北京理工大学

编译原理课程设计实验报告

实验三 语法分析器设计

姓　名：廖汉龙

学　号：1120151880

学　院：计算机学院

班　级：07111507

邮　箱：liamliaohl@gmail.com

2018 年 5 月 14 日 星期一

目录

实验报告与代码链接：https://github.com/HanlongLiao/Course/tree/master/Compiling

一、实验目的

1.通过基于 BIT-Mini-CC 编译原理框架，在上一个实验——C 语言子集词法分析器的基础上，实现 C 语言子集的语法分析器

2.通过动手实践，对语法分析等理论知识有一个更加深刻的理解与认识，熟练掌握语法分析器的实现过程。

3.通过本实验，逐步增强自主动手实践的能力。

二、实验内容

在 BIT-MiniCC 框架下，可以按照如下步骤完成语法分析实验：

1.参照 实验指导书给出的文法， 扩充定义自己希望实现的 C 语言语法子集。

参考文法只给出了函数定义以及简单的表达式相关的文法。局部变量声明、分支语句以及循环语句等需要自己进行扩充。主要采用自顶向下的分析方法时，不能有左递归，避免文法产生式的多个候选式存在公共因子。如果出现左递归或者公共因子，则可以通过文法等价变换进行消除。

2.从递归下降分析方法、 LL(1)分析方法、算符优先分析方法、 LR 分析方法中选择一种，作为 BIT-MiniCC 框架中设计并实现语法分析器的指导方法。

3.构建语法分析器， BIT-MiniCC 中已经定义了一个类 CMyMiniCCParser，并定义了 run 方法，实验以该类为主进行。语法分析的输入为词法分析的出，因此语法分析器首先要读入 xxx.token.xml 文件；在分析的过程中构建语法树；

4.将语法树输出为 xml 文件；目前已有的分析方法包括递归下降、 LL(1)和 LR 等多种分析方法，可以选择其中的一种实现，递归下降更为直观。例如，基于框架自带的文法及其对应的实现，当输入为如下的程序时：

该实验选择 C 语言的一个子集， 基于 BIT - MiniCC 构建 C 语法子集的语法分析器，该语法分析器能够读入 XML 文件形式的属性字符流，进行语法分析并进行错误处理，如果输入正确时输出 XML 形式的语法树，输入不正确时报告语法错误。需要说明的是，能够分析的输入程序依赖于选用的语法子集，而输出的语法树的结构又与文法的定义密切相关。

三、实验过程

3.1 对于文法的改写与扩充

在实验指导书中给出的基本文法如下图-1 所示：

```
CMPL_UNIT     : FUNC_LIST
FUNC_LIST     : FUNC_DEF FUNC_LIST | ε
FUNC_DEF      : TYPE_SPEC ID ( ARG_LIST ) CODE_BLOCK
TYPE_SPEC     : int | void
PARA_LIST     : ARGUMENT | ARGUMENT , PARA_LIST | ε
ARGUMENT      : TYPE_SPEC ID
CODE_BLOCK    : { STMT_LIST }
STMT_LIST     : STMT STMT_LIST | ε
STMT          : RTN_STMT | ASSIGN_STMT
RTN_STMT      : return EXPR
ASSIGN_STMT   : ID = EXPR
EXPR          : TERM EXPR2
EXPR2         : + TERM EXPR2 | - TERM EXPR2 | ε
TERM          : FACTOR TERM2
TERM2         : * FACTOR TERM2 | / FACTOR TERM2 | ε
FACTOR        : ID | CONST | ( EXPR )
```

