编译原理实验二

实验报告

151250100

刘宇翔

实验目的：

结合第一次实验一起，编写一个类似 yacc 的程序，可以自己自定义一系列 CFG 上下文无关文法，用来分析输入的文件。具体采用了由 cfg 到 LR(1)的方法来分析字段。程序模拟了如何将一个 cfg 文法转化为一个 LR(1)分析表，然后通过分析表很简单的便能够将输入的字段匹配相应的文法。

内容描述：

这一次实验我结合了上一次实验的内容一起，模拟了一个从 input.cpp 文件,lex 文件，yyac 文件再到最后导出的分析结果所有内容整合在一起的一个程序。

此程序用 c++实现，输入的内容由三个部分组成接下来将介绍这三个部分。

第一个部分为一个自定义的类 Lex 文件。文件中具体内容可以参考第一次实验的内容。在这里将第一次实验的文档也复制到了 regex 目录下。这里将 lex 文件的内容作为第一部分输入进行处理，从而获取各个 token 的属性名，为接下来处理 cfg 时匹配生成式中的终结字符串。

第二个部分为一个自定义的类 yacc 文件。该文件的输入同样划分为两个部分，两个部分通过”$$”符号划分。第一个部分为自定义的 cfg(上下文无关文法），在我自己的测试中主要定义了两个 cfg，在 input/目录下的 cfg1 和 cfg2，其中 cfg2 是不包含移入-规约冲突的 cfg，cfg1 是包含了移入规约冲突的 cfg。具体格式可以参考 cfg1 与 cfg2。第二部分为定义终结字符串的优先级和是否是左结合

如：

$$

{+,L}|{-,L}

{*,L}|{/,L}

则定义了两个等级的优先级，低等级的为+，-符号，两个符号都是左结合，(定义中 L 代表左，R 代表右)

高等级为*，/符号，两个符号同样也是左结合

第三个部分为需要分析的文本，如输入 3+5*6 的话。便会先通过 Lex 文件转化为

(num,3)(+)(num,5)(*)(num,6)

然后根据 cfg 如：

E->num 等通过 lr（1）分析表进行分析判断是否符号文法。暂时还没有加入对于 token 内数值的计算(包含了第 5 章的内容)

由于 yyac 文件和 lex 文件全为自己定义，所以定义了的语法并不是很完善，如果想要分析复杂的文本可自行修改 lex/mylex.lex 与 input/cfg1 来达到提高可分析的文本的复杂度。.

方法：

由于 lex 的实现为第一次实验内容，这里不做描述，可以查看 regex/下的实验报告。由于目标是将一段普通的代码先转化为 token。再将 token 中的属性名对应于 cfg 中的各个终结字符来分析是否相符。具体分析表采用 lr（1）的形式。分析 cfg 产生的 lr1 的状态中只存储了核心项，因为非核心项可以说是同样的，所以省略节省内存。同时对于 lr(1)分析表中的冲突情况，处理了规约-移入冲突。这是通过运算符的优先级和左右结合性来解决的。如果有终结符号没有定义，则自动默认为最低优先级并且左结合。

总的来说，就是模仿书上 cfg->lr(1)的流程，参照 yyac 文件的输入，实现了这一些过程，并解决了部分二义性的状况。

假设：

假定所有当做lex的输入文件语法上都是正确的。同时正则表达式只支持
^, $, +, [,], *, |, ?, (,)，-(不支持[^])
用来分析的文件可以为 c++或者 java
同时 cfg 文件中的内容定义按照格式，为每个字段中间用空格隔开，同时终结符号如果是未定义的则用单引号扩起。已经声明的终结符号(在 lex 中定义好的)则不用单引号。

