编译原理报告

姓名：李昕
学号：2017302629

目录

编译原理报告 1
一、文法说明 2
1.1 文法定义 2
1.2 部分文法说明 3
1.3 文法部分的问题及解决方案 6
二、总体设计 6
三、详细设计 8
3.1 词法分析 8
3.1.1 函数与接口功能说明 8
3.1.2 关键变量和数据结构设计 8
3.1.3 设计思路 9
3.2 语法分析 9
3.2.1 函数与接口功能说明 9
3.2.2 关键变量和数据结构设计 10
3.2.3 设计思路 11
3.3 符号表 12
3.3.1 函数与接口功能说明 12
3.3.2 关键变量和数据结构设计 13
3.3.3 设计思路 13
3.4 语义检查 15
3.4.1 函数与接口功能说明 15
3.4.2 关键变量和数据结构设计 15
3.4.3 设计思路 15
3.5 生成四元式 16
3.5.1 函数与接口功能说明 16
3.5.2 关键变量和数据结构设计 16
3.5.3 设计思路 17
3.6 代码优化 18
3.6.1 函数与接口功能说明 18
3.6.2 优化效果举例 18
3.6.2.1 四元式优化部分 18

3.6.2.2 AST 优化部分 21
3.6.3 关键变量和数据结构设计 22
3.6.4 设计思路 22
3.7 代码生成 22
3.7.1 函数与接口功能说明 22
3.7.2 关键变量和数据结构设计 23
3.7.3 设计思路 23
3.7.3.1 运行栈结构的设计 23
3.7.3.2 寄存器分配策略 25
3.7.3.3 指令选择和翻译策略 25
3.8 错误处理 27
3.8.1 函数与接口功能说明 27
3.8.2 关键变量和数据结构设计 27
3.8.3 设计思路 27
四、版本结构 27
五、测试 28
六、总结与感想 32

一、文法说明

1.1 文法定义

此文法为扩充的 C0 文法，支持数组，无实型，支持 for 语句和 do-while 语句，不支持 switch-case 语句

< 加法运算符 > ::= + | -
< 乘法运算符 > ::= * | /
< 关系运算符 > ::= < | <= | > | >= | != | ==
< 字母 > ::= _ | a | ... | z | A | ... | Z
< 数字 > ::= 0 | < 非零数字 >
< 非零数字 > ::= 1 | ... | 9
< 字符 > ::= ' < 加法运算符 > ' | ' < 乘法运算符 > ' | ' < 字母 > ' | ' < 数字 > '
< 字符串 > ::= " {十进制编码为 32,33,35-126 的 ASCII 字符} "
< 程序 > ::= [< 常量说明 >] [< 变量说明 >] { < 有返回值函数定义 > | < 无返回值函数定义 > } < 主函数 >
< 常量说明 > ::= const < 常量定义 > ;{ const < 常量定义 > ;}
< 常量定义 > ::= int < 标识符 > = < 整数 > {, < 标识符 > = < 整数 > }| char < 标识符 > = < 字符 > {, < 标识符 > = < 字符 > }
< 无符号整数 > ::= < 非零数字 > { < 数字 > }
< 整数 > ::= [+ | -] < 无符号整数 > | 0
< 标识符 > ::= < 字母 > { < 字母 > | < 数字 > }
< 声明头部 > ::= int < 标识符 > |char < 标识符 >

< 变量说明 > ::= < 变量定义 > ;{ < 变量定义 > ;}
< 变量定义 > ::= < 类型标识符 > (< 标识符 > | < 标识符 > '[' < 无符号整数 > ']'){,(< 标识符 > | < 标识符 > '[' < 无符号整数 > ']') }
< 无返回值函数定义 > ::= void < 标识符 > '(' < 参数 > ')''{' < 复合语句 > '}'
< 有返回值函数定义 > ::= < 声明头部 > '(' < 参数 > ')' '{' < 复合语句 > '}'
< 复合语句 > ::= [< 常量说明 >] [< 变量说明 >] < 语句列 >
< 参数 > ::= < 参数表 >
< 参数表 > ::= < 类型标识符 > < 标识符 > {, < 类型标识符 > < 标识符 > }| < 空 >
< 主函数 > ::= void main'('')' '{' < 复合语句 > '}'
< 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 > | < 标识符 > '[' < 表达式 > ']' | < 整数 > | < 字符 > | < 有返回值函数调用语句 > |'(' < 表达式 > ')'
< 语句 > ::= < 条件语句 > | < 循环语句 > | '{' < 语句列 > '}' | < 有返回值函数调用语句 > ;| < 无返回值函数调用语句 > ; | < 赋值语句 > ; | < 读语句 > ; | < 写语句 > ; | < 空 > ; | < 返回语句 > ;
< 赋值语句 > ::= < 标识符 > = < 表达式 > | < 标识符 > '[' < 表达式 > ']'= < 表达式 >
< 条件语句 > ::= if '(' < 条件 > ')' < 语句 > [else < 语句 >
< 条件 > ::= < 表达式 > < 关系运算符 > < 表达式 > | < 表达式 >
< 循环语句 > ::= do < 语句 > while '(' < 条件 > ')' |for'(' < 标识符 > = < 表达式 > ; < 条件 > ; < 标识符 > = < 标识符 > (+|-) < 步长 > ')' < 语句 >
< 步长 > ::= < 无符号整数 >
< 有返回值函数调用语句 > ::= < 标识符 > '(' < 值参数表 > ')'
< 无返回值函数调用语句 > ::= < 标识符 > '(' < 值参数表 > ')'
< 值参数表 > ::= < 表达式 > {, < 表达式 > } | < 空 >
< 语句列 > ::= { < 语句 > }
< 读语句 > ::= scanf '(' < 标识符 > {, < 标识符 > }')'
< 写语句 > ::= printf'(' < 字符串 > , < 表达式 > ')'|printf '(' < 字符串 > ')'|printf '(' < 表达式 > ')'
< 返回语句 > ::= return['(' < 表达式 > ')']
< 类型标识符 > ::= int | char

1.2 部分文法说明

< 程序 > ::= [< 常量说明 >] [< 变量说明 >] { < 有返回值函数定义 > | < 无返回值函数定义 > } < 主函数 >

分析：一个程序可以有常量说明和变量说明和函数定义说明，也可以没有，如果都有的话就必须得遵从先声明常量再声明变量再声明函数的顺序，不能先声明变量再声明常量。在程序中主函数是必须存在的

示例：

```
Const int c = 1;
Int b;
Void test()
{
}
```

```
Void main()
{
Printf("hello C0");
}
```

< 常量定义 > ::= int < 标识符 > = < 整数 > {, < 标识符 > = < 整数 > }| char < 标识符 > = < 字符 > {, < 标识符 > = < 字符 > }

分析：常量的定义是必须给定初始值的，常量有两种类型，不可以把数组定义为常量

示例：

```
Const int a = 1 ,  b = 2;
Const char c = 'c';
```

< 整数 > ::= [+	−] < 无符号整数 >	0

分析：一个整数可以是一个无符号整数也可以是带上正负号的无符号整数 或者 为 0

示例：

```
a = +10;
a = -10;
a = +0;
a = -0;
a = 10;
```

< 标识符 > ::= < 字母 > { < 字母 >	< 数字 > }

分析：标识符必须以字母开头或者下划线开头，后面可以跟数字或者字母或者下划线

示例：

```
int _a;
Int a_div;
```

< 复合语句 > ::= [< 常量说明 >] [< 变量说明 >] < 语句列 >

分析：复合语句类似于程序，也可以有常量和变量说明，但是不可以有函数定义

示例：

```
Void test()
{
    const char c = 'c';
    int b;
    b = 1;
}
```

< 主函数 > ::= void main'('')' '{' < 复合语句 > '}'

分析：main 函数必须用 void 修饰，不可以用 int 修饰

示例：

```
Void main()
{
    Printf("hello C0");
}
```

< 表达式 > ::= [+ | −] < 项 > { < 加法运算符 > < 项 > }

分析：表达式前面可以带正负号来作用于整个表达式

示例：

```
a = + (1+2*4);
a = - (1+2*4);
```

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

分析：赋值语句包括普通变量的赋值，还包括数组元素的赋值

示例：

```
a = 1;
a[2] = 5;
```

< 条件 > ::= < 表达式 > < 关系运算符 > < 表达式 > | < 表达式 >

分析：条件可以是两个表达式的关系运算，也可以只有单个的表达式，若只有单个的表达式，表达式的值为真则表示条件为真，反之为假

示例：

```
if(a)
{}
If(b+c)
{}
If(a>d)
{}
```

< 读语句 > ::= scanf '(' < 标识符 > {, < 标识符 > }')'

