

東南大學

编译原理课程设计

设计报告

成员： 09013430 任杰文

09013429 黄路遥

09013413 钱鑫

东南大学计算机科学与工程学院

二 016 年 5 月

设计任务名称		SeuYACC	
完成时间		<u>2016/5/30</u>	验收时间
本组成员情况			
学 号	姓 名	承 担 的 任 务	成 绩
09013430	<u>任杰文</u>	<u>Lex 全部任务：分析，设计，实现。</u> <u>相关文档报告编写。</u>	
09013429	<u>黄路遥</u>	<u>YACC：完成 yySeuYacc 分析程序的生成，利用 PDA 配合 Lex 完成测试程序的读入和分析，采用语法制导的翻译（SDT）的方式进行中间代码生成</u>	
09013413	<u>钱鑫</u>	<u>YACC: YACC 文件的读取，实现了求 FIRST、CLOSUER 等函数，完成了 LR(1)项目集的构造及相关 table 的生成，LALR（1）项目集的构造及相关 table 的生成。</u>	

注：本设计报告中各部分如果页数不够，请自行扩页。原则是一定要把报告写详细，能说明本组设计的成果和特色，能够反映小组中每个人的工作。报告中应该叙述设计中的每个模块。设计报告将是评定各人成绩的重要依据之一。

1 编译对象与编译功能

1.1 编译对象

(作为编译对象的 C 语言子集的词法、语法描述)

Minic.y 的内容如下

```
%{
/*  minic.y(1.9)  17:46:21  97/12/10
*
*   Parser demo of simple symbol table management and type checking.
*/

#include <stdio.h> /* for (f)printf() */
#include <stdlib.h> /* for exit() */

extern int line = 1; /* number of current source line */
extern int seulex(); /* lexical analyzer generated from lex.l */
char *yytext; /* last token, defined in lex.l */ /* current symbol table, initialized in lex.l */
char *base; /* basename of command line argument */

void
yyerror(char *s)
{
    fprintf(stderr, "Syntax error on line %d: %s\n", line, s);
    fprintf(stderr, "Last token was \"%s\"\n", yytext);
    exit(1);
}

%}

%union {
    char*      name;
    int        value;
    T_LIST*    tlist;
    T_INFO*    type;
    SYM_INFO*  sym;
    SYM_LIST*  slist;
}

%token  INT FLOAT NAME STRUCT IF ELSE RETURN NUMBER LPAR RPAR LBRACE
RBRACE
%token  LBRACK RBRACK ASSIGN SEMICOLON COMMA DOT PLUS MINUS TIMES
DIVIDE EQUAL

%type  <name>  NAME
```

```

%type <value>  NUMBER
%type <type>   type parameter exp lexp
%type <tlist>  parameters more_parameters exps
%type <sym>   field var
%type <slist>  fields

%left      ELSE
%right     EQUAL
%left      PLUS  MINUS
%left      TIMES DIVIDE
%left      UMINUS
%left      DOT   LBRACK

%%
program      : declarations
              ;

declarations : declaration declarations
              |
              ;

declaration  : fun_declaration
              | var_declaration
              ;

fun_declaration : type NAME { /* this is $3 */
                          $<sym>$ = symtab_insert(scope,$2,0);
                          scope = symtab_open(scope); /* open new scope */
                          scope->function = $<sym>$; /* attach to this function */
                          }
              LPAR parameters RPAR { /* this is $7 */
                          $<sym>3->type = types_fun($1,$5);
                          }
              block { scope = scope->parent; }
              ;

parameters   : more_parameters { $$ = $1; }
              |
              { $$ = 0; }
              ;

more_parameters : parameter COMMA more_parameters
                { $$ = types_list_insert($3,$1); }
              | parameter
                { $$ = types_list_insert(0,$1); }
              ;

```

```

parameter: type NAME {
    symtab_insert(scope,$2,$1); /* insert in symbol table */
    $$ = $1; /* remember type info */
}
;

block      : LBRACE      { scope = symtab_open(scope); }
            var_declarations statements RBRACE
            { scope = scope->parent; /* close scope */ }
;

var_declarations: var_declaration var_declarations
|
;

var_declaration : type NAME SEMICOLON { symtab_insert(scope,$2,$1); }
;

type      : INT          { $$ = types_simple(int_t); }
| FLOAT    { $$ = types_simple(float_t); }
| type TIMES { $$ = types_array($1); }
| STRUCT LBRACE fields RBRACE
            { $$ = types_record($3); }
;

fields      : field fields { $$ = symtab_list_insert($2,$1); }
|           { $$ = 0; }
;

field       : type NAME SEMICOLON { $$ = symtab_info_new($2,$1); }
;

statements  : statement SEMICOLON statements
|
;

ifhead      : IF LPAR exp RPAR
;

ifstatement : ifhead statement
;

statement : ifstatement
| ifstatement ELSE statement
| lexp ASSIGN exp { check_assignment($1,$3); }