图-1

本实验中，我采用了 LL(1) 的方法，LL(1)文法的特点如下：

（1）LL(1) 文法不具有二义性；

（2）若一文法中非终结符号含有左递归，则必然是非 LL(1)文法。

（3）非 LL(1)文法语言是存在的。

（4）存在一种算法，能够判断任一文法是否为 LL(1)文法。

（5）存在一种算法，能够判定任一两个 LL(1)文法是否产生相同的语言。

（6）不存在这样的算法，能判定上下文无关语言能否由 LL(1)文法产生。

根据以上特点以及具体的实验要求，由于本实验需要实现的是识别 C 语言子集的语法，文法相对不复杂，可以不需要通过代码的方式实现手写文法，且可以保证不包含左递归文法，通过对以上的文法进行扩充，加入包括多种条件判断语句与全局变量的定义等文法，并且为了易于程序的实现，不含有类似于 A -> B | C | D…的文法形式，如果出现相应的文法，则拆分开来表示成 A -> B， A -> C, A -> D…的方式，文法中不含有直接左递归文法和间接左递归文法，下面是具体扩充之后的文法：

```
/ 函数列表,包括 main 函数
FUNC_LIST -> FUNC_DEF FUNC_LIST
FUNC_LIST -> @

// 函数定义声明
FUNC_DEF -> TYPE_SPEC ID ( ARG_LIST ) CODE_BLOCK
FUNC_DEF -> INIT_STMT ;

// 类型符号
TYPE_SPEC -> int
TYPE_SPEC -> void
TYPE_SPEC -> float
TYPE_SPEC -> double
TYPE_SPEC -> char

// 参数列表
ARG_LIST -> ARGUMENT
ARG_LIST -> ARGUMENT , ARG_LIST
ARG_LIST -> @

// 参数，类型和标识符
ARGUMENT -> TYPE_SPEC ID

// 代码段
CODE_BLOCK -> { STMT_LIST }

//  ……
```

图-2

文法的其他部分将以附录附在实验报告后。

3.2.语法分析器的程序实现

3.2.1 LL（1）语法分析器实验原理

LL(1)分析法是一种自上而下的分析法，使用显示堆栈而不是递归调用来实现分析。所谓 LL(1)是指语法分析按照自左向右的顺序扫描输入字符串，并在分析过程中产生句子的最左推导。“1”表示在分析过程中，每一步推导，最多只要向前查看（从右扫描）输入一个字符。既能确定查看当前推导所应用的文法规则。

LL（1）分析法的实现是由一个总控程序控制输入字符串在一张 LL(1)分析表（也称为预测分析表）和一个分析栈上运行而完成语法分析的任务。所以一个 LL(1)分析器的逻辑结构如下图所示，由总控程序、LL(1)分析表和分析栈三部分完成。

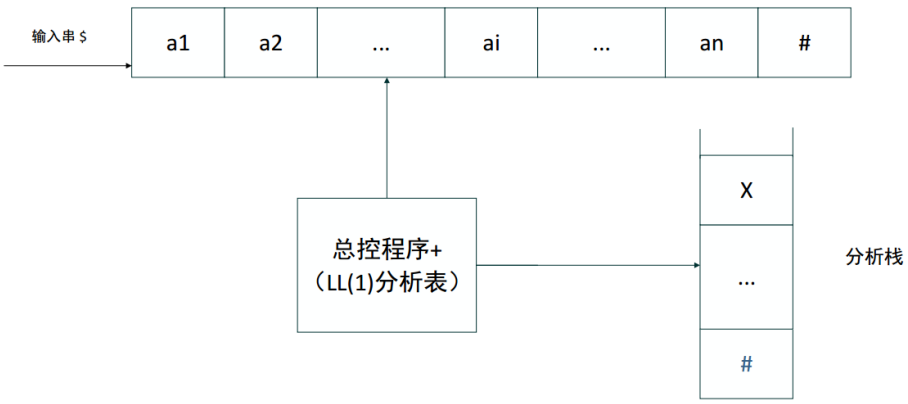

图-3

其中：

（1）LL(1)分析表：用 M[A, a]形式的矩阵来表示。其中 A 是文法的非终结符号，a 是文法的终结符号或者“#”。矩阵元素 M[A, a]存放一条关于非终结符号 A 的产生式(实际为 A 的一个侯选式)或用空白来给出一个出错标志。矩阵元素实际是相应的分析动作（即所选的推导的产生式），即对[A, a] = Ai -> a 表示当前栈顶为 Ai，输入字符为 a 时，应选用 Ai -> a 进行推导。