分析：读语句以关键字 scanf 开头，可以将内容依次读入多个变量内

示例：

```
scanf(a,b,c);
```

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

分析：写语句以关键字 printf 开头，可以只单独写一个字符串或者一个表达式，也可以写一个字符串加表达式

示例：

```
printf("the value is : ",a);
```

printf(“hello C0”);
printf(a);

1.3 文法部分的问题及解决方案

1.文法没有左递归，但是存在 first 集合冲突的情况，为了不增加复杂性，我没有选择修改文法，而是采用预读两个单词的方式来解决这个问题。
需要预读的文法部分如下：

< 程序 > ::= [< 常量说明 >] [< 变量说明 >] { < 有返回值函数定义 > | < 无返回值函数定义 > } < 主函数 >

这里的变量说明和函数定义说明存在冲突的情况，需要预读两个符号来确定是变量说明还是函数定义

< 因子 > ::= < 标识符 > | < 标识符 > ‘[’ < 表达式 > ‘]’ | < 整数 > | < 字符 > | < 有返回值函数调用语句 > |‘(’ < 表达式 > ‘)’

这里变量和数组以及函数调用存在冲突，读到标识符后需要预读一个符号来判断是变量还是数组还是函数调用
2.文法有一个部分让我的词法分析很难做，即对+符号的识别和对整数+1 的识别，如果在表达式 1+2 中，词法分析器会把其拆解为两个单词一个是整数 1，一个是整数+2，即无法识别出+符号，这给我的语法分析带来了问题，在这种地方一直会报错，这个问题也一直困扰我到现在，我不想修改文法，就先规定类似于 1+2 这种表达式必须分开写：1 + 2 这样我的词法分析器就可以将其识别为 3 个单词，所以我的测试程序中表达式中间都带有空格

二、总体设计

开发采用增量开发的方式，将项目划分为词法分析，语法分析，语义分析，中间代码生成，代码优化，目标代码生成六个阶段，每一个阶段严格贯彻从需求分析到详细设计到编码到单元测试的方式，把控代码质量，在开发前期找到尽可能多的 bug，每一个阶段集成前面阶段的代码做集成测试，最后整个系统做系统测试。

中间代码生成之前我用的是 C 语言，后面感觉到开发的不便，需要使用大量复杂的数据结构辅助，因此之后我把代码改成了 C++，开发环境为 Windows10,IDE 为 CodeBlocks。

项目模块划分如下：

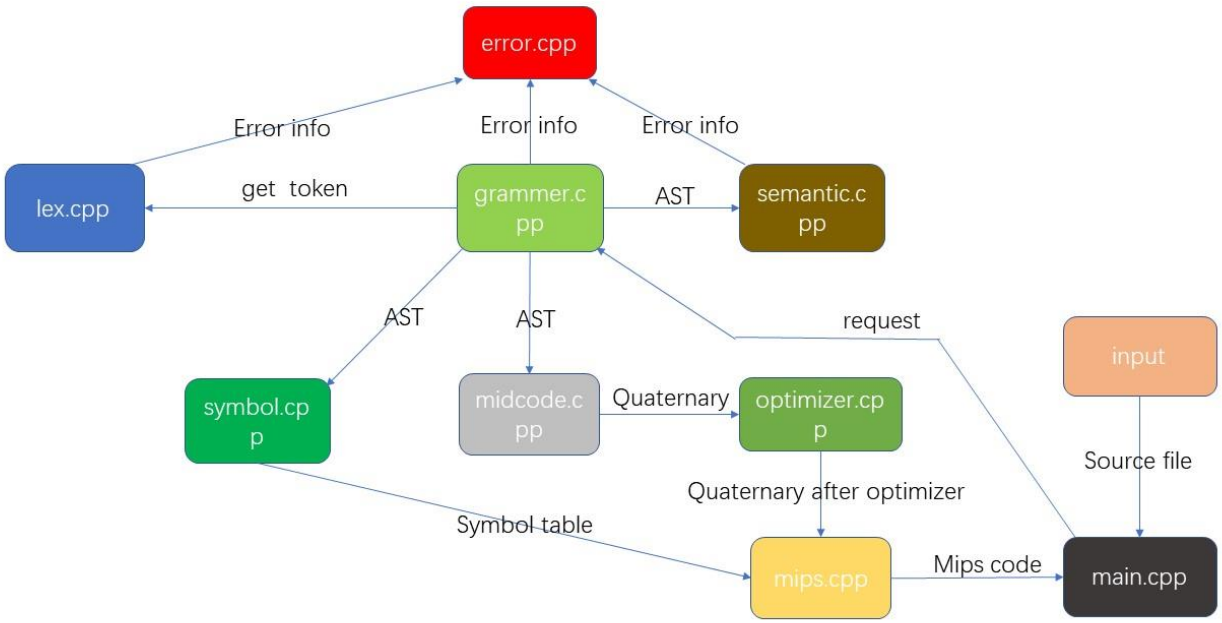
模块	功能
词法分析	将源程序中的字符串划分为一个个的单词
语法分析	根据文法检测源程序的正确性，并转化为抽象语法树
符号表	根据抽象语法树生成符号表
语义检查	在抽象语法树上检查源程序的语义
中间代码	根据抽象语法树生成四元式序列
代码优化	对中间表示抽象语法树和四元式进行优化
代码生成	将四元式序列转化成对应的 MIPS 汇编代码

主函数	将各个模块链接起来并设计用户接口
错误处理	对错误进行统一处理

各个模块对应的源文件即代码行如下:

文件	模块	代码行
lex.cpp+lex.h	词法分析	443
grammer.cpp+grammer.h	语法分析	1788
Symbol.cpp+symbol.h	符号表	513
Semantic.cpp+semantic.h	语义检查	260
Midcode.cpp+midcode.h	中间代码	510
Optimizer.cpp+optimizer.h	代码优化	907
Mips.cpp+mips.h	代码生成	978
Main.cpp+global.h	主函数	139
Error.cpp+error.h	错误处理	105
合计		5643

模块调用关系图如下:



三、详细设计

3.1 词法分析

3.1.1 函数与接口功能说明

函数原型	功能说明
int isDigit(int);	判断是否是 0-9 之间的数
int isLetter(int);	判断是否是字母 a-z|A-Z
int isNonZeroDigit(int);	判断是否是 1-9 之间的数
int isSymbol(int);	判断是否是+|-|*|/符号
int getnextchar();	读源文件中的下一个字符
void ungetnextchar();	回退一个字符
tokentype lookup(tokentype);	查找保留字表判断当前字符是否为保留字
void errorhander(int ,int);	词法部分的错误处理
tokentype getToken();	读一个单词送给语法部分

3.1.2 关键变量和数据结构设计

保留字表如下：

```
struct
{
    char* name;
    tokentype type;
}reservedwords[MaxReservedLength]={
{"void",VOID},{"int",INT},{"char",CHAR},{"if",IF},{"else",ELSE},{"const",CONST},{"while",WHILE},{"for",FOR},
{"do",DO},{"return",RETURN},{"break",BREAK},{"continue",CONTINUE},{"printf",PRINTF},{"scanf",SCANF},{"main",MAIN}
};
```

符号类型定义如下：

```
typedef enum
{
    ID,PLUS,MINUS,MULTIP,DIVD,LT,LTAE,BT,BTAE,EQ,NEQ,CHR,STRING,LBB,RBB,LB,RB,DOT,SEMI,ASSIGN,INTGE,
    VOID,INT,CHAR,IF,ELSE,CONST,WHILE,FOR,DO,RETURN,BREAK,CONTINUE,PRINTF,SCANF,MAIN,ERR,ENDFILE,LSB,RSB
}tokentype;
```