```

```

| RETURN exp
    { check_assignment(scope->function->type->info.fun.target,$2); }
| block
;

lexp    : var          { $$ = $1->type; }
| lexp LBRACK exp RBRACK { $$ = check_array_access($1,$3); }
| lexp DOT NAME         { $$ = check_record_access($1,$3); }
;

exp      : exp DOT NAME    { $$ = check_record_access($1,$3); }
| exp LBRACK exp RBRACK    { $$ = check_array_access($1,$3); }
| exp PLUS exp            { $$ = check_arith_op(PLUS,$1,$3); }
| exp MINUS exp           { $$ = check_arith_op(MINUS,$1,$3); }
| exp TIMES exp           { $$ = check_arith_op(TIMES,$1,$3); }
| exp DIVIDE exp          { $$ = check_arith_op(DIVIDE,$1,$3); }
| exp EQUAL exp           { $$ = check_relop(EQUAL,$1,$3); }
| LPAR exp RPAR           { $$ = $2; }
| MINUS exp
    { $$ = check_arith_op(UMINUS,$2,0); }
| var          { $$ = $1->type; }
| NUMBER       { $$ = types_simple(int_t); }
| NAME LPAR RPAR { $$ = check_fun_call(scope,$1,0); }
| NAME LPAR exps RPAR { $$ = check_fun_call(scope,$1,&$3); }
;

exps     : exp          { $$ = types_list_insert(0,$1); }
| exp COMMA exps        { $$ = types_list_insert($3,$1); }
;

var       : NAME          { $$ = check_symbol(scope,$1); }
;

%%

int
main(int argc,char *argv[])
{
base = "test.c";
yyparse();
}

```

1.2 编译功能

(所完成的项目功能及对应的程序单元)

FileReader.h 中

FileReader 类: **mimicry** 文件的读入, 并将内容分割为 C 语言头定义段, **yacc** 定义段, 规则段, 语义定义段和用户子程序段。

ItemStructure.h 中

Symbol 类, 存放终结符以及非终结符。

Rule 类, 存放单条语法规则。

RuleSet 类, 存放所有的语法规则, 为其他部分提供查询功能, 并通过 **calFF()** 成员函数完成 **First** 集合的计算。同时存放优先级结合律等信息, 供其它部分使用。

Item 类, 存放语法规则的编号和点所在位置以及预测符号集, 同时实现了两种判断相等依据以供上层使用。还提供了项层面上的合并函数, 供上层使用。

Itemset 类, 存放项目集, 同时实现了两种判断相等依据以供 **LR** 和 **LALR** 使用, 提供了项目集层面上的合并函数, 供上层使用。提供了项目集层面上的 **hash** 方法。完成闭包的计算, 提供了移进项目集的计算。

CallItemSets 类, 完成 **LR** 项目集和 **LALR** 项目集的计算, 完成冲突处理以及 **ActionTable** 和 **GotoTable** 的生成。

SeuYacc 文件生成部分: **SeuYacc** 分析 **minic.y** 后生成 **yySeuYacc.h**、**actionTable**、**gotoTable**、**translation.h** 四个文件。

通过 **SeuYacc** 根据 **minic.y** 生成 **yySeuYacc** 后, 还需要一些额外的头文件进行语法分析和语义分析, 这些头文件包括:

PDA.h 中

PDA 类, **PDA** 维护一个下推栈和一个状态栈, 实现了规约和移进的算法, 并初步实现了错误分析程序, 可以根据语法分析错误类型, 给出错误位置提示, 并打印错误处的栈状态方便进行分析。其中 **PDA** 根据随 **yySeuYacc** 生成 **actiontable** 和 **gototable** 进行推导, 并在每次规约的时候调用 **minic.y** 预定义好的语义程序段进行语义分析, 生成四元式

SDT.h 和 **Quadurple.h** 中

实现了四元式的结构, 并定义了语义栈、**TC** 栈和 **FC** 栈。语义栈与语法分析同步进行了, 将分析到的 **token** 存储, 并在规约的时候弹栈生成对应的中间代码表示。**TC** 栈和 **FC** 栈分别在遇到分支语句的时候用于存储真链和假链, 当完成 **if** 语句中 **then** 和 **else** 部分的分析后, 通过对 **TC** 和 **FC** 弹栈进行真链和假链的拉链回填操作。

2. 主要特色

实现了所有要求，通过标记的方式减少了求 CLOSUER 过程中的消耗。

可以通过 LEFT、RIGHT 等定义优先级结合律从而处理冲突。

求 LALR 项目集的过程中做到了边求边合并，节省了空间消耗。

完成了基本的语义分析，可以根据在 minic.y 中自定义的语义程序进行语义分析，并生成四元式作为中间代码。

可以进行布尔语句和分支语句的正确翻译，并用回填技术进行拉链操作。

3 概要设计与详细设计

（由总到分地介绍 **SeuLex** 和 **SeuYACC** 的设计，包括模块间的关系，具体的算法等。采用面向对象方法的，同时介绍类（或对象）之间的关系。在文字说明的同时，尽可能多采用规范的图示方法。）

3.1 概要设计

（以描述模块间关系为主）

SeuYACC 主要包含 9 个类及模块

1) **Symbol** 类，存放终结符以及非终结符。

class Symbol//终结符与非终结符

```
{  
public:  
    string _name;  
    int _type;//0 非终结符，1 终结符  
}
```

2) **Rule** 类，由 **Symbol** 组成

class Rule//单条规则

```
{  
public:  
    Symbol Left;//产生式左部  
    vector<Symbol> Right;//产生式右部  
}
```

3) **RuleSet** 类，由 **Rule** 组成，提供计算 **First** 的功能

class RuleSet//规则集 id->rule，是数据主要存放的地方

```
{  
public:  
    vector<Rule> rules;//规则  
    map<Symbol, set<Symbol>> Firstset;//First集合  
    //在读yacc文件时存放的额外信息  
    set<string> Token;  
    set<string> Left;  
    set<string> Right;  
    set<string> Nonassoc;  
    map<string, int> priority;  
    //符号集合  
    set<Symbol> Symbols;  
private:  
    map<Symbol, vector<int>> queryMap;  
    int ruleNum;  
};
```

