**概述**

（这篇文档整体是重构过的，甚至，在我开始重构这篇文档的时候，仍然有一定数量的测试样例过不去——这意味着某些时候，可能仍然需要部分或全部重构这篇文档。）

语法分析的输入是词法分析的输出的token流，输出则是一棵语法树，对于语法分析阶段的工作，请务必结合符号表的文档一起看（我在符号表这个内容上面吃了很大的亏）

以下，我将分为算法，符号表相关内容，错误处理等几个部分来进行详解。

另外，在预处理阶段我曾经提到过，预编译问题，我认为，在这一步也需要进行考虑，直接把一些头文件的语法树通过某种方式进行输出即可——毕竟从某种角度来说，语法树的搜索才是最耗时的。

至于为什么不把预编译的文件进行读入这件事情换到之后的阶段，是因为在后面阶段再引入那么这一阶段的很多事情就没法做了，举个例子，假设我想使用printf，如果没有引入了stdio的符号表和语法树，那么在这个阶段就会报错。因此，在这一阶段，需要完成将语法树转录到文件的工作，便于之前的预处理器读取的时候，直接读取已经编译好的文件的语法树。

**相关parser算法以及文法方面的记录**

编译原理课上涉及过的典型算法我就不想讲了。总不会有人最经典的LL和LR，LALR什么的都一点也不了解，然后来看我这个编译器怎么搞的吧？这倒并不是说得精通他们，但是，总该有个概念。

（这段文字内容写的我一言难尽，需要查资料，并且满满的翻译腔，并且，我表示仍然需要有太多东西去补充他，尽管我无意在一个开发日志文档里面去写文献综述，笑。。。）

* **GCC、Clang的语法分析阶段的情况**

Clang一直都是手工编码的递归下降分析。

对于gcc，在某个版本之前他使用的是一个自动化的yacc bison解析器，然而到4.x之后，则使用了一个新的手写的parser（详见 <https://gcc.gnu.org/wiki/New_C_Parser> ）。

这里面存在一些历史原因，但从自动解析器改到手工编码，可能还是手工编码的优势更大一些吧。据说，原先使用的LALR方法，在错误处理上面存在一些问题。即使在某个地方出错了，仍然要多次递归返回之后，才能确定到底错在哪里——当然，我的mcc由于其记忆化搜索的设计，同样出现了这个问题。

在某种不知从哪里来的“定理”看来，似乎对于工业级别的C和C++编译器就应该使用手工编码来完成，而不是所谓的自动词法分析器。（对于C++来说，这样似乎没错，C++的文法和语义，对于编译器玩家来说，简直是一种灾难）

另外，GCC和clang他们的解析器本身作为一个C和C++（应该还包括C#）都尽量兼容的东西，相对于只考虑较为简单的C文法的编译器来说，这两者的难度并不是一回事，例如GCC试图在不同语言间共享代码，而这显然很容易在一些自动解析器中出现一些问题。所以他们试图手工编码这点我觉得可以理解。

就实践而言，一个通用性的文法解析器，例如GLR可能能够分析C和C++本身的上下文敏感问题和文法中的二义性问题（当然会在语义分析阶段变得无歧义，不然怎么玩对吧）。然而我没做过，也没找到足够多的实验来表明这样做，在最坏情况下，仍然有足够优秀或者最起码相近的性能，因为通用性的文法解析器，其最坏性能一定是超过O（n）的。这来源于处理文法的二义性，相比之下虽然确定性的做法也许不优雅，但是，他们起码可以保证O（n）。

* **C本身的文法限制问题**

一个不可否认的一点是，C是一个上下文敏感的文法，而不是一个CFG文法，就比如typedef name，这需要做一件符号表hack的事情，在语法分析阶段就建立符号表然后去查找来确定语义。正如之前提到的，gcc在3.x.x某个版本之前仍然使用的parser自动生成器。这里面当然有解析器速度和debug的考虑，但是显然，bison去解析C文法，这么做据说虽然难看，但是仍然是可行的。

当然，使用GLR等等算法，貌似可以解决这种（不考虑上下文带来的）二义性问题，但是这带来了额外的代价——当存在二义性时，会生成多个栈，在某些情况下会导致显著的性能问题（不过我个人觉得，如果出现了，那么是设计文法的人脑子进水了的缘故）。