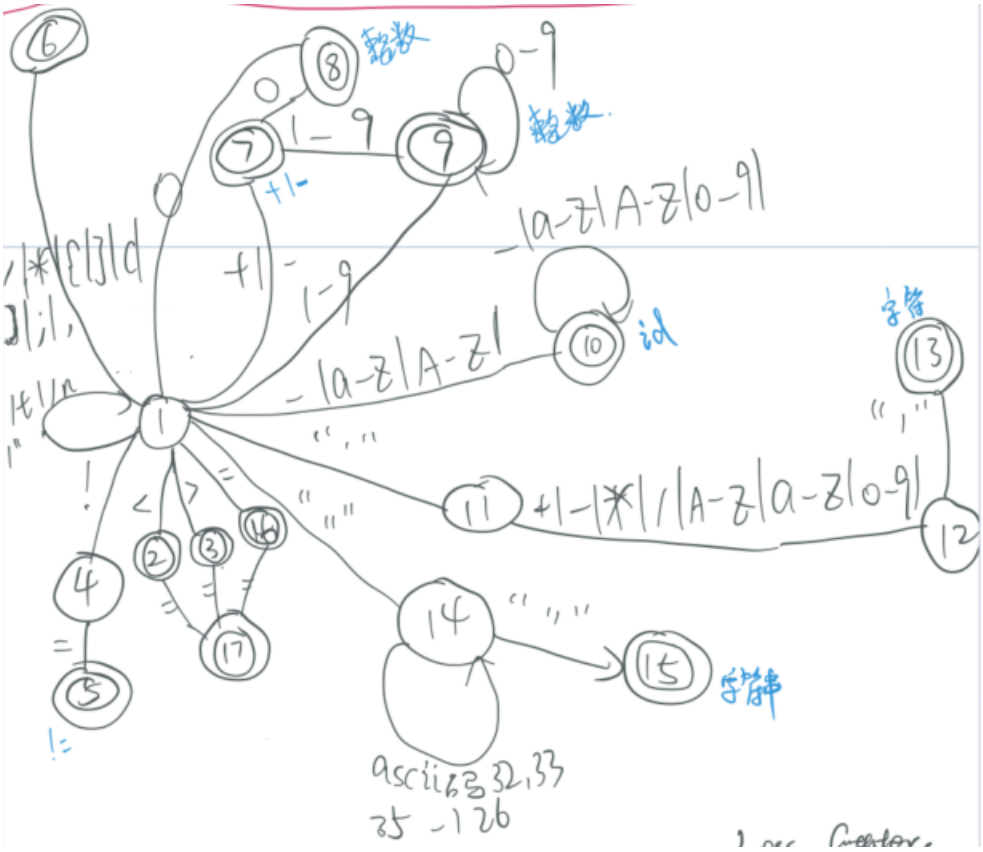
DFA 状态定义如下：

```
typedef enum
{
    START,END,EXCLAM,LESS,GREATER,EQUAL,CHARIN,CHARON,STR,POSANDNEG,IDM,INTGER
}statetype;
```

词法分析单词的字面量保存如下：

```
extern char tokenvalue[MaxTokenLength+1];
```

DFA 设计手稿如下：



3.1.3 设计思路

我首先在文法中找到跟词法分析相关的部分提取出来，然后划分出所有的保留字，为保留字设计保留字表，然后划分符号的类别，具体有标识符整数等类型，然后画出 DFA，划分 DFA 的有用状态，然后根据手稿用 switch-case 的结构实现 DFA。

值得一提的是我借鉴了 louden 的思想，在 DFA 中只有 ID 类型没有各个保留字，即所有的保留字先是识别出来为 ID 类型，然后查找保留字表，如果有匹配的就改成相应的保留字类型，这样设计减少了实现的复杂度。

3.2 语法分析

3.2.1 函数与接口功能说明

函数原型	功能
ASTNODE* nodeconstructor(NodeType);	生成一个抽象语法树节点
ASTNODE* const_dec();	常量声明部分
ASTNODE* vari_dec();	变量声明部分
ASTNODE* no_void_fun_dec();	有返回值的函数定义部分
ASTNODE* void_fun_dec();	无返回值的函数定义部分
ASTNODE* compond_stmt();	符合语句部分
ASTNODE* arg_lists();	参数表部分

ASTNODE* arg_list();	单个参数部分
ASTNODE* main_dec();	主函数定义部分
ASTNODE* exp();	表达式部分
ASTNODE* item();	项部分
ASTNODE* factor();	因子部分
ASTNODE* stmt();	语句部分
ASTNODE* assgin_stmt();	赋值语句部分
ASTNODE* condition_stmt();	If 语句部分
ASTNODE* condition();	条件部分
ASTNODE* for_stmt();	For 语句部分
ASTNODE* while_stmt();	Do-while 语句部分
ASTNODE* fun_call();	函数调用部分
ASTNODE* arg_values();	形参声明部分
ASTNODE* stmt_seq();	语句序列部分
ASTNODE* scanf_stmt();	Scanf 读语句部分
ASTNODE* printf_stmt();	Printf 读语句部分
ASTNODE* return_stmt();	Return 语句部分
void match(tokentype);	检查当前单词类型是否匹配并读入下一个单词
void copypre();	为预读一个单词辅助
void copyprepre();	为预读两个单词辅助
void eval(char* ,char*);	去掉字符串的双引号
void getASTNODE_STR(int);	得到节点类型的字符串如"PROGRAM"
void tranverse(ASTNODE*);	前序遍历语法树
ASTNODE* program()	构造 program 节点,得到语法树

3.2.2 关键变量和数据结构设计

语法树节点类型定义:

```
typedef enum
{
    PROGRAM , CONSTDECLARE , CONSTSUBDECLARE , IDTYPE , CONSTNUM , CONSTCHAR , CONSTSTRING,
    VARIDECLARE , SUBVARIDECLARE , ARRAYTYPE , FUNCDECLARE , ARGS , ARG , MAINTYPE , COM_STMT ,
    STMT_SEQ , STMT , OPTYPE ,  CALL , ASSIGN_STMT , IF_STME ,
    FOR_STMT , RELATION , VALUE_ARG, WHILE_STMT , SCANF_STMT , PRINTF_STMT , RETURN_STMT
}NodeType;
```

语法树节点定义:

```
typedef struct ASTNODE
{
    NodeType type;//AST节点的类型标记

    union{
        char* name ;//对IDTYPE类型节点，标识符的值进行存储，用于填符号表||对函数定义节点，保存函数的名字，用于填符号表
        int val;//对于CONSTNUM类型节点，存储数字的值 || 让value_arg节点记录函数调用时有多少个参数
        char ch;//对于CONSTCHAR类型节点，存储字面量
        char* str;//对于CONSTSTRING类型节点，存储字符串的值
    } attr;
    tokentype ttype;//记录是+,-,<,>,<=,>=,==,!= || 在for中记录步长是加还是减
    struct ASTNODE* childen[MAXCHILDREN];//子节点
    struct ASTNODE* sibling;//兄弟节点
    int lineno;//如果是id需要记录所在行数，用于填符号表
    Expkind kind;//用来记录id是void,int,char类型||对于CONSTDECLARE VARIDECLARE这种节点也使用这个变量来保存声明的类型
    int useful;//这个属性是用来记录对于exp构造的0+和0-，如果为false则表示是构造出来的辅助节点，不需要生成代码和类型检查
}ASTNODE;
```

3.2.3 设计思路

语法分析部分由于我用的是 C 语言，没有 vector 这样的数据结构，为了解决无法预知的常量定义，变量定义，函数定义，语句等的数量，我使用了兄弟节点加子节点的策略，然后考虑到对于复杂的源程序，语法树的规模可能会非常大，我划分了不同的语法树节点，然后尽可能地复用一些变量，对于不会同时使用的变量我用一个 union 进行定义，尽量降低空间复杂度。

我先把文法中跟语法分析相关的部分提取出来，经过反复观察之后，我开始设计我的语法树，划分我的语法树节点的类型，然后根据情况，思考这个语法树节点需要什么样的内容说明，比如对于一个常数节点，必要的内容是常数的类型包括 int 和 char，节点的类型为常数类型，常量的值，所在的行号等信息。分析完所有的节点之后，将其综合为一个抽象语法树节点的结构体。

接下来是利用递归下降的方法进行语法分析，这里的主要问题是存在无法判别的情况，比如读到的单词为 INT 无法判断是变量定义还是函数定义，所以我在这里使用了预读的方法，但也正是因为这个预读的方法让我的程序逻辑非常复杂，因为预读导致递归下降进入某些函数时当前符号不确定，不能统一处理，虽然最后用比较复杂的逻辑解决了这个问题，但是也导致了代码的冗余，这是我以后需要注意的地方。

