Compiler Principles Lab-2

wacky6 / Jiewei Qian (C)

This is the original lab report. I hope it can help you understanding wtf this (supposedly interesting but very badly-taught) lesson is talking about.

Copy with caution if your programming skill is weak or having no experience with non-VS environment, or your teacher will realize that you “learn” from someone else’s report.

: (

一、 实验原理

语法分析是编译系统的重要部分，是词法分析的下一个环节，它接受词法分析器输出的符号串（Tokens），按照一定的规则，将符号串识别有意义的结构（指令＋操作数）。

实验要求：

- 检测左递归,如果有则进行消除
- 求解 FIRST 集和 FOLLOW 集
- 构建 LL(1)分析表
- 构建 LL 分析程序，对于用户输入的句子，显示出分析过程。

思路：整体上按照书上的分析器构造，但在设计上有一些改动：

1. A =>* epsilon 的处理：

   在构造 First/Follow 与 LL 分析表的时候，需要判定非终结符能否推到出空（epsilon），因此在构建 First/Follow 之前，现对所有非终结符进行判定，用 epsilon_closure 记录结果方便以后使用。对应代码中对 produceEpsilonClosure

2. 规则或的处理：

   例如 E => E+T | T

   这实际上表示两条推倒规则，只要任意一条满足，是典型的或关系。显然，在程序内部用’|’字符分割两条规则是可行的，但这在代码编写上会引起很多不必要的麻烦，这样做也不能处理规则中含有’|’字符的情形。因此，程序采用 multimap（一个键可以对应多个值）保存规则，自然就能够表示非终结符的多种推倒，在程序实现上也相当简便。

3. 消除左递归过程中生成符号的表示：

   因为需要分析的文法比较简单，为了方便调试，采用大写字母表示非终结符。利用 ASCII 码有效值为 7 位二进制位的特性，将第 8 位作为 flag 表示该符号是否为生成符号（比如 E 对应 E’，与书上的内容一致）。在输出时，用 PrintSym 函数将符号转换为可打印字符串。

First 集：表示某非终结符可能的第一个符号。从规则 A 的第一个符号开始处理，直到当前符号不能推倒出空（符号为终结符、或非终结符不属于 epsilon_closure）。如果符号为终结符，将符号加到 First(A)中。如果符号 B 为非终结符，将 First(B)加到 First(A)中，如果 B 可以推倒到空，继续处理下一个符号。

First(终结符)为终结符本身

Follow 集：表示某非终结符后可以紧跟着的下一个符号。可以采取以下方式求解：

+= 表示集合取并集

- 对开始符号 S，Follow(S) = #;
- 对 A=>aBb，Follow(B) += b
- 对 A=>aBC，Follow(B) += First(C)
- 对 A=>aB，Follow(A) += Follow(B)
- 对 A=>aBC、C=>*epsilon，Follow(A) += Follow(B)

First/Follow 集的求解采用了离散数学中闭包的概念。在实现上，为了避免递归操作，采用循环求解，直到 First/Follow 集不再扩大。这种方式简化了代码实现。

LL 分析总控程序，LLParse：

根据分析栈顶元素与当前扫描符号进行对应操作：

- 栈顶、符号均为终结符#：扫描成功
- 栈顶、符号相同，不为#：表示当前符号得到匹配，出栈并将扫描位置后移
- 栈顶为非终结符，查找分析表 LLTable[stack.top()][token]，如果分析表有规则，则按照这个规则推倒，把规则元素反向入栈。如果没有规则，表示输入串不匹配，此时提示出错。

LL 分析的思路是按照根据当前扫描到的符号，推测整个符号串的语法形式，分析器现推测出一个语法结构，然后用这个结构与输入串去进行匹配。显然，这样的方式只能满足一部分语言的需求，对复杂的语言就难以实现匹配了。