4) **Item** 类，由 **ruleId** 和 **dot** 两个整数表示产生式和点的位置，以及 **Symbol** 的集合表示预测符集

class Item

```
{
public:
    int ruleId;//产生式编号
    int dot;//点的位置
    set<Symbol> predict;//预测符集合
};
```

5) **ItemSet** 类，由 **Item** 组成，提供计算 **Closure** 的功能

```
class ItemSet
{
public:
    set<Item> items;//项的集合
};
```

6) **CalItemSets** 类，由 **ItemSet** 组成，计算过程中会使用 **RuleSet** 类中的数据，提供计算项目集族，填actiontable和gototable的功能

```
class CalItemSets
{
public:
    vector<ItemSet> vItemSet;//项目集族
    map<pair<int, Symbol>, int> edgeSet;//<id, Symbol>->id
    map<pair<int, Symbol>, string> actionTable;
    map<pair<int, Symbol>, int> gotoTable;
    unordered_map<string, int> itemHash;//hash表
    ItemSet start;//起点
    int index;
};
```

7) **seuYacc** 的 **main** 函数

主要进行 **minic.y** 文件的读取分析，输出 **action table** 和 **goto table**，生成 **yyseuYacc** 通用部分（**yyparse**（）实现）并和用户程序段组合形成完整的分析程序。再根据 **minic.y** 中语义定义部分，生成语义分析辅助程序，并保存为 **translation.h**。

生成的 **yyparse** 如下，主要完成调用 **yySeuLex** 生成 **token** 并调用 **PDA** 进行语法分析。

```
void yyparse()
{
    FileReader fd;
    fd.ParseYFile("minic.y");
    vector<Symbol> s;
    ifstream fin(base);
    string str;
    while (1)
    {
        string t = seuLex(fin);
        if (t == "ERROR") break;
        if (t != "SPACE"){
```

```

        Symbol sym(t,l,seuLexLastLex);
        s.push_back(sym);}
    }
    Symbol sym("$",1);
    s.push_back(sym);
    fin.close();
    PDA pda(rs);
    pda.input(s);
    pda.readinTables("actionTable","gotoTable");
    pda.parse();
}

```

8) PDA 类，主要辅助生成的 yySeuYacc 进行语法分析，其中封装了标准的 shift 和 reduce 过程。根据 LALR 生成的 action table 和 goto table 即可进行语法分析。

```

class PDA
{
public:
    PDA();
    PDA(ruleSet RS);
    PDA(vector<rule> RS);
    ruleSet rs;
    vector<Symbol> sentence; //Sentence waiting to be parsed
    int reader;              //where the PDA is reading
    stack<int> stateStack;
    stack<Symbol> PDstack;
    vector<Symbol> stackData;
    map<pair<int, Symbol>, string> actionTable;
    map<pair<int, Symbol>, int> gotoTable;
public:
    void readinTables(string actionfile, string gotofile);
    void parse();           //true yyparse()
    void input(vector<Symbol> s);
    void shift();
    Symbol reduce(rule r);  //reduce with a given rule
    bool isSuccess();
    bool isError();
    void success();
    void error(int);
};

```

9) 语法制导翻译和四元式部分

主要提供了一个四元式生成、临时变量生成的函数，并定义了语义栈、真链和假链，可以在语义程序中使用，用于分析分支程序。

```

Quadurple.h:
class quadurple
{

```

```

public:
    string sym;        //the symbol of the operation
    string result;
    string i;          //third element of a quadruple
    string j;          //fourth element of a quadruple
    quadruple(string SYM,string RESULT,string I,string J);
    friend ostream& operator << (ostream& out, quadruple);
};

SDT.h
stack<string> SDTStack;
stack<int> TC;
stack<int> FC;
vector<quadruple> quadSet;
int tempID = 0;
int lineo = 0;
ofstream fout("IntermediaCode.c");
string intToString(int a)
{
    char c[100] = "";
    itoa(a,c,10);
    string C(c);
    return c;
}
string newTemp()    //produce a new temp variable of Tx
{
    char id[20] = "";
    tempID++;
    itoa(tempID,id,10);
    string ID(id);
    return "T"+ID;
}
void gen(string sym,string result,string i, string j) //generate a new quadruple
{
    quadruple q(sym,result,i,j);
    quadSet.push_back(q);
    lineo++;
}
void quadOutput()
{
    for (int i = 0;i<quadSet.size();i++)
    {
        fout<<i<<" : ";
    }
}

```

```

        fout<<quadSet[i];
    }
}

```

3.2 详细设计

(以描述数据结构及算法实现为主)

求 Closure 算法

龙书上的算法:

```

SetOfItems CLOSURE(I) {
    repeat
        for ( I 中的每个项  $[A \rightarrow \alpha \cdot B\beta, a]$  )
            for (  $G'$  中的每个产生式  $B \rightarrow \gamma$  )
                for ( FIRST( $\beta a$ ) 中的每个终结符号  $b$  )
                    将  $[B \rightarrow \gamma, b]$  加入到集合  $I$  中;
    until 不能向  $I$  中加入更多的项;
    return I;
}

```

实际处理过程中, 应该要注意对于每一个项只会被扩展一次, 但是扩展过程中它的预测符集可能会被修改多次, 所以采用了一个队列对要修改的项进行维护。

void Closure (项集 I)

```

{
    队列 q;
    将初始项加入q尾部。
    while (!q.empty())
    {
        取出q的队首元素t;
        if (t中点已经到达结束位置)
            continue;
        Symbol st = t中点所在位置的符号;
        if (st是终结符)

