C Compiler

作者：罗跃骜

目录

1.C-语言的语法图描述

- 关键字：

else if int return void while所有的关键字都是保留字。

- 专用符号：

+ - * / < <= > >= == != = ; , ( ) [ ] { } /* */

- 其它标记：

ID = letter letter*

NUM = digit digit*

Letter = a|..|z|A|..|Z

Digit = 0|..|9

- 空格有空白、换行符和制表符组成。空格通常被忽略，除了它必须分开ID、NUM关键字
- 注释用通常的C语言符号/.../围起来。注释可放在任何空白出现的位置(即注释不能放在标记内)上，且可以超过一行，注释不能嵌套。
-

以下为C-语法的BNF：

```
1.  program -> declaration-list
2.  declaration-list -> declaration-list declaration | declaration
3.  declaration -> var-declaration | fun-declaration
4.  var-declaration -> type-specifier ID ; | type-specifier ID [ NUM ] ;
5.  type-specifier -> int | void
6.  fun-declaration -> type-specifier ID ( params ) | compound-stmt
7.  params -> params-list | void
8.  params-list -> params-list , param | param
9.  param -> type-specifier ID | type-specifier ID [ ]
10. compound-stmt -> { local-declarations statement-list }
11. local-declarations -> local-declarations var-declaration | empty
12. statement-list -> statement-list statement | empty
13. statement -> expression-stmt | compound-stmt | selection-stmt |
iteration-stmt | return-stmt
14. expression-stmt -> expression ; | ;
15. selection-stmt -> if ( expression ) statement | if ( expression )
statement else statement
16. iteration-stmt -> while ( expression ) statement
17. return-stmt -> return ; | return expression ;
18. expression -> var = expression | simple-expression
19. var -> ID | ID [ expression ]
20. simple-expression -> additive-expression relop additive-expression |
additive-expression
21. relop -> <= | < | > | >= | == | !=
22. additive-expression -> additive-expression addop term | term
23. addop -> + | -
24. term -> term mulop factor | factor
25. mulop -> * | /
26. factor -> ( expression ) | var | call | NUM
27. call -> ID ( args )
28. args -> arg-list | empty
29. arg-list -> arg-list , expression | expression
30
```

因为LL（1）文法无法处理S->AB,A->a...,B->a...的问题，所以对原文法进行了一些修改：

```
1.  S -> program
2.  program -> declaration-list
3.  declaration-list -> declaration-list declaration | declaration
4.  declaration -> var-declaration | fun-declaration
5.  var-declaration -> type-specifier ID ; | type-specifier ID [ NUM ] ;
6.  type-specifier -> int | void
7.  fun-declaration -> type-specifier IDF ( params ) | compound-stmt
8.  params -> params-list | void
9.  params-list -> params-list , param | param
10. param -> type-specifier ID | type-specifier ID [ ]
11. compound-stmt -> { local-declarations statement-list }
12. local-declarations -> local-declarations var-declaration | empty
13. statement-list -> statement-list statement | empty
14. statement -> expression-stmt | compound-stmt | selection-stmt |
iteration-stmt | return-stmt
15. expression-stmt -> expression ; | ;
```

```
16. selection-stmt -> if ( expression ) statement | if ( expression ) statement else statement
17. iteration-stmt -> while ( expression ) statement
18. return-stmt -> return ; | return expression ;
19. expression -> var1 = expression | simple-expression
20. var -> ID | ID [ expression ]
21. var1 -> ID1 | ID1 [ expression ]
22. simple-expression -> additive-expression relop additive-expression | additive-expression
23. relop -> <= | < | > | >= | == | !=
24. additive-expression -> additive-expression addop term | term
25. addop -> + | -
26. term -> term mulop factor | factor
27. mulop -> * | /
28. factor -> ( expression ) | var | call | NUM
29. call -> IDF ( args )
30. args -> arg-list | empty
31. arg-list -> arg-list , expression | expression
32
```

2.系统设计

2.1系统的总体结构

① 基于LL（1）文法的编译器（手写实现）；

② 基于LALR文法的编译器——使用Flex和Bison工具实现；

2.2主要功能模块的设计

①基于LL（1）文法的编译器（主要功能）

(1) 程序读入两个文件后，首先进行词法分析，在这之前，先预处理一下，对源程序去除注释，去除多余空格，随后对处理过后的源程序进行，得到一个token流，供语法分析时使用，生成栈以及语法分析树。

(2) 对语法规则消除左递归以及公共左因子，并且根据|拆分语法推导式，对处理后的语法规则，首先计算出所有的终结符以及非终结符，并求出非终结符的first,follow集，生成预测分析表，根据后序语法分析发现的问题后回过来对语法做的一些细微的修改也在这一步完成。