主程序：

removeDirectLeftRecuesion() 消除左递归

initializeFirstFollow() 初始化 First、Follow 集

produceEpsilonClosure() 创建 epsilonClosure，方便判定 A=>*epsilon

produceFirst() 构建 First 集

produceFollow() 构建 Follow 集

produceLLTable() 创建 LL 分析表

LLParse(LLTable, “符号串”) 根据生成的 LLTable 分析符号串

程序采用 C++编写，大量使用了 STL 模版，同时使用了函数对象，需要 C++11 编译器。

源代码：

```
/****************
 * lab-compiler/parser.cpp
 *
 * flag: -std=c++11
 *
 ***************/

#include <set>
#include <map>
#include <iostream>
#include <cstdio>
#include <iomanip>
#include <algorithm>
#include <cstdlib>
#include <cctype>
#include <string>
#include <stack>
#include <functional>
using namespace std;

typedef char Symbol;
// gSym: generated symbol, used to remove direct left recursion
// sSym: source / original symbol, used to print pretty rule table
```

```
#define epsilon          ('$')
#define terminator       ('#')
#define is_term(sym)      (!isupper(sSym(sym)))
#define is_non_term(sym)   (isupper(sSym(sym)))
#define is_gSym(sym)      (sym&0x80)
#define gSym(sym)         ((char)(sym|0x80))
#define sSym(sym)         ((char)(sym&0x7F))

template<typename F>
void for_each_keyed_value(multimap<Symbol,string> m, Symbol sym, F f) {
   for (auto i=m.begin(); i!=m.end(); ++i) {
      if (i->first == sym)
         f(i->second);
   }
}
template<typename F>
void for_each_kv(multimap<Symbol,string> m, F f) {
   for (auto i=m.begin(); i!=m.end(); ++i) {
      f(i->second, i->first);
   }
}
long numOfSymbols(map<Symbol, set<Symbol> > ff);
const char* PrintSym(Symbol sym);
const char* PrintRule(string rstr);
multimap<Symbol,string> removeDirectLeftRecuesion(multimap<Symbol,string> rules);
void produceFirst();
void produceFollow();
void produceEpsilonClosure();
void produceLLTable();
void LLParse(string lltable[256][256], string input);

multimap<Symbol, string> rules;
map<Symbol, set<Symbol> > first;
map<Symbol, set<Symbol> > follow;
map<Symbol, bool> epsilon_closure;  // eclosure[B]=true  if  B =>* epsilon
string ll_table[256][256];        // ll_table[NonTerm][Term] = rule
Symbol target;

const char* PrintSym(Symbol sym) {
   static char buf[256];
   if (sym==epsilon)
      sprintf(buf, "%s", "ε");
   else if (is_gSym(sym))
      sprintf(buf, "%c'", sSym(sym));
```

```
   else
      sprintf(buf, "%c", sym);
   return buf;
}
const char* PrintRule(string rstr) {
   static char buf[512];
   int pos = 0;
   for(int i=0; i!=rstr.size(); ++i)
      pos += sprintf(buf+pos, "%s", PrintSym(rstr[i]));
   buf[pos++] = 0;
   return buf;
}

const char* PrintParseStack(stack<Symbol> s) {
   static char buf[65536];
   int pos = 0;
   stack<Symbol> t;
   while (!s.empty()) {
      t.push(s.top());
      s.pop();
   }
   while (!t.empty()) {
      pos += sprintf(buf+pos, "%s", PrintSym(t.top()));
      t.pop();
   }
   return buf;
}

long numOfSymbols(map<Symbol, set<Symbol> > ff) {
   long sz = 0;
   for (auto it=ff.begin(); it!=ff.end(); ++it)
      sz += it->second.size();
   return sz;
}

void LLParse(string llt[256][256], string input) {
   printf("LL Parse:\n");
   stack<Symbol> stk;
   input = input+terminator;
   stk.push(terminator);
   stk.push(target);
   int pos  = 0;
   int step = 0;
   while (true) {
```

```
      // print parse process
      printf("%-3d   %-15s   %15s    ",
           ++step, PrintParseStack(stk), input.substr(pos).c_str()
      );
      if (stk.top()==terminator && input[pos]==terminator) {
        printf("SUCCESS\n");
        break;
      }
      if (stk.top()==input[pos] && stk.top()!=terminator) {
        stk.pop();
        ++pos;
        printf("\n");
        continue;
      }
      if (is_non_term(stk.top())) {
        Symbol sym = stk.top();
        string rule = llt[(unsigned char)sym][(unsigned char)input[pos]];
        if (!rule.length()) {
          printf("ERROR!\n");
          break;
        }
        stk.pop();
        if (rule[0]!=epsilon)
          for (auto it=rule.rbegin(); it!=rule.rend(); ++it)
            stk.push(*it);
        printf("%-2s => ", PrintSym(sym));
        printf(" %s\n",  PrintRule(rule));
        continue;
      }
    }
}


multimap<Symbol,string> removeDirectLeftRecuesion(multimap<Symbol,string> rules)
{
    set<Symbol> recursed;
    multimap<Symbol, string> ret;
    for_each_kv(rules, [&](string s, Symbol sym){
      if (s[0]==sym)
        recursed.insert(sym);
    });
    for_each_kv(rules, [&](string rule, Symbol sym){
      if (recursed.find(sym) != recursed.end()) {
        if (rule[0]==sym)
```

```
            ret.insert(make_pair(gSym(sym), rule.substr(1)+gSym(sym)));
         else
            ret.insert(make_pair(sym, rule+gSym(sym)));

      }else{
         ret.insert(make_pair(sym, rule));
      }
   });
