****

*编译原理课程设计实验报告*

实验三 中间代码生成

姓 名：廖汉龙

学 号：1120151880

学 院：计算机学院

班 级：07111507

邮 箱：[liamliaohl@gmail.com](mailto:liamliaohl@gmail.com)

2018年5月14日 星期一

目录

[一、实验目的 3](#_Toc514099603)

[二、实验内容 3](#_Toc514099604)

[三、实验过程 4](#_Toc514099605)

[3.1 对于文法的改写与扩充 4](#_Toc514099606)

[3.2.语法分析器的程序实现 5](#_Toc514099607)

[四、实验结果与总结 10](#_Toc514099608)

[4.1.基本测试： 10](#_Toc514099609)

[4.2 循环与条件语句的测试 11](#_Toc514099610)

[五、附录 13](#_Toc514099611)

**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，在上一个实现——C语言子集语法分析器的基础之上，实现C语言子集的代码的中间代码生成。

2.通过动手实践，对编译过程中的中间代码生成的内容有更加深刻的了解与认识，能够熟练掌握根据语法生成树来中间代码生成的实现过程。

3.通过本实验，增强自主实践的能力

## 二、实验内容

1. 实验要求

该实验选择 C 语言的一个子集， 基于 BIT-MiniCC 构建 C 语法子集的代码生成模块，该语法分析器能够读入 XML 文件形式的语法分析树，进行寄存器分配，并生成 MIPS 或者 X86 汇编代码。需要说明的是，生成汇编程序是该部分的基本要求；进一步地，如果生成的是 MIPS 汇编， 则要求汇编代码能够在 BIT-MiniCC 集成的 MIPS 模拟器中运行（注意 config.xml 的最后一个阶段“simulating”的“skip”属性配置为“false”）；如果生成的是 X86 汇编，则要求使用 X86 汇编器生成 exe 文件并运行。

2.实验具体内容

在 BIT-MiniCC 框架下，按照如下步骤完成代码生成实验：

（1）在语法分析实验的基础之上，对获得的语法树进行变换，得到抽象语法树（见 2-1），变换的过程主要是将左递归导致的左斜树转换为平衡结构。如下图所示为一输出的分析树：