(3) 根据token流计算出符号表

(4) 根据token流以及预测分析表完成整个程序的出入栈以及语法树的建立。

(5) 使用深度遍历语法树遇到相应节点，即进行语义动作，产生三地址代码。

(6) 最后对产生额三地址代码进行优化。

②基于LALR文法的编译器

(1) 编写lex.l文件。该文件用于flex词法分析，文件内容包含C/C++语言声明与和定义正则表达式，匹配模式和事件。

(2) 编写Syntax.y文件，用于bison语法分析和语义分析，内容对文法的终结符和非终结符做一些相关声明，并段定义了文法的非终结符及产生式集合，以及当归约整个产生式时应执行的操作

(3) 输入命令g++ -o Syntax.tab.exe Syntax.tab.c lex.yy.c -w编译得到C-语法分析器Syntax.tab.exe，设计测试用例test.txt，输入命令Syntax.tab.exe < test.txt即可运行并输出相应文件。

(4) 使用Makefile以简化编译流程。

3.系统实现

3.1主要函数说明

基于LL（1）文法的编译器——词法分析

- 删除注释

void remove_comments(string infile, string outfile)

- 删除空白符

void remove_blank(string infile, string outfile)

词法分析

void lexical_analysis(string infile ,string outfile)
void sup_lexical_analysis(string infile, string outfile)

状态转换图：

种别码	单词符号	助记符
0	else	RESERVED
1	if	RESERVED
2	int	RESERVED
3	return	RESERVED
4	void	RESERVED
5	while	RESERVED
6	+	OOP
7	-	OOP
8	*	OOP
9	/	OOP
10	<	COP
11	<=	COP
12	>	COP
13	>=	COP
14	==	COP
15	!=	COP
16	=	AOP
17	;	EOP
18	,	SOP
19	(	SOP
20	)	SOP
21	[	SOP
22	]	SOP
23	{	SOP
24	}	SOP
25	letter|letter*	ID
26	digit|digit*	NUM

基于LL（1）文法的编译器——语法分析

初始化终结符表

void init_tstype(map<string, string>& tstype);

初始化终结符类型表使用string-string类型的map表，键为原始终结符符号，值为其类型。

初始化预测分析表

void init_patable(struct Pre_Ana_Table &patable);

初始化预测分析表使用专用定义的结构体，初始化时将预测分析表的第一列填充为所有非终结符，第一行填充为所有终结符，并且所有位置预设初始值0即不能推导。

消除左递归与公共左因子

void del_Ltre_Cfactor(string infi, string outfi);

函数用于消除语法的左递归以及公共左因子。

将一个推导记录左侧字符串为left，右侧为right。下一步将right根据|拆分为多个部分，该操作调用函数splitblank，返回一个数组其中存储了各部分的字符串值和一个int值表示有几个部分。下一步由于观察语法推导式可以发现所有推导式出现左递归的情况只会出现在右侧第一个推导，所以只判断第一部分是否出现了左递归，如果出现了则按照P->Pα|β P->βP'和P'->αP'|ε。随后判断公共左因子，也很容易发现存在公共左因子的情况也只会出现在右侧只有两个部分时，那么也可以简化算法，判断前两个是否出现了公共的左因子，也按照类似左递归进行消除。

计算终结符与非终结符

void cal_ns_ts(string infi, string outfi);

该函数计算出所有的非终结符以及终结符。

逐条读入推导式，对于左边部分，判断是否已经出现在非终结符表中，如果没有出现则加入，并判断是否出现在终结符表中，如果出现则从终结符表中删除，对于右边部分，判断是否出现在终结符表中，如果未出现则加入。

构建First集和Follow集

void cal_ns_ts ();

该函数计算所有非终结符的first集, follow集。

cal_first()计算所有非终结符的first集。First、follow集均采用string-vector存储

\1. 循环初始flag=1 逐条分析推导式。判断推导式第一个元素是否为终结符没如果为终结符则判断该终结符是否在当前非终结符的first集中，如果不在则加入并且flag=0；

\2. 如果第一个元素为非终结符，则将该非终结符的first集中的所有元素加入推导式首部的非终结符的first集中；

\3. 如果推导式前部分的非终结符的first集中有empty，则将他们所有的first集中的所有元素加入至首部的非终结符的first集中，且flag=0，直到遇到一个不能推导出empty的非终结符或是终结符。

cal_follow()计算所有非终结符的follow集。

思路同上

构建预测分析表

void cal_pre_ana_table(string outfi);