* **GLR GLL parser algorithm和Early**

对于编译原理课上讲的LR和LL系列的分析方法，这都是**确定性的分析方法**，这些文法都是假定不存在二义性的，而这里所谓的GLR系的，更确切的讲，应当是通用性的分析方法，换而言之，主要是针对文法中的二义性，才衍生出了不确定性的问题。

在不确定性的文法中。深度优先和广度优先是两种不同的方法，然而深度优先在我看来只不过是自顶向下和自底向上加了个回溯。GLR和Early则是广度优先的算法，两者貌似一脉相承。

我并不想讨论Ungar这样的性能O（n^3）的算法，因为他们在实践中毫无意义，当然他应该存在学术上的意义吧（比如，可以让文献综述多写几个字来凑字数）。

这是通用文法解析的一些大致情况。就实践而言，正如我上面所述，虽然确定性的做法也许不优雅，但是，他们起码可以保证O（n）。

当确定性的时候，就是LR，这不必多说，然而当存在二义性的时候，分析表中，存在多个可能的动作，而分析栈也会随之分裂，因为两种情况都可能存在，这就是所谓的宽度优先，而不是先沿着某条路线往下走。如果高层的结果相同，还可以进行压缩

GLL的思想我感觉和GLR差不多，只不过它是基于LL情况下的。

**记忆化搜索**

* **算法**

对于算法，从理想的算法来看，应当使用成熟的LL或者LR等等分析方法，但是，由于我是个人编译器玩家，又不打算使用yacc等自动工具之后，我所能做的选择就极为有限了。如果想要采用LR或者LL分析法，那么我不得不手动构建一个自动化的工具。

即便如此，无论是yacc还是其他，都需要在符号表以及作用域方面做一定的处理，因此在yacc源文件后面都会附加上一个源代码段。当这些部分的内容相互耦合之后，调试所需要的成本实在是个人玩家无法承受的（C的文法着实又臭又长）

综合考虑，我不得不抛弃这些算法，另一种考量方法是手工编码，gcc和ucc等不少编译器都是这么做的，但我仍然希望能有一定程度的自动化，而不是按照每条规则来进行编码，最后我选择了递归下降分析——并不是预测分析那个，而是最原始的暴力递归。

当然，这么做带来的问题显而易见，那就是性能下降。

**记忆化搜索算法第一部分，剪枝**

在提升性能方面，考虑一下LR分析法的过程，他先是构建项目集规范族，然后转化为一个表格——显而易见的是，前者构建项目集规范族，是一个暴力搜索的过程，而后面构建的表格，则是可以看做对于前者的一个记忆，因此我认为，如果给暴力递归加上一个记忆，那么一定可以显著改善他的性能，这也是记忆化搜索的思想所在。

然而，记忆化搜索同样存在递归层数的限制问题，这个问题，由于文法树的特殊性，我会在第二部分的搜索算法中去描述。

对于第一部分的记忆，我认为更恰当的描述是，记住当前读取的token到底要往哪个或者哪几个子树去搜索。对于某棵子树，一定搜不到的情况，那么可以直接舍弃。这样剪枝可以极大的减少搜索空间。

~~具体算法如下，一开始，对于某个终结符的某条搜索路径，记忆中，一定是没有记录的，因此使用暴搜，在这个过程中，如果搜索到其他终结符的路径，也同样记录下来。~~

~~随后，当再次需要用到这个终结符的该搜索路径时，可以直接从记忆中寻找，并通过简单判断路径上相关节点是否符合来决定是否使用，而无需再次暴力搜索。~~

先暴搜建立记忆，然后再根据记忆去匹配。

从渐进的时间复杂度来看，当一个文法树确定下来后，那么只有第一次需要暴搜，之后的都可以根据记忆中的内容去判断，因此渐进复杂度应当是O（n）——尽管由于记忆的查找，这个O（n）可能是LR的O（n）的十几倍。在实际测试当中，每个token平均约需要查询100个节点左右（具体数据有所浮动），才能确定下来，尽管如此，parser代码看似很多，但是，里面包含了大量的switch case或者if else 语句，因此每个节点实际运行的代码并没有那么夸张，以现在的芯片性能而言，毛毛雨啦，就我实机运行的感觉而言，当扔掉了输出打印之后，测试样例并没有感觉到明显的延迟。

