**《编译原理》实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | | **2018级计科5班** | | **姓名** | **李林轩** |
| **实验题目** | 语法分析程序的设计与实现 | | | | |
| **实验时间** | **2021/4/24** | | **实验地点** | **DS1421** | |
| **实验成绩** |  | | **实验性质** | **□验证性 □设计性 ■综合性** | |
| 教师评价：  □算法/实验过程正确；□源程序/实验内容提交 □程序结构/实验步骤合理；  □实验结果正确； □语法、语义正确； □报告规范；  其他：  评价教师签名： | | | | | |
| 一、实验目的  🡪了解编译器中语法分析环节的作用和意义  🡪掌握语法分析的方法和步骤  🡪熟悉自底向上、自顶向下等各类语法分析方法；掌握语法树的构造方法 | | | | | |
| 二、实验项目内容  一）实验内容  设计并实现实验一中的C语言子集的语法分析程序，要求实现以下功能：   1. 必做项： 2. 可以识别出用C语言子集中数据类型和语句集编写的源程序，并以语法分析树的形式输出分析结果。   （基本数据类型至少包含整数、布尔类型；至少包含赋值语句、变量申明语句、算数加/减法表达式、布尔表达式、循环语句、分支语句）   1. 检查源程序中存在的语法错误，并报告错误所在的位置。 2. 可选项：   需在给定C语言子集的基础上进行扩充,实现下列要求：   1. 能识别函数调用语句 2. 对源程序中出现的错误进行适当的恢复，使得语法分析可以继续进行，检查并报告源程序中存在的所有语法错误及错误位置   二）实验要求：   1. 自由选用程序设计语言（java，python，c/c++）作为实现语言，手工编写语法分析程序。 2. 提交实验报告及源代码。实验报告需严格遵循学校文档规范，内容包含对应文法、语法分析测试用例。   三）C语言子集  数据类型: **int**, 无符号整数, 取值范围0-9999  int a;  int a,b;  int a = 1;  算术运算符: **+**,**-**  a = b + 1;  a = b + c;  赋值运算符: **=**  a = 1;  关系运算符: **==** ,**>**,**<,<>,>=,<=**  a = (b==c);  a = (b>c);  a = (b<c);  逻辑运算符: **&&**, **||, !**  a = (b&&c);  a = (b||c);  a = (!b);  条件语句: **if**  if(a==b)  {  };  循环语句: **while**  while(a==b)  {  };  输入,输出: **get**,**put**  get(a);  put(a);  语句结束符: **;**  **加分项：**  条件语句 **if else**  if(a==b)  {  };  else  {  }; | | | | | |
| 三、实验过程或算法 | | | | | |
| 内容要点，供参考：   1. **语言说明**   本语法分析器在实验一词法分析器的基础上，实现了C语言子集的语法分析器。  能够识别基本数据类型的修饰、定义、声明（含指针以及多维指针），对变量的赋值，表达式以及逻辑表达式，输入输出系统函数（根据测试样例，将输入输出函数更改为get与put），函数的定义声明，函数块内部，if-else语句，for循环，while循环等，下面对设计的文法进行说明。  本语法分析器采用的程序设计语言为Python，设计的文法为了方便程序读入将文法格式规定成  的形式，为了方便理解，方便自己改动将文法使用中文进行设计，没有再改为符号。使用空格将文法符号分开，这样在Python中只需要使用split()进行分割，再后续处理，非常方便。  下面开始对文法进行说明。  **1.1 能够推出空的文法集合**  由于在求符号的First集与Follow集时，对能够推出空（即ε，在文法中使用@进行表示）的非终结符都有相应的处理，因此在设计文法完成后，对顺序进行了调整，将能推出ε的非终结符文法放在最开始。   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **修饰词闭包** | **@** | | **星号闭包** | **@** | | **数组下标** | **@** | | **项** | **@** | | **因式递归** | **@** | | **参数声明** | **@** | | **变量闭包** | **@** | | **常量闭包** | **@** | | **赋初值** | **@** | | **声明闭包** | **@** | | **函数块闭包** | **@** | | **声明语句闭包** | **@** | | **参数闭包** | **@** | | **否则语句** | **@** | | **因式递归** | **@** |   后续在写道对应非终结符以及其文法的设计时再详细说明。  **1.2 语法分析器开始文法集合**  接着这部分是程序的开始，也是语法分析器的开始部分，即类似于一般的文法中的非终结符S。