```

```

        continue;
    对于形如  $A \rightarrow \alpha.B\beta, a$  的项, 计算预测符号集  $b = \text{First}(\beta a)$ 
    for (st 为产生式左部的每个产生式  $X \rightarrow \gamma$ )
        if ( $X \rightarrow \gamma$  已经加入了 I, 且那项为 T)
            将 b 合并到 T 的预测符号集合里
            if T 的预测符号集变化了
                将合并后的 T 加入 q 尾部
        else
            将  $\{X \rightarrow \gamma, b\}$  加入到 I 里, 并加入 q 尾部
    }
}

```

求 First 算法

龙书上的算法

计算各个文法符号 X 的 $\text{FIRST}(X)$ 时, 不断应用下列规则, 直到再没有新的终结符号或 ϵ 可以被加入到任何 FIRST 集合中为止。

1) 如果 X 是一个终结符号, 那么 $\text{FIRST}(X) = X$ 。

2) 如果 X 是一个非终结符号, 且 $X \rightarrow Y_1 Y_2 \dots Y_k$ 是一个产生式, 其中 $k \geq 1$, 那么如果对于某个 i , a 在 $\text{FIRST}(Y_i)$ 中且 ϵ 在所有的 $\text{FIRST}(Y_1), \text{FIRST}(Y_2), \dots, \text{FIRST}(Y_{i-1})$ 中, 就把 a 加入到 $\text{FIRST}(X)$ 中。也就是说, $Y_1 \dots Y_{i-1} \Rightarrow \epsilon$ 。如果对于所有的 $j = 1, 2, \dots, k$, ϵ 在 $\text{FIRST}(Y_j)$ 中, 那么将 ϵ 加入到 $\text{FIRST}(X)$ 中。比如, $\text{FIRST}(Y_1)$ 中的所有符号一定在 $\text{FIRST}(X)$ 中。如果 Y_1 不能推导出 ϵ , 那么我们就不会再向 $\text{FIRST}(X)$ 中加入任何符号, 但是如果 $Y_1 \Rightarrow \epsilon$, 那么我们就加上 $\text{FIRST}(Y_2)$, 依此类推。

3) 如果 $X \rightarrow \epsilon$ 是一个产生式, 那么将 ϵ 加入到 $\text{FIRST}(X)$ 中。

实际处理中, 因为要考虑到成环的情况, 所以通过一个数组记录了每个产生式的使用情况, 避免了无限递归。

```

int getFirstSet(int ruleId, set<Symbol> &firstSet, set<Symbol> &isUsed)
{
    if 该规则的右部为  $\epsilon$ 
        返回 0
    else {
        对于 ruleID 对应的产生式 R;
        i=0;
        if R 形如  $A \rightarrow \alpha B$  ( $\alpha$  为终结符)
            把  $\alpha$  加入 firstSet
        else
        {
            那么 R 形如  $A \rightarrow X_0 X_1 \dots X_n$ 
            当 i 小于 R 右部长度 n 时
                对于每个以  $X_i$  为产生式左部的产生式 r 计算 getFirstSet(r, firstSet, isUsed)
                if 对于  $X_i$  的每个产生式出的返回值都不为 0, 即其对应 First 都不包含  $\epsilon$ 
                    break
                i++
            if i==R 右部长度 n
                把  $\epsilon$  加入 firstSet 并返回 0
        }
    }
}

```

```

    }
    返回 1;
}
}
生成项集族算法 (LR (1))
void CallItemSets::genPDALR()
{
    index=1
    将文法开始符号及对应产生式加入项集 start, 并计算其 Closure
    将 start 加入项集族 vItemSet
    将 start 加入队列 q 尾部
    讲<start,index>存入 hash 表 itemHash 中
    index++
    while (!q.empty())
    {
        取出 q 队首的项集 temp
        根据 item 中各项点的位置, 所对应的非终结符, 组成候选扩展符号集 cand
        for (cand 中的每个符号 s)
        {
            计算 temp 接受 s 后会移进到的项集 p
            if p 不在 hash 表 itemHash 中
            {
                将 p 加入项集族 vItemSet
                讲<p,index>存入 hash 表 itemHash 中
                index++
                将<<temp,s>,p>存入 edgeSet (此处为了节省空间使用的是 temp 和 p 所对
应的 index)
                将 p 加入队列 q 尾部
            }
            else
            {
                找到 p 所对应编号 pindex
                将<<temp,s>,p>存入 edgeSet (此处为了节省空间使用的是 temp 和 p 所对
应的 index, p 的 index 也就是 pindex)
            }
        }
    }
}
这部分的 hash 函数是利用项集的每个项及预测符号集合计算出来的

生成项集族算法 (LALR (1))
void CallItemSets::genPDALR()
{
    index=1

```

```

将语法开始符号及对应产生式加入项集 start，并计算其 Closure
将 start 加入项集族 vItemSet
将 start 加入队列 q 尾部
讲<start,index>存入 hash 表 itemHash 中
index++
while (!q.empty())
{
    取出 q 队首的项集 temp
    根据 item 中各项点的位置，所对应的非终结符，组成候选扩展符号集 cand
    for (cand 中的每个符号 s)
    {
        计算 temp 接受 s 后会移进到的项集 p
        if p 不在 hash 表 itemHash 中
        {
            将 p 加入项集族 vItemSet
            讲<p,index>存入 hash 表 itemHash 中
            index++
            将<<temp,s>,p>存入 edgeSet（此处为了节省空间使用的是 temp 和 p 所对
应的 index）
            将 p 加入队列 q 尾部
        }
        else
        {
            找到 p 所对应编号 pindex
            将 p 与在 itemHash 中记录的同心项集 i1 合并得到 i2，并保存到 i1 中
            将<<temp,s>,p>存入 edgeSet（此处为了节省空间使用的是 temp 和 p 所对
应的 index，p 的 index 也就是 pindex）
            if i2!=i1
                讲 i2 存入 q 尾部
        }
    }
}
}