（2）分析栈：用于存放分析过程中的文法符号。分析栈初始化时，在栈低压入一个“#”，然后在次要底放入文法的开始符号。

（3）总控程序的功能是依据分析表和分析堆栈联合控制输入字符串的识别和分析，它在任何时候都是依据当前分析栈的栈顶符号 X 和当前的扫描字符 a 来执行控制功能。

下面先给出 LL(1)分析表的构造算法与实现，然后给出其中的 FIRST 集合和 FOLLOW 集合的算法及其实现过程。

3.2.2 LL(1)分析表的构造方法

算法：

```
输入：文法 G，G 的 FIRST 和 FOLLOW 集合
输出：文法 G 的 LL(1)分析表
for 文法 G 的每个产生式 A -> y1 | y2 | y3 | … | ym
{
  if a in FIRST(yi) 置 M[A, a] 为 “A -> yi”;
  if @ in FIRST(yi)
    for 任何 a in FOLLOW(A) {置 M[A, a] 为 “A -> yi”}
}
置所有无定义的 M[A, a]为出错
```

图-4

根据以上算法，发现如果不是 LL(1)文法，则一定会出现重复定义的情况，在构造分析表的过程中发现需要使用到 first 函数与 follow 函数，这两个函数的实现将在本节后续内容进行说明，下面是该部分的核心代码：

```
class LL1():
        # ...
        def createTable(self):
            vtt = self.vt | set(["#"])    #(非终结符号集合 - '@') | '#'
            vtt = vtt - set(["@"])
```

```
            for vn in self.vn:
                for vt in vtt:
                    for g in self.grammar: # 对于每一个形如 A -> B 的产生式，提取左部
分和右部分
                        left, right = g.split("->")
                        left = left.split()  # <list>
                        right = right.split() # <list>
                        if vn == left[0]:
                            first_beta = self.first(right)
                            if vt in first_beta:
                                self.table[(vn, vt)] = g
                            if "@" in first_beta: # '@' in first（产生式右部）
                                follow_beta = self.follow(vn)
                                for f in follow_beta:
                                    self.table[(vn, f)] = g


        # ...
```