该函数计算出预测分析表。

函数遍历推导式表，将每一个非终结符作为预测分析表的行，再取出箭头右侧第一个符号，对于它的所有的first集，将该推导式编号填入first集内容所对应的列中，如果遇到empty，判断推导式左侧的非终结符是否能退出空，如果有则将那条推导式编号填入该非终结符follow集元素所对应的列中。

基于LL（1）文法的编译器——语义分析

构建符号表

```
// 初始化符号表
 void init_symbol_table()
 //给定函数名字，查询是否在函数表中，返回位置
 int func_exist_in_functable(struct func functable[255], int funcnum, string str)
 //在函数表中添加函数符号表
 struct symbol* add_func(struct func functable[255], int &funcnum, string str, string ret)
 // 给定变量名字，查询是否在该函数的符号表中，返回位置
 int var_exist(struct func functable[255], struct symbol *st, int n, string str)
 // 给定变量，添加到本函数的符号表中
 void add_var(struct func functable[255], struct symbol *st, int curnum, int num, string name,
string property, string type)
 // 给定函数名，添加到本函数的符号表中
 void add_funcname(struct func functable[255], struct symbol *st, int curnum, int num)
 // 生成符号表
 void create_symbol_table(string infile, string outfile)
```

初始化树函数

void init_tree()

初始化语法分析树

申请根节点，名称为root，类型为非终结符，随后为根节点构造三个子节点分别为#、S、#，其中S为定义的开始符号

建立语法分析树函数

void make_tree(fstream &in, fstream &out)

根据词法分析结果，自上而下利用符号栈来构建语法分析树，并为每个非终结符加入语义动作节点，名称为'$'+节点编号，类型为todo

局部变量

\1. token：从词法分析结果中读取的每个符号及其信息

\2. cur：树节点指针，指向当前正在操作的树节点，开始时指向

\3. str：表示待展开字符串

\4. num：当前token在预测分析表中的列数

\5. line：当前非终结符str，遇到token表示的终结符时应使用的推导式在ad_grammer.txt中的编号

实现步骤

\1. 初始化语法分析树（init_tree()），符号栈（#、S依次进栈）；

\2. 从文件中读取一个符号的信息存入token；

\3. 栈顶元素赋值给str；

\4. 若str为非终结符，则表明需要向下构建子树，进入下一步，否则跳过，到步骤11；

\5. 根据str找到num的值，再根据str和num找到line的值

\6. 找到line后表明该符号可以展开，将栈顶元素出栈，再将line所表示的推导式右部每个符号依次逆向入栈，使栈顶为推导式右部的第一个符号

\7. 为当前cur节点构造子节点并设置相关指针属性，子节点从左到右依次为该推导式右部的每个符号，同时将该子节点编号nodenum加入placelistmap内容暂为空

\8. 构造完符号子节点后，在最后插入一个语义动作节点，名称为'$'+节点编号，类型为todo

\9. cur指向第一个子节点，若为空则找下一个子节点，所有子节点都为空或找到语义动作节点则找最近兄弟的最左子节点，以此类推直到使cur指向最近的非终结符节点

\10. 重复步骤3，继续先序构造语法分析树；

\11. 表明已经为这个token构造出一条路径，将token的表示的符号名称赋给当前的终结符节点，同时需要将cur指针回溯到最近的非空的兄弟节点或更上层的节点，栈顶元素出栈；

\12. 重复步骤2，为下一个符号构造路径，直到文件结束

错误检查：

每次取到一个非终结符，先检查预测分析表中该非终结符所在行对应的token所在列的值是否为零，若为零则表明不能出现该语法

当为一个终结符构建完路径后，栈顶元素是这个终结符，此时再判断栈顶元素和token是否匹配，不匹配则表明出现语法错误。

生成中间代码函数

void midcode(fstream &in, fstream &out)