Related FA descriptions：

下面是我自己定义的类 lex 文件

```
ws  {delim}+
letter  [A-Za-z]
digit   [0-9]
id  {letter}({letter}|{digit})*
number  {digit}+(.{digit}+)?(E([+-])?{digit}+)?
punctuation \(|\)|\{|\}|\[|\]|\.|\,|\:|\;
compareop   <|>|<=|>=|<>|==
basicop \+|\-|\*|\/
sumop   {basicop}+=
incrementop \+\+|\-\-
logicop \|\||!|&&|\||&|\^
scopeop \:\:
operator    <<|>>
op  {punctuation}|{basicop}|{compareop}|{sumop}|{incrementop}|{logicop}|
type    int|double|float
ioclass cout|cin
$$
{ws}    DEFAULT
NULL    null
,   split
;   delimiter
{head}  include_head
int main\(\)    start
for for
endl    iostring
if  IF
else    ELSE
std StandardScope
{ioclass}   ioclass
\{  {
\}  }
\(  (
\)  )
{scopeop}   scope_op
{punctuation}   punctuation
{compareop} compare_op
{sumop} sum_op
{incrementop}   increment_op
{logicop}   logic_op
{operator}  io_op
\-|\+   plus_minus
[*/]    muti_divide
\=  =
{type}  type
{id}    var
{number}    num
```

下面是我自己定义的类 yyac 文件

```
S->start BLOCK


BLOCK->'{' Content '}'
Content->Content STATEMENT | STATEMENT
STATEMENT-> B|D|F|IO|IFELSE

B->type var '=' C
C->E delimiter| var '=' C

D->type var delimiter| type var E' delimiter
E'->split var | split var E'

E->E plus_minus E | E muti_divide E | '(' E ')' | num | var

F->for '(' forstate ')' BLOCK
forstate-> B var compare_op num delimiter increment_op var

IO->IOCLASS IO_VALUE delimiter
IOCLASS->StandardScope scope_op ioclass | ioclass
IO_VALUE->io_op io_var | IO_VALUE io_op io_var
io_var->E|iostring

IFELSE->IF '(' judgestatement ')' BLOCK | IF '(' judgestatement ')' BLOCK ELSE BLOCK
judgestatement->num|judgevar compare_op judgevar
judgevar->num|var

$$
{io_op,L}
{plus_minus,L}
{muti_divide,L}
{=,R}
{delimiter,L}
```