代码-1

3.2.2 FIRST 集合生成方法：

算法：

输入：待求 FIRST 集合的字符列表 ss

输出：FIRST 集合字符列表 ans

If ss[0] 是终结字符：

  ans |= {ss[0]}

  返回 ans

If len(ss) == 1 and ss[0] 是非终结字符：

  对于以 ss[0]为左部的所有产生式的右部的每一个字符 s：

    ans = (ans | {fist(s)}) - { '@' }

    if '@' not in first(s): break

  如果所有 s 都是能推导到 '@'，则 ans |= { '@' }

对于字符列表中的每一个字符 S，实行上述两个过程

    If @ not in fist(S): break

图-5

对于 first 的算法，主要是要考虑到当前字符如果能够推导到‘@’，则需要考虑下一个字符的 FIRST 集合

First 函数的实现代码如下：

```
    class LL1():
    # ...
        def first(self,ss):
            ans = set()
            if ss[0] in self.vt:    # 如果当前的首个字符是单个终结符
                ans |= set([ss[0]])
                return list(ans)

            if ss.__len__() == 1:  # 如果当前是单个非终结字符
                for g in self.grammar:
                    left, right = g.split("->")
                    left = left.split()[0]  # <char>
                    right = right.split()  # <list>

                    if left == ss[0]:   # 找到所有以当前字符为文法左部的产生式
                        times = 0
                        for i in right: # 对于产生式的右部的每个符号，求 FIRST 集合
                            temp = (ans | set(self.first([i]))) - set(["@"])
                            ans = ans | temp
                            if "@" not in self.first([i]) and i != "@": #如果当前的
                            #字符的 FIRST 集合不包含空字符，则停止搜索
                                break
                            times += 1
                        if times == right.__len__(): # 如果产生式右部的所有字符都可以
推导出'@'，那么@属于 ans
                            ans.add("@")
                return list(ans)
            times = 0
            for i in ss: # 当前是由字符列表，对每个字符求 FIRST 集合
                temp = (ans | set(self.first([i]))) - set(["@"])
                ans = ans | temp
                if "@" not in self.first([i]) and i != "@":
                    break
                times += 1
            if times == ss.__len__():
                ans.add("@")
            return list(ans)
    # ...
```

代码-2

3.2.3 FOLLOW 集合生成方法：

算法：

```
输入：待求 FOLLOW 集合的非终结符号 A
输出：FOLLOW 集合 ans
对于开始符号 S,   ans |= { “#” }
for 产生式 g in 文法 Grammar:
  if A in 产生式右部：
    if B -> aAb:
      ans |= {first(b) - “@” }
    if B -> aA 或者 B -> aAb 且 ‘@’ in first(b):
      ans |= follow(B)
```

图-6

下列是 follow()函数的核心代码

```
    class LL1():
   # ...
       def follow(self, ss):
           ans = set()
           if ss == "S":      # 如果是开始符号
               ans.add("#")
               return list(ans)
           for g in self.grammar:  #对于文法中的每个产生式的右部
               left, right = g.split("->")
               left = left.split()[0]
               right = right.split()
               # 对应算法中的第一种情况
               if ss in right and right.index(ss) + 1 != right.__len__() and
left != ss:
                   ans = ans | set(self.first(right[right.index(ss) + 1:]))
                   if "@" in self.first(right[right.index(ss) + 1:]):
                       ans = ans | set(self.follow(left))
               # 对应于算法中的第二种情况
               if ss in right and right.index(ss) + 1 == right.__len__() and
left != ss:
                   ans = ans | set(self.follow(left))
           ans -= set(["@"])
```

```
            return list(ans)
    # ...
```

代码-3

代码中和算法中有不同且容易忽略的是,在判断条件时，在实际过程中会出现这样的情况：

A -> aA

在实际过程中，互不断循环找 follow(A),导致程序无法正常停止，堆栈空间消耗完报错。在实际的过程中，还有以下的情况：

A -> aB

B -> cC

C -> aA

在实际过程中，可能会陷入一直求 follow(A)，follow(B)，follow(C)的过程，最终报错，在这个实验过程中发现了这个问题，但是由于在构造分析表的时候，由于当前“@”不在产生式右部 first 集合中，所以没有出现求左部 FOLLOW 集合的情况，没有出现错误。

四、实验结果与总结

4.1.基本测试：

当输入为：

```c
#define NUM 4

/* this is a demo program */
void fun(){
    int a;
}

int main(){
    char b;
    int a = 0;
    a = NUM * 5 + 6 - 7; //here is a macro

    return a;
}
```

代码-4

输出为：

```
<STMT>
  <ASSIGN_STMT>
    <TYPE_SPEC>
      <value>char</value>
      <value>b</value>
      <value>;</value>
    </TYPE_SPEC>
    <ASSIGN_STMT2/>
  </ASSIGN_STMT>
</STMT>
<STMT_LIST>
  <STMT>
    <ASSIGN_STMT>
      <TYPE_SPEC>
        <value>int</value>
        <value>a</value>
      </TYPE_SPEC>
      <ASSIGN_STMT2>
        <EXPR>
          <TERM>
            <FACTOR>
              <value>0</value>
              <value>;</value>
            </FACTOR>
            <TERM2/>
          </TERM>
          <EXPR2/>
        </EXPR>
        <value>=</value>
      </ASSIGN_STMT2>
    </ASSIGN_STMT>
```