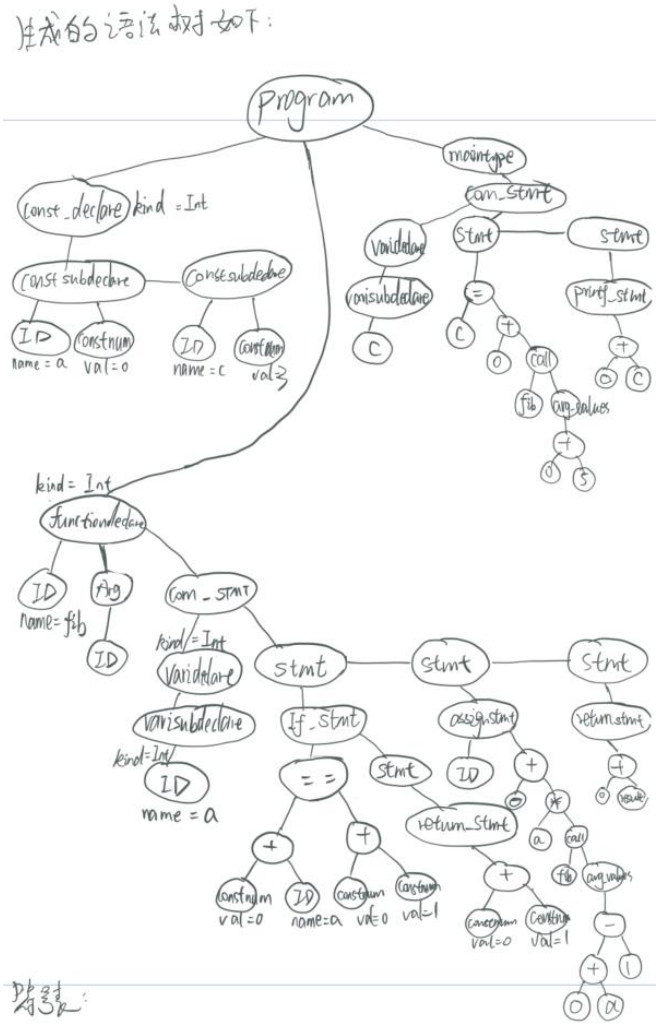
针对如下文法处理的时候，我用到了一个 trick

<表达式> ::= [+ | -] <项>{<加法运算符> <项>}

因为有可能是+|-号开头，为了程序统一处理，对于有+|-开头的表达式，我添加一个常数节点 0+exp | 0-exp，而对于没有以+|-开头的表达式，则添加一个常数节点 0+exp。这样会便于程序统一处理，但是对我后续工作带来了不便，因为如果对这种为 0 的常数节点进行翻译，会增加中间代码的复杂性降低速度，所以后面针对这种节点，我做了 AST 的剪枝操作，保证其不会生成无用的四元式。

代码完成后，我针对一个斐波那契的程序，画出了其抽象语法树，代码和对应的手稿如下：

```
const int a = 0 , c = 3;

int fib(int a)
{
    int result;
    if(a == 1)
        return (1);
    result = a * fib(a - 1);
    return result;
}

void main()
{
    int c;
    c = fib(5);
    printf(c);
}
```



3.3 符号表

3.3.1 函数与接口功能说明

函数原型	功能
entry* entryconstructor(entrytype);	构造一个符号表的条目
symboltable* symboltableconstructor(char*);	构造一个符号表
void insert_const_char(char* in_name , char in_val);	往符号表中插入一个常量字符条目
void insert_const_int(char* in_name ,int in_val);	往符号表中插入一个常量常数条目
void insert_func(char* in_name,Expkind in_kind,int in_paranum);	往符号表中插入一个函数条目
void insert_array(char* in_name,Expkind in_kind,int in_size);	往符号表中插入一个数组条目
void insert_arg(char* in_name,Expkind in_kind);	往符号表中插入一个参数条目
int hash(char*);	根据 id 执行 hash 操作
entry* find_table(char* in_name);	根据 id 在符号表中查找对应的条目

void build_table(ASTNODE* tree);	遍历语法树，得到符号表
void printtable();	输出符号表
void insert_vari(char* in_name,Expkind in_kind);	往符号表中插入一个变量条目

3.3.2 关键变量和数据结构设计

符号表条目的类型：

```
typedef enum{
TConst , TVariable , TArray , TFunc , TArg
}entrytype;//符号表条目的类型
```

符号表条目数据结构：

```
typedef struct entry//符号表的条目
{
    entrytype type;//条目的类型
    char* name;//identifier的名字，a()的a | int a 的a | const int a 的a | arg
    int addr;//条目的相对偏移地址
    Expkind kind;//identifier的类型，int a 的int | const int a 的int | char a()的char | arg的类型
    union
    {
        char ch_val;
        int int_val;
    }val;//对于常量来说，保存常量的值
    int paranum ;//对于function来说保存参数的个数，以便类型检测
    int array_size;//对于数组来说，保存数组的大小，防止越界
    int used;//用来表示定义过后是否使用了
    struct entry* next;//对于哈希表来说，指向下一个节点
}entry;
```

符号表的数据结构：

```
typedef struct symboltable
{
    entry* bucket[MaxBukSize];//hash table
    char* name;//对于全局的符号表为global ，函数的局部符号表为function的名字，
    int totaladdr;//总的地址

    struct symboltable* funchild;//对于global来说指向第一个function
    struct symboltable* mainchild;//对于global来说指向main
    struct symboltable* sibling;//对于funtion来说指向第二个function
    struct symboltable* parent;//指向父节点
}symboltable;
```

Hash 桶的大小：

```
#define MaxBukSize 211
```

全局符号表指针和当前符号表指针：

```
symboltable* top ;//指向global符号表的指针
symboltable* cur;//当前处在的符号表
```

3.3.3 设计思路

符号表这个数据结构由于在之后的生成汇编代码起到很关键的作用，于是在这里我设计了很久，力求设计一个便于实现也使用方便的结构，最后我是参考了龙书的一个符号表的例子和 louden 的 tiny 语言的符号

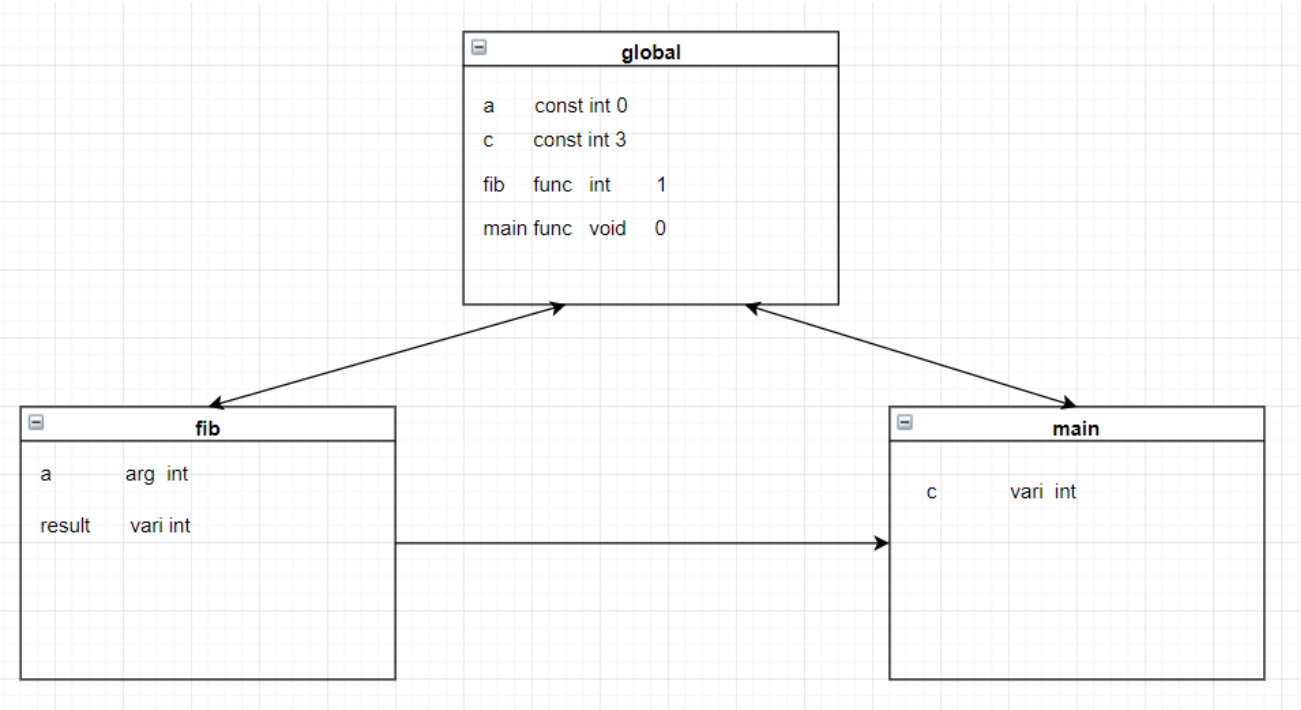
表的结构。

一开始设计的时候由于粗心大意，步入了一个误区，我一开始以为 C0 文法可以支持在语句块内进行变量常量定义，类似于 C++，想要实现这样的需求就非常难，要考虑到符号表的嵌套和同层符号表关系等一些问题，我先初步设计了一下符号表结构，但是非常复杂，我就比较怀疑其正确性，于是我去参考了国产开源 UCC 编译器的源码，发现他的符号表也非常复杂，结构和我之前设计的很类似，给了我很大的信心。但是后来发现 C0 文法并不支持块内常量变量定义，定义只出现在全局和函数的开始，于是我对我之前的设计进行了简化，为每一个函数分配一个符号表，它们属于同一层，由一个全局符号表统领，第一个函数的符号表作为全局符号表的子节点，所有函数的符号表用兄弟节点进行连接，语法分析和符号表我都用到了兄弟节点，好处就是完全不限制大小，可以有任意多张符号表。

符号表作为一个代码生成时经常被访问的数据结构，加快其查找效率能有效提高编译器效率，所以我选择将其设计为一个 hash 表结构，解决冲突的方法是链地址法，hash 函数是根据查找的 id 名字依次取出各个字符左移相加再取余，hash 桶的大小我选择的是 211，因为它是大于 200 的最小素数，选择素数作为 hash 桶的大小可以有效地减少 hash 冲突。

