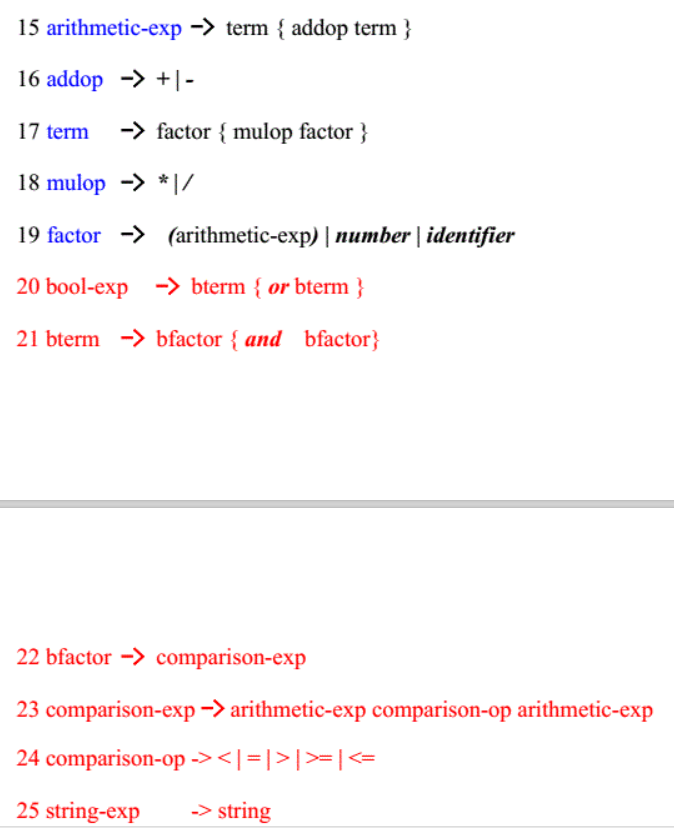
语法分析实验报告

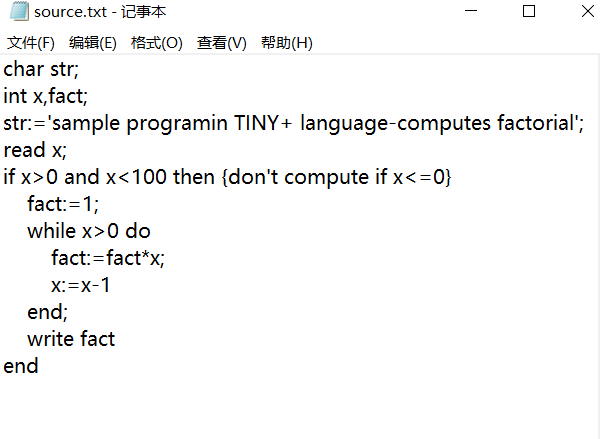
构造原理：



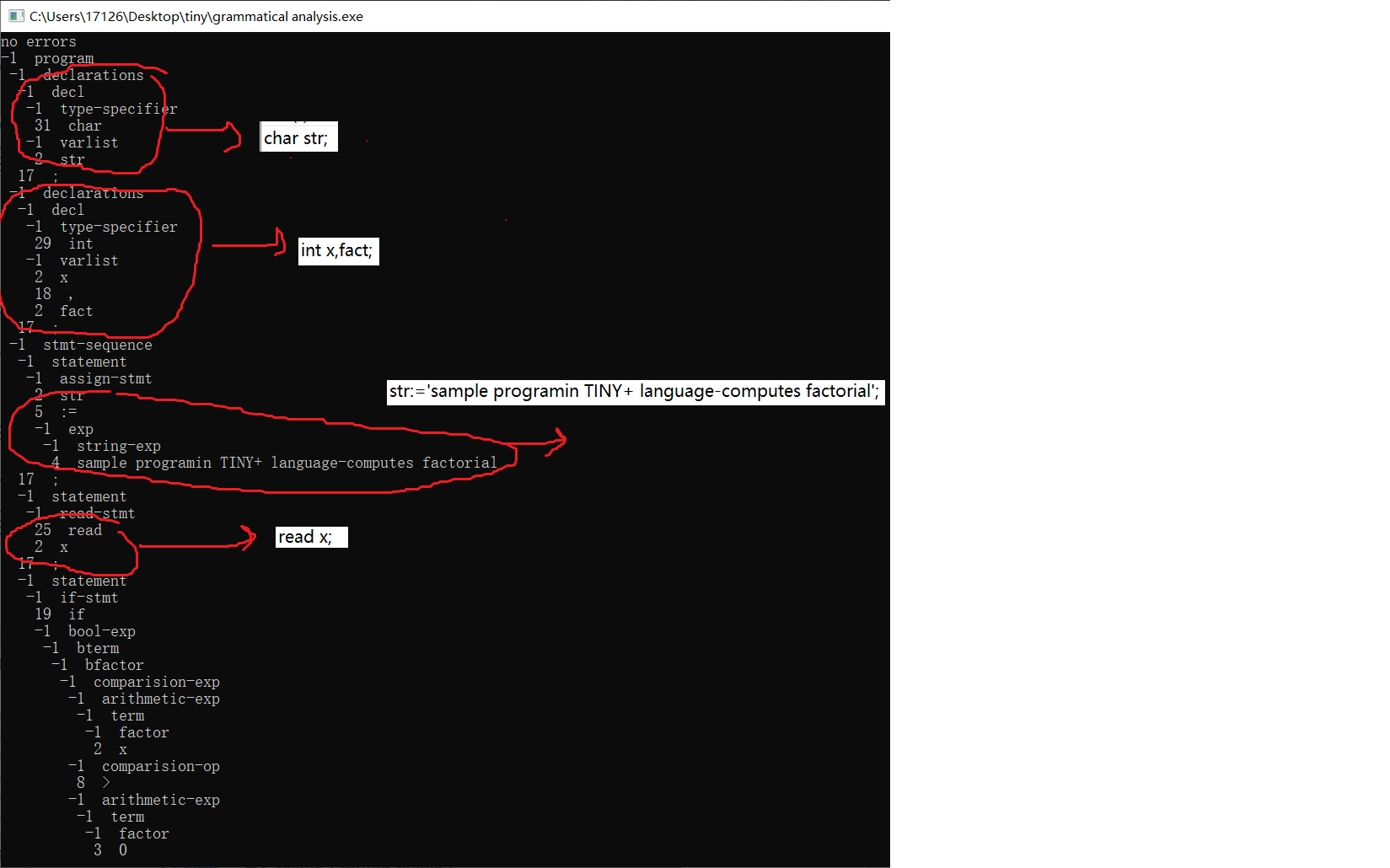


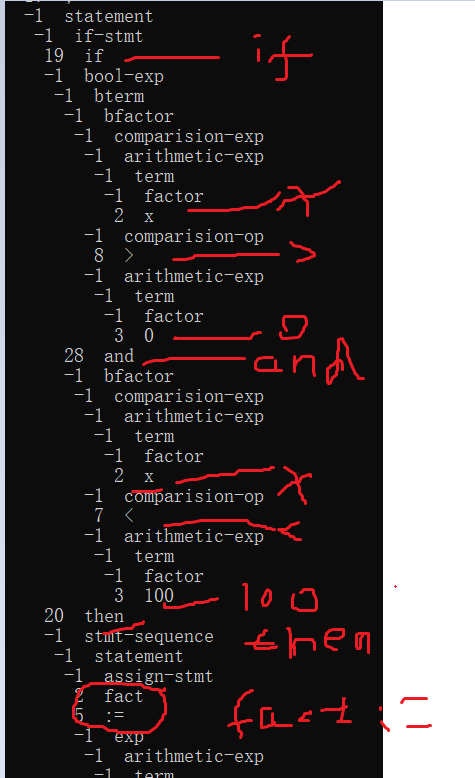