另外我不得不指出，expression这一长串长得还差不多的子树节点对于性能的影响。很多时候，这些都是空节点，没有任何实际意义，而且还会增加递归层数。

在这一点上，对于ucc，声明和expr分开的做法，我表示非常羡慕，因为可以节省非常多的性能，专门处理expr类型。

对于记忆化搜索的优点，我认为主要还是在保持性能的同时，尽量做到代码和调试方面的简单明了——这对于整个项目同样是相当重要的。

* **数据结构和具体算法**

对于暴搜的部分我想没什么好说的，主要是记忆，这个记忆我也用一个表格来记忆，一个轴表示当前遇到的非终结符，另一个轴表示想要搜索的终结符，也就是目前的输入token。

有没有觉得和LL里面的预测分析表差不多？但一个巨大的差别在于，LL分析方法里面的预测分析表是确定的，而在这里，则不是，因为可能存在多条路径从一个非终结符到某个特定的终结符。因此在表格的每一项里面我都用一个vec来使用，表示当前非终结符想要搜索某个终结符可以选择的路径上下一个节点，并且保证，下一个节点一定是当前非终结符所代表的规则的子节点。

为了达成这个记忆，还需要一个额外的trace\_vec来记录当搜到某个终结符时所经过的路径。从而能够完成一个记忆的更新。

在实际写代码的过程中，我发现了一个巨大的问题在于搜索，如果这是一棵标准的树，那么不会存在这样的问题，可惜并不是，在实际的代码中，这是一棵存在回路的树，举个例子，expression遍历到最后的primary expression下有个（expression）的表达式子。从而造成环路问题。同时，link规则要求规则下必须存在前置的符号才能遍历后继的符号，如果只是暴搜那么没这么麻烦，在发现link规则中某个符号中断之后不再搜索后继符号，可是加了记忆之后，这必然导致后继符号没有搜索到，这种情况下，记忆肯定是残缺的。如果中断之后继续搜索，那么环路问题又会造成无限递归（在原本暴搜的版本中没有环路问题是因为保证了没有左递归，当左边符号不对可以直接返回，但是现在加上后继符号了，事情大条了）。

代码会变得非常难以处理，然而，我认为，这里的问题在于，试图将记忆的过程和匹配的过程混合起来，这种耦合导致了代码难以处理。

因此，我做了如下改进，**先暴搜做记忆，然后再根据记忆去做匹配（终究是功力不够没本事写进一个递归里面去）**。

实际代码中，我将规则设置为4种规则，因此还需要分别对四种规则情况下的算法进行一个处理。然而在匹配过程中，只需要考虑link和或类型的规则即可，否则遇到star或者？类型出现次数为0的情况就会有问题，并且这两种规则的子规则肯定只有一个，两种类型碰到记忆中没有的，就返回，对于或类型，则可以根据记忆中的方向进行搜索。

当然，理论上可以根据下一个符号进行进一步的判别，优点是，搜索方向必然确定，但是那跟LL有啥区别呢。

另外，即使做了上述的处理，仍然需要考虑在暴搜记忆的过程中，需要面对的环路问题，环路的确定很容易，由于记录了trace，所以只需要和trace中的记录进行对比即可，但是，环路的处理则不容易。显然，当碰到环路的时候，对于该条规则以下的规则必须停止递归，否则必然无限递归了，随后，需要更新该条路径上，所有可能搜索到的符号——在这一步我遇到了麻烦。

* **想法一**

最开始的想法是，当碰到递归的符号时，沿着trace往回，将环路的所有后续符号集合加入进去做更新，但这存在问题。

Well，来个例子吧。（同样，大写，非终结符，小写终结符）

|  |
| --- |
| A –> B|C  B -> a|b|A  C -> c|d|A |

显然，从ABC均能够通过某种方式搜索到abcd（尽管正常情况下我不会那么蠢的去写成这样）。

从A出发先搜索B，那么a和b都能作为AB的后续符号集合，当碰到B下面的A的时候，停止并更新，这没问题，此时AB都有a和b的记忆，但是当搜了C之后，A已经搞定了，C的部分我可以通过trace，发现碰到了A，那么我可以把此时A已有的可能性（因为这里检测了环路），a和b均加入到C的集合中，也没问题，那么B怎么办呢，我对此毫无办法，他仍然是a和b，而缺少c和d。