```

这部分的 hash 函数是利用项集的每个项所用规则以及点的位置计算出来的

LALR(1)和 LR(1)的计算项目集族的方法还是比较像的，都采用了类似 BFS+Hash 的方法，避免重复扩展。通过这个过程同时也得到了 Goto 函数，也就是 edgeSet 中的内容。

冲突处理及填表算法

```

int recur(int state ,int*flag)
{
    if 该 state 已经处理过
        返回 0;
    flag[state] = 1;
}

```



```

根据 is 中每个项中点的位置，把这些项归到可规约集合 reduceV 和可移进集合 cand
if (可归约集合 reduceV 不为空)
{
    for (可规约集合 reduceV 里的每一项 it)
    {
        根据 it 的 ruleId 和预测符号集 predict，往 actionTable 中填入相应规约动作，
        如果 actionTable 对应位置已有动作则不覆盖，并加入已处理符号集 produced，且 ruleID 为 0
        时填入 acc
    }
}
for (可移进集合 cand 每一项 s)
{
    根据 edgeSet 得到在当前 state 和 s 下会进入的状态 next
    if s 不在已处理符号集 produced 中
    {
        根据当前状态 state 和 s 还有 next 往 actionTable 和 gototable 中填入相应动作
    }
    else
    {
        根据 s 和已有的优先级以及结合性等信息，确定是否覆盖原来的规约动作，例
        如本状态的操作优先级低于下个状态的，那么就移进，如果本状态优先级更高就选择规约。
        如果优先级都一样，s 是左结合的就移进，是右结合的就规约
    }
    递归处理 recur(next, mem, flag);
}
返回 0;
}

```

总体上来说是一个 dfs 遍历所有项集的过程，在每一个项集按照规则进行处理。

PDA 根据 action table 和 goto table 进行语法分析：

(1) shift (移进)

将当前读头下的 token 压下推栈（语义值同时压栈），读头向后移动一个。若超出分析的程序长度，报错 reader 越界。

(2) reduce (规约)

根据给定规则 R 进行规约，记 $|R.right|$ 为 R 规则产生式右部规则包含的 token 个数，将当前下推栈中弹出 $|R.right|$ 个元素。每次弹出的时候检查弹出的 token 是否与当前规则匹配，若不匹配则报错规则不匹配。否则弹栈结束后，将 R 左侧 token 压栈。

(3) parse (分析主程序)

初始状态栈栈顶为 0，读头指向第一个 token。

While (1)

```

{
    s = actionTable(stateStack.top, Sentence[reader])
    if (s == error)

```

```

    error();//错误处理程序
if (s==acc)
    success();//提示分析成功
if (s 为 s+n (n 为数字) 的结构 (即进行移进))
    PDStack.push(n)
    Shift()
else
    //s 为 r+n 的结构 (即进行规约)
    Reduce(n);
    将 stateStack 栈顶 | R.right | 个元素弹栈, 并将 gotoTable(stateStack.top, L)压栈
    运行语义分析程序
}

```

中间代码生成部分：该部分主要分析四则运算语句和条件分支语句，并产生他们的四元式。
四则运算：（以加法为例）

语义程序：T = newTemp () //产生一个新的临时变量
 Gen ('+', T, \$1, \$3)
 SDTStack.push(T)//新产生的临时变量进入语义栈

赋值语句

语义程序：Gen ('=', \$1, _, \$3)

分支语句：(if-then-else 为例)

ifhead ---> IF LPAR exp RPAR

语义程序： gen("j", \$1, \$3, "0");//目标地址记 0，待回填
 TC.push(lineo); //当前行数记为真链回填位置，下一行为假链回填位置
 FC.push(lineo+1);
 gen("j", "_", "_", "0");//下一句为一个无条件跳转语句
 quadSet[TC.top()-1].j = lineo; //本句已经可以回填，进行回填
 TC.pop(); //弹出已经回填的行号

ifstatement ---> ifhead statement

语义程序： gen("j", "_", "_", "0");//分析完 then 部分进行一次无条件跳转，跳转到
 if-then-else 模块后一句，暂时不知道地址，待回填

 TC.push(lineo); //将当前行号压入真链，待回填
 quadSet[FC.top()-1].j = lineo; //同时该位置为上一个假链的跳转位置（马上
 上执行 else 语句），因此进行一次回填
 FC.pop();

statement :---> ifstatement ELSE statement

语义程序： quadSet[TC.top()-1].j = lineo; //当前行号为真链栈顶语句应跳转的位置，进
 行回填

 TC.pop();

语义分析和语法分析同步进行，每次规约的时候进行一次四元式的产生（对于可以产生四元式的规则而言）。在进行语法分析同时维护一个 SDTstack，其中存储与语法分析 token 对应的源程序的符号。