具体的做法是：先创建全局 global 符号表，遍历语法树，每找到一个函数定义就为这个函数创建一个符号表，设置好该符号表与其他符号表的关系，继续遍历语法树，对于定义节点，根据当前符号表指针插入到相应的符号表中，语法树遍历完成，符号表也建立好。

仍然以斐波那契函数为例，其符号表结构大致如下：



```
Global table:
EntryType       name            addr    kind    value   paranum         arrarysize
    TFunc       fib             2       1       -----   1               -------
   TConst       a               0       1       0       -------         -------
   TConst       c               1       1       3       -------         -------
    TFunc       main            3       0       -----   0               -------

fib:
EntryType       name            addr    kind    value   paranum         arrarysize
TVariable       $t0             2       1       -----   -------         -------
TVariable       $t1             3       1       -----   -------         -------
TVariable       $t2             4       1       -----   -------         -------
TVariable       $t3             5       1       -----   -------         -------
TVariable       result          1       1       -----   -------         -------
     TArg       a               0       1       -----   -------         -------

main:
EntryType       name            addr    kind    value   paranum         arrarysize
TVariable       $t4             1       1       -----   -------         -------
TVariable       c               0       1       -----   -------         -------
```

Global 符号表中存放的是全局定义的常量和变量以及程序中所有函数的信息，其余的函数包括 main 函数每一个都有一张独立的符号表，包含了自己的参数和局部变量和常量的信息，下面这张图是后面中间代码生成后的符号表，由于中间代码生成的时候会有临时变量的辅助，我也把这些临时变量放入到了符号表中。

另外需要指出的是我这个符号表没有设计释放函数，也就是说通过 AST 遍历得到的符号表是一直存在着的，不会释放掉，变化的只是符号表指针，即后面四元式生成和代码生成甚至是语义检查，需要用到符号表的时候只是在进入函数时设置好其指针即可。

3.4 语义检查

3.4.1 函数与接口功能说明

函数原型	函数功能
void typecheck(ASTNODE * tree);	基于 AST 检查程序的语义

3.4.2 关键变量和数据结构设计

这个部分没有辅助数据结构

3.4.3 设计思路

这个部分我设计得比较简单，先规划了一下需求，需要检查的内容有：算数运算符和比较运算符两边的表达式的类型是否一致，函数调用时实参数量和定义时的形参数量是否一致，变量是否未定义就使用，数组的非法访问，数组越界，对常量的赋值，scanf 函数参数中是否有常量，数组下标是否为整数，赋值语句两边的类型是否一致。

这里没有考虑类型转化，即如果发生类型错误就会报错退出。

语义检查是建立在语法树的前序遍历和后序遍历的基础上的，建立符号表的时候主要使用的是语法树的定义部分，而语义检查的时候主要使用的是语法树除了定义部分的其他部分。在前序遍历的基础上，对于特定类型的节点做特定的处理，比如遍历到函数定义的节点时，设置好对应的符号表指针，方便查找遍历，遍历到赋值语句的时候，后序遍历两个子节点，然后比较两个子节点的类型是否一致，一致则返回，遍历到

函数调用节点时，遍历其实参节点计算个数，遍历到数组节点时，如果其下标是一个常数就检查符号表看是否超过其定义的大小，如果遍历到运算节点或者比较节点则后序遍历子节点，然后比较子节点的类型是否一致，一致则返回等等类似的思路。

这一部分内容和上一部分内容就体现出来 AST 这种中间表示的意义了，因为如果不构建 AST，就只能在递归下降的过程中构建符号表，类型检查和生成四元式，也就是把语法分析这个模块的功能复杂化了，构建 AST 后就可以把这些部分拆解出来变成独立的模块，这样做也比较合理。

3.5 生成四元式

3.5.1 函数与接口功能说明

函数原型	函数功能
midcode* midcodeconstructor(fourth_type input);	创建一个四元式
char* getLable();	得到一个新的可以使用的标号
char* getTemp();	得到一个新的可以使用的临时变量
void emit_midcode(fourth_type in_type,char* in_op1,char* in_op2,char* in_op3);	根据四元式类型和 3 个操作数创建一个四元式
void gen_code(ASTNODE* tree);	根据 AST 得到四元式
void output_midcode();	将得到的四元式从内存输出到屏幕上

3.5.2 关键变量和数据结构设计

四元式的类型如下：

```
typedef enum{
    PARA , FUNC_BEGIN , FUNCC_END , RET , LABLE , JMP , JNP , JEP , VOID_FUNC_CALL ,NONVOID_FUNC_CALL,
    F_PLUS , F_MINUS , F_MULTIPY , F_DIVD , ARRA_ASSIGN_R , ARRA_ASSIGN_L ,
    F_BT , F_BAET , F_LT , F_LAET , F_EQ , F_NEQ , F_ASSIGN , F_PRINTF , F_SCANF,
    NEG
}fourth_type;
```

一条四元式的数据结构：

```
typedef struct
{
    fourth_type type;//四元式的类型
    char op1[MaxTokenLength];
    char op2[MaxTokenLength];
    char op3[MAXTEMPLENGTH];
}midcode;
```

这里我已经把代码从 C 搬运到 C++上了，因为本来按我的想法，是用链表把四元式连起来的，但是考虑到后面优化对四元式序列有很多删减操作，如果用链表则要自己做一些操作函数，而且后面的内容需要用到很多复杂的数据结构，如果全部从零开始设计会很复杂，为了不让程序过于复杂，我还是决定用 C++，有了 STL 就方便了很多，四元式序列结构如下：

```
extern std::vector<midcode*> midcode_list;//中间代码表
```

四元式语句即含义如下：

Type	Op1	Op2	Op3	Example	含义
Para	Src1			Para a	定义一个实参 src1 入栈
Func_begin	Src1			Func_begin fib	标识一个函数 src1 的开始
Func_end	Src1			Func_end fib	标识一个函数 src2 的结束
Ret	(Src1)			Ret a | Ret	函数的返回或者带返回值的函数返回
Lable	Src1			Lable L1	标志一个标识符号 src1
JMP	Src1			JMP L1	无条件跳转到一个标识 src1 处
JNP	Src1		Condition	JNP L1 , ,a	根据条件 condition，如果条件为假则跳转到 src1
JEP	Src1		Condition	JEP L1 , ,a	根据条件 condition，如果条件为假则跳转 src1
CALL	Src1		Src3	Call F, ,a	调用函数 src1，并把结果赋给 Src3
Void_CALL	Src1			Void_CALL F	调用函数 src1
+|-|*|/	Src1	Src2	Result	+ , a , b , $t1	将 src1 和 src2 做算数运算后结果赋值给 result
<|<=|>|>=|==|!=	Src1	Src2	Result	< , a , b , $t1	将 src1 和 src2 做关系运算后结果赋值给 result，以供跳转四元式作为参考
[]=R	Src1	Index	Result	$t1 = a[i]	将数组元素 src1[index]的值赋给 result
=	Result	Src1		$t1 = a	将 src1 赋值给 result
[]=L	Src1	Index	Src2	a[i] = $t1	将 src2 的值赋给数组元素 src1[index]
Printf	(src1)	(src2)	Type	Printf “hello C0” , a, int	将 src1 输出，根据 type 的类型输出 src2
Scanf	Src1			Scanf a	从标准输入中读一个内容到 src1 中
Neg	Result	Src1		Neg $t1 , a	将 src1 的值取负后赋值给 result

3.5.3 设计思路

生成四元式阶段是我遇到的第一个平台期，大概花了一周多才完成，不像前面几个阶段，有明确的实现思路，这个阶段一开始没有任何思路，而且我找了很多资料，很少有讲从 AST 到四元式转化的，顿时感觉生成 AST 是给自己挖了个坑。

一开始没有思路主要有三个方面的原因，第一个方面是我一开始觉得虚拟寄存器是有限的，临时变量不能无限多，所以生成临时变量的那个函数一直没有想法，第二个方面是我之前看的网课学理论看的是语法制导的翻译是基于 LR 的，当时看的时候就觉得拉链和回填很难，没有意识到用递归下降的方法不需要拉链和回填，实现起来都是递归的过程。最后一个方面是我不知道在递归的时候怎么样传递信息，比如 if 语句

翻译的时候，对于 then 部分和 else 部分的代码都需要标号的信息，算数运算左右操作数计算出来的结果去哪里取没有想清楚。

后来有一天我在 leetcode 写算法题用到栈的时候突然想到可以使用栈来保存计算的结果，比如算数运算节点翻译的时候，先递归地翻译左操作数再递归地翻译右操作数，然后他们的结果分别保存在了栈中，递归返回到算数运算节点时再取出来这两个操作数，这个想法转化为代码如下：

```
gen_code(tree->childen[0]);
gen_code(tree->childen[1]);

result2 = result_stack.top();
result_stack.pop();
result1 = result_stack.top();
result_stack.pop();
temp_vari = getTemp();
```