* **想法二**

第二个想法是，对每条规则都建立一个vec，第一遍先按照暴搜的办法去处理，在这个过程中，碰到非环路的非终结符，往下继续搜索，碰到环路的非终结符，放入trace里面环路之后的规则的vec中，并且同样不继续往下搜索，碰到非环路的终结符，则将trace中的记忆也更新。第二遍，按照树的方式遍历第二遍，碰到非环路的非终结符，往下搜索，遇到环路的非终结符，直接返回就行了，碰到非环路的终结符，啥也不做返回。

在这两遍更新中，同样用了记忆化的思想，使用一个向量表示是否搜索过该规则，如果搜索过，那么直接将该规则的结果向trace传递即可，而不继续往下搜索。

以前面举的例子为例，第一遍之后，在表格中A行的abcd均有对应的搜索方向，而B只有a和b，C有c和d，另外，BC的vec中均有A，继续按照树的方式遍历，A的vec中没有A，更新（啥也没做），然后搜子节点B和C，然后B和C发现有另外两个符号还有方向，从而在对应的格子里面填入A，同时trace里面节点的记忆——但是只更新到环路节点就不更新了，因为再往上肯定是不会有变化的。

这样一来，应该是稳健的算法了。（PS：后面又出错了，真的是写bug而不是写代码）

对于记忆的大体算法就是这样了，当然实践过程中的代码可能有些出入，请结合实际代码理解。

**记忆化搜索算法第二部分，记忆化搜索**

（两个月后继续回来考古）

今天遇到一件非常非常尴尬的事情。迫使我不得不重新思考这算法，emmm，大概就是，遇到了前文所述的递归层数限制的问题，我递归弄了n个条件表达式，然后，gcc正常搞定，mcc运行了半天（其实内存足够大应该是算的出来的吧），不巧的是，炸栈以及内存极度膨胀了（据目测，应该是每次弄表达式，都弄很深一大堆，所以非常不客气的炸了），并且即使不炸，速度和性能也确实算不上很高。

然后我看了一下约束要求，大概可以嵌套63层是最起码的要求。那么我是大概多少层炸的栈呢。Emm15、6层吧。我估计性能要提升一个n或者起码logn才行。

我查看了内存申请和释放的次数，当递归13层的时候，235096291——而实际上ast总共只有1k多个节点，代码行数只有40多行。

先看看还能不能拯救一下吧。简而言之，导致这么严重问题的根源在于树的深度太深了，以至于深度遍历的规模变得非常巨大。

就目前而言，这个多层嵌套表达式之后炸栈的问题虽然严重，但是语法树中表达式子树节点太深的问题，早就有过印象，只不过当初没有处理罢了，而就目前来看，别的貌似并不受到太大的影响。所以要做的事情如下：

第一个步骤，列举一下C标准说明了的翻译数量限制。

第二个步骤，分别验证一下这些限制，到底能不能通过，把不能通过的情况列举出来。

第三个步骤，分析一下，这些因为限制引起的炸栈问题到底是广泛存在还是就表达式的这个特例。

第四个步骤，如果是广泛存在的这个问题，好吧，算法可以重新设计了。如果只是表达式这一类节点单独存在的问题，那么可以专门为此做一个树遍历的压缩算法去改进。

另外还存在貌似内存泄漏问题。

（已经改完内存池，估计是最终版本的内存池了，已经非常快了）。

C标准对于翻译的各种数量限制如下：

|  |
| --- |
| 127层嵌套block  63层嵌套conditional inclusion（不知道是啥，看了一下，反正预编译里面的）  12层用point array function来修饰基本类型（这个不用测试，轻轻松松）  Declarator可以嵌套63层  表达式嵌套63层（就是前面出事情的那个地方）  宏名称或者internal的标识符起码有63个字符（不用测试，可以有）  External的标识符起码有31个字符（这里还有什么Unicode字符算几个的说明，不管他）  一个翻译单元内4095个external标识符  一个block里面511个标识符（不用测试，用的可扩展的hash，而且那个hash数量我给的非常大）  一个翻译单元内4095个宏定义  Function定义和call里面127个参数（parser里面肯定没问题，因为是list）  宏定义和使用中127个参数  一行内4095个符号  连接之后一个string最大4095个字符  65535 bytes in an object（没懂，应该说的是object文件）  #include可以递归15层  Switch可以有1023个case（parser这里不会出问题）  Struct里面可以有1023个对象  Enum里面至少1023个enum  Struct里面63层嵌套。（测了一半32，没啥延迟，那基本没问题了，有问题已经像那个表达式嵌套炸栈加很高延迟了，那个数量级增长非常快的） |