   for_each(recursed.begin(), recursed.end(), [&](Symbol sym){
      ret.insert(make_pair(gSym(sym), string("")+epsilon));
   });
   return ret;
}

void produceEpsilonClosure() {
   // produce epsilon closure, it is used in production (3)
   for_each_kv(rules, [&](string rule, Symbol sym){ epsilon_closure[sym]=false; });
   for (int i=0; i!=first.size(); ++i)
      for_each_kv(rules, [&](string rule, Symbol sym){
         if (rule.length()==1 && rule[0]==epsilon) {
            epsilon_closure[sym] = true;
            return;
         }
         for (auto ich=rule.rbegin(); ich!=rule.rend(); ++ich) {
            if (is_term(*ich) || !epsilon_closure[*ich])
               break;
            epsilon_closure[*ich] = true;
         }
      });
}

/* assume first, follow are initialized! */
void produceFirst() {
   int before, after;
   do{
      before = numOfSymbols(first);
      set<Symbol> prodSyms;
      for_each_kv(rules, [&](string rule, Symbol sym){ prodSyms.insert(sym); });
      for_each(prodSyms.begin(), prodSyms.end(), [&](Symbol sym){
         for_each_keyed_value(rules, sym, [&](string rule){
            int pos = 0;
            while (1) {
               Symbol cur = rule[pos];
               if (is_term(cur))
```

```
                first[sym].insert(cur);
             if (is_non_term(cur))
                for_each(first[cur].begin(), first[cur].end(), [&](Symbol fsym){
                   first[sym].insert(fsym);
                });
             if (is_term(cur)) break;
             if (!epsilon_closure[cur]) break;
             if (pos==rule.size()-1) break;
          }
       });
     });
     after = numOfSymbols(first);
   }while(before != after);
}

void produceFollow() {
   // add # to target
   follow[target].insert(terminator);
   int before, after;
   do {
     before = numOfSymbols(follow);
     for_each_kv(rules, [&](string rule, Symbol sym){
       int len = rule.length();
       for (int i=0; i!=len; ++i) {
         Symbol cur  = rule[i];
         Symbol next = i+1<len ? rule[i+1] : 0;
         if (is_non_term(cur) && next) {
           // Follow production (2)
           if (is_term(next)) {
             follow[cur].insert(next);
           }else{
             for_each(first[next].begin(), first[next].end(), [&](Symbol fsym){
               if (fsym!=epsilon)
                  follow[cur].insert(fsym);
             });
           }
         }
         if (is_non_term(cur) && (!next || epsilon_closure[next])) {
           // Follow production (3)
           for_each(follow[sym].begin(), follow[sym].end(), [&](Symbol fsym){
             follow[cur].insert(fsym);
           });
         }
       }
```

```
    });
    after = numOfSymbols(follow);
  }while(before != after);
}

void produceLLTable() {
  #define IDX(ch) ((unsigned char)ch)
  #define LLT(_sym, _ch) (ll_table[IDX(_sym)][IDX(_ch)])
  for_each_kv(rules, [&](string rule, Symbol sym){
    set<Symbol> firstA;
    for (int i=0; i!=rule.length(); ++i) {
      Symbol a = rule[i];
      if (is_term(a))
        firstA.insert(a);
      if (is_non_term(a))
        firstA.insert(first[a].begin(), first[a].end());
      if (a==epsilon || (is_non_term(a) && epsilon_closure[a]))
        continue;
      break;
    }
    for_each(firstA.begin(), firstA.end(), [&](Symbol t){
      LLT(sym, t) = rule;
    });
    if (firstA.find(epsilon)!=firstA.end())
      for_each(follow[sym].begin(), follow[sym].end(), [&](Symbol t){
        LLT(sym, t) = rule;
      });
  });
}


int main() {
  #define RULE(sym,rule) rules.insert(make_pair<Symbol, string>(sym, rule));
  /* E  ->  E+T | T
   * T  ->  T*F | F
   * F  ->  (E) | i
   */
  RULE('E', "E+T");
  RULE('E', "T");
  RULE('T', "T*F");
  RULE('T', "F");
  RULE('F', "(E)");
  RULE('F', "i");
  target = 'E';
```

```
    rules=removeDirectLeftRecuesion(rules);

    // initialize first/follow
    for_each_kv(rules, [&](string rule, Symbol sym){
        first[sym]  = set<Symbol>();
        follow[sym] = set<Symbol>();
    });

    produceEpsilonClosure();
    produceFirst();
    produceFollow();
    produceLLTable();

    LLParse(ll_table, "i*(i+i)");
    LLParse(ll_table, "i+i*i");
    LLParse(ll_table, "i*+i");

}
```

四、实验小结

思考题：能否不采用预先定义的文法,而是允许用户输入文法的若干规则,生成文法?

显然，这个功能是可以实现的，只需要处理好文字规则到内部规则（计算机表示）的转换就可以了。按照本次实验的代码，将 RULE()定义部分改为一个规则生成函数。这里需要注意判定输入文法是否为 LL(1)文法，否则在分析表生成过程中会出现问题。

通过本次实验，加深了对 LL(1)分析的理解。同时，感受手动编写分析器是一个耗时的繁琐过程。事实上也确实如此，当语言规则变复杂时，需要一个完善的框架才能正确的写出分析程序。现在已有一些开源的语法分析器生成方案（如 bison、flex），这些软件可以根据输入自动生成语法分析器（LL、LR 均可）。利用这些软件可以快速构建所需要的词法、语法分析程序，缩短了软件开发时间。

在编译系统设计上，语法、词法、语义分析是互相独立但不可分割的，三者必须同时使用才能对语言进行解析。三者互相独立简化了各自的实现，也方便了功能的扩展。在实际使用中，手写分析器是一件繁琐的事情，如果语言满足一定条件，最好用语法分析器声称程序生成代码。