相关重要的数据结构：

由于文件中没有注释，所以部分结构在这边进行解释。

主要的类有 yyac，cfg，LR_ParsingTable，LR_Status，LR_Generation，State，下面将作出描述

```
typedef string terminal; //终结符号
typedef string non_terminal; //非终结符号
typedef pair<string,bool> symbol; //符号(true为终结符号，false为非终结)
typedef pair<int,bool> lr_skip;//lr分析表中每一栏的属性（bool值为是否规约，int值为跳转的路径）
typedef pair<int, bool> op_property;//操作符的优先级(bool为是否是左结合)
const int ERROR = -1;//代表空路径
const string split_symbol = "->";//分割字符
```

```
struct Generation
{
    non_terminal left_value;//生成式左值 为E->T*B中的E
    Formula right_value;//生成式右端 如 E * E等
    int number;//生成式号
    Generation(const non_terminal & _l,const string & _r, int _num) :left_value(_l),
right_value(_r), number(_num) {}
    friend ostream & operator << (ostream & out, const Generation & g);
};
struct First
{
    bool use;//代表这个非终端符号的first集合是否被探索
    non_terminal val;//非终端符号
    set<non_terminal> nonterminal_set;//该非终端符号first集合中的非终端符号
    set<terminal> terminal_set;//该非终端符号的first集合
    First() :val(), nonterminal_set(), terminal_set(),use(false) {}
    First(const non_terminal & _val) :val(_val), nonterminal_set(), terminal_set(),
use(false) {}
    bool insert(const non_terminal & value);
};
struct ClosedSet
{
    set<terminal> inner_forward;//一个非终端符号的闭合集中产生的向前看字符
    non_terminal val;//非终端符号
    vector<Generation> right_value;//该非终端符号对应的所有的生成式
    set<symbol> directions;//方向
    ClosedSet(const non_terminal & _val = "") :val(_val), right_value(), directions(),
inner_forward() {};
    void push_back(const Generation & t)
    {
        right_value.push_back(t);
        directions.insert(t.right_value.sym[0]);
    }
};

class CFG
{
private:
    vector<Generation> __cfg_expressions;//一个cfg中的所有表达式
    vector<First> __first_nonterminal;//一个cfg中所有非终端符号的first集合
    unordered_map<terminal, int> __terminal_list;//一个cfg中的所有终结符号
    unordered_map<non_terminal, int> __nonterminal_list;//cfg中的所有非终结符号
    vector<ClosedSet> __closed_set;//cfg中的所有终结符号对应的闭合集合
    unordered_map<terminal, op_property> op;//在yyac文件中定义的cfg中所有终结符号的优先
```

```
级和左右结合性
    LR_ParsingTable __mytable;//cfg对应的lr(1)分析表
    int __terminal_num;//cfg中总共的终结符号个数
    int __nonterminal_num;//cfg中所有非终结符号的个数
    void analysis_first(First & sym);//分析first集合
    vector<int> state_stack;//分析文本时的状态栈
    vector<symbol> symbol_stack;//分析文本时的符号栈
    void printNowState(ofstream & out, const vector<Token> & str,int index);
public:
    CFG(ofstream & out,const vector<string> & cfgs,const vector<string> & _op);
    friend ostream & operator << (ostream & out, const CFG & cfg);
    friend void LR_ParsingTable::generate(ofstream & out,const non_terminal & start,
CFG & cfg);
    friend void LR_ParsingTable::analysis_status(int index, CFG & cfg);
    friend bool LR_ParsingTable::analysis_generation(int & now_num,
unordered_map<symbol, LR_Status, pairhash> & directions, LR_Generation & g, LR_Status &
status, CFG & cfg);
    friend void LR_ParsingTable::analysis_closedset(const set<terminal> & ori_forward,
const non_terminal & nt, unordered_map<non_terminal, ClosedSet> & closedset, CFG &
cfg);
    friend int LR_ParsingTable::move_step(CFG & cfg, const non_terminal & next_sym);
    friend lr_skip LR_ParsingTable::next_step(CFG & cfg, const terminal & next_sym);
    set<terminal> getFirst(const symbol & sym);//获得first集合
    void analysis(ofstream & out, const vector<Token> & str);//分析文本
};

/*
添加hash<T1,T2>的结构
*/
struct pairhash
{
    template<class T1, class T2>
    size_t operator() (const pair<T1, T2> &x) const
    {
        hash<T1> h1;
        hash<T2> h2;
        return h1(x.first) ^ h2(x.second);
    }
};

struct LR_Generation
{
    Generation generation;//LR生成式对应的生成式
    set<terminal> forward;//LR生成式的向前看字符
```

```
    int dot_index;//点的位置
    LR_Generation(const Generation & _g,int _index=0) :generation(_g),
forward(),dot_index(_index) {};
    LR_Generation(const non_terminal & _l, const string & _r, int _num,int _index =
0) :generation(Generation(_l, _r, _num)), forward(), dot_index(_index) {};
    bool operator ==(const LR_Generation & _val);
    friend ostream & operator << (ostream & out, const LR_Generation & g);
};
struct LR_Status
{
    vector<LR_Generation> raw;//LR状态的核心生成式
    int status_num;//状态号
    terminal * now_symbol;//如果LR状态可以规约，则now_symbol指向规约式子中的最后一个终
结字符(用来比较优先级和左右结合性)
    vector<lr_skip> terminal_skip;//终结符号的跳转
    vector<int> non_terminal_skip;//非终结符号的跳转
    LR_Status(int _num = -1) :raw(), status_num(_num), terminal_skip(),
non_terminal_skip(), now_symbol(nullptr) {}
    void push_back(const LR_Generation & _val);
    bool operator ==(const LR_Status & _val);
    friend ostream & operator << (ostream & out, const LR_Status & LR_Status);
};

class CFG;
class LR_ParsingTable
{
private:
    friend class CFG;
    enum {default_dot_index=0};
    vector<LR_Status> __state;
    int __number;
    void analysis_closedset(const set<terminal> & ori_forward, const non_terminal & nt,
unordered_map<non_terminal, ClosedSet> & closedset, CFG & cfg);//分析闭合集合并加入状态
    bool analysis_generation(int & now_num, unordered_map<symbol, LR_Status, pairhash>
& directions,LR_Generation & g, LR_Status & status, CFG & cfg);//分析生成式并加入状态
    void analysis_status(int index, CFG & cfg);//分析状态
public:
    LR_ParsingTable() : __number(0), __state() {};
    void generate(ofstream & out,const non_terminal & start,CFG & cfg);//生成LR分析表
    lr_skip next_step(CFG & cfg, const terminal & next_sym);//下一次跳转
    int move_step(CFG & cfg, const non_terminal & next_sym);//移动
};

class Yacc
```

```
{
private:
    const string path;//输出文件的路径
    CFG * __cfg;
    const string filename;//输入的类yyac文件
public:
    static Lex lex;//进行匹配的lex文件
    Yacc(const string & path);
    void analysis(const string & path);//要分析的文本的路径
};
```