在实验报告给出的测试样例中，并没有像C语言一样引入stdio.h或是规定主函数并在之前做一些声明。因此此处定义的文法给出了两种情况，一种直接推出函数块，用于实验报告中的样例测试，另一种更接近于C语言的文法，可以在后续自己设计的类似C语言测试程序中进行测试。   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **程序开始** | **函数块** | | **程序开始** | **修饰词闭包 类型 变量 ( 参数声明 ) { 函数块 }** |   接下来对修饰词进行说明，C语言中修饰词可以叠加使用，即类似以下这种写法： 为了使设计出的文法能够满足这种情况，引入闭包的概念，非常类似右递归。修饰词部分文法设计如下：   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **修饰词闭包** | **修饰词 修饰词闭包** | | **修饰词** | **const** | | **修饰词** | **unsigned** | | **修饰词** | **signed** | | **修饰词** | **static** | | **修饰词** | **extern** |   可以看到， 这一条文法以及在最开始的空文法中定义的 满足了叠加使用修饰词的程序，并能够正确的终结。  **1.3 类型部分文法集合**  类型，包括int，float等基本数据类型，也包括auto等自动判断变量类型的声明，同时也包括指针以及多维指针。文法设计如下：   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **类型** | **auto 取地址** | | **类型** | **int 取地址** | | **类型** | **double 取地址** | | **类型** | **float 取地址** | | **类型** | **char 取地址** | | **类型** | **string 取地址** | | **类型** | **long 取地址** | | **类型** | **register 取地址** | | **类型** | **short 取地址** | |  |  | | **取地址** | **星号闭包** | | **星号闭包** | **星号 星号闭包** | | **星号** | **\*** |   通过类型，可以得到只能有一次的类型声明以及取地址。取地址则可以推出星号闭包，来进行多维指针的生成，类似 同样，星号闭包也可以推出ε，已经在最开始定义过了，保证了这个闭包可以终结。  **1.4 变量以及变量引出的文法集合**   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **变量** | **标识符 数组下标** | | **标识符** | **id** | | **数组下标** | **[ 因式 ]** | | **因式** | **( 表达式 )** | | **表达式** | **因子 项** | | **项** | **+ 因子 项** | | **项** | **- 因子 项** | | **项** | **& 因子 项** | | **项** | **| 因子 项** | | **因子** | **因式 因式递归** | | **因式递归** | **\* 因式 因式递归** | | **因式递归** | **/ 因式 因式递归** | | **因式** | **变量** | | **因式** | **常量** | | **常量** | **digit** |   变量包含单个变量以及该变量类型的数组，因此有文法 而推出单个变量的情况也存在，因此有 标识符推出id，作为终结。词法分析中的结果（符号表和token序列）在读入语法分析器后会将标识符的词法编号统一重命名为id，方便后续处理。  而当推出数组的情况，数组下标将会推出因式，因式可以直接推出变量或者常量，也可以推出表达式，在此处文法规定，在数组中使用表达式，必须在表达式外部添加括号。这样能够与后面的文法定义复用同样的文法，而不用再去添加。因式推出表达式用于识别以下这种情况： 后续因式会推出项以及运算符来实现闭包，为了使设计出的文法满足左递归，将不同的运算符分配到了不同的文法。但是功能依旧能够实现。  因式递归类似闭包，是一个右递归的情况，在前面定义的 使得该递归（闭包）能够正常完结。  **1.5 参数声明部分文法集合**   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **参数声明** | **声明 声明闭包** | | **声明** | **修饰词闭包 类型 变量 变量闭包 赋初值** | | **变量闭包** | **, 变量 变量闭包** | | **赋初值** | **= 右值** | | **右值** | **表达式** | | **右值** | **{ 多个数据 }** | | **多个数据** | **常量 常量闭包** | | **常量闭包** | **, 常量 常量闭包** | | **声明闭包** | **声明 声明闭包** | | **声明语句闭包** | **声明语句 声明语句闭包** | | **声明语句** | **声明 ;** |   参数声明指的是在定义函数时进行的声明，使用右递归的方式实现声明闭包。