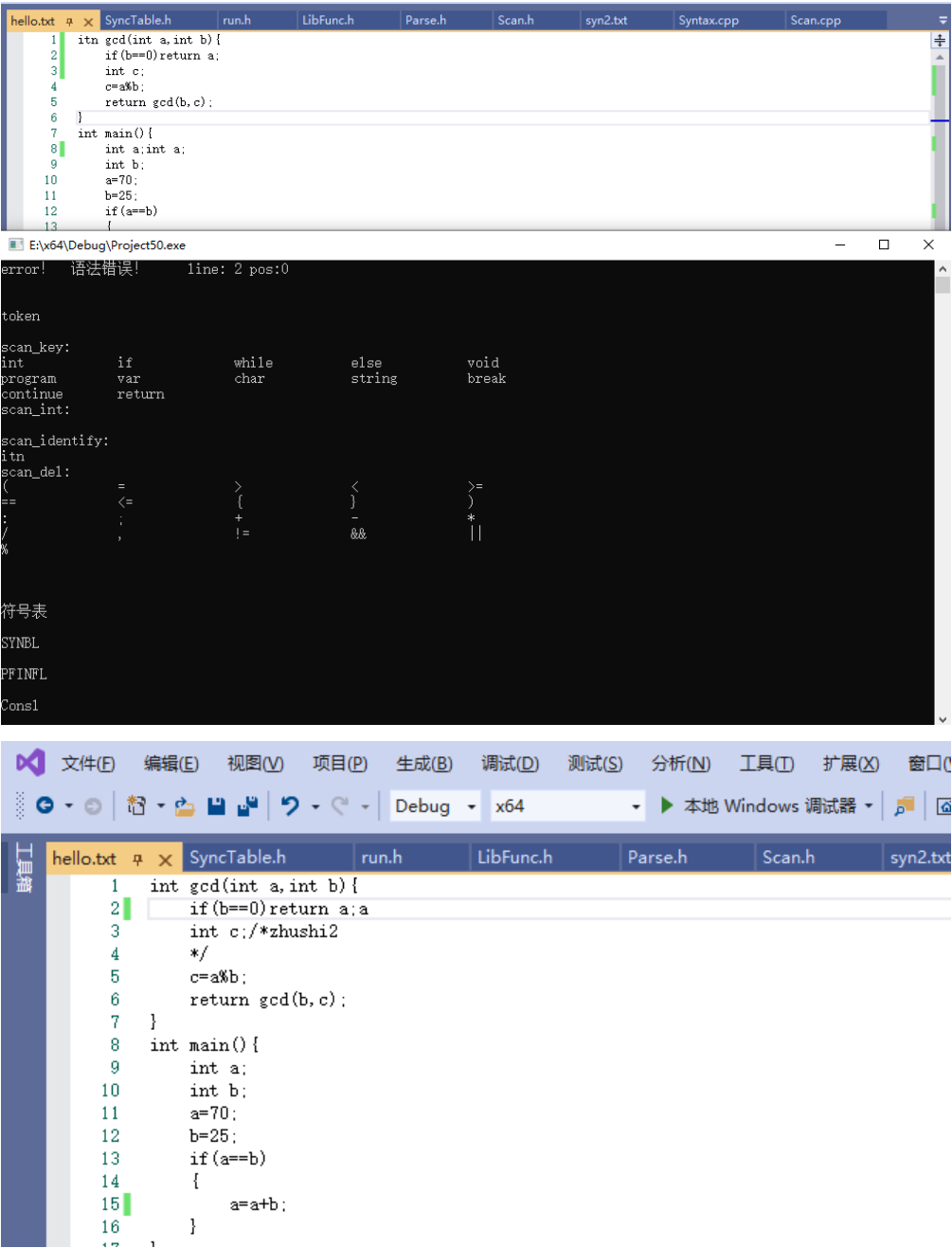
token
(key, 0) (del, 0) (key, 0) (identify, 1) (del, 16)
(key, 0) (identify, 2) (del, 9) (del, 7) (key, 1)
(del, 0) (identify, 2) (del, 5) (del, 9) (key, 11)
(identify, 1) (del, 11)

```

语法错误报错

功能：出现语法错误时进行报错

示例



```

last_expression      -> id
multiplicative_expression -> last_expression
additive_expression  -> multiplicative_expression
relational_expression -> additive_expression
equality_expression   -> relational_expression
last_expression      -> int_num
multiplicative_expression -> last_expression
additive_expression  -> multiplicative_expression
relational_expression -> additive_expression
equality_expression   -> equality_expression == relational_expression
logical_and_expression -> equality_expression
logical_or_expression -> logical_and_expression
assignment_expression -> logical_or_expression
expression            -> assignment_expression
last_expression      -> id
multiplicative_expression -> last_expression
additive_expression  -> multiplicative_expression
relational_expression -> additive_expression
equality_expression   -> relational_expression
logical_and_expression -> equality_expression
logical_or_expression -> logical_and_expression
assignment_expression -> logical_or_expression
expression            -> assignment_expression
jump_statement       -> return expression ;
statement             -> jump_statement
selection_statement   -> if ( expression ) statement
statement             -> selection_statement
error!  语法错误!      line: 3 pos:19

```

语义分析报错

功能

能处理两种语义错误:

- 一、使用未定义的标识符
- 二、标识符重复定义

算法

语法分析到 `aa = $ + b;` 的时候, 顺便检查 `aa` 和 `b` 是否有定义。发现 `b` 没有定义, 不符合使用规则, 报错。

语法分析到声明标识符时, 查符号表, 若标识符已经存在, 报错。

示例

hello.txt	SyncTable.h	run.h	LibFunc.h	Parse.h	Scan.h
1	int gcd(int a,int b){				
2	if(b==0)return a;				
3	int c;/*zhushi2				
4	*/				
5	c=a*b;				
6	return gcd(b,c);				
7	}				
8	int main(){				
9	int a;				
10	int b;				
11	a=70;				
12	b=25;				
13	if(a==b)				
14	{				
15	a=a+b+d;				
16	}				
17	}				