最终结果

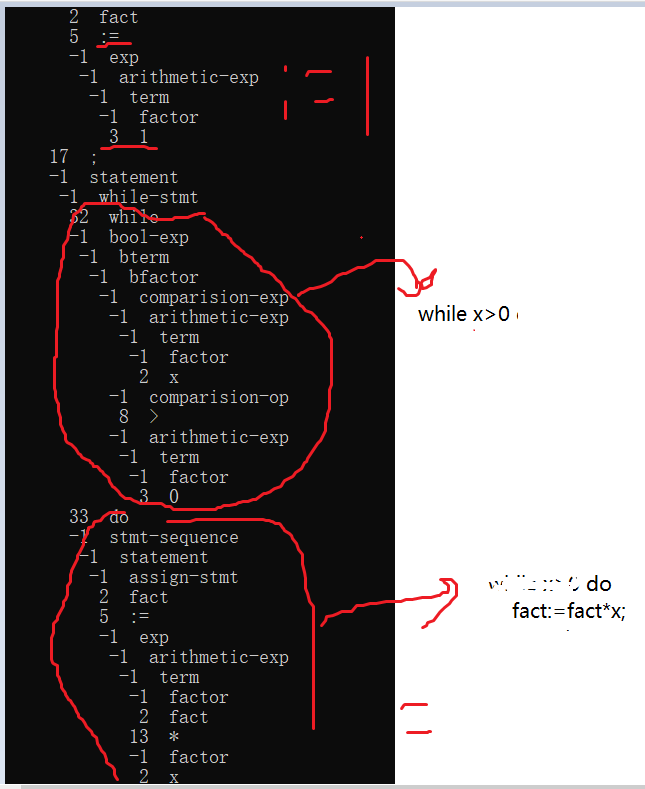
原文件：

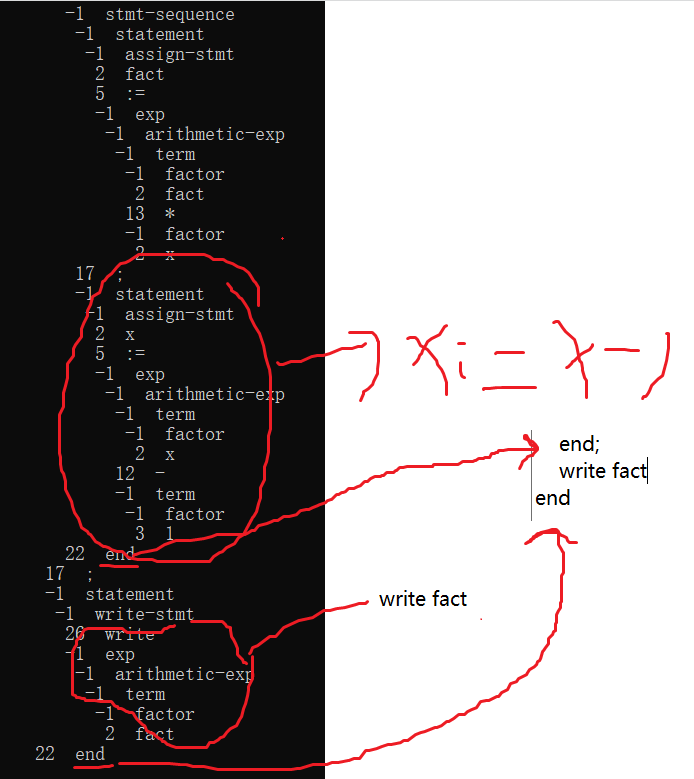


语法分析：

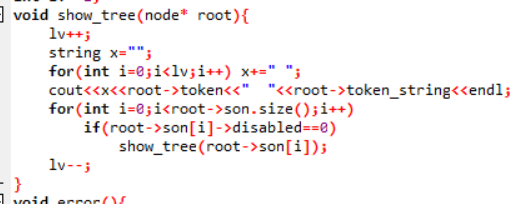