上述是我遇到的问题。最后为了解决这些问题，我对代码做了非常多的变动。

* **记忆化搜索**

因此，各种魔改之下，最后得到了记忆化搜索的第二部分的算法（我感觉，这才是相对而言更加贴合记忆化搜索这个名词的算法）

首先我们来定义一下，什么算作可重用的模块？

我认为，当我们试图对token序列中，某一个token开始的后续符号，按照某一个规则rule来规约的时候，就算做可重用的模块——如果，**从某个确定的符号按照某个确定的规则来规约，必然只有一种最优的结果**，那么，对每一个token都记录下，在该token按照各种规则rule来进行规约的子树结果——由于rule数量是确定的，其上限最多不超过非终结符的数量（终结符直接就拿来用了，没必要算他）。因此这个空间复杂度是O(n)的。

当重复遍历到该token，并试图使用同样的rule进行规约的时候，那么我们无需重复进行递归遍历，只需要在该token的rule列表中找到对应rule的那个子树。这个子树的记忆就是记忆化搜索的内涵所在。

使用这种方法，对于多层expr嵌套的情况，实际的遍历时间是令我相当满意的。

关于这种方法的一些细节。

首先，由于对子树的遍历存在大量的重复遍历，如果我在代码中删除了某个节点的数据，非常容易造成重复删除等等复杂内存问题。最后我迫于无奈，所有的这些节点我都选择了保留，哪怕其中有不少节点我并未使用。

如果想要不保留也很简单——重新开一棵树，按照原有的内容建立一遍，然后把遍历过程中的那些全都删掉就完了。

当然，我这么做同样存在一个问题，就是，很多节点可能最终并未被使用。那么，这个未被使用的节点，并且未被销毁掉的节点，意味着可能存在内存泄漏（使用内存池可以避免这个问题，但是，所占用一定空间是不可避免的）

在实际使用中，我对此进行空间的计量，样例程序中，未使用节点约是使用节点的三分之一左右。还是可以接受的范围。实际上，对于这棵树，我每个节点的token值都是其最左子树节点的token值（以此递归）。所以，每个token对应的节点值都是某一条由最左孩子构成的链，这个深度远远低于rule的总数，所以每个token的rule链也是相当有限的，即使其全部都是未使用节点，那么也是O(n)的大小。更何况有大量节点是被使用的，因此，一个O（n）上限（实际应该低于这个）的未使用空间数量，本身是可以接受的。

因此，没必要重开一棵树复制过去。浪费时间。

第二个难点在于符号表的问题。符号表和抽象语法树节点并非一一对应。而且在语法树生成阶段是不可能省略的。比如typedef name这种，对于使用他的节点而言，必然是在某个上层节点的兄弟节点所定义的，如果没有符号表，这访问不到啊。所以不可能省略。

吊诡的事情发生了，由于语法树多条路径都可能调用某个节点，如果还是原来那样暴力递归，没问题，全都扔掉再全都建立起来。可是现在不是这样了。比如ABCD都是不同的符号表，A走路径B到D，这时候建立起了一堆符号表的父子关系。D的爸爸符号表是B，A还可以走C到D，那么C和D之间的符号表，没有任何关系，假设ACD是实际采用的路径，那么这符号表访问就出现问题了。实际的节点和符号表的关系远比这复杂的多。这样一来，如何处理符号表就成了大问题。

这里符号表的作用，只在于查找typedef和enum的id，由于这些定义的特殊性，肯定不可能在function prototype scope里面被定义——尽管这在语法分析过程中没有任何问题，虽然后面会报错，然而，他们本身不应该在函数声明之外被访问到，所以这没有问题。

对于剩下三个scope，本身是相互包含的。而在typedef和enum的查找中，不涉及横向的查找，只有向上不断的查找，这并不会导致任何的问题。所以，尽管符号表看上去因为这种到处拼装的算法，看上去支离破碎，但是语法树本身没问题的。