```

E:\x64\Debug\Project50.exe
equality_expression -> equality_expression == relational_expression
logical_and_expression -> equality_expression
logical_or_expression -> logical_and_expression
assignment_expression -> logical_or_expression
expression -> assignment_expression
primary_expression -> id
last_expression -> id
multiplicative_expression -> last_expression
additive_expression -> multiplicative_expression
last_expression -> id
multiplicative_expression -> last_expression
additive_expression -> additive_expression + multiplicative_expression
last_expression -> id
multiplicative_expression -> last_expression
additive_expression -> additive_expression + multiplicative_expression
error! 语义分析错误 未定义的标识符 d line:15 pos:9

token
(key,0) (del,0) (key,0) (identify,1) (del,16)
(key,0) (identify,2) (del,9) (del,7) (key,1)
(del,0) (identify,2) (del,5) (del,9) (key,11)
(identify,1) (del,11) (key,0) (identify,3) (del,11)
(identify,3) (del,1) (identify,1) (del,20) (identify,2)
(del,11) (key,11) (identify,0) (del,0) (identify,2)
(del,16) (identify,3) (del,9) (del,11) (del,8)
(key,0) (identify,4) (del,0) (del,9) (del,7)
(key,0) (identify,1) (del,11) (key,0) (identify,2)
(del,11) (identify,1) (del,1) (int,1) (del,11)
(identify,2) (del,1) (int,2) (del,11) (key,1)

```

4.3 文法的数值化及 LR（1）分析表的生成

该部分负责人：周雨桓

4.3.1 功能

文法是为了深入研究语言的内在性质，而构造语言的方法。换句话说，给定一个文法，就能从结构上唯一的确定语言（形式语言理论可以证明此结论为真）。

一个文法必须由 4 部分组成：

字母表，表中的字符成为终结符。因为通过文法规则，最终得到的句子只能含有这些字符，这种字母称为终结符集合，记为 `termanal`。

一个中间字母集，称为非终结符，记为 `nontermanal`，一般出现在规则左部的符号都是非终结符。

文法规则集合。规则形如 `type_specifier -> void`。读作“导出”、“产生”、“生成”或者“定义为”。

文法的开始符号 `S0`。`S0` 为特殊的非终结符。

下面是总结出来的 c 语言的文法，总共有 57 条规则：

- 1 `S0 -> translation_unit`
- 2 `translation_unit -> external_declaration`
- 3 `external_declaration -> function_definition`
- 4 `external_declaration -> function_definition external_declaration`
- 5 `declaration -> type_specifier id ;`

```

6 type_specifier -> void
7 type_specifier -> int
8 type_specifier -> double
9 type_specifier -> string
10 type_specifier -> char
11 function_definition -> type_specifier function_name parameter
compound_statement
12 function_name -> id
13 parameter -> ( )
14 parameter -> ( function_declaration_list )
15 compound_statement -> { statement_list }
16 declaration_list -> declaration
17 declaration_list -> declaration_list declaration
18 statement_list -> statement
19 statement_list -> statement_list statement
20 statement -> compound_statement
21 statement -> expression_statement
22 statement -> selection_statement
23 statement -> iteration_statement
24 statement -> jump_statement
25 statement -> declaration_list
26 expression_statement -> ;
27 expression_statement -> expression ;
28 expression -> assignment_expression
29 assignment_expression -> primary_expression = assignment_expression
30 assignment_expression -> logical_or_expression
31 logical_or_expression -> logical_or_expression ||
logical_and_expression
32 logical_or_expression -> logical_and_expression

```

```

33 logical_and_expression -> logical_and_expression && equality_expression
34 logical_and_expression -> equality_expression
35 equality_expression -> equality_expression == relational_expression
36 equality_expression -> relational_expression
37 relational_expression -> relational_expression < additive_expression
38 relational_expression -> relational_expression > additive_expression
39 relational_expression -> relational_expression <= additive_expression
40 relational_expression -> relational_expression >= additive_expression
41 relational_expression -> additive_expression
42 additive_expression -> additive_expression - multiplicative_expression
43 additive_expression -> additive_expression + multiplicative_expression
44 additive_expression -> multiplicative_expression
45     multiplicative_expression     ->     multiplicative_expression     *
last_expression
46     multiplicative_expression     ->     multiplicative_expression     /
last_expression
47     multiplicative_expression     ->     multiplicative_expression     %
last_expression
48 multiplicative_expression -> last_expression
49 primary_expression -> id
50 primary_expression -> double_num
51 primary_expression -> int_num
52 primary_expression -> ( expression )
53 selection_statement -> if ( expression ) statement
54     selection_statement     ->     if     (     expression     )     statement
selection_continue_statement
55 iteration_statement -> while ( expression ) statement
56 jump_statement -> continue ;
57 jump_statement -> break ;

```



```

58 jump_statement -> return ;
59 jump_statement -> return expression ;
60 function_declaration_list -> function_declaration
61     function_declaration_list    ->     function_declaration_list    ,
function_declaration
62 function_declaration -> type_specifier id
63 selection_continue_statement -> else end_selection_statement
64 selection_continue_statement -> else selection_statement
65 end_selection_statement -> compound_statement
66 end_selection_statement -> expression_statement
67 end_selection_statement -> iteration_statement
68 end_selection_statement -> jump_statement
69 s_declaration_list -> s_declaration
70 s_declaration_list -> s_declaration_list , s_declaration
71 s_declaration -> id
72 last_expression -> function_name ( s_declaration_list )
73 last_expression -> id
74 last_expression -> double_num
75 last_expression -> int_num
76 last_expression -> ( expression )
77 last_expression -> function_name ( )

```

终结符		非终结符	
符号	编号	符号	编号
"#"	1	"S0"	35
"%"	2	"translation_unit"	36
"&&"	3	"external_declaration"	37
"("	4	"declaration"	38
")"	5	"type_specifier"	39
"*"	6	"function_definition"	40
"+"	7	"function_name"	41
"_"	8	"parameter"	42
"/"	9	"compound_statement"	43

";"	10	"decalration_list"	44
"<"	11	"statement_list"	45
"<="	12	"statement"	46
"="	13	"expression_statement"	47
"=="	14	"expression"	48
">"	15	"assignment_expression"	49
">="	16	"logical_or_expression"	50
"break"	17	"logical_and_expression"	51
"continue"	18	"equality_expression"	52
"double"	19	"relational_expression"	53
"double_num"	20	"additive_expression"	54
"id"	21	"multiplicative_expression"	55
"if"	22	"primary_expression "	56
"int"	23	"selection_statement"	57
"int_num"	24	"iteration_statement"	58
"return"	25	"jump_statement"	59
"void"	26	"function_declaration_list"	60
"while"	27	"function_declaration"	61
"{"	28	"selection_continue_statement ,"	62
" "	29	"end_selection_statement"	63
"}"	30	"s_declaration_list"	64
","	31	"s_declaration"	65
"string"	32	"last_expression"	66
"char"	33		
"else"	34		

我们可以从表中阅读出本实验 C 的语法规则：C 程序只是一个语句序列，它共有 5 种语句：if 语句、while 语句、赋值语句。

另外还可以看出：C 的表达式有两类：布尔表达式和算术表达式。布尔表达式使用比较运算符 “=” 和 “<”，通常用在 if 语句和 while 语句中作为测试条件；算术表达式使用整型运算符 “+”、“-”、“*”、“/”，它们具有左结合和常规的优先关系。与此不同，比较运算是非结合的（每个没有括号的表达式只允许一种比较运算）。比较运算符的优先权都低于算术运算符。

另外，我们也可以把 C 的表达式看作三类：算符表达式、常量表达式和标识符表达式。

C 中的标识符指的是简单整型变量，它没有类似数组或记录等类型的变量。C 中也无需

显式的变量声明：任何一个变量只是通过出现在赋值语句左边或者 read 关键字的右边来隐式地声明。另外，变量只有全局作用域。

C 的语句序列是指用 N 个分号分隔开来的 N 条语句。

4.3.2 特点

1. 77 条文法产生式，100 多个状态，文法规则丰富、鲁棒性较强，
2. 数值化算法直接从手动输入文法到 LR (1) 分析表的全自动生成，可以快捷拓展文法语句，如文法不符合 LR (1) 规则会自动报错，
3. 能实现几乎所有文法的自动分析。如左递归
4. 相比递归下降 LL (1) LR (0)，LR (1) 分析法对文法的限制更少，效率更高。

4.3.3 功能