然后有了这个想法之后之前的三个问题就都解决了，仍然是利用 AST 的前序遍历，在特定的节点进行特定的操作，比如遇到函数定义节点时，先设置好符号表指针，然后生成一个函数开始，递归遍历函数主体节点，然后生成函数结束；遇到算数运算或者关系运算，先递归左子树再递归右子树然后从栈中取出结果的临时变量进行相应的操作后保存到结果中；如果遇到常量则直接把它的值 push 到栈中供上层使用；遇到 if 语句先生成条件部分，然后生成一个跳到 else 部分的标号，再生成一条条件为假调到 else 部分的语句，然后递归生成 then 部分，然后生成一个 if 语句的出口，然后在 then 部分结束后生成一条无条件挑战到出口，然后生成 else 部分的开始标志，再递归生成 else 部分，最后生成出口的标志，其他的结点都是类似的思路这里就不再赘述。

有了思路之后这一部分的代码还是比较好写的，很多都是递归实现的，自己想的方式奏效了，还是挺有成就感的。

3.6 代码优化

3.6.1 函数与接口功能说明

函数原型	函数功能
void merge_const_value();	常量的+|-|*|\直接计算后改为赋值
void delete_temp_value();	删除无用的临时变量
void const_replace();	对于 const 修饰的变量直接用其值进行替换
void delete_useless_jmp();	删除无用跳转
void merge_same_exp();	合并局部公共子表达式(包括代数恒等式)
void deadcode(ASTNODE*);	基于 AST 删除死代码
void weaken_exp();	代数削弱
void const_propagate();	块内常量传播
void optimizer();	优化封装函数，将四元式优化封装在一起

3.6.2 优化效果举例

3.6.2.1 四元式优化部分

代数削弱：

$t1 = a * 1	➔	$t1 = a
$t1 = a * 0	➔	$t1 = 0
$t1 = a * 2	➔	$t1 = a + a
$t1 = a + 0	➔	$t1 = a
$t1 = a - 0	➔	$t1 = a
$t1 = 0 / a	➔	$t1 = 0
$t1 = a / 1	➔	$t1 = a

常量传播：

情况 1:

```
a = 5
c = 1
b = a + c
```

优化后:

```
a = 5
c = 1
b = 5 + 1
```

情况 2:

```
a[1] = 5
b = 1 + a[1]
```

优化后:

```
a[1] = 5
b = 1 + 5
```

合并局部公共子表达式：

情况 1:

```
$t1 = a + c
$t2 = a + b
$t3 = a + c
```

优化后:

```
$t1 = a + c
$t2 = a + b
$t3 = $t1
```

情况 2(代数恒等式情况):

```
$t1 = c + a
$t2 = a + b
$t3 = a + c
```

优化后:

```
$t1 = c + a
$t2 = a + b
$t3 = $t1
```

合并常量：

```
$t1 = 5 + 7
```

优化后:

```
$t1 = 12
```

删除无用的临时变量:

情况 1:

```
$t1 = a + c
b = $t1
```

优化后:

```
b = a + c
```

情况 2:

```
$t1 = 5
b = $t1
```

优化后:

```
b = 5
```

情况 3:

```
$t1 = fib(5)
b = $t1
```

优化后:

```
b = fib(5)
```

(这些情况的出现是由从 AST 翻译到四元式的方法的特性决定的)

常量替换:

```
const int a = 5;
b = a;
```

优化后:

```
const int a = 5;
b = 5;
```

删除无用跳转:

```
jmp     Lable3
Lable   Lable2
Lable   Lable3
```

优化后:

```
Lable   Lable2
```

(这种情况出现是因为 if 语句没有 else 分支)

最后用 optimizer 综合所有的四元式优化方法，循环优化直到无法优化为止

```
void optimizer()
{
    int len = midcode_list.size();
    do
    {
        len = midcode_list.size();

        const_replace();//优化3 常量替换

        const_propagate();//优化6 变量的常量传播

        merge_const_value();//优化1 常量加减乘除改赋值

        delete_temp_value();//优化2 删除冗余局部变量

        delete_useless_jmp();//优化4 删除无用的跳转和标号

        merge_same_exp();//优化5 合并局部公共子表达式

        weaken_exp();//优化7 代数削弱

    }while(len != midcode_list.size());
}
```

3.6.2.2 AST 优化部分

基于 AST 的死代码删除

情况 1:

```
If(1)
{
   a = 1;
}
else{
   a = 2;
}
```

优化后:

```
a = 1;
```

情况 2:

```
If(0)
{
   a = 1;
}
else{
   a = 2;
}
```

优化后:

```
a =2;
```

情况 3:

```
const int a = 1;
If(a)
```

```
{
   a = 1;
}
else{
   a = 2;
}
```

优化后:

```
a = 1;
```

3.6.3 关键变量和数据结构设计

该部分无数据结构

3.6.4 设计思路

一开始我觉得这一部分很简单，想着赶紧写完，但是往往心急吃不了热豆腐，写了半天优化得不怎么样还有很多 bug，于是我就把龙书的优化部分看了一遍，然后看了一遍中科大的优化部分的网课，渐渐思路才明朗起来。

我这里的优化没有使用额外的类似于 CFG,DAG 这样的数据结构，主要的原因是我觉得 C0 文法比较简单，然后生成的四元式类型也不是很多，基本上可以从四元式的类型来判断出来是否在一个基本块内，可以通过遍历四元式的方式来达到优化效果，所以我觉得用 DAG 可能把这个问题复杂化了。由于我不打算用图着色和线性扫描的寄存器分配，所以我就没做数据流分析，我的寄存器分配策略是引用计数的策略，不需要数据流信息。这样的简化便于实现，缺点就是使得编译器的算法复杂度变高。

实现方式也是比较直观，我先进行的是需求分析，我输入测试用例，然后观察四元式的结构，看看哪些地方是冗余的，带来了优化的机会，将其全部总结出来得到需求。这些需求如消除冗余跳转，消除无用局部变量等等都是利用的窥孔优化的思想，扫描四元式表，符合优化条件的四元式进行修剪。对于基本块的划分我的想法是遍历四元式只要不遇到 lable，跳转，return，函数结束，函数调用，其他的四元式都是一个块内的四元式，就可以通过不断遍历的方式进行块内的各种优化，对于块内的常量传播，只需要检查变量的定义和使用之间的四元式序列，如果它们没有对这个变量进行重新定义即可进行常量传播，局部公共子表达式也是类似的思想。

为了让我费力生成的 AST 发挥一些作用，我考虑了在它上面进行修剪，前面提到过我为了让代码更加统一，导致 AST 上有一些无用节点，在转化成四元式的时候我将其跳过了。还有一个对于 if 语句，如果其条件是永真我就删掉假分支，如果条件是永假我就删掉真分支，这样可以减少四元式和目标代码的数量。

3.7 代码生成

3.7.1 函数与接口功能说明

函数原型	函数功能
void lw(char* reg_name,char* vari_name);	将一个变量放入某一个寄存器中
void sw(char* reg_name,char* vari_name);	将某一个寄存器中的内容存回某一个变量
int find_addr(char* vari_name);	找到某一个变量相对于 fp 的偏移地址

int gettotalsize(char* func_name);	计算栈空间的大小，便于 sp 减去大小
void gen_data();	产生 data 段的描述
void gen_text();	产生 text 段的描述
bool is_const_value(char* input);	判断输入是否是一个常量
void gen_mips();	生成 mips 汇编
void init();	将变量的引用计数信息进行初始化
void cntpp(char*);	对变量的引用计数得分加一
int find_refn(char*);	辅助 cntpp 找改变量是否在寄存器引用计数描述符中
int find_in_refn(char*);	在寄存器引用计数描述符中找到其位置
bool cmp(std::pair<char*,int> num1 , std::pair<char*,int> num2);	对寄存器引用计数描述符进行排序时用到的排序函数
void save_reg();	保存 s0 到 s7
void return_reg();	还原 s0 到 s7

3.7.2 关键变量和数据结构设计

辅助处理 printf 字符串的变量如下：
产生 data 段时，用这个变量依次给字符串命名，使用时也利用这个变量来定位

```
static int num_str = 1;//用来表明当前是处理到了哪一个字符串，(字符串仅在printf中出现)
```

用于实现引用计数的寄存器分配的数据结构：
这里用到的是 STL 的 vector 和 pair 的组合，pair 的第一个值表示变量的名字，第二个值表示该变量在这个函数里面被引用的次数