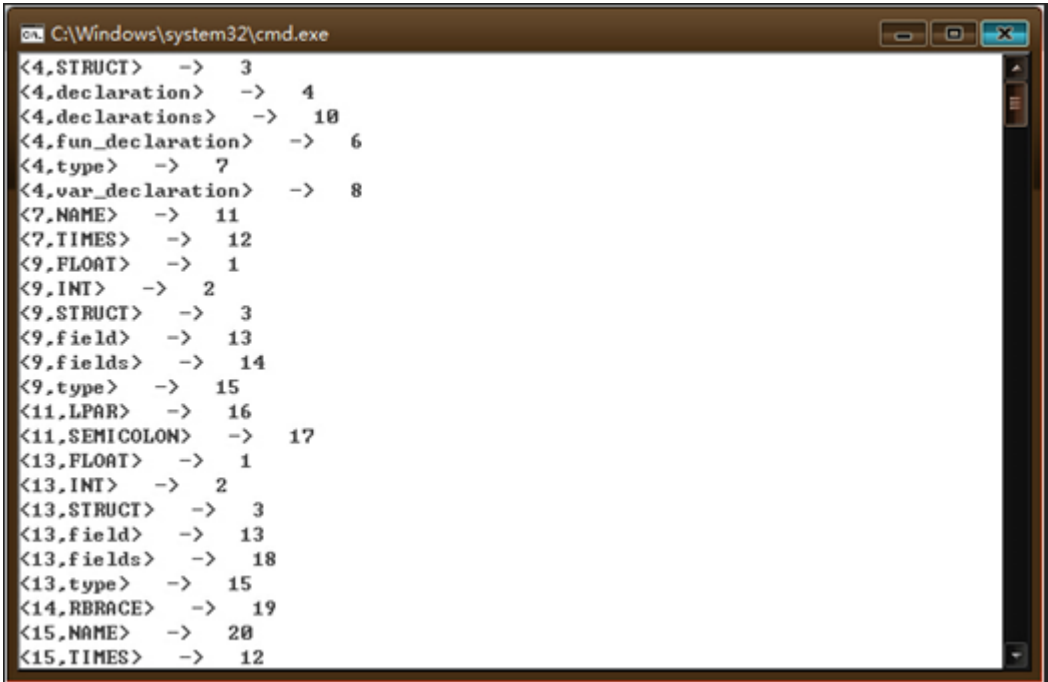
4 使用说明

4.2 SeuYacc 使用说明

- (1) 准备好 yacc 文件，如 **minic.y**，运行 **seuYacc**。
- (2) 若运行成功会在子文件夹 **genCode** 中生成 **actionTable**、**gotoTable**、**translation.h**、**yySeuYacc.cpp** 四个文件。
- (3) 要进行语法分析，还需要准备 **FileReader.h**、**ItemStructure.h**、**PDA.h**、**quadruple.h**、**SDT.h**、**yySeuLex.h** 以及它们对应的 **cpp** 文件。（可以将这些文件重新组成一个工程文件）
- (4) 准备待分析的主程序，如 **test.c**（根据 **minic.y** 中用户程序决定）
- (5) 运行分析程序，若运行成功会得到分析成功提示，并生成 **intermediateCode.c** 文件。其中包含的即是根据语法分析产生的中间代码文件；若分析失败会提示分析失败时读头位置，打印出当前栈状态方便分析，同时错误信息会保存在 **ParsingLog** 文件中。

5 测试用例与结果分析

(1) 采用的 minic.y 如前文所示，运行 SeuYacc 程序部分结果如图



```
C:\Windows\system32\cmd.exe
<4,STRUCT> -> 3
<4,declaration> -> 4
<4,declarations> -> 10
<4,fun_declaration> -> 6
<4,type> -> 7
<4,var_declaration> -> 8
<7,NAME> -> 11
<7,TIMES> -> 12
<9,FLOAT> -> 1
<9,INT> -> 2
<9,STRUCT> -> 3
<9,field> -> 13
<9,fields> -> 14
<9,type> -> 15
<11,LPAR> -> 16
<11,SEMICOLON> -> 17
<13,FLOAT> -> 1
<13,INT> -> 2
<13,STRUCT> -> 3
<13,field> -> 13
<13,fields> -> 18
<13,type> -> 15
<14,RBRACE> -> 19
<15,NAME> -> 20
<15,TIMES> -> 12
```



```
C:\Windows\system32\cmd.exe
<87,DIVIDE> -> 65
<87,DOT> -> 66
<87,EQUAL> -> 67
<87,LBRACK> -> 68
<87,MINUS> -> 69
<87,PLUS> -> 70
<87,TIMES> -> 71
<88,DIVIDE> -> 65
<88,DOT> -> 66
<88,EQUAL> -> 67
<88,LBRACK> -> 68
<88,MINUS> -> 69
<88,PLUS> -> 70
<88,TIMES> -> 71
<90,LPAR> -> 46
<90,MINUS> -> 47
<90,NAME> -> 48
<90,NUMBER> -> 49
<90,exp> -> 80
<90,exps> -> 93
<90,var> -> 51

Parsing succeed!
请按任意键继续. . .
```

同时 genCode 文件夹下生成如下文件。

名称	修改日期	类型	大小
actionTable	2016-5-31 21:12	文件	34 KB
gotoTable	2016-5-31 21:12	文件	30 KB
translation.h	2016-5-31 21:12	C++ Header file	2 KB
yySeuYacc.cpp	2016-5-31 21:12	C++ Source file	2 KB

actionTable 内容（稀疏方式存储的 actionTable 内容）

```
$ 0 r2
ASSIGN 0 error
COMMA 0 error
DIVIDE 0 error
DOT 0 error
ELSE 0 error
EQUAL 0 error
FLOAT 0 s1
IF 0 error
INT 0 s2
LBRACE 0 error
LBRACK 0 error
LPAR 0 error
MINUS 0 error
NAME 0 error
NUMBER 0 error
PLUS 0 error
RBRACE 0 error
RBRACK 0 error
RETURN 0 error
RPAR 0 error
SEMICOLON 0 error
STRUCT 0 s3
TIMES 0 error
```