核心算法：

核心算法主要在生成LR(1)分析表中计算闭包集合和计算某个方向的集合和向前看字符。

```
void analysis_closedset(const set<terminal> & ori_forward, const non_terminal & nt, unordered_map<non_terminal, ClosedSet> & closedset, CFG & cfg);//分析闭合集合并加入状态
bool analysis_generation(int & now_num, unordered_map<symbol, LR_Status, pairhash> & directions,LR_Generation & g, LR_Status & status, CFG & cfg);//分析生成式并加入状态
void analysis_status(int index, CFG & cfg);//分析状态
```

analysis_status为分析一个状态。首先会为该状态的建立好表格，初始化为(ERROR,FALSE)。然后遍历该状态的所有核心生成式，同时该方法中有两个map。

```
unordered_map<symbol, LR_Status,pairhash> directions;
unordered_map<non_terminal,ClosedSet> closedset;
```

第一个为对应的该状态跳转到下一个状态的方向和下一个状态的集合。
第二个为对应的该状态内的所有非终结符号的闭合集合。
对于核心状态的跳转会添加到directions中的对应state中。
同时会将当前点为非终结符号的情况添加到closedset中。
接下来会分析所有closedset中的生成式，并重复这个步骤，直到没有新的非终结符号的添加或是对应的闭合内的向前看符号的变化。
接下来会添加所有closedset中的生成式到对应的下一个跳转的状态中。
最后会分析所有跳转的状态和之前的状态是否相等，如果不相等则生成新的状态。
并且在填入跳转指示中会处理规约-移入冲突。判断对应的符号与当前状态的最晚终结符号的优先级的比较达到处理的效果。

analysis_generation为分析表达式，如果该表达式的节点已经指到最末端。则会将所有向前看符号对应的跳转指示栏填充为该表达式，否则添加到对应的下一个状态中。

analysis_closedset为分析闭合集合以及添加新的闭合集合。将统计出该状态下的所有由核心表达式生成出来的初始表达式和向前看符号。
由于部分实现过于繁琐，长度不短这里便只做描述而不将代码复制到这边。