生成中间代码

输出文件为中间代码文件 middlecode.txt 、三地址形式中间代码threeadd_code.txt

深度遍历语法分析树，对每一个todo节点，执行其对应的语义动作函数，将中间代码输出到文件

语义动作

一共有82条产生式，故有82个对应的语义动作函数，分为以下几类：

一. 无需额外语义动作

对于这些产生式的双亲节点，其推导出的子树对应的中间代码已经全部产生，不需要额外的信息传递，故其对应的语义动作函数为空

- **formula*1 ~ formula*38**

二. 传值且生成证中间代码，将中间代码存入map

遍历到这些节点时，其子节点的信息已经足够进行中间代码的生成，但其中特殊语句如if或while等生成的中间语句不完整，需要留空等待回填

- **13 fun-declaration -> type-specifier IDF (params)**

  函数声明，则产生中间代码为标记函数开头*(Prog, -, -, name)*

- **43 return-stmt -> return return-stmt'**

  推出返回语句，产生中间代码return语句*(ret, -, -, name(无返回值为空))*

- **46 expression -> var1 = expression**

  可能推出等式，产生赋值语句中间代码 *(:=, arg1, arg2, result)*

- **54 simple-expression -> additive-expression simple-expression'**

  可能推出逻辑表达式，产生中间代码 *(relop, arg1, arg2, result)*

- **77 call -> IDF (args)**

  可能推出调用表达式，若函数有参数，对每个参数产生中间代码*(param, -, -, arg)*，再产生中间代码 *(call, -, -, name)*

if else while特殊处理

- **39 特殊处理**

  selection-stmt -> if (expression $39*1) statement $39*2 selection-stmt $39_3'

$39_1遍历过判断条件后，立刻设置真出口为当前位置+2行，当前位置+1行则是假出口，先填写转移语句，转移目标暂缺等待回填

39_2真语句结束，设置跳转语句，跳转至至if的下一条语句，待回填

39_3else句结束，若存在else语句，将下一条地址回填至真语句结束的goto后面；若不存在else语句，则去除之前留下的假出口的goto语句，将当前地址回填到真出口

- **41 特殊处理**

  41 selection-stmt' -> else $41*1 statement $41*2

$41_1 else的入口地址 在41_3时填入

41_2记录else的出口地址 在41_3时填入

- **42 特殊处理**

  42 iteration-stmt -> while ($42*1 expression $42*2) statement $42_3

$42_1判断语句前的动作，记录while入口地址 42_3时跳转回此地址

42_2判断语句后的动作，填写真出口为真强制跳转至当前地址+2， 下一行填写假出口，目的地址未求出等待回填

42_3循环语句后的动作，回填假出口值，且插入跳转回42_1位置的中间代码

加减的运算

由于语义动作按照深度优先遍历，故连续加法中语义动作的顺序是右结合，违反了计算顺序，这里用一个数组存产生的临时变量Tn（顺序取用），一个栈存放每个中间代码的右部分（倒序取用），每个64号产生式都将新产生的元素加入对应容器，在63号产生式时统一计算，达到从左到右计算的目的。

- **63 additive-expression -> term additive-expression'**

  可能推出加法表达式，产生中间代码 *(addop, arg1, arg2, result)*

- **64 additive-expression' -> addop term additive-expression'**

乘除的运算

思路同上，用两个容器实现了同优先级操作符左结合的规则

三. 传值，不生成中间代码

- **剩余的formula函数**

剩余的语义动作操作为信息传递，同时设置arraryflag、returnflag、emptyflag等信息

基于LL(1)文法的编译器——代码优化

```
// 初始化操作符表
 void init_op()
 // 删除字符串前后的空格
 void remove_blank(string str)
 // 获取中间代码，修改格式
 void get_intermediate_code(string infile)
 // 复写传播优化
 void facsimile_transmission(int start, int end)
 void optimize_intermediate_code(string infile, string outfile)
```

主要功能

对中间代码进行复写传播优化

\1. 逐行读取四元式代码，处理字符串，按照空格分开，存到map中

\2. 划分基本块，对每一基本块进行优化

\3. 当四元式代码有三个字符串，如果第三个是数字，则将这一代码加入常量表。如果第三个字符串能在常量表中找到，就把他替换

基于LALR文法的编译器——词法分析

正则表达式定义

```
KeyWord     else|if|int|return|void|while
Identifier  [A-Za-z][A-Za-z]*
Constant    ([0-9])|([1-9][0-9]*)
Operator    <=|>=|==|!=|[\+\-\*\/\<\>\=]
Delimiter   \/\*|\*\/|[\;\,\[\]\{\}]|\(|\)
Space       [\ \t\n]
Comment     \/\*(.|\n|\t|\ )*\*\/
Other       [^{KeyWord}{Identifier}{Constant}{Operator}{Delimiter}{Space}
{Comment}]

```

主要功能

将输入文件内容识别出token

输入/输出

输入测试文件，输出各token名称、类型、行列号信息

基于LALR文法的编译器——语法分析

终结符与非终结符声明

```
%start program
%token ID
%token NUM
%token PLUS MINUS MUL DIV
%token LESSEQUAL LESS GREATEREQUAL GREATER EQUAL NOTEQUAL
%token CONST
%token ELSE IF INT RETURN VOID WHILE
%token LSPAREN RSPAREN LMPAREN RMPAREN LBPAREN RBPAREN
%token COMMA SEMI

%right CONST
%left PLUS MINUS
%left MUL DIV
%nonassoc ELSE
%left LSPAREN RSPAREN
%left LBPAREN RBPAREN
%left LMPAREN RMPAREN
%left LESSEQUAL GREATEREQUAL LESS GREATER EQUAL NOTEQUAL