完成对形如一下代码的识别：  赋初值是可以推出空的，其实就是程序中函数参数的默认值。 中间的右值即为赋值给变量的具体的值。右值可能是表达式，也可能是在对数组赋值的时候右值包含着的多个数据，通过大括号将其包住。观察之前定义的产生式，通过 这条产生式易得，表达式能够推出变量。即右值也可以是变量（包括数组）。 这段文法后半部分是较为宏观的声明闭包，即一段程序中声明语句可以有多条，这些声明语句通过’;’进行分割，同样，声明闭包与声明语句闭包也能够推出空值。  **1.5 函数块文法集合**  接下来这部分是相当于最顶层的文法设计，即函数块推出的后续文法仍然是非常大的部分，这部分的后续仍然能够推出很多东西，这些后续文法如下表所示：   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **函数块** | **声明语句闭包 函数块闭包** | | **函数块闭包** | **赋值函数 函数块闭包** | | **函数块闭包** | **流输入输出 函数块闭包** | | **函数块闭包** | **for循环 函数块闭包** | | **函数块闭包** | **while循环 函数块闭包** | | **函数块闭包** | **条件语句 函数块闭包** | | **函数块闭包** | **函数返回 函数块闭包** |   声明语句相关文法以及赋值函数相关文法在前面已经介绍，接下来按照这部分产生式推出的相关右部产生式的顺序进行说明  **1.6 赋值函数类文法集合**   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **赋值函数** | **变量 赋值或函数调用** | | **赋值或函数调用** | **( 参数列表 ) ;** | | **赋值或函数调用** | **= 右值 ;** | | **参数列表** | **参数 参数闭包** | | **参数闭包** | **, 参数 参数闭包** | | **参数** | **标识符** | | **参数** | **常量** | | **参数** | **字符串** | | **字符串** | **str** |   赋值函数可以使用第三条产生式指变量被赋值推出变量被右值赋值，也可以指一个有着返回值的函数将其返回值付给了前面的一个变量，此处的变量则指这个函数调用的函数名。 由于不能确定函数中参数列表的参数数量，因此参数列表同样使用闭包实现，参数可以是标识符（变量），常量，字符串。语法分析器将把在词法分析中得出的token序列中的所有字符串都重命名为str，用于匹配。  **1.7 流输入输出文法集合**   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **流输入输出** | **get ( 参数列表 ) ;** | | **流输入输出** | **put ( 因式 ) ;** |   流输入输出按照测试样例中的形式，使用get(),put()函数，其中get函数参数列表中可以有多个参数，比如： 这样我们将推出的get设为参数列表，通过上一部分定义的参数闭包对get中的参数进行匹配； 而put函数在两个测试样例中都只有一个参数，因此语法分析器中规定为put函数只允许有一个参数，该参数使用因式，既能输出变量，常量，也能输出表达式的计算结果。  **1.8 for循环及相关文法集合**   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **for循环** | **for ( 赋值函数 ; 逻辑表达式 ; 后缀表达式 ) { 函数块 }** | | **逻辑表达式** | **bool** | | **逻辑表达式** | **表达式 运算符 表达式** | | **逻辑表达式** | **! 表达式** | | **运算符** | **>** | | **运算符** | **<** | | **运算符** | **==** | | **运算符** | **>=** | | **运算符** | **<=** | | **运算符** | **!=** | | **运算符** | **%** | | **运算符** | **&&** | | **运算符** | **||** | |  |  | | **后缀表达式** | **变量 后缀运算符** | | **后缀运算符** | **++** | | **后缀运算符** | **--** |   C语言中for循环语法与下面类似： 首先是赋值函数，为循环变量赋初值，接着是判断循环何时终结的逻辑表达式。逻辑表达式可以是运算的结果，也可以是一个布尔值，比如true，false，词法分析的布尔常量将在语法分析器里统称为bool。此处定义了运算符所能够推出的终结符，这些终结符均为二元运算符。而最后的后缀表达式时比较常见的for循环循环变量改变的方式，这里定义两个一元后缀运算符，++，--用于循环变量的自增与自减。这里的后缀表达式不能全面的表达所有情况，循环变量也存在其他赋值的方式，但是为了表达的清晰简便，并且考虑到大部分写法均为自增自减，这里将循环变量改变的方式固定为自增与自减。  