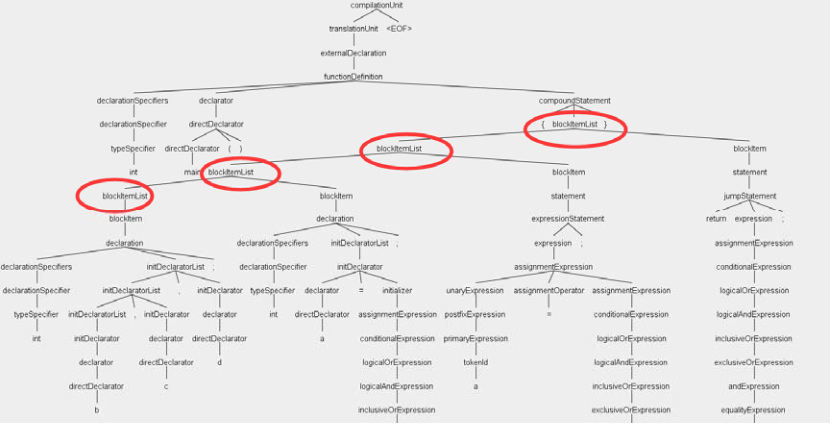


图2-1

转换后的抽象语法树为：

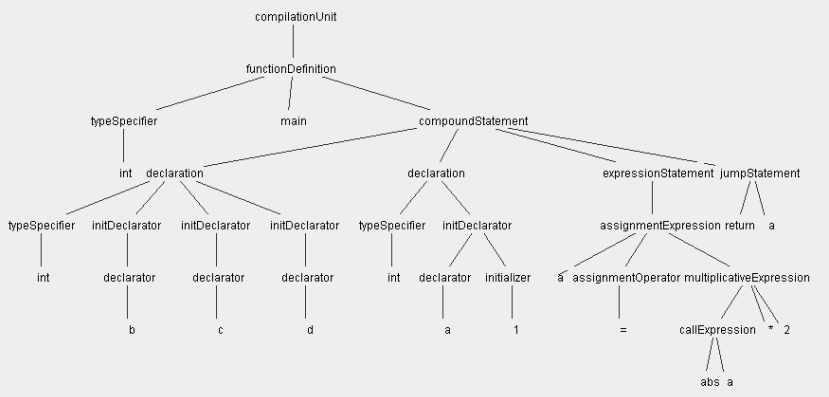


图2-2

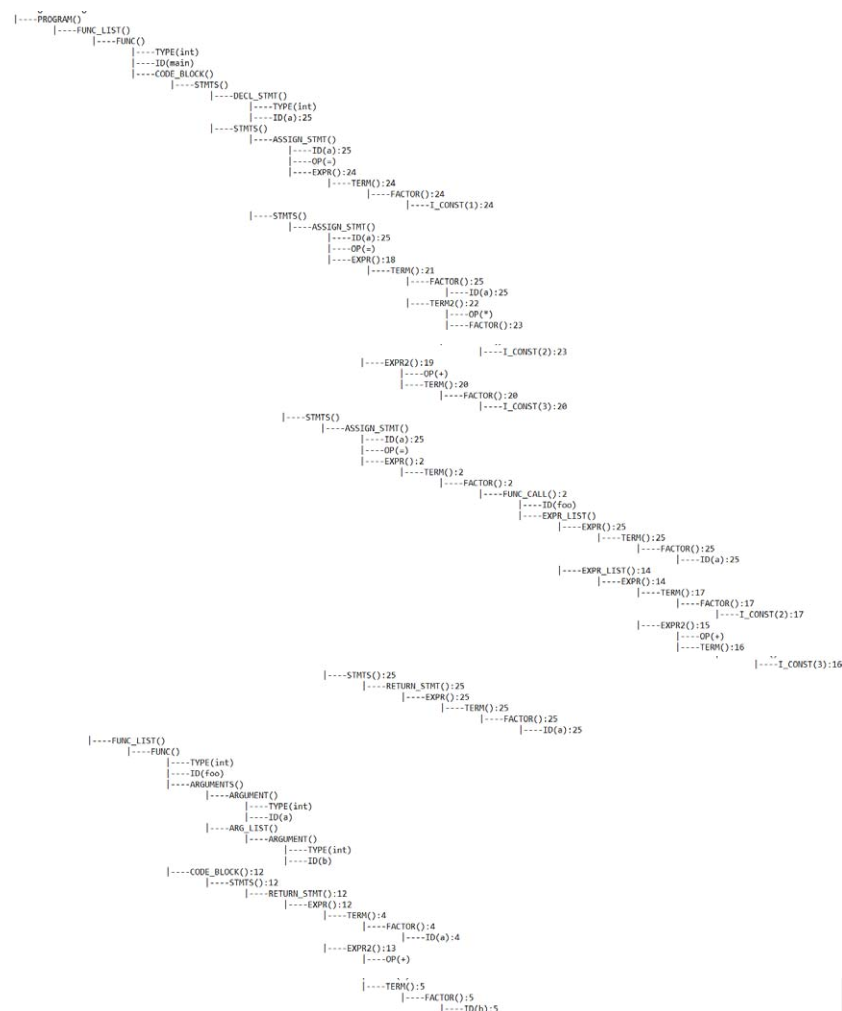
（2）为每个标识符以及中间临时变量分配寄存器。当变量和临时变量较少  
时，可以直接用为每个标识符以及中间临时变量分配唯一的标识符。