```
static map<string, int> symbol;//终结符、非终结符的字符串到整型映射

string    s[]    =    {    "    #", "%", "&&", "!=" , "(" , ")" , "*" , "+" , "-" , "/" , ";" , "<" , "<=" , "=" , "==" , ">" , ">=" , "break" , "continue" , "double" , "double_num" , "id" , "if" , "int" , "int_num" , "return" , "void" , "while" , "{", "||", "}" , ",", "string" , "char" , "else"};//终结符集

class item_node//项目点
{
public:
    int        cfg_no;                //产生式编号
    int        dot_pos;                //加点位置
    set<int>    possible_prefix;        //可能出现输入符号的集合
    item_node(int a, int b, set<int> c);
    item_node();
};

class sentence { //文法产生式的句子
public:
    string left;
    int left_num;//左部的总序号
```

```

vector<string> right;
vector<int> right_num;//右部的总序号
vector<vector<int>> right_int;//右部根据 symbol 表的分词序号
void divide();
sentence();
};
class SyncTable {
public:
    vector<vector<int>> cfg_list;//存放产生式的左右部序号
    int action_goto[500][offset + ooffset]);//LR(1)分析表，纵坐标是状态（从0
开始），横坐标是字符编号，正值是跳转到的状态，负值是规约的产生式序号（非|版，从1
开始）
    vector<sentence> syntax;//文法（句子集合）
    int sen_num = 1;//全局句子序号
    string guocheng;//存储过程产生式
    int l_num;
    void make_map();//终结符、非终结符 map 制作
    int is_terminal(string a);
    int is_nonterminal(string a);
    void show_set(set<int> &a);
    void load_syntex();//从文件中读取文法并数值化存储
    void show_syntex();
    void find_first(set<int> & first_list, int left_num, int right_num, int
dot);//first 集合算法的子算法
    set<int> first(int left_num, int right_num, int dot);//first 集合算法
    void show_symbol();
    vector<vector<item_node>> closure_item;//项目集簇
    vector<item_node>& get_closure(vector<item_node> &closure);//项目集的闭
包算法

```

```

vector<item_node> get_goto(const int i, const int x); //项目集的 goto 算法
void show_item(vector<item_node> &tem);
void show_item2(item_node &tem);
int item_equal(item_node &a, item_node &b); //判断项目是否相等
int items_equal(vector<item_node> &a, vector<item_node> b); 判断项目集是
否相等

int is_exisit(int n, vector<item_node> tem);
int make_items(); //LR (1) 分析表生成算法
void show_itemarray();
void init(); //初始化
SyncTable();

int state;

string w;

int num;

void R(int b, string a);

};

```

4.3.4 功能

1. 文法右部字符串的分词：文法右部存储按空格分开，每次查找第一次出现空格的位置 n ，读取字符串 0 到 n 的字串，然后删除 0 到 $n+1$ 的字串，循环直到字符串中没有空格

2. 终结符、非终结符的 string 到 int 映射：先依次读取字符串数组中每个字符串，插入 map，然后对每个文法 i 结点遍历，取每个产生式的左部 string，按 i +终结符数量插入到 map 中。

3. 文法的读取：按行读取文件，文法存储按空格，“-》”，“|”，查找第二个空格的位置，从而提取出序号，查找“-》“的位置，提取文法左部，查找”|“位置，提取文法的每一个产生式右部。

4. first 集合算法：传入目标产生式的左部序号和右部序号还有打点位置，若点后字符为终结符，往集合中插入该终结符，若打点后为非终结符且该产生式不是左递归产生式，则递归调用 first 函数

5. get_closure 算法：传入一个文法节点数组，取出每个文法节点的信息，若加点位置下个符号为非终结符且该符号没有存在集合中，创建一个打点位置为 0 的以该符号为

左部的产生式节点,用 first 算法求出该产生式可能碰到的符号集合并压入文法节点数组,集合中压入该符号。若该符号为非终结符且已经存在集合中,则求出该产生式的 first 集合,并且用该集合与文法节点数组中所有有相同产生式右部的可能符号集合求并集。

6. goto 算法:传入状态 i 和读取到的字符序号 x,遍历第 i 个项目集中的每个项目,如果它的下一个字符为 x,且 x 不为 1,则新建一个项目集,并把当前项目的打点位置加一的项目压入项目集中。再对该项目集用 get_closure 算法

7. LR (1) 分析表生成算法:把状态数组全部初始化为-1000,新建一个初始化项目集,新建一个初始化项目 S0,压入项目集中,求出该项目集的闭包压入项目集簇中,对项目集簇中每个项目集求:对项目集中每个项目求当前项目的下一个字符 x 如果该 x 不在已有集合中且不为 1,传入 goto (i, x) 求得新的项目集压入项目集簇,并把新项目集的序号填入状态数组【i】【x】中。如果该符号为 1,则遍历该项目的所有可能符号集合 iter,若状态数组【i】【iter】=-1000,填入-cfg_no,若!=-1000 则出现规约归约冲突,进行报错。如果该产生式为 1 且符号为 1,则状态数组【i】【0】填入 1000,表示接受该语言。

4.3.5 实验结果截图

#	%	&&	!=	()	*	+	-	/	;	<	<=	=	==			
0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
1	OK	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
2	-2	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
3	-3	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
4	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
5	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
6	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
7	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
8	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
9	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
10	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
11	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
12	-4	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
13	-5	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
14	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
15	x	x	x	x	18	x	x	x	x	x	x	x	x	x	x	x	x
16	x	x	x	x	-13	x	x	x	x	19	x	x	x	x	x	x	x
17	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

转移表