函数块是能够通过声明语句闭包，函数块闭包而推出空的，也就是说当程序有一个for循环并且循环内部并没有代码时，程序是允许这种情况的，并不会产生语法错误。  **1.9 while循环及相关文法集合**   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **while循环** | **while ( 逻辑表达式 ) { 函数块 } ;** |   由于在前面将逻辑表达式与函数块都定义过了，在此处不再需要其他文法，只需要按照C语言子集的文法设计while循环即可。  **1.10条件语句及相关文法集合**   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **条件语句** | **if ( 逻辑表达式 ) { 函数块 } ; 否则语句** | | **否则闭包** | **elif语句 否则闭包** | | **否则闭包** | **else { 函数块 } ;** | | **elif语句** | **elif ( 逻辑表达式 ) { 函数块 } ;** |   条件语句包括if，elif（else if），else三种，其中后两种都可以通过 推出空，即可以存在只有if一条的情况。  **1.11 函数返回及相关文法集合**   |  |  | | --- | --- | | **产生式左部** | **产生式右部** | | **函数返回** | **return 因式 ;** |   函数返回通过函数块闭包推出，当出现return时使用这条产生式进行分析。   1. **选用语法分析方法及相应设计**   本语法分析器选择的语法分析方法是LL(1)，对此最重要的一点就是对文法的设计：即需要保证文法满足LL(1)文法。   1. 文法不含左递归 2. 文法中每个非终结符A的各个产生式的首终结符集两两不相交，即 3. 文法中每个非终结符A若其首字符集中含有ε，且   为了满足上述条件，进行了大量的试错。左递归是很好消除的，在设计文法时注意一下即可，即使出现了左递归也可以进行消除。但是要满足后两个条件则需要一定的尝试。最终确定了使用闭包等来消除左递归，并用符号推出闭包，再用闭包推出空而不是直接让某个符号推出空避免了一些文法的问题，当文法是错误的时候，文法的顺序甚至会影响action table。好在最终设计出的文法满足LL(1)条件，能够正确完成语法分析的任务。  语法分析器的结构如下图所示：    **action\_table.xls**是生成的action table，将其存储到excel中，方便查看和调试程序中的错误。 **class\_methodDefine.py**定义了一系列语法分析需要用到的函数与类，以及这些函数/类的测试函数 **first\_follow.py**和名字一样，定义了通过文法求得first集，follow集的相关函数 **main\_grammarAnalysis.py**是将所有文件的函数整合起来，完整的进行语法分析的主函数 **output\_temp\_result.py**用于将各部分生成的集合进行格式化输出，比如第一个文件action\_table.xls就由该文件中定义的函数格式化输出导excel中 **read\_lexical\_analyse\_result.py**将实验一中词法分析器的词法序列单元与符号便按格式化读入，并转为语法分析器所需的格式 **symbol\_id.py** 该文件类似于C/C++中的头文件，定义了关键字以及关键字的词法编号。用于上一个文件读入词法单元序列并将其通过编号转变为对应的语法终结符 **tree.dot** 该文件为存储语法树信息的文件，最终将会通过graphvis的dot命令生成tree.png tree.png 最终的语法树图片，通过命令 将tree.dot生成图片 **tree\_plot\_by\_graphvis.py**该文件为绘制语法树的文件，通过将使用到的产生式按顺序递归求得生成语法树，再前序遍历生成的语法书得到节点以及边的情况，并绘制语法树。绘制语法树的库一开始想要采用的是ete3，但是这种库只能在Linux环境下运行，并且需要生成对应格式的字符串，实现起来较为繁琐。因此最后采用了graphvis  **wenfa.txt** 定义的文法   1. **程序数据结构说明**   这里将按照主程序中的执行顺序，对各个阶段使用到的数据结构进行说明。主程序中的执行流程如下：  **构建语法分析器**  这部分使用到的数据结构为列表（数组），集合，字典（哈希表）。  列表使用的地方太多了，不在此一一说明  集合主要是保存终结符和非终结符。由于使用python自带的函数set()会使列表变为元组，不可修改，使得后续一些操作会受到限制。因此实现方法就是在将元素append进入列表前进行判断即可。python实现也非常方便。  字典则是生成first集合，follow集合，action table时需要使用。每个终结符/非终结符作为字典的key，而他们对应的first/follow集则是字典key对应的value。action table同理，只不过它是一个二维的字典，通过非终结符先索引到一个字典，再通过该字典的每个keys——非终结符，索引到相应的产生式或是Error。  **读取词法分析器执行结果**  此处使用到的数据结构为列表，字典。  字典是在头文件中定义的keywords。