图-7

且识别显示为成功

4.2 循环与条件语句的测试

```
const int N;
int A;
int B;
int value[N][N];
int value[100][100];
float  function(int a,float b)
{
    float x = 100LLU;
    return a+b;
}

int main()
{
    int str = 1+2*3%4;
    str[100][100] = memset(1,1,size(str));
    if (a==function(1,strcpy("abc"))+2)
    {
        str[i][j] = rand() % 100 + 1.2e100;
        while(str[i]<2.)
        {
            printf("%d\n",a);
        }
    }
    while(str[i]<2.)
        memset(1);
    return .1;
}
```

代码中包含了条件，循环等较为复杂的语言，经过测试，得到了理想的结果。

图-8

如上图所示，循环语句，全局变量以及条件语句等都可以实现，实验是成功的。

实验总结：

这个实验是我在自己学习 python 不久后实现的第一个还算有点难度的代码段，自己实现 fitst 集合，follow 集合，分析表的构建以及总控程序的实现，期间出现了较多的错误，大部分已经得到了有效的解决，但是有部分是仍然需要花时间去再看看的。

通过这个实验，我学习到了 LL(1)语法分析器的构造，同时，也深入地了解了自下向上与自上向下的语法分析器的构造方法，收获颇丰。

五、附录

LL(1)文法

S -> FUNC_LIST

FUNC_LIST -> DEF FUNC_LIST

FUNC_LIST -> @

DEF -> TYPE_SPEC identifiers FUNC_OR_VAL

FUNC_OR_VAL -> (ARG_LIST) CODE_BLOCK

FUNC_OR_VAL -> ARRAY

TYPE_SPEC -> int

TYPE_SPEC -> void

TYPE_SPEC -> float

TYPE_SPEC -> char

TYPE_SPEC -> const int

ARG_LIST -> ARGUMENT ARG_LIST2

ARG_LIST -> @

ARG_LIST2 -> , ARGUMENT ARG_LIST2

ARG_LIST2 -> @

ARGUMENT -> TYPE_SPEC identifiers

CODE_BLOCK -> { STMT_LIST }

STMT_LIST -> STMT STMT_LIST

STMT_LIST -> @

STMT -> RTN_STMT ;

STMT -> ASSIGN_STMT ;

STMT -> IF_STMT

STMT -> ;

RTN_STMT -> return EXPR

ASSIGN_STMT -> TYPE_SPEC identifiers ASSIGN_STMT2

ASSIGN_STMT -> EXPR

ASSIGN_STMT2 -> = EXPR

ASSIGN_STMT2 -> @

EXPR -> TERM EXPR2

EXPR2 -> + TERM EXPR2

EXPR2 -> - TERM EXPR2

EXPR2 -> = TERM EXPR2

EXPR2 -> == TERM EXPR2

EXPR2 -> > TERM EXPR2

EXPR2 -> < TERM EXPR2

EXPR2 -> @

TERM -> FACTOR TERM2

TERM2 -> * FACTOR TERM2

TERM2 -> / FACTOR TERM2

TERM2 -> % FACTOR TERM2

TERM2 -> @

FACTOR -> identifiers CALL

FACTOR -> constants

FACTOR -> string

FACTOR -> (EXPR)

CALL -> (CALL_LIST)

CALL -> [EXPR] ARRAY

CALL -> @

CALL_LIST -> FACTOR CALL_LIST2

CALL_LIST -> @

CALL_LIST2 -> , FACTOR CALL_LIST2

CALL_LIST2 -> @

ARRAY -> [EXPR] ARRAY

ARRAY -> @

IF_STMT -> if (ASSIGN_STMT) FOLLOW_STMT

IF_STMT -> while (ASSIGN_STMT) FOLLOW_STMT

FOLLOW_STMT -> STMT

FOLLOW_STMT -> CODE_BLOCK