```

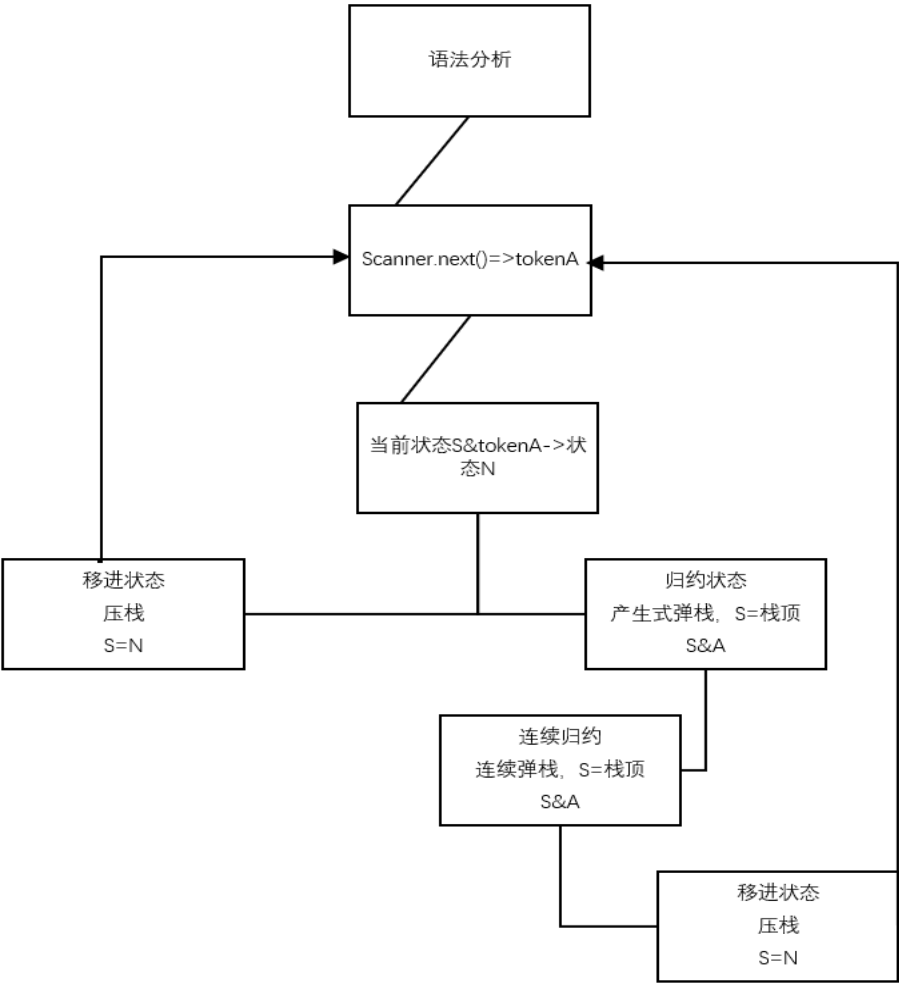
syn.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
1 S0 -> translation_unit
2 translation_unit -> external_declaration
3 external_declaration -> function_definition | function_definition external_declaration | deca
4 declaration -> type_specifier id ;
5 type_specifier -> void | int | double | string | char
6 function_definition -> type_specifier function_name parameter compound_statement
7 function_name -> id
8 parameter -> ( ) | ( function_declaration_list )
9 compound_statement -> { statement_list } | { }
10 decalration_list -> declaration | decalration_list declaration
11 statement_list -> statement | statement_list statement
12 statement -> compound_statement | expression_statement | selection_statement | iteration_sta
13 expression_statement -> ; | expression ;
14 expression -> assignment_expression
15 assignment_expression -> primary_expression = assignment_expression | logical_or_expression
16 logical_or_expression -> logical_or_expression || logical_and_expression | logical_and_expre
17 logical_and_expression -> logical_and_expression && equality_expression | equality_expressio
18 equality_expression -> equality_expression == relational_expression | equality_expression !=
19 relational_expression -> relational_expression < additive_expression | relational_expression >

```

4.4 语法分析、语义分析控制器，四元式与符号表生成

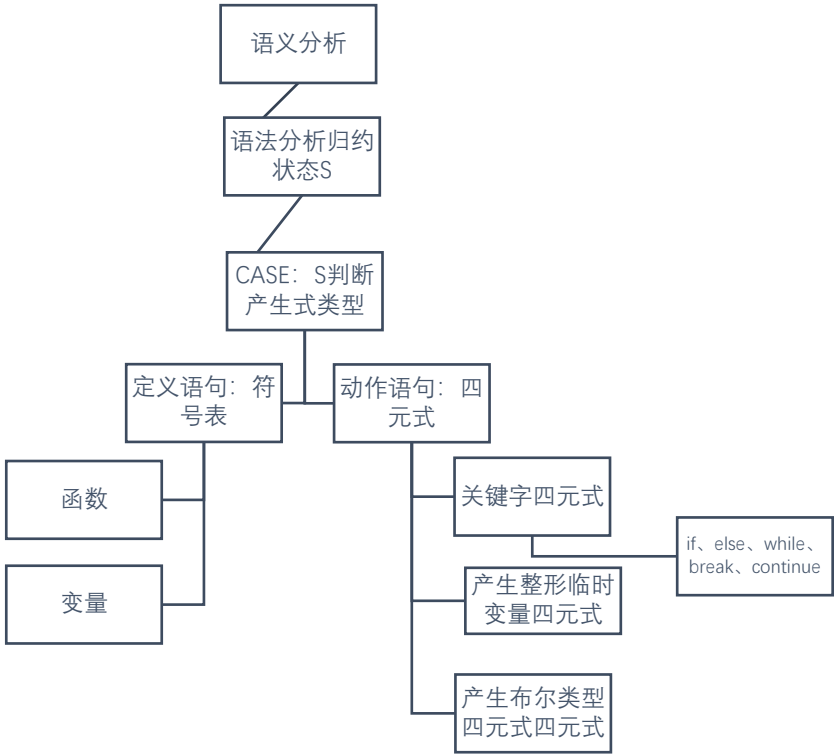
该部分负责人：王惟

4.4.1 语法分析结构图



4.4.2 语义分析结构图

特点分析：LR（1）文法，只在产生式最右侧归约的时候有语义动作



1. 语义分析处理，状态共有 112 个，分类为 26 中语义动作。
2. LR（1）是自底向上的分析，动作都在产生式的最右端，故用产生式序号作为动作标志。
3. 地址的分配，每个函数是一个连续的地址单元，地址的前四位是用来链接区域，由于用户定义类型只有整形，所以 int 定义语句，或整形临时变量，每次分配 4 个存储单元，布尔类型只产生于布尔表达式中，临时变量申请 2 个存储单元。

4.4.3 实验结果图

测试代码：辗转相除最大公因数

SyncTable.h	Token.cpp	SyncTable.cpp
1	int main() {	
2	int a;	
3	int b;	
4	a=32;	
5	b=18;	
6	while(b>0) {	
7	int c;	
8	c=a%b;	
9	a=b;	
10	b=c;	
11	}	
12	}	

语法分析，产生式归约全过程：

```

C:\Users\王博\source\repos\Project50\Debug\Project50.exe
type_specifier          -> int
function_name           -> id
parameter               -> ( )
type_specifier          -> int
declaration             -> type_specifier id ;
decalration_list        -> decalration
type_specifier          -> int
declaration             -> type_specifier id ;
decalration_list        -> decalration_list declaration
statement               -> decalration_list
statement_list          -> statement
primary_expression      -> id
last_expression         -> int_num
multiplicative_expression -> last_expression
additive_expression     -> multiplicative_expression
relational_expression   -> additive_expression
equality_expression     -> relational_expression
logical_and_expression  -> equality_expression
logical_or_expression   -> logical_and_expression
assignment_expression   -> logical_or_expression
assignment_expression   -> primary_expression = assignment_expression
expression              -> assignment_expression
expression_statement    -> expression ;
statement               -> expression_statement
statement_list          -> statement_list statement
primary_expression      -> id
last_expression         -> int_num

```