它将词法分析序列的token与编号对应起来，能够进行转换。  **语法分析**  这部分使用到的数据结构为栈，由于python中没有定义好的栈，因此自己实现了栈，并添加了一些方便调试，查看的函数    栈主要应用在语法分析中的符号栈。  **构建并绘制语法树**    语法书相关的数据结构自然就是树节点以及边的数据结构。 树节点的属性包括id，child\_list语法树的子节点列表，也就是产生式右部，value代表当前节点的值，也就是该节点的符号，is\_leaf就是是否为叶节点，非终结符一定不是叶节点，而终结符一定是叶节点，可以通过这个属性来判断。father即为该节点的父节点，但是在后续实现中并没有使用到。  边的数据结构很简单，只有边的起始节点与结束节点，这两个节点类型都是TreeNode。   1. **语法树以及输出说明**   **输出说明**  按照3中程序的执行顺序，按照以下顺序在控制台产生输出：   * **非终结符集合**      * **终结符集合**      * **First集**      * **Follow集**      * **生成的action table**   当成功把action table格式化输出到excel中时，控制台会输出:  Excel中action table格式如下：     * **语法分析过程（栈内元素 & 元素在程序中所在行数 & 当前输入符号）**     最终，当语法分析成功，输出如下：     * **语法分析中依次调用的产生式**      * **部分语法树节点编号以及value**      * **部分语法树边节点：边的两个节点编号以及value**     **语法树说明**  生成的语法树保存在dot文件中，格式如下： 首先是对于树节点的定义：前面的数字代表着该节点的编号，后面的”lable=”代表着在打印出的语法树图像中在节点上写的内容。    接下来是边的写入格式：根据之前定义的语法树唯一的编号（即不会出现两个节点编号相同，但是会出现两个节点label相同），将节点与节点之间连接构成边    其中一个测试样例生成的语法树图像如下所示，该语法树过于庞大，对该测试样例生成的语法树的详细说明见实验报告第四部分，实验测试结果。   1. **错误处理**   按照老师的PPT，将错误类型分成了两类：   1. 栈顶的终结符与当前输入符不匹配   遇到该错误时，弹出当前输入的终结符，此时相当于把所有的符号都看作同步符号   1. 分析表中此终结符与非终结符的内容为空   遇到该错误时，设A为当前栈中的非终结符，将FOLLOW(A)中的所有符号作为A的同步符号；并跳过输入串中的一些词法单元，直至遇到FOLLOW(A)中的元素，再把A从栈中弹出  错误处理程序见下一部分：关键程序说明；对错误程序的处理见第四部分测试用例中的说明   1. **关键程序说明**   按照选用语法分析方法及相应设计部分介绍的文件顺序进行关键程序说明。   1. **class\_methodDefine.py**   下面这部分是建立预测分析表的函数。首先将t\_set（终结符集合）中添加代表完结的符号（$），接着将该二维字典进行初始化，将分析表中所有项设为ERROR（ERR）。  接着对于每条产生式： 对于FIRST(α)中的每个终结符号b，将产生式A→α加入到M[A, b]中 如果ε在FIRST(α)中，那么对于FOLLOW(A)中的每个终结符号b，将产生式A→α加入到M[A, b]中  由于一开始就将没有产生式的部分设为了ERR，因此不需要再做处理，最后返回生成的预测分析表    下面这部分是错误处理。大部分是对错误信息的输出。而根据传进来的类型参数有不同的处理方法，可以看注释。最终将输入符号串新的index返回，而由于symbol\_stack是自定义的类型对象，在函数内的修改能直接映射到外部变量。因此不需作为返回值返回       1. **测试样例以及结果展示**   在此处列出所有的测试样例，在第四部分试验测试结果中对这些测试样例的运行结果吗，生成的语法树进行逐个说明。   1. **简易计算器**   //输入数据num1,num2,op，根据op确定操作进行运算，最后输出运算结果ans  int num1,num2,op,ans;  get(num1,num2,op);  if(op==0)  {  ans = num1 + num2;  };  if(op==1)  {  ans = num1 - num2;  };  if(op==2)  {  ans = num1 & num2;  };  if(op==3)  {  ans = num1 | num2;  };  put(ans);   1. **跑马灯**   //循环输入op，改变输出结果out，输入0则结束程序  int num0,num1,out,op;  num1 = 3333;  num2 = 6666;  num3 = 9999;  op = 1;  while(op>0)  {  if(op==1)  {  out = num1;  };  if(op==2)  {  out = num2;  };  if(op==2)  {  out = num3;  };  put(out);  get(op);  };   1. **自定义测试一：附加语法测试**   