24,1 顶端

gotoTable 内容（稀疏存储的 gotoTable 内容）

```
block 0 0
declaration 0 4
declarations 0 5
exp 0 0
exps 0 0
field 0 0
fields 0 0
fun_declaration 0 6
ifhead 0 0
ifstatement 0 0
lexp 0 0
more_parameters 0 0
parameter 0 0
parameters 0 0
program 0 0
statement 0 0
statements 0 0
type 0 7
var 0 0
var_declaration 0 8
var_declarations 0 0
block 1 0
declaration 1 0
declarations 1 0
<R\SEUyacc\SEUyacc\genCode\gotoTable" 1974L, 30605C
```

1,1 顶端

Translation.h（四元式生成程序）

```

#pragma once
#include "ItemStructure.h"
#include "SDT.h"
#include "quadruple.h"
void translation(rule R,int ruleID)
{
    string temp;
    string sym[20];
    switch (ruleID)
    {
case 24 :
for (int i = 0;i<R.Right.size();i++)
{
    sym[i] = SDTStack.top();
    SDTStack.pop();
}
gen("j"+sym[2],sym[1],sym[3],"0");TC.push(lineo);FC.push(lineo+1);gen("j","_","_",
"0");quadSet[TC.top()-1].j=intToString(lineo);TC.pop();
break;
case 25 :
for (int i = 0;i<R.Right.size();i++)
{
    sym[i] = SDTStack.top();
    SDTStack.pop();
}
gen("j","_","_", "0");TC.push(lineo);quadSet[FC.top()-1].j=intToString(lineo);FC.pop(
);
break;
case 27 :
for (int i = 0;i<R.Right.size();i++)
{
    sym[i] = SDTStack.top();
    SDTStack.pop();
}
quadSet[TC.top()-1].j=intToString(lineo);TC.pop();
break;
case 28 :
for (int i = 0;i<R.Right.size();i++)
{
    sym[i] = SDTStack.top();
    SDTStack.pop();
}
gen(sym[1],sym[2],"_",sym[0]);
break;

```

```

case 36 :
for (int i = 0;i<R.Right.size();i++)
{
    sym[i] = SDTStack.top();
    SDTStack.pop();
}
temp=newTemp();gen(sym[1],temp,sym[0],sym[2]);SDTStack.push(temp);
break;
case 37 :
for (int i = 0;i<R.Right.size();i++)
{
    sym[i] = SDTStack.top();
    SDTStack.pop();
}
temp=newTemp();gen(sym[1],temp,sym[0],sym[2]);SDTStack.push(temp);
break;
case 38 :
for (int i = 0;i<R.Right.size();i++)
{
    sym[i] = SDTStack.top();
    SDTStack.pop();
}
temp=newTemp();gen(sym[1],temp,sym[0],sym[2]);SDTStack.push(temp);
break;
case 39 :
for (int i = 0;i<R.Right.size();i++)
{
    sym[i] = SDTStack.top();
    SDTStack.pop();
}
temp=newTemp();gen(sym[1],temp,sym[0],sym[2]);SDTStack.push(temp);
break;
default:
break;
}
}
yySeuYacc.cpp (语法分析主程序)
#include "PDA.h"
#include <vector>
#include "yyseuLex.h"
#include "FileReader.h"
/*  minic.y(1.9) 17:46:21    97/12/10
*
*   Parser demo of simple symbol table management and type checking.

```

```

*/
#include<stdio.h>    /* for (f)printf() */
#include<stdlib.h>   /* for exit() */
extern int    line = 1; /* number of current source line */
extern int    seulex(); /* lexical analyzer generated from lex.l */
char*yytext; /* last token, defined in lex.l */ /* current symbol table,
initialized in lex.l */
char    *base;      /* basename of command line argument */
void
yyerror(char *s)
{
    fprintf(stderr,"Syntax error on line #%d: %s\n",line,s);
    fprintf(stderr,"Last token was \"%s\"\n",yytext);
    exit(1);
}
extern ruleSet rs;
void yyparse()
{
    FileReader fd;
    fd.ParseYFile("minic.y");
    vector<Symbol> s;
    ifstream fin(base);
    string str;
    while (1)
    {
        string t = seuLex(fin);
        if (t == "ERROR") break;
        if (t != "SPACE"){
            Symbol sym(t,1,seuLexLastLex);
            s.push_back(sym);}
    }
    Symbol sym("$",1);
    s.push_back(sym);
    fin.close();
    PDA pda(rs);
    pda.input(s);
    pda.readinTables("actionTable","gotoTable");
    pda.parse();
}

int
main(int argc,char *argv[])
{

```



```
base = "test.c";
```

```
yyparse();
```

```
}
```

(2) 根据使用说明中的方法，将若干其他辅助分析程序头文件和 `cpp` 文件组合为语法分析程序后运行：

采用的 `test.c` 如下：

```
int main()
```

```
{
```

```
    int k;
```

```
    float m;
```

```
    if (a == 10)
```

```
    {
```

```
        if (c == 8)
```

```
        {
```

```
            a = 6;
```

```
        }
```

```
    else
```

```
    {
```

```
        c = 9;
```

```
    };
```

```
    }
```

```
    else
```

```
    {
```

```
        b = 8;
```

```
    };
```

```
    v = i;
```

```
    k = k + c;
```

```
    m = m + 2;
```

```
    c = 2 - 3;
```

```
    if (m == 5)
```

```
    {
```

```
        k = 8;
```

```
    };
```

```
    return 0;
```

```
}
```