```

last_expression          -> int_num
multiplicative_expression -> last_expression
additive_expression      -> multiplicative_expression
relational_expression    -> additive_expression
equality_expression      -> relational_expression
logical_and_expression   -> equality_expression
logical_or_expression    -> logical_and_expression
assignment_expression    -> logical_or_expression
assignment_expression    -> primary_expression = assignment_expression
expression               -> assignment_expression
expression_statement     -> expression ;
statement                -> expression_statement
statement_list            -> statement_list statement
last_expression          -> id
multiplicative_expression -> last_expression
additive_expression      -> multiplicative_expression
relational_expression    -> additive_expression
last_expression          -> int_num
multiplicative_expression -> last_expression
additive_expression      -> multiplicative_expression
relational_expression    -> relational_expression > additive_expression
equality_expression      -> relational_expression
logical_and_expression   -> equality_expression
logical_or_expression    -> logical_and_expression
assignment_expression    -> logical_or_expression
expression               -> assignment_expression
type_specifier           -> int

```

```

expression               -> assignment_expression
type_specifier           -> int
declaration              -> type_specifier id ;
decalration_list         -> declaration
statement                -> decalration_list
statement_list            -> statement
primary_expression       -> id
last_expression          -> id
multiplicative_expression -> last_expression
last_expression          -> id
multiplicative_expression -> multiplicative_expression % last_expression
additive_expression      -> multiplicative_expression
relational_expression    -> additive_expression
equality_expression      -> relational_expression
logical_and_expression   -> equality_expression
logical_or_expression    -> logical_and_expression
assignment_expression    -> logical_or_expression
assignment_expression    -> primary_expression = assignment_expression
expression               -> assignment_expression
expression_statement     -> expression ;
statement                -> expression_statement
statement_list            -> statement_list statement
primary_expression       -> id
last_expression          -> id
multiplicative_expression -> last_expression
additive_expression      -> multiplicative_expression
relational_expression    -> additive_expression

```

```

logical_or_expression    -> logical_and_expression
assignment_expression    -> logical_or_expression
assignment_expression    -> primary_expression = assignment_expression
expression               -> assignment_expression
expression_statement     -> expression ;
statement                -> expression_statement
statement_list            -> statement_list statement
primary_expression       -> id
last_expression          -> id
multiplicative_expression -> last_expression
additive_expression      -> multiplicative_expression
relational_expression    -> additive_expression
equality_expression      -> relational_expression
logical_and_expression   -> equality_expression
logical_or_expression    -> logical_and_expression
assignment_expression    -> logical_or_expression
assignment_expression    -> primary_expression = assignment_expression
expression               -> assignment_expression
expression_statement     -> expression ;
statement                -> expression_statement
statement_list            -> statement_list statement
compound_statement       -> { statement_list }
statement                -> compound_statement
iteration_statement       -> while ( expression ) statement
statement                -> iteration_statement
statement_list            -> statement_list statement
compound_statement       -> { statement_list }

```

```

assignment_expression    -> primary_expression = assignment_expression
expression               -> assignment_expression
expression_statement     -> expression ;
statement                -> expression_statement
statement_list           -> statement_list statement
compound_statement       -> { statement_list }
statement                -> compound_statement
iteration_statement       -> while ( expression ) statement
statement                -> iteration_statement
statement_list           -> statement_list statement
compound_statement       -> { statement_list }
function_definition       -> type_specifier function_name parameter compound_statement
external_declaration     -> function_definition
translation_unit         -> external_declaration
LR(1)分析成功

```

```

符号表
SYNBL
main / p 1
a i v 5
b i v 9
c i v 15

PFINFL
main 0 4 0

Const
32 32 1000
18 18 1004
0 0 1008

VALL
a 5 8
b 9 12
t1 13 14
c 15 18
t2 19 22

```

```

四元式
func main _ _
= 32 _ a
= 18 _ b
wh _ _ _
> b 0 t1
do t1 _ _
% a b t2
= t2 _ c
= b _ a
= c _ b
we _ _ _
end func _ _

```

```

四元式地址形式
func main _ _
= 1000 _ 5
= 1004 _ 9
wh _ _ _
> 9 1008 13
do 13 _ _
% 5 9 19
= 19 _ 15
= 9 _ 5
= 15 _ 9
we _ _ _
end func _ _

```

四元式符号表

5 编译器后端设计

5.1 四元式的优化

该部分负责人：马超、王惟

5.1.1 基本块的划分

负责人：马超

方法：每次遇到关键字四元式，则将前后四元式划分为两个基本块。

5.1.2 常值表达式优化

负责人：王惟

方法：查询四元式，第二个和第三个单元，若都为常数，则删除第三个单元，将运算结果保存在第二个单元

5.1.3 公共子表达式优化

负责人：王惟

方法：查询四元式，若两个四元式前三个单元都相同，便利两个四元式中间的四元式，若中间两个单元未在这些四元式第四个单元出现，在第一个四元式前添加前三个单元运算结果的临时变量，并修改这两个相同的四元式为直接临时变量赋值，减少运算量。

5.1.4 删除无用赋值

负责人：王惟

方法：查询四元式，若 A 出现在两个四元式的第四个单元，且 A 在两个四元式之间的四元式没有引用，则删除第一个四元式。

5.1.5 实验结果图

5.2 目标代码具体实现

该部分负责人：秦立国

多重 while 与 if, else, break 结合实现统计质数个数：

源代码

四元式

地址形式

<pre> int main() { int i; int j; int k; int m; int n; i=100; m=0; while(k==1) { while(i>1) { j=1; while(j<i) { j=j+1; if((i%j)==0) { m=m+1; break; } else { n=n+1; } } i=i-1; } break; } return 0; } </pre>	<pre> func main -- = 100 i -- = 0 m -- == k 1 t1 -- wh -- -- do t1 -- > i 1 t2 -- wh -- -- do t2 -- = 1 j -- < j i t3 -- wh -- -- do t3 -- + j 1 t4 -- = t4 j -- % i j t5 -- == t5 0 t6 -- ifl t6 -- + m 1 t7 -- = t7 m -- br -- -- el -- -- + n 1 t8 -- = t8 n -- iel -- -- we -- -- - i 1 t9 -- = t9 i -- we -- -- br -- -- we -- -- re -- 0 -- end func -- </pre>	<pre> func main -- = 1000 5 -- = 1004 17 -- == 13 1008 25 -- wh -- -- do 25 -- > 5 1008 27 -- wh -- -- do 27 -- = 1008 9 -- < 9 5 29 -- wh -- -- do 29 -- + 9 1008 31 -- = 31 9 -- % 5 9 35 -- == 35 1004 39 -- ifl 39 -- + 17 1008 41 -- = 41 17 -- br -- -- el -- -- + 21 1008 45 -- = 45 21 -- iel -- -- we -- -- - 5 1008 49 -- = 49 5 -- we -- -- br -- -- we -- -- re -- 1004 -- end func -- </pre>
---	--	---

汇编代码生成:

第一部分