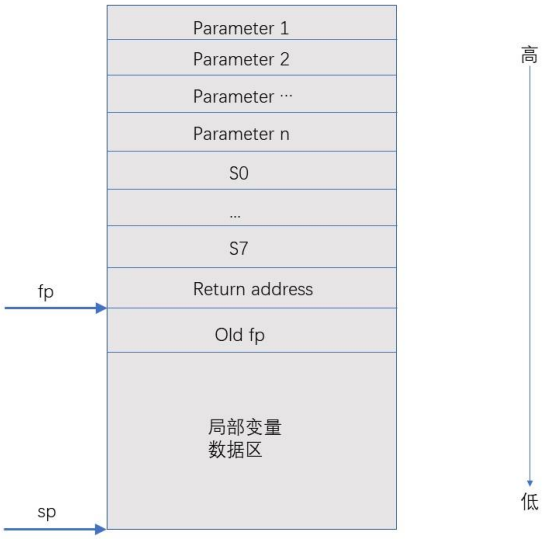
```
std::vector<std::pair<char*,int> > refn;//引用计数记录
```

3.7.3 设计思路

3.7.3.1 运行栈结构的设计

设计这一部分的时候首先要解决的问题就是运行栈结构的设计，我选择生成 MIPS 汇编，在 Mars 上仿真，但是我对 MIPS 不是很熟悉，之前学 x86 汇编的时候觉得其运行栈结构很巧妙，就仿照其来设计。

我用 fp 来作为帧指针，用 sp 来作为栈指针，使用临时寄存器 t0-t7 保存函数内的临时变量，使用 s0-s7 保存引用计数得分高的 8 个变量，每次函数调用时由调用方保存和还原 s0-s7，我在设计的时候没有严格区分栈的边界，我的目标就是能够方便地定位我想要找的变量即可，运行栈结构示意图如下：



与运行栈相关的四元式翻译操作如下:

四元式	功能	翻译操作
Para	参数入栈	sp = sp-4 参数->sp 所指的栈内位置
Void_call	无返回值函数调用	保存 s0-s7 sp = sp – 4 jal 函数标号(返回地址在 ra 中) 还原 s0-s7 sp 加上参数对应栈空间
Call	带返回值的函数调用	保存 s0-s7 sp = sp – 4 jal 函数标号(返回地址在 ra 中) 还原 s0-s7 sp 加上参数对应栈空间 返回值 v1 保存到变量中
Func_begin	函数开始	设置好符号表指针指向本函数 计算本函数内的所有变量的引用次数，将其排序 产生函数开始标号 ra->sp 所指的栈内位置 fp->sp-4 所指的栈内位置(旧 fp 入栈) fp = sp sp = sp – (旧 fp 和局部数据区占用的空间大小) 查找符号表，把局部常量依次放入栈内对应位置 将引用次数多的参数和全局变量依次放入对应的寄存器
Ret	函数返回	若有返回值先将返回值放入 v1 sp = fp 还原返回地址给 ra sp = sp – 4 还原旧的 fp

		jr ra 返回
Func_end	函数结束	sp = fp 还原返回地址给 ra sp = sp – 4 还原旧的 fp jr ra 返回

3.7.3.2 寄存器分配策略

分配策略我用的是引用计数的寄存器分配策略，即在函数的开始先为函数内的每一个变量分配一个初始分值 0，然后扫描改函数，变量出现一次就将其得分加一，最后函数扫描结束，得分最高的 8 个变量说明是在该函数中最经常使用的变量，将其分别分配给 s0-s7.之后每次需要使用变量时，先在寄存器描述符中找，该变量是否已经在 s0-s7 中了，是的话就直接使用，如果没有就使用临时寄存器 t0-t7，如果是对其的赋值，还需要更新到内存中，寄存器描述符表举例如下：

对应寄存器	变量名	得分
S0	Temp	9
S1	a	6
S2	b	6
S3	test	5
S4	children2	4
S5	tree	4
S6	children1	3
S7	g_a	2

在函数开始的时候检查这几个描述符，只有变量为参数和全局变量会被 load 到寄存器中，之后翻译时任何时候如果遇到变量就寻找是否在这 8 个寄存器中，如果有就直接拿来用，可以大大减少访存操作。

3.7.3.3 指令选择和翻译策略

指令选择为了方便我选取了 MIPS 汇编的一些伪指令，具体选择的指令如下：

指令	举例	功能
li	li $t1,100 || li $t1,'c'	往寄存器里存一个立即数
lw	lw $t1,0($fp)	从内存中往寄存器里存值
sw	sw $t1,0($fp)	从寄存器往内存中存值
la	la $t1,msg1 || la $t1,12($fp)	取地址
subi	subi $t1,$t1,10	减立即数
addi	addi $t1,$t1,10	加立即数
jar	jar $ra	跳转并链接(返回地址存入$ra)
move	move $t1,$t2	寄存器间的赋值
beq	beq $t1,$t2,lable1	相等跳转
j	j lable1	无条件跳转
bne	bne $t1,$t2,lable1	不等跳转
bge	bge $t1,$t2,lable1	大于等于跳转

bgt	bgt $t1,$t2,lable1	大于跳转
ble	ble $t1,$t2,lable1	小于等于跳转
blt	blt $t1,$t2,lable1	小于跳转
add	add $t1,$t1,$t2	加
sub	sub $t1,$t1,$t2	减
mul	mul $t1,$t1,$t2	乘
div	div $t1,$t1,$t2	除
mflo	mflo $t1	取$lo 赋值到寄存器
sll	sll $t1,$t2,$t3	逻辑左移
syscall	syscall	系统调用

其余的四元式翻译策略如下:

四元式	功能	翻译操作
LABLE	产生一个标号	产生一个标号
JMP	无条件跳转	生成无条件跳转语句
JNP	条件为假跳转 (只在 if 语句条件只有一个变量时出现)	如果条件为一个常量，判断是否为 0 如果为 0 生成无条件跳转 如果条件为一个变量，生成 beq 指令与$0 比较，等于 0 跳转
JEP	条件为真跳转 (只在 while 语句条件只有一个变量时出现)	如果条件为一个常量，如果不为 0 生成无条件跳转 如果条件为一个变量，生成 bnq 指令与$0 比较，不等于 0 跳转
<|<=|>|>=|==|!=	进行比较运算	根据紧挨着的后面一条跳转四元式生成对应的跳转语句
+|-|*|/	进行算数运算	直接翻译为对应的算数运算 除法运算要多加一条 mflo 指令
=	赋值语句	把值赋给对应的变量，如果该变量在 s 寄存器中，直接 move 到这个寄存器中，否则 sw 到对应的内存位置
[]=R	数组元素赋值	先找到数组的首地址 再根据偏移计算数组元素的地址 把该元素 lw 到寄存器中 进行赋值
[]=L	数组元素被赋值	先找到数组的首地址 再根据偏移计算数组元素的地址 将值赋给这个内存单元
Printf	输出语句	如果有字符串，使用 v0=4 的系统调用 如果有变量，根据其类型选择打印数字还是字符
Scanf	输入语句	根据变量的类型，选择对应的系统调用输入字符或者数字
Neg	取负运算	做先把等号右边的操作数取出来，然后用 0 减掉它赋值给左边的操作数

用到的系统调用如下:

v0 的值	功能
1	打印一个数字
4	打印一个字符串
5	读入一个数字
11	打印一个字符
12	读入一个字符

3.8 错误处理

3.8.1 函数与接口功能说明

函数原型	功能
void error_msg(char* , int);	打印错误字符串，根据错误类型进行跳读

3.8.2 关键变量和数据结构设计

跳读方式定义如下：

```
#define skip_to_void_int_char 0
#define skip_to_void 1
#define skip_to_endfile 2
#define skip_to_RBB 3
#define no_skip 4
#define skip_to_next 5
#define skip_to_RB_next 6
#define skip_to_SEMI_next 7
#define skip_to_SEMI 8
#define skip_to_RBB_next 9
#define skip_to_RB 10
```

3.8.3 设计思路

我在这里主要处理的是语法分析时出现的错误，根据其处在语法分析的阶段，进行对应的跳读，比如如果在处理常量时遇到错误，则跳到 void，int，char 三个符号之一，如果在处理 main 函数时遇到错误则直接调到文件结束，如果在处理复合语句时出现错误，则跳到}，如果在处理某一个单一语句出现错误，则跳到；，其余都是类似的思想。

四、版本结构

所有的代码文件如下：

名称	修改日期	类型	大小
scan	2020/4/12 23:52	文件夹	
V1.0.0	2020/4/18 18:20	文件夹	
V1.0.1	2020/5/19 11:07	文件夹	
V2.0.0	2020/5/19 11:07	文件夹	
V2.0.1	2020/5/19 11:08	文件夹	
V3.0.0	2020/5/19 11:06	文件夹	
V4.0.1(废除)	2020/5/10 10:33	文件夹	
V4.1.1	2020/5/19 11:04	文件夹	
V4.2.1	2020/5/21 10:30	文件夹	
V4.2.2	2020/5/24 18:24	文件夹	
V4.3.2	2020/5/25 22:06	文件夹	
词法分析	2020/4/6 21:53	文件夹	
递归下降计算器	2020/4/2 12:38	文件夹	