Use cases on running

输入的lex文件路径为lex/mylex.lex

```
ws  {delim}+
letter  [A-Za-z]
digit   [0-9]
id  {letter}({letter}|{digit})*
number  {digit}+(.{digit}+)?(E([+-])?{digit}+)?
punctuation \(|\)|\{|\}|\[|\]|\.|\,|\:|\;
compareop   <|>|<=|>=|<>|==
basicop \+|\-|\*|\/
sumop   {basicop}+=
incrementop \+\+|\-\-
logicop \|\||!|&&|\||&|\^
scopeop \:\:
operator    <<|>>
op  {punctuation}|{basicop}|{compareop}|{sumop}|{incrementop}|{logicop}|
type    int|double|float
ioclass cout|cin
$$
{ws}    DEFAULT
NULL    null
,   split
;   delimiter
{head}  include_head
int main\(\)    start
for for
endl    iostring
if  IF
else    ELSE
std StandardScope
{ioclass}   ioclass
\{  {
\}  }
\(  (
\)  )
{scopeop}   scope_op
{punctuation}   punctuation
{compareop} compare_op
{sumop} sum_op
{incrementop}   increment_op
{logicop}   logic_op
{operator}  io_op
\-|\+   plus_minus
[*/]    muti_divide
\=  =
{type}  type
{id}    var
{number}    num
```

输入的类 yyac 文件路径为 input/cfg1

```
S->start BLOCK


BLOCK->'{' Content '}'
Content->Content STATEMENT | STATEMENT
STATEMENT-> B|D|F|IO|IFELSE

B->type var '=' C
C->E delimiter| var '=' C

D->type var delimiter| type var E' delimiter
E'->split var | split var E'

E->E plus_minus E | E muti_divide E | '(' E ')' | num | var

F->for '(' forstate ')' BLOCK
forstate-> B var compare_op num delimiter increment_op var

IO->IOCLASS IO_VALUE delimiter
IOCLASS->StandardScope scope_op ioclass | ioclass
IO_VALUE->io_op io_var | IO_VALUE io_op io_var
io_var->E|iostring

IFELSE->IF '(' judgestatement ')' BLOCK | IF '(' judgestatement ')' BLOCK ELSE BLOCK
judgestatement->num|judgevar compare_op judgevar
judgevar->num|var

$$
{io_op,L}
{plus_minus,L}
{muti_divide,L}
{=,R}
{delimiter,L}
```

输入的分析文本的内容，路径为input/input.cpp

```cpp
#include<iostream>
int main()
{
    double temp1 = 3 + 5 * 6 - 7 / 8;
    int i1, i2, i3;
    int i4 = i1 = i2 = i3 = 3 - 7 * (5 + 6);
    for (int i = 0; i < 10; ++i)
    {
        if (i < 5)
        {
            std::cout << i << endl;
        }
        else
        {
            std::cout << i + 5 << endl;
        }
    }
}
```

输出结果：

（由于分析的文本较为复杂，输出的结果大小都有一定规模，所以只截取部分截图，具体内容在output中查看）

输出的LR分析表 input/table_cfg1部分截图

```
*************************************************************
state:0 :
-1.init :{S,nonterminal}
,dot=0,[null,]
[2,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
*************************************************************
*************************************************************
state:1 :
-1.init :{S,nonterminal}
,dot=1,[null,]
[-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
*************************************************************
*************************************************************
state:2 :
0.S :{start,terminal}{BLOCK,nonterminal}
,dot=1,[null,]
[-1,s][4,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s
-1,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
*************************************************************
*************************************************************
state:3 :
0.S :{start,terminal}{BLOCK,nonterminal}
,dot=2,[null,]
[-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,s][-1,
-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
*************************************************************
*************************************************************
```