```

DSEG      SEGMENT
RUL        DB          100 DUP (0)
DSEG      ENDS
CSEG      SEGMENT
ASSUME    CS:CSEG, DS:DSEG
START:    MOV     AX, DSEG
          MOV     DS, AX
          MOV     AX, SSEG
          MOV     SS, AX
          MOV     [1000H], 1
          MOV     [1004H], 2
          MOV     [1008H], 100
          MOV     [1012H], 5
          MOV     [1016H], 0
          MOV     CX, [1000H]
          MOV     [0005H], CX
          MOV     CX, [1004H]
          MOV     [0011H], CX
          MOV     AX, [000DH]
          MOV     BX, [1008H]
          CMP     AX, BX
          JNE     UNEQUAL0
          MOV     AL, 1
          JMP     END_EQUAL0
UNEQUAL0:  MOV     AL, 0
END_EQUAL0: MOV     [0019H], AL

```

第二部分

```

WHILE0:    MOV     AH, [0019H]
          CMP     AH, 1
          JNE     ENDWHILE0
          MOV     AX, [0005H]
          MOV     BX, [1008H]
          CMP     AX, BX
          JNA     UNABOVE1
          MOV     AL, 1
          JMP     END_ABOVE1
UNABOVE1:  MOV     AL, 0
END_ABOVE1: MOV     [001BH], AL
WHILE1:    MOV     AH, [001BH]
          CMP     AH, 1
          JNE     ENDWHILE1
          MOV     CX, [1008H]
          MOV     [0009H], CX
          MOV     AX, [0009H]
          MOV     BX, [0005H]
          CMP     AX, BX
          JNB     UNBELOW2
          MOV     AL, 1
          JMP     END_BELOW2
UNBELOW2:  MOV     AL, 0
END_BELOW2: MOV     [001DH], AL

```

第三部分

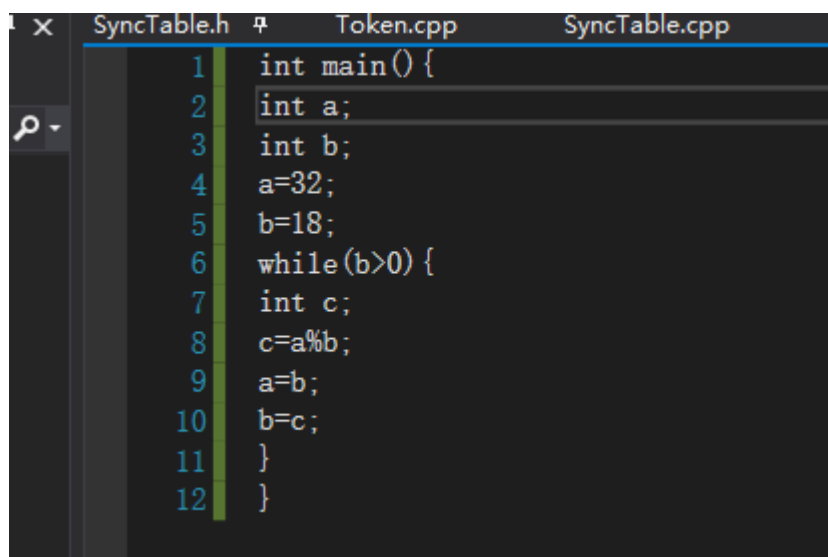
第四部分

UNBELOW2:	MOV	AL, 0	JNE	ELSE0
END_BELOW2:	MOV	[001DH], AL	MOV	AX, [0011H]
WHILE2:	MOV	AH, [001DH]	MOV	BX, [1008H]
	CMP	AH, 1	ADD	AX, BX
	JNE	ENDWHILE2	MOV	[0029H], AX
	MOV	AX, [0009H]	MOV	CX, [0029H]
	MOV	BX, [1008H]	MOV	[0011H], CX
	ADD	AX, BX	JMP	ENDWHILE2
	MOV	[001FH], AX	JMP	ENDIF_ELSE0
	MOV	CX, [001FH]	ELSE0:	MOV
	MOV	[0009H], CX		AX, [0015H]
	MOV	AX, [0005H]		BX, [1008H]
	MOV	BX, [0009H]		ADD
	MUL	BX		AX, BX
	MOV	[0023H], DX		MOV
	MOV	AX, [0023H]		[002DH], AX
	MOV	BX, [1004H]		CX, [002DH]
	CMP	AX, BX		MOV
	JNE	UNEQUAL3		[0015H], CX
	MOV	AL, 1	ENDIF_ELSE0:	JMP
	JMP	END_EQUAL3		WHILE2
UNEQUAL3:	MOV	AL, 0	ENDWHILE2:	MOV
END_EQUAL3:	MOV	[0027H], AL		AX, [0005H]
	MOV	AH, [0027H]		BX, [1008H]
	CMP	AH, 1		SUB
	JNE	ELSE0		AX, BX
				MOV
				[0031H], AX
				MOV
				CX, [0031H]
				MOV
				[0005H], CX
				JMP
				WHILE1
			ENDWHILE1:	JMP
				ENDWHILE0
				WHILE0
			ENDWHILE0:	MOV
				AH, 4CH
				INT
				21H
			CSEG	ENDS
			END	START

5.3 在 DOS 下运行汇编代码，并用 debug 验证正确性

该部分负责人：王惟

如下程序辗转相除计算最大公因数



```

1  int main() {
2  int a;
3  int b;
4  a=32;
5  b=18;
6  while(b>0) {
7  int c;
8  c=a%b;
9  a=b;
10 b=c;
11 }
12 }

```

得到的目标代码：

```

C:\Users\王雅\source\repos\Project50\Debug\Project50.exe

目标代码
SSEG          SEGMENT
STK            DB                      2000 DUP (0)
SSEG          ENDS
DSEG          SEGMENT
RUL            DB                      2000 DUP (0)
DSEG          ENDS
CSEG          SEGMENT
ASSUME        CS:CSEG, DS:DSEG, SS:SSEG
START:        MOV          AX, DSEG
               MOV          DS, AX
               MOV          AX, SSEG
               MOV          SS, AX
               MOV          WORD PTR DS:[1000H], 32
               MOV          WORD PTR DS:[1004H], 18
               MOV          WORD PTR DS:[1008H], 0
               MOV          CX, DS:[1000H]
               MOV          DS:[0005H], CX
               MOV          CX, DS:[1004H]
               MOV          DS:[0009H], CX