```

主要功能

声明BNF文法中的终结符，定义操作符的结合规则和优先级

输入/输出

输入测试文件，输出语法树、规约过程；

实现思想

\1. 将BNF中的终结符提取，用%token声明；

\2. 定义操作符的结合规则，=号为右结合，四则运算符、界符与比较运算符都是左结合，且从上到下优先级递增。

构建规约栈

```
typedef struct{
    int top;
    elem index[maxsize];
}Stack;                                 //符号栈
void stack_init();                      //初始化符号栈
elem stack_pop();                       //出栈
void stack_push(elem p);                //进栈
void Process(char* temp);               //移进
void Reduce(char* name, int num);       //规约
```

构建语法树

```
typedef struct node{
    char* data;
    struct node* left;
    struct node* right;
}Node;                                          //树节点（左孩子-右兄弟）
typedef struct{
    Node* root;
}Tree;                                          //树
```

构建一颗孩子-兄弟树T，最后先序遍历，跟据所处层数为k输出到文件

\1. 在规约过程中进行建树操作，调用left_insert与right_insert插入孩子和兄弟；

\2. 在规约到最后时，调用先序遍历函数。

\3. 先序遍历函数为递归函数，先判断节点是否为空，非空则输出第k层的该节点信息；

\4. 该节点输出完成后，继续调用该函数，传入该节点的左孩子，最后传入该节点的右兄弟，形成先序遍历的顺序。

基于LALR文法的编译器——语义分析

构造符号表

```
struct symbol {                         //符号表一个元素
    string name;                            //变量名
    string type;                            //变量类型(void, int, *)
    string property;                        //变量属性(func, var)
    int num;                                //func的参数个数
};
map<string, map<string, symbol> > st;//符号表(函数名->变量名->变量信息)
```

判断变量或函数是否被定义，根据规则向符号表中添加变量或函数

输入测试文件，输出符号表、中间代码

\1. 初始化符号表时，向其中添加基本输入、输出函数input和output；

\2. 遇到定义符号行为，判断其在当前域是否被定义，若是，则报错，若不是，则定义该符号；

\3. 遇到使用符号行为，判断其在所有域是否被定义，若是，则正常使用，若不是，则报错；

\4. 最后，将符号表全部信息输出到文件中。

中间代码生成

```
stack<int> truestack, falsestack;           //true栈和false栈
vector<string> code;                            //输出的中间代码
vector<string> code_qua;                        //输出的中间代码(四元式)
void backpatch(stack<int> &S);              //回填
void emit(string content1, string content2);    //产生中间代码
void Output();                                  //输出符号表和中间代码到文件
```

在规约过程中执行相应操作，产生两种中间代码

\1. emit函数两个参数分别对应两种形式的中间代码，将中间代码存入vector中，待后续操作；

\2. 遇到if...else...或while语句时，需要回填：在条件产生式后进行falsestack.push()，在“真语句”的产生式后进行truestack.push()，在else产生式后进行backpatch(falsestack)，待“假语句”执行完成后再进行backpatch(truestack)；对于while语句类似，不同的是“真语句”执行完成后需要指回条件产生式处；

\3. 待规约到最后，将中间代码输出到文件中。

语义动作实现

```
selection-stmt  :       IF LSPAREN expression RSPAREN {
                            string dest = to_string(code.size() + 102);
                            emit("if " + $3 + "==1 Goto " + dest, "(jnz, " + $3 + ", -, " + dest);
                            falsestack.push(code.size());
                            emit("Goto ", "(j, -, -, ");
                        }
                        statement {
                            truestack.push(code.size());
                            emit("Goto ", "(j, -, -, ");
                        }
                        selection-stmts {
                            fprintf(fi, "selection-stmt -> IF ( expression ) statement selection-stmts\n");
                            Reduce("selection-stmt", 6);
                        }
                ;
.....

```

主要功能

在BNF中插入{语义动作}，执行相关操作

输入/输出

输入测试文件，输出符号表、中间代码

实现思想

\1.为简单算术表达式、赋值语句、数组元素的引用、条件控制语句等添加语义动作进行翻译；

\2. 较为复杂的回填操作通过建立truestack与falsestack，并对其进行操作来实现（上面由相关的说明）。