软件版本号设定为 A.B.C

A 为第一级，表示重大重构

B 为第二级，表示重大功能改进

C 为第三级，表示小的功能添加和 bug 的修复

每一个版本号文件夹内的 README.md 文件描述了本版本的内容和相较于上一版本的改进。

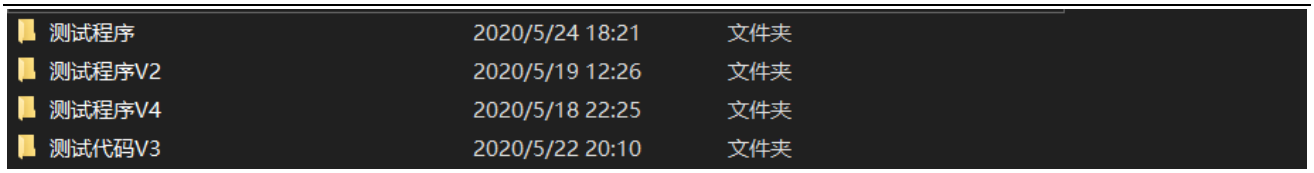
各个版本的大致内容如下：

版本号	描述
1.0.0	完成了词法分析和语法分析
1.0.1	完成了词法分析语法分析语义检查和符号表生成
2.0.0	将原本的 C 语言改为 C++语言，内容与上一版本相同
2.0.1	在原来的版本基础上增加了生成中间代码
3.0.0	完成不带优化的版本，在前一版本基础上增加 MIPS 汇编代码生成
4.0.1	尝试进行寄存器分配，写完发现逻辑错误，需要修改的地方过多，废弃不用
4.1.1	在 3.0.0 基础上增加四种四元式优化手段
4.2.1	在前一版本基础上增加一种四元式优化，完成寄存器分配
4.2.2	在上一版基础上再增加四元式优化，完成所有的优化策略
4.3.2	最终版本，在上一版本基础上完善错误处理

另外几个文件是我开始工作之前的一些练习项目，有 louden 书上的递归下降计算器等。

五、测试

测试代码文件如下：



我主要选用的测试程序，测试程序 V2，测试程序 V3 中的代码，一共 56 份测试代码，其中 46 份正确的代码，10 份错误的代码.对于正确的代码可以正确输出结果，对于错误的代码可以指出错误位置与原因。

具体可见测试报告文件

下面举列正确和错误的代码各一份：

正确代码：

```
void main()
{
	const int max = 1000;
	int i,j;
	i = 1;
	do
	{
		j = 1;
		do
		{
			if(j*j==i) printf(" ",i);
			j = j + 1;
		 }while(j<=i)
	  i = i + 1;
	}while(i<=max)
}
```

优化后的 MIPS 汇编如下：

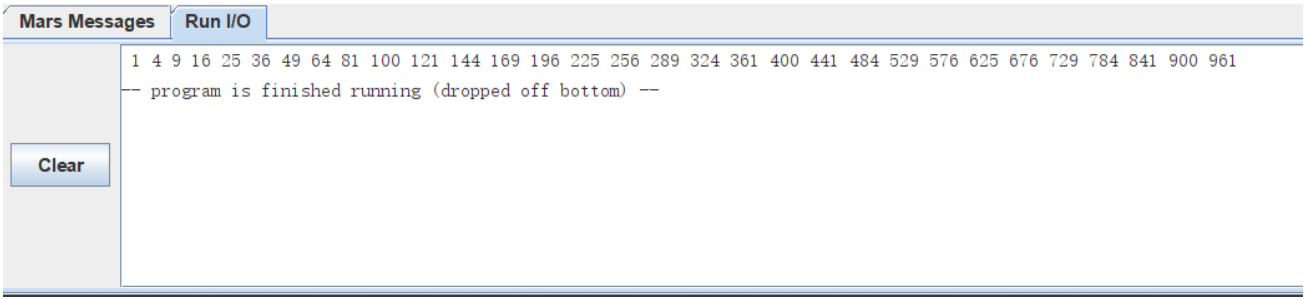
```
.data
	msg1:	.asciiz " "
.text
	j	main
main:
	sw	$ra,0($sp)
	sw	$fp,-4($sp)
	add	$fp,$sp,$0
	subi	$sp,$sp,40
	li	$t2,1
	move	$s0,$t2
Lable0:
	li	$t2,1
	move	$s1,$t2
Lable1:
	mul	$s2,$s1,$s1
	bne	$s2,$s0,Lable2
	li	$v0,4
```

```
    la   $a0,msg1
    syscall
    move    $a0,$s0
    li   $v0,1
    syscall
Lable2:
    li   $t2,1
    add $s1,$s1,$t2
    ble  $s1,$s0,Lable1
    li   $t2,1
    add $s0,$s0,$t2
    li   $t2,1000
    ble  $s0,$t2,Lable0
```

中间产物如下:

```
E:\编译原理程序记录\V4.3.2\bin\Debug\V4.3.exe
Please input the absolute path of your source file:E:\编译原理程序记录\V4.3.2\测试程序V2\test5.txt

Would you like to optimize the final code?[default:Y](Y/n):Y

Semantic Analyze Complete...

Global table:
EntryType        name            addr    kind    value   paranum         arrarysize
    TFunc        main            0       0       -----   0               -------

main:
EntryType        name            addr    kind    value   paranum         arrarysize
TVariable        j               2       1       -----   -------         -------
TVariable        $t0             3       1       -----   -------         -------
TVariable        $t1             4       1       -----   -------         -------
TVariable        $t2             5       1       -----   -------         -------
TVariable        $t3             6       1       -----   -------         -------
TVariable        $t4             7       1       -----   -------         -------
TVariable        $t5             8       1       -----   -------         -------
   TConst        max             0       1       1000    -------         -------
TVariable        i               1       1       -----   -------         -------

midcode :
Func_begin       main
=                i       1
Lable            Lable0
=                j       1
Lable            Lable1
*                j       j       $t0
==               $t0     i       $t1
jnp              Lable2          $t1
printf                   i       int
Lable            Lable2
+                j       1       j
<=               j       i       $t3
jep              Lable1          $t3
+                i       1       i
<=               i       1000    $t5
jep              Lable0          $t5
Func_end         main

mips code already generated...
```

运行结果如下:



优化测试效果如下:

方式	运行时间
没有寄存器分配，没有优化	4.51s
寄存器分配，没有优化	2.01s
寄存器分配，优化四元式	1.76s

错误代码如下：

```
const    a = 0 , c = 3 ;
int b , c , d;

int fib(int a)
{
        int result;
        if(a == 1)
        {
                return (1);
        }
        result = a * fib(a - 1);
        return (result);
}

void main()
{
        const int t = 1;
        const int kk = -1;
        char b ;
        int c ;
        int d ;
        c = fib(void);
        d = c * 1 ;
        c = c / a;
        c = int;
        if(t)
      {
        c    = c * 2;
             if(t>3)
```

```
    {
    c = 5 - 1 ;
    }
        else
    {
    b = 'A' + 'A';
    }
   }
    printf('c');
}
```

结果如下：

```
E:\编译原理程序记录\V4.3.2\bin\Debug\V4.3.exe
Please input the absolute path of your source file:E:\编译原理程序记录\V4.3.2\测试程序\1.txt

Would you like to optimize the final code?[default:Y](Y/n):
Error at line 1 :
 a is not expected , expect token as int , char

Error at line 24 :
 void is not expected , expected token as + , - , id , integer , charactor , ( , )

Error at line 27 :
 int is not expected , expected token as integer , charactor , id , (


Total error:3

Process returned 0 (0x0)   execution time : 3.609 s
Press any key to continue.
```

六、总结与感想

通过本次编译器的编写，我的各个方面的能力都有了很大的提升，包括编码和测试的能力，自学能力，工程思维能力等等，通过亲身实现从高级语言到汇编语言的转换，让我受益匪浅，对理论知识和计算机系统的认识又有了很大程度的提高，这些都是理论学习达不到的效果。通过自己思考设计最终完成自己的想法，很有成就感，也体会到了设计师的快乐。同时这也是我大学期间第一个一个人完成的大型项目，以前学汇编的时候觉得写编译器的人很厉害，编译器居然可以编译出这么巧妙的汇编代码，现在完成这个编译器项目后觉得编译器也没有那么神秘了。然而我所实现的文法虽然是扩充的 C0 文法，但是也只是实现了 C 语言的很小一部分的功能，让我体会到 GCC 编译器的强大，这个领域还有很多我需要学习的地方。