本测试样例主要测试指针，数组的声明，以及if，else if，else语句结构的识别：  int num0 = 10;  int num1,out,op;  int testArray[100];  char \* str;  num1 = 3333;  num2 = 6666;  num3 = 9999;  op = 1;  while(op>0)  {  if(op==1)  {  out = num1;  } ; elif (op==2)  {  out = num2;  } ; else  {  out = num3;  } ;  put(out);  get(op);  };   1. **自定义测试二：附加语法测试**   本测试主要测试主函数定义，for循环以及修饰符闭包，数组的识别  for (int i = 0 ; i < 3 ; i++){  const static int a[5] = {1, 2, 3, 4, 5} ;  for ( int j = 0 ; j < i ; j ++){  a[i] = i \* j ;  put(a) ;  while (true) {  put(a[i]) ;  } ;  } ;  } ;   1. **自定义测试三：附加语法测试** 2. **自定义测试四：错误测试**   本测试在简易计算器的基础上修改，主要测试第一种错误类型能否被识别，以及被识别之后进行错误处理后能否使得语法分析继续进行。  错误包括：  第二行中，op后添加+号；  第五行中，num2;后面添加;  第九行中，  int num1,num2,op,ans;  get(num1,num2,op+);  if(op==0)  {  ans = num1 + num2;;  };  if(op==1)  {  ans = num1 - num2;  };  if(op==2)  {  ans = num1 & num2;  };  if(op==3)  {  ans = num1 | num2;  };  put(ans);   1. **自定义测试五：错误测试**   本测试主要测试第二种错误类型能否被识别，以及被识别之后进行错误处理后能否使得语法分析继续进行。   1. 测试样例及结果展示：给出能涵盖全部要素的测试样例 | | | | | |
| 四、实验测试结果  内容要点，供参考：  测试样例及结果展示：给出能涵盖全部要素的测试样例，包括正确源码样例及包含各种错误类型的样例   1. **简易计算器**   该测试样例生成的语法树如下：    通过语法树可以看出，分析结果事实上分成了几个较大的子树，对应源程序的几段语句：    第一段较大的子树如上图所示，是通过声明语句闭包生成的子树。对应到原程序中即为：    这一段通过最后的叶节点可以推出get(id, id, id);，也就是程序中的    后面三段if语句非常相似，选择其中一组进行展示 最后是put函数输出。控制台有输出grammar analyse success！，语法分析成功完成，没有出现错误。  可以看出，每一棵较大的子树均通过一行语句或是一个较大的闭包生成（比如for循环，while循环），而只要读取这棵子树的所有终结符，将其连在一起，就还原成为了这一行程序的语句。   1. **跑马灯**   该测试样例生成的语法树如下：    下面对各部分进行说明：    观察所有叶节点，连在一起可以发现，该句为  而原句为  分析成功。    原程序中，后面四句为给前面定义的四个变量赋值。此处的语法分析书成功的通过函数块闭包完成了对这部分的分析。将叶节点连起来就为：  语法分析成功。    后面的部分是while循环内包含if语句，语法分析器很好的分析出了这两种句型。if语句由于没有else if和else，所以否则闭包推出空。    最终是get，put的流输入输出。至此，语法分析，语法树绘制完成。   1. **自定义测试一：附加语法测试**     下面对主要测试的语法子树部分分别进行说明    该部分是对声明变量时声明数组的测试，图中红框内叶节点按顺序输出为  与测试程序中  对应，该文法测试成功    这部分是对后面的指针声明生成的语法子树。将叶节点拼接得到  与源程序中的  对应，该部分语法分析成功        后面的if，elif，else成功在树中显示出来，语法分析成功   1. **自定义测试二：附加语法测试**     下面对主要测试的语法子树部分分别进行说明    外圈的for循环以及相应符号成功识别    将叶节点连接起来得到：  成功与源程序对应：  该部分语法分析成功    见图中红框标记，while（true）被成功识别，put（a[i]）也被成功识别  该测试样例语法分析成功。  **自定义测试三：附加语法测试**      在测试程序中添加错误：op后面增加运算符。语法分析产生错误，在控制台输出错误类型为：分析表中此格为空，并同时输出当前符号为+，产生错误的行数为2 。按照错误处理方法，跳过+号，得到）。此时能够进行后续语法分析，输出Error Solver Finished，语法分析继续进行。   1. **自定义测试四：附加语法测试** 2. **自定义测试五：错误测试** | | | | | |
| 五、实验总结  1、实验过程中所遇到的问题及解决办法  2. 实验心得 | | | | | |