（3）代码生成。根据语法树以及寄存器分配结果，生成相应的代码。

以如下的 C 语言代码为例：

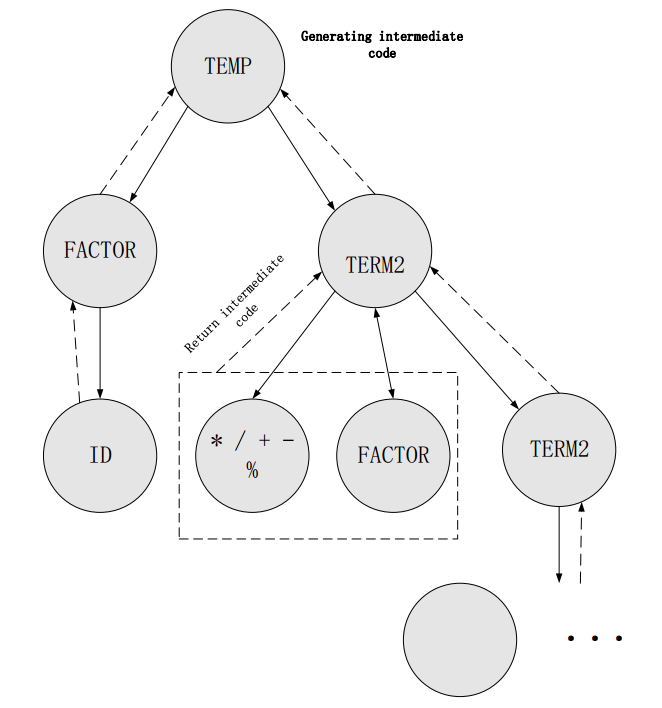
|  |
| --- |
| #include <stdio.h>  int main()  {  int a;  a = 1;  a = a \* 2 + 3;  a = foo(a, 2+ 3);  return a;  }  int foo(int a, int b)  {  return a + b;  } |