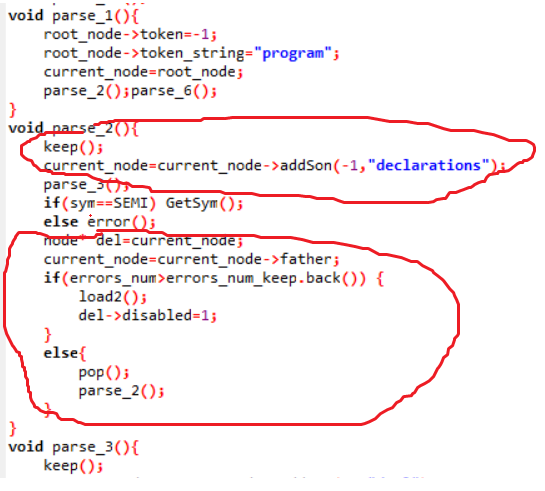


每行前面每有一个空格说明树多一层。树沿着从左到右的子树的顺序展开。输出能够确定唯一的树。



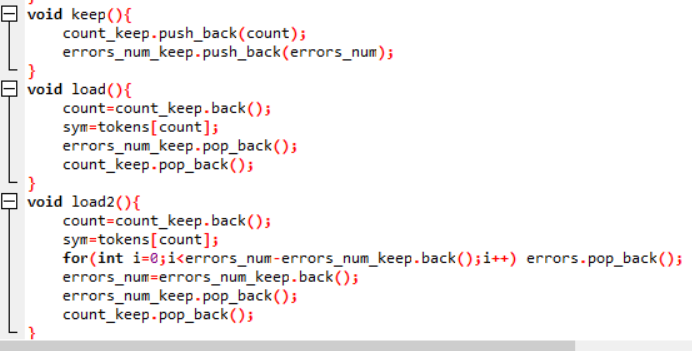
这里采用的是disable子节点的方法，之前采用的直接删除的方法，可能是因为删除不干净，导致树很混乱。