1、`test.c` 无语法错误

部分程序结果截图

```
E:\学习资料\大三下课程及资料\Compiler\Test\Debug\Test.exe
exp-->NUMBER
statement-->lexp ASSIGN exp
statements-->
statements-->statement SEMICOLON statements
block-->LBRACE var_declarations statements RBRACE
statement-->block
ifstatement-->ifhead statement
statement-->ifstatement
exp-->NUMBER
statement-->RETURN exp
statements-->
statements-->statement SEMICOLON statements
statements-->statement SEMICOLON statements
statements-->statement SEMICOLON statements
statements-->statement SEMICOLON statements
statements-->statement SEMICOLON statements
statements-->statement SEMICOLON statements
statements-->statement SEMICOLON statements
block-->LBRACE var_declarations statements RBRACE
fun_declaration-->type NAME LPAR parameters RPAR block
declaration-->fun_declaration
declarations-->
declarations-->declaration declarations
This sentence has been accepted
```

生成的中间代码为 intermediaCode.c, 如下:

```
0: (j==, 10, a, 2)
1: (j, _, _, 8)
2: (j==, 8, c, 4)
3: (j, _, _, 6)
4: (=, a, _, 6)
5: (j, _, _, 7)
6: (=, c, _, 9)
7: (j, _, _, 9)
8: (=, b, _, 8)
9: (=, v, _, i)
10: (+, T1, c, k)
11: (=, k, _, T1)
12: (+, T2, 2, m)
13: (=, m, _, T2)
14: (-, T3, 3, 2)
15: (=, c, _, T3)
16: (j==, 5, m, 18)
17: (j, _, _, 20)
18: (=, k, _, 8)
19: (j, _, _, 0)
2、test.c 存在语法错误
（第 20 行缺少一个分号）
int main()
{
    int k;
    float m;
```

```

if (a == 10)
{
    if (c == 8)
    {
        a = 6;
    }
    else
    {
        c = 9;
    };
}
else
{
    b = 8;
};
v = i
k = k + c;
m = m + 2;
c = 2 - 3;
if (m == 5)
{
    k = 8;
};
return 0;
}

```

程序运行结果：

```

E:\学习资料\大三下课程及资料\Compiler\Test\Debug\Test.exe
statements-->
statements-->statement SEMICOLON statements
block-->LBRACE var_declarations statements RBRACE
statement-->block
statement-->ifstatement ELSE statement
statements-->
statements-->statement SEMICOLON statements
block-->LBRACE var_declarations statements RBRACE
statement-->block
ifstatement-->ifhead statement
var_declarations-->
var-->NAME
lexp-->var
exp-->NUMBER
statement-->lexp ASSIGN exp
statements-->
statements-->statement SEMICOLON statements
block-->LBRACE var_declarations statements RBRACE
statement-->block
statement-->ifstatement ELSE statement
var-->NAME
lexp-->var
ERROR!
More information in ParsingLog.txt

```

生成的 ParsingLog 文件：

ERROR!

Reader at : k

Error info:Shift into error state!

state stack:

48

54

39

57

40

32

30

PD stack:

NAME

ASSIGN

lexp

SEMICOLON

statement

var_declarations

标明在读到变量 **k** 之前遇到了错误，即在 **k** 之前缺少分号

6 课程设计总结（包括设计的总结和需要改进的内容）

本次实验比较完整地实现了全部要求，通过本次实验对 LR 文法和下推自动机有了更深刻的认识，加强了编码及调试的能力。在调试 LR 文法的过程中，因为中间过程和数据结构比较复杂，充分意识到了一个好的思路和设计可以减少很多不必要的问题。完成代码过程中也应当注意一些用于调试的接口函数的实现，这样可以极大地方便调试过程。

我们的 SEUYacc 仍然有些问题需要解决，

第一，部分代码不够简洁。因为部分代码在编写过程中思路不太清晰，改了又改，最后 debug 时出了问题又调整了，导致有一些冗余代码。如果有时间重构一遍，性能和代码可读性还能更好。

第二，在冲突处理上，默认采用对于规约冲突采用先规约序号小的项，移进规约冲突采用先移进的策略。仅提供了全局的结合性以及优先级的控制，没提供对于单条规则的例外的冲突处理功能。

语法分析程序生成和四元式生成部分：

总结：这部分主要完成了 yySeuYacc 程序的生成，并构造了辅助分析的下推自动机配合 LALR 部分生成的两个表格进行表格制导的语法分析。同时，在生成了四元式生成的辅助程序，可以根据用户预定义的语义动作进行语义分析，并提供了语义栈、真链栈、假链栈辅助进行分析。针对分析失败（即输入程序有语法错误）的情况，进行了简单的错误检查，并输出错误信息和错误时的堆栈信息。

需要改进的部分：

（1）语义动作程序部分不能形式化表示，对于用户编写其他 yacc 程序的语义动作时有一定的困难（需要考虑语义栈、真链假链的填入弹出、回填等操作

（2）语义动作部分的函数并未完全封装，调用的时候会有困难（例如 backpatch 函数、merge 函数）

（3）分支语句没有进行并链，有可能产生一些问题。

（4）错误分析程序还不完善，目前只能输出一些简单类型的错误，还不能分析错误原因。例如上文测试样例中，缺少分号只能提示在分号后的一个字符处有问题，不能提示是缺少分号导致的。

7 教师评语

签名：_____

附：包含源程序、可运行程序、输入数据文件、输出数据文件、答辩所做 PPT 文件、本设计报告等一切可放入光盘的内容的光盘。