采用递归下降分析法，获得的语法树及其寄存器分配结果为：



## 三、实验过程

### 3.1 实验原理



3.1.1 符号表

在实验指导书中给出的基本文法如下图-1所示：

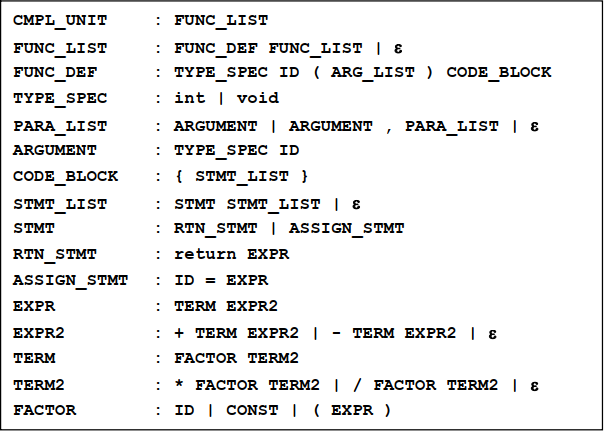


图-1

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：

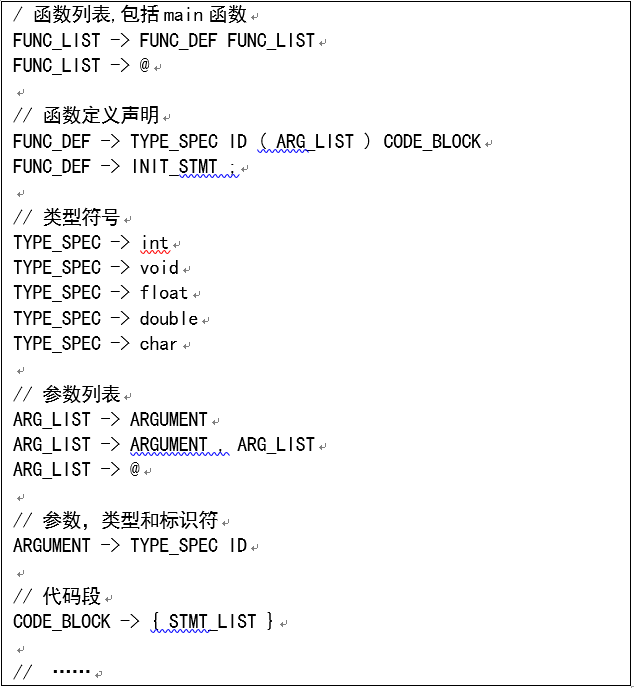


图-2

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。

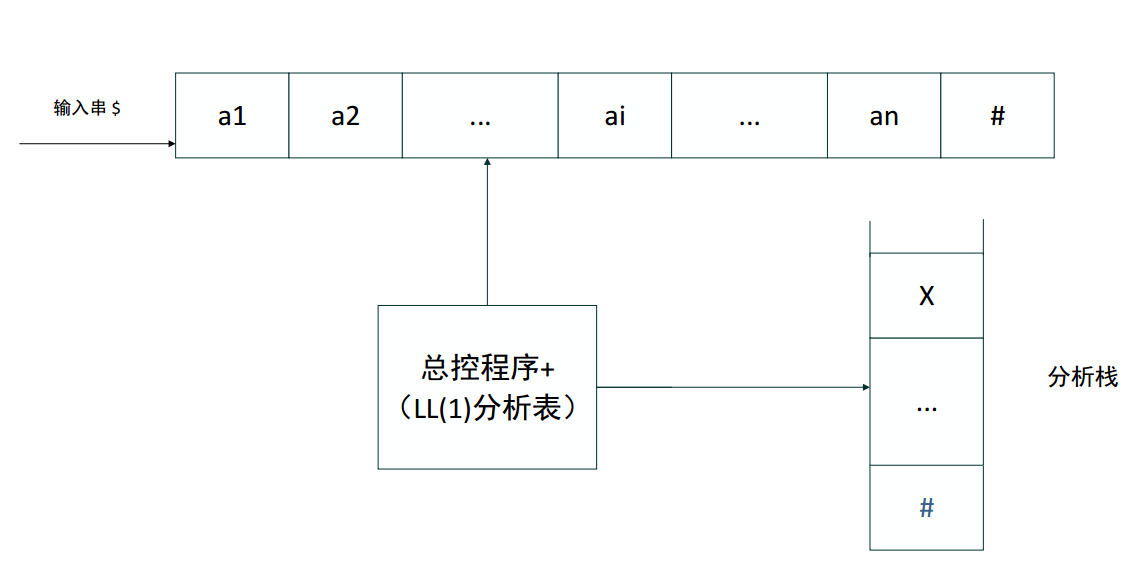


图-3

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：

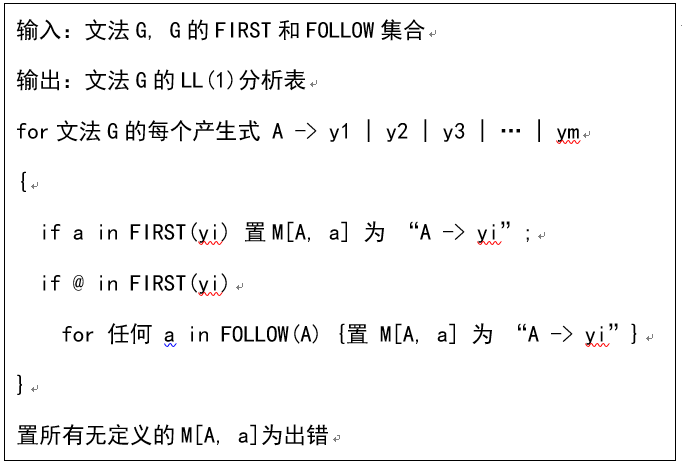


图-4

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：