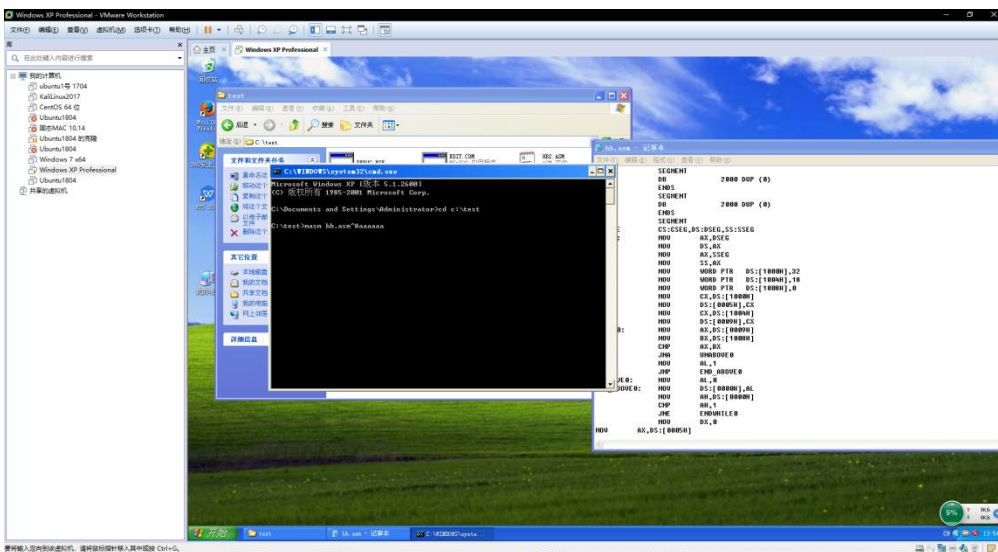
```

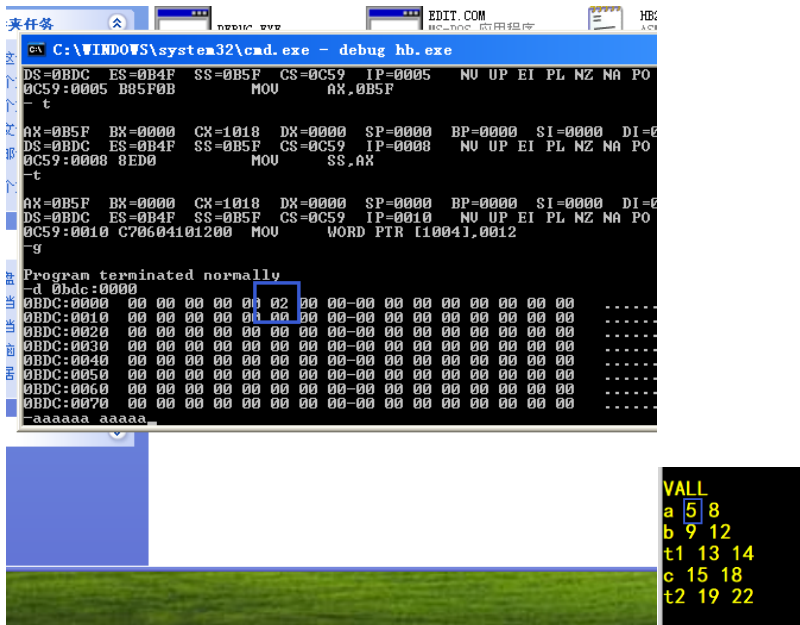
```