输出的对于cfg1的分析文本input/result_cfg1

```
stack:{0,}
    symbol:
    input:(start,int main()) ({,{) (type,double) (var,temp1) (=,=) (num,3) (plus_minus,+) (num,5) (m
stack:{0,2,}
    symbol:start
    input:({,{) (type,double) (var,temp1) (=,=) (num,3) (plus_minus,+) (num,5) (muti_divide,*) (num,
stack:{0,2,4,}
    symbol:start {
    input:(type,double) (var,temp1) (=,=) (num,3) (plus_minus,+) (num,5) (muti_divide,*) (num,6) (pl
stack:{0,2,4,6,}
    symbol:start { type
    input:(var,temp1) (=,=) (num,3) (plus_minus,+) (num,5) (muti_divide,*) (num,6) (plus_minus,-) (n
stack:{0,2,4,6,18,}
    symbol:start { type var
    input:(=,=) (num,3) (plus_minus,+) (num,5) (muti_divide,*) (num,6) (plus_minus,-) (num,7) (muti_
stack:{0,2,4,6,18,23,}
    symbol:start { type var =
    input:(num,3) (plus_minus,+) (num,5) (muti_divide,*) (num,6) (plus_minus,-) (num,7) (muti_divide
```

```
stack:{0,2,4,5,14,20,32,52,74,8,19,27,49,67,87,74,90,}
    symbol:start { Content for ( forstate ) { IF ( judgestatement ) BLOCK ELSE { Content
    input:(},}) (},}) (},})
stack:{0,2,4,5,14,20,32,52,74,8,19,27,49,67,87,74,90,100,}
    symbol:start { Content for ( forstate ) { IF ( judgestatement ) BLOCK ELSE { Content }
    input:(},}) (},})
stack:{0,2,4,5,14,20,32,52,74,8,19,27,49,67,87,97,}
    symbol:start { Content for ( forstate ) { IF ( judgestatement ) BLOCK ELSE BLOCK
    input:(},}) (},})
stack:{0,2,4,5,14,20,32,52,74,10,}
    symbol:start { Content for ( forstate ) { IFELSE
    input:(},}) (},})
stack:{0,2,4,5,14,20,32,52,74,7,}
    symbol:start { Content for ( forstate ) { STATEMENT
    input:(},}) (},})
stack:{0,2,4,5,14,20,32,52,74,90,}
    symbol:start { Content for ( forstate ) { Content
    input:(},}) (},})
stack:{0,2,4,5,14,20,32,52,74,90,100,}
    symbol:start { Content for ( forstate ) { Content }
    input:(},})
stack:{0,2,4,5,14,20,32,52,73,}
    symbol:start { Content for ( forstate ) BLOCK
    input:(},})
stack:{0,2,4,5,13,}
    symbol:start { Content F
    input:(},})
stack:{0,2,4,5,17,}
    symbol:start { Content STATEMENT
    input:(},})
stack:{0,2,4,5,}
    symbol:start { Content
    input:(},})
stack:{0,2,4,5,16,}
    symbol:start { Content }
    input:
stack:{0,2,3,}
    symbol:start BLOCK
    input:
stack:{0,1,}
    symbol:S
    input:
accept
```

i) Problems occurred and related solutions

问题1：对于每一个lr分析表中的状态，如果记录全部lr生成式则会导致内存占用量太大

解决方案：

只记录核心生成式，其余的生成式通过分析闭合集合来达成。

问题2：

对于二义性文法处理以及lr分析表中会存在的冲突如何取舍。

解决：

采用定义符号的优先级和左右结合性处理

问题3：

生成lr分析表时，对于重复的状态应当进行排除和用之前的状态进行替代。

解决：

对于LR_Status重载==号。但是这种方法需要比较每一个LR_Status，这导致时间复杂度会很高。但是暂时不清楚有什么更便捷的做法，因为LR_Status中有LR生成式的集合等各个属性都不方便比较。所以不好重载<号，同时也不方便写一个hash函数作业键值来比较。所以只能采用效率最低的逐个比较方法。

j) Your feelings and comments

通过这次实验，我结合了第一次实验的内容和这一次实验一起，构成了一个简单的可以做到分析文本的程序。让我明白了编译原理从词法分析到语法分析的过程以及语法分析中如何将cfg文法转

化为相应的LR(1)分析表的过程。通过这两次的实验，让我更加深刻的理解到了编译原理开学到现在所讲的内容。通过类似lex和yyac文件的实现，也大致明白了lex文件和yyac文件的工作流程。