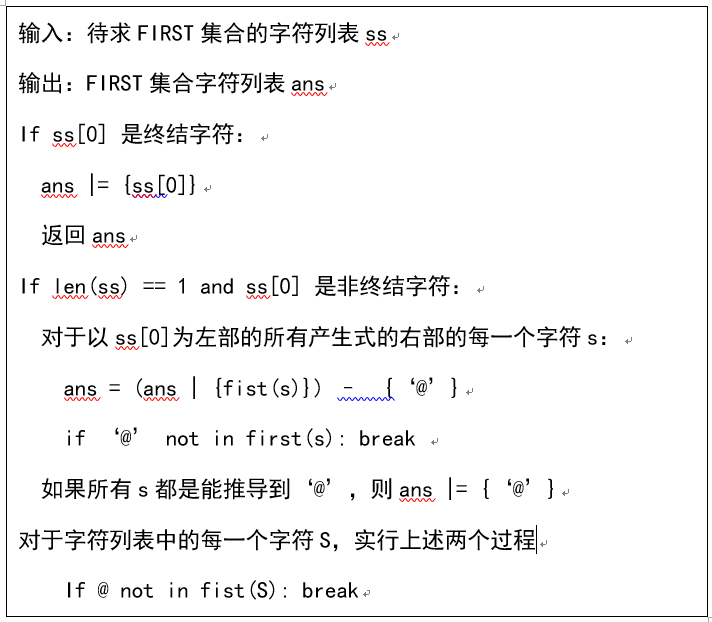


图-5

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：

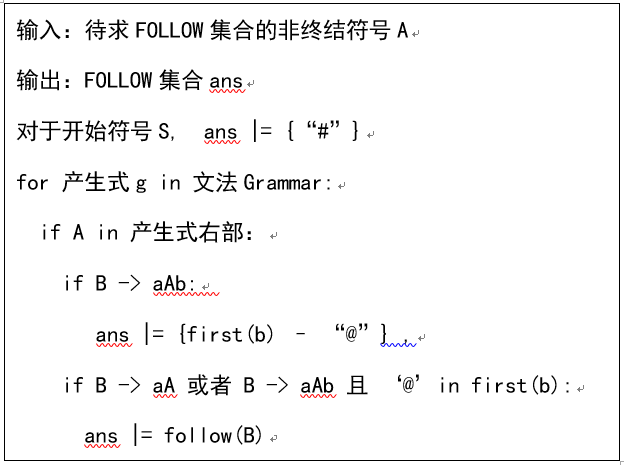


图-6

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：

|  |
| --- |
| #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：



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。

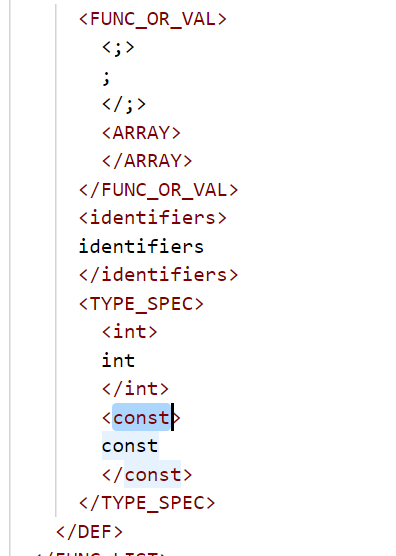
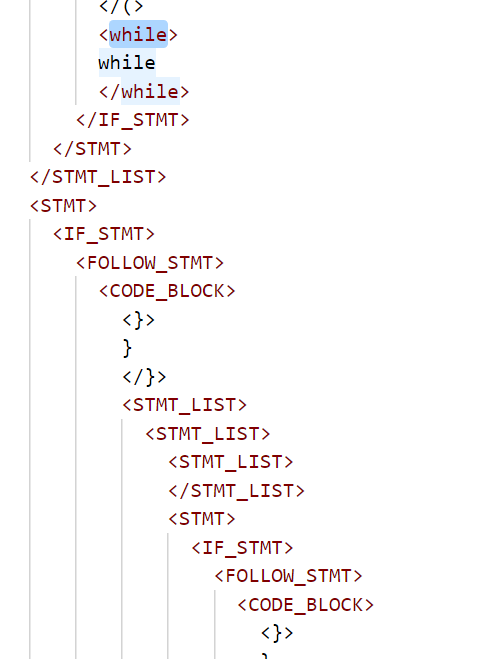


图-8

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**