C:\Users\王雅\source\repos\Project50\Debug\Project50.exe

WHILE0:        MOV          AX, DS:[0009H]
               MOV          BX, DS:[1008H]
               CMP          AX, BX
               JNA          UNABOVE0
               MOV          AL, 1
               JMP          END_ABOVE0
UNABOVE0:      MOV          AL, 0
END_ABOVE0:    MOV          DS:[0000H], AL
               MOV          AH, DS:[0000H]
               CMP          AH, 1
               JNE          ENDWHILE0
               MOV          DX, 0
MOV            AX, DS:[0005H]
               MOV          BX, DS:[0009H]
               DIV          BX
               MOV          DS:[000DH], DX
               MOV          CX, DS:[000DH]
               MOV          DS:[0000H], CX
               MOV          CX, DS:[0009H]
               MOV          DS:[0005H], CX
               MOV          CX, DS:[0000H]
               MOV          DS:[0009H], CX
               JMP          WHILE0
ENDWHILE0:    MOV          AH, 4CH
               INT          21H
CSEG          ENDS
END

```





可看到，0bdc: 0005 为 2，这是 a 所在的地址，即 32 与 18 最大公因数为 2，正确

6 结论

编译原理课程设计是本课程重要的综合实践教学环节，是对平时实验的一个补充。通过编译器相关子系统的设计，使学生能够更好地掌握编译原理的基本理论和编译程序构造的基本方法和技巧，融会贯通本课程所学专业理论知识；培养学生独立分析问题、解决问题的能力，以及系统软件设计的能力；培养学生创新能力及团队协作精神。编译程序是一种翻译程序，特指把某种高级程序设计语言翻译成具体计算机上低级程序设计语言。

LR（1）分析结果正确，编译实验成功。

7 参考文献

- 陈火旺.《程序设计语言编译原理》（第 3 版）. 北京：国防工业出版社. 2000.
- 美 Alfred V.Aho Ravi Sethi Jeffrey D. Ullman 著. 李建中，姜守旭译.《编译原理》. 北京：机械工业出版社. 2003.
- 美 Kenneth C.Louden 著. 冯博琴等译.《编译原理及实践》. 北京：机械工业出版社. 2002.
- 金成植著.《编译程序构造原理和实现技术》. 北京：高等教育出版社. 2002.

8 收获、体会和建议

王惟：

这次课设，我负责的主要是需求分析，任务分配，头文件编写，语法控制器、语义、四元式、优化、目标代码运行以及整合调试修改，其中对我感触最深的是任务分配。在刚开始时，任务分配不均匀，组员之间没有实现进度并行，相互等待前面任务的完成和突破才能进行自己接下来的任务。听了老师的意见之后，我们立刻修改了组员之间的任务分配，提前编制、规定好模块之间的接口，每个人都有输入模式、输出模式，变“串行”为“并行”，每一节课，每一个人都有进度。其他的具体任务加深了对编译原理，也同时复习了课程，最主要的还是，加深了对团队合作的认识和理解，在分工合作中受益匪浅。

建议：建议编译原理课程设计可以与课程上课进度并行，这样可以在学习的同时进行应用，时间安排更为合理，与期末考试、其他课设不冲突，将编译课设做的更加全面，更加精致。

周雨桓：

这次编译原理实验真的让我收获许多也让我感触良多。

刚开始听到课设的事的时候，我是和另一个编程大佬 A 组的队，那时候我们以为这次课设任务较为轻松，时间充裕，两个人足够完成。但是在听到老师的课设动员的时候我们发现这次课设其实是一个十分复杂且艰巨的任务，模块很多，而且我们要从纯理论型的编译原理课转换成代码，需要非常深的理解和应用，再加上老师说 3-5 人组队，而且一个队伍会限制优的数量，这让我们原本的两人队伍变得不稳定，急需有能力又愿意让出组长位置的队员，同时我发现落在组长身上的担子会很大——工作量最多，且本次课设很难。原本是想当组长的，后来和 A 调和了一下，我们决定做出一个完整且优良的编译器，从而让一个组能至少有两个优，这样我让出了组长位置，但同时为了我自己能得一个满意的成

绩，我依旧需要为整个课设付出很大的努力。然后我去找剩下的组员，C、D。本来最初的期望是他们能提供一点帮助，在一些简单却繁杂的工作上帮忙，主要工作又我和A来做，但往后事情的发展着实出乎我的预料。

首先是基础工作的开展，如文法制定，结构体的确定，相关资料的查询，组员分工、PPT和相关资料的阅读，这里是由我和A共同完成，但即便如此这里也花费了比预料时间远远长的时间。特别是符号表的活动记录的理解。

完成这些基础工作后，因为A出去比赛，他打算交给我一个“简单”的工作：LR（1）分析表的制定，这样他回来以后我们就能快速开始语法和语义部分。

当初选定LR（1）分析法的原因也是我们觉得LR（1）分析法的优越性能成为我们编译器的一大亮点，在递归下降和LL（1）中脱颖而出。但是当我真正开始这项工作的時候，我发现我们远远低估了LR(1)分析表的制作难度。

首先为了了解LR（1）分析法，我花了大半天时间阅读PPT，阅读网络资料，在确定对LR（1）的理解已经较为准确后，我从文法的读取开始，制作LR（1）算法。

首先是文法的读入，因为我们之前已经把“每个左部对应一个右部”的产生式转换成“每个左部对应多个右部”的产生式——即用“|”连接，这让我的文法读取、文法数值化工作增添了很多不必要的负担，当我用了大约一天的时间用来做这项工作的時候，我才发现我们当初产生式的转换，不仅仅对阅读的用处非常小，而且增加工作量，十分的不值得。但时间已经花费了，已经没办法了。接下来，我上网寻找LR（1）算法的相关资料并且阅读理解——关于“可能的符号”的理解中途还出现了错误，这又花费了半天时间。

在着手实现算法的过程中，我发现为了算法的实现，文法需要高度的数值化，以及有序的数据结构，于是我在文法进一步数值化上又花费了不少精力。同时因为这些数值，算法中用到了大量的一维数组、二维数组、甚至有用到三维数组，这大大拖慢了我实现算法

的速度——为了不让之后的程序出现莫名其妙的 bug，我必须把每个数组的标号对应无误，又要把每个数值和相关文法对应无误。再加上算法本身有一定的复杂性、有递归调用、边界条件判断条件较难理解，我用了将近一天半的时间理解实现它们，但是因为它这个算法是递归性生成 130 多条状态，边界条件的理解困难。而且每个状态对应的都是纯数字——数值化的文法、序号等，导致 bug 排查工作进展缓慢。工作初期出现的 bug 还较为简单，大概花费了大半天时间去完成。在我拿了 PPT 的样例 LR（1）文法和分析表对比，以为工作全部完成了。

但是在整合的时候又出现了很多逻辑上、难排查的 bug，如左递归文法的特殊性、判断边界条件的下标、“可能的符号”是一个集合、集合不能更新、重复压入项目，这些 bug，因为高度数值化、数组的复杂、算法本身逻辑较为复杂、缺少标准的 LR1 文法和分析表来及时发现生成表的不准确等等原因，每一个都需要我花费大量的精力去检查，最终这里竟然又花费了将近三天时间才得出了一个完美的、有自信的、正确的算法。这时小组的工作已经被我拖慢了许多，而我连续五六天耗费在 700 行代码上，还排查 bug 这么慢，让我筋疲力尽，没办法较快加入剩下的工作中。

我们小组在验收的前四天，周三的时候，连一个完整的前端都没做出来，那时候我的 bug 也没排完，遭到组长的质疑：我怕你排不出来，到时候我们全组的工作都将白费。再加上之前其他组员比较佛，其他工作进展缓慢，我被一个“简单的工作”——这个工作甚至不用 LR（1）的组完全不用做，我也很担心这个工作会不会被老师承认，也被繁杂的 bug 弄得信心全无、筋疲力尽，整个组的气氛都充满了焦虑，现状不容乐观。

当时我也很焦虑，觉得是自己拖慢了整个组，自己的 bug 还不知道能不能排出来。这时候我向组员保证一件事：没有排不出来的 bug，这个 bug 我一定排的出来，也许时间会长一点，一定能做得到，你们其他工作抓紧做，最后和我完整的合到一起，就一切来得

及。如果就这样放弃，用别人的代码，就要重新阅读别人的代码，同时还会有一定的风险，虽然也可能节省出来复习期末考的时间，但这等于否认之前的决心、信心和努力，太憋屈了。这时我问组长——这样你决定继续做还是用别人的代码？——继续做。

在这样的气氛下，我动员了佛系同学 C，他也明白了事情的急迫性，于是虽然前途未卜，我们不问东西。

最后大发力的是 C 和 A，最后几天他们把精力都放在了课设上，特别是原本我们不报特别大期望的 C 同学，也十分的努力和尽心，最后我们终于顺利的完成了整个课设，一个完整的，支持很多语法的编译器，有 LR (1) 分析表全自动生成的特色。每个人的努力，面临危机时的决心，让我们从周三晚上焦虑的一个组变成了周日第一个验收完毕舒坦走回宿舍的一个组。

但还是很有遗憾的，我在 LR (1) 分析上花费了大量的时间、精力，导致后来没有能继续为组做其他的主要工作。但我的工作都是实打实的，必要的，也的确花费了我大量的精力，也算是问心无愧了吧。

秦立国：

为期两周的课程设计有些匆忙，此次课程我负责的任务是词法分析与目标代码的生成。词法分析部分进行的比较顺利，但是也发现了一些问题，比如说忘了去掉由 Tab 产生的空格，还有一些行数位置的统计错误。在目标代码生成刚开始的时候无从下手，用了一段时间学习符号表以及四元式的形式，花费了很长时间才有所构思各种四元式的处理，由于汇编语言与教学 ppt 上的伪指令代码有所差异，并没有太多的寄存器可用，所以对于没能使用寄存器的分配感到遗憾。同时通过组长的帮助也顺利的解决了许多困扰许久的问题，并且最终目标代码能够在 Dos 上成功运行。此次编译原理课程设计不仅让我熟悉了课本上的内

容，还增强了自己的动手操作能力，同时跟组员们学到了很多。

马超：

经过两周的课程设计，终于也看到的团队的成果，虽然自己写的仅仅是这个编译器中很小的一部分，但毕竟完全是自己一条条代码写出来，心里还是蛮有成功感的。学习了一学期的编译原理课程，平时也基本能够跟上老师的讲课。但要真正实践去写代码，还是要费许多功夫的。通过实验过程中不断复习老师所讲，不断查找资料。到现在终于明白真正的编译器的工作过程是这样的，不得不承认，仅仅学过不一定会掌握，实践才是最重要的！

我还总结了写代码的经验：那就是多写注释，这样调试时就容易很多。方便自己也方便组员。

总之，本次课设让我对编译原理的理论知识有了更深的认识，而且让我对 c++ 的使用变得更加熟练，除此之外，团队沟通也是课程设计中非常重要的环节，以前写代码只用理清自己的思路，这次却必须进行组员之间的协调。以后的学习工作中这种能力是不可或缺的，所以提升了团队沟通能力也是我在这次课设中最重要的收获之一。