采用递归下降分析法，



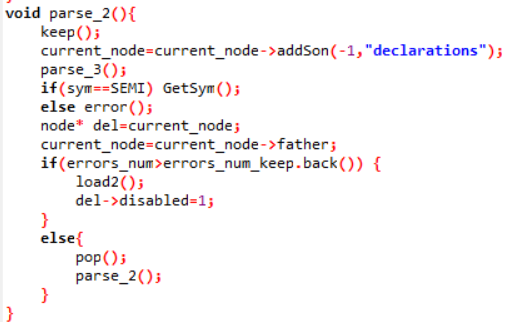
实现时，最关键的问题是需要尝试错误的递推方法，此时的结果不应加入树中。

因此，一般地，要通过红圈里的代码，使得如果在本parse里面发生了错误，就删除此节点，当前节点退回到父节点。读取之前的token指针的位置。

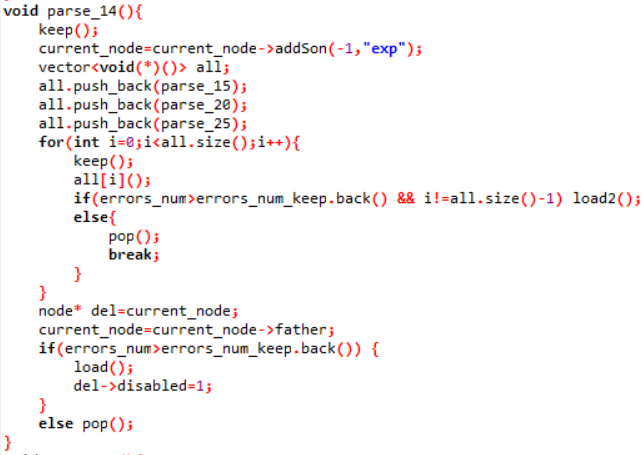


Load和load2的区别是load2会消除keep和该函数间产生的错误。而load却会保留这些错误。

下面通过具体parse分析：

例如parse\_2函数，

每次尝试decl;declaration时前先keep保存，如果尝试时发生错误就load2并停止尝试（期间产生的错误记录也会被消除），如果没有发现错误，则调用自己，直到发生错误，变为空符号。

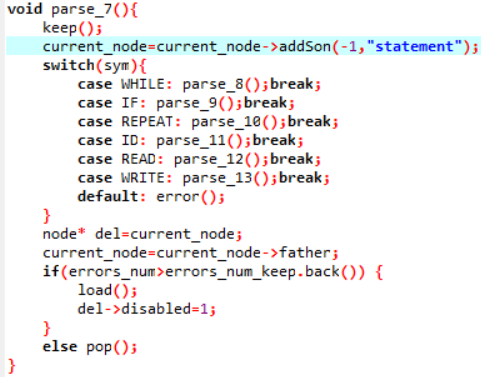
又比如parse\_14，

先把3个要尝试的对象放到vector里面，遍历尝试之，每次尝试之前keep保存，如果发生了错误，而且不是尝试的最后一个，则读取重新尝试。如果期间没有错误，则结束尝试，采用此结果。这样，即使全部都会出错，也会采取最后一种尝试方案的展开方式。而如果有不会出错的展开，则会采取之。

最后根据选用的展开方式，再次检验中间有没有错误，如果有，则放弃该节点。

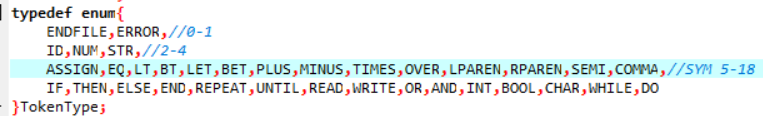
一言以蔽之，就是parse函数运行过程中有错误时，在语法树上删除该函数对应节点。即使有的尝试中函数运行时没有error，因为这条路径不可行，其某个祖先节点一定会删除，导致这个函数也被忽略。

也就是说，一种可行的展开，一定对于中间没有error的展开路径。而不存在一种错误的展开，其展开路径上没有error。

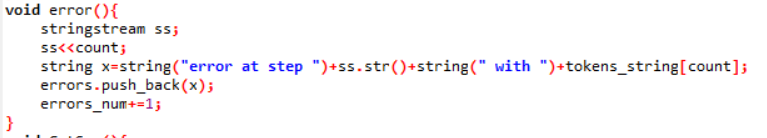
在parse\_7里，

因为其后面的展开选择第一个都是终止符号，因此通过switch的方式简化了选择过程。

获得tokens的方法和实验一一样，只不过，实验一把所有key都简单归类为key，而本实验，应该细分每一种key，每个key对应一个tokentype



Error函数



，存入error信息，errors\_num加一

Getsym函数，每次读一个终止符（不可再展开的符号），直接加到当前字符的子节点里。因为每次parse调用完都会返回调用前的节点，因此，在一个parse函数里读取终止符时，其一定可以直接加入到当前节点的子节点中。所有的叶节点的加入都是通过该函数。