当然，如果没有enum和typedef的语法——尤其是后者的乱七八糟的判定，这在很多语法中根本不会存在，那么这个算法可以应用的语法应该更多一些。

随后，我设计了一个步骤，让他在抽象语法树构建起来之后，根据ast去重建了符号表树。这就好了。

（有一说一，正常人做这个事情都是直接拿成熟的算法和工具去构建的吧）

* 这样算对于文法的规则要求

不说符合什么文法的语法分析器算法都是臭流氓。对于这件事情，我必须承认，C所描述的文法可以满足这个算法，但是并不等于这个算法可以符合所有文法。

举个例子，还记得前面所描述的假定，对于某个token开始，按照某个rule来弄最长匹配只有一个的条件吗，LR文法里面有个移入-规约冲突。而在处理这个冲突的时候，按照最长匹配的原则来使用，永远选择的都是较长的那种，而事实上，较长的那种，后面多出来的token，有可能被后续的其他兄弟节点所正确规约，这样就麻了爪子。

我在实际分析中也遇到了这个问题。

举个例子吧继续

|  |
| --- |
| Typedef int c\_test;  Int t(signed long int c\_test)  {} |

问题主要在于函数的参数列表，建议参照parameter-list的C标准中的文法去了解。

这里的c\_test 标识符显然发生了overlap，但是对于函数参数列表的语法分析过程中，按照我所使用的文法（其实只要是照着C标准抄，不管文法写成啥样都无所谓），由于那个typedef的存在，c\_test也会被识别为某种specify。就是由于最长匹配规则，才导致的参数列表的四个token都是specify。而后面应该有的declarator则是被识别成了可选的abstract-declarator。显然这是不对的对吧。

其原因，就在于c\_test不加约束的，被当做specify那条规则——因为这样在分析specify的时候，用到了4个token，肯定比3个token要长，而后面的declarator就没有token用了，偏偏有一条规则，在里面，后面的abstract-declarator是可选的，于是悲剧了。

好在，C文法里面有个约束限制，就是说，specify只能在某几种类型中选择一种，signed long int是其中一种，而c\_test作为typedef name，作为另一种，因为这样的文法分析，使用了两种类型，所以一定是出错了的。

假设没有这样一条限制约束，没有任何办法能够得到c\_test是一个重载的变量名称这一点。

上面这些问题，都是由于所谓的最长匹配规则导致的。这在移入规约冲突的那些样例中，体现的非常明显。

当然，如果没有这条约束，他本身是一个二义性的文法。

在此，由于这不是一篇学术类的文档，我不对这个算法做相关更加细致的文法分析，什么LL、LR啦，关我啥事儿，但需要举出这个样例，来表达一件事情，那就是这个算法本身存在一定的文法方面的约束，尽管他对于我目前采用的C文法是能够使用的（我的测试样例应该会涵盖了所有的非终结符的规则，所以不太存在没考虑到的情况），却不意味着他可以用于所有的文法分析。可能存在一些例外，导致合法的文法无法使用这种方法进行文法分析的情况存在。（包括上面这个样例，我也是在每个节点的后处理中加了一条特殊规则才解决的）

**抽象语法树**

这个概念本身我想对编译器有所了解的人都知道，但是我仍然要把一些问题放在这边。

虽然我给树取名为ast，但是严格而言这就只是一个parse tree。毕竟很多处理基本就是没做。但是，对于抽象语法树到底该长什么样，我仍然觉得非常的迷惑。我查阅了一定数量的资料，大部分网络资料对于具体语法树（也就是语法分析树）和抽象语法树的区别讲的语焉不详，让人恼火。

我可以确定的几点为，首先是逗号分号这些分隔符不应当存在，括号什么的也不应该有。这些是可以确定的。

而有些更加优化的ast设计中，比如if语句，会指定其条件部分，满足条件后的语句部分，还比如表达式语句中，会像我所需要的树压缩一样，把那些乱七八糟的节点都砍掉，直接用一个运算符来代替。

一个让我迷惑的地方就在于，其优化程度需要达到怎样的程度才算作抽象语法树，这貌似是编译器自己决定的。换句话说，由于树形状也是一种中间表示的形式，抽象语法树和中间表示的区分在哪里呢。优化的越少，越接近原生的具体语法树，而优化的很多的情况下，这和一个较高层级的中间表示又有什么差别呢？反正我没查到资料。如果为此而再设计一种抽象语法树的表示，我认为大可不必。（如果有人能够清晰地给出抽象语法树和这两者的区别，欢迎告诉我）

而且现代编译器比如gcc都会有多层的中间表示，按照鲸书的说法，HIR到MIR到LIR。我想所说的中间表示都是MIR吧。LIR则大多数情况下都是被当做和目标代码一对一的表示的。

另一个让我迷惑的地方在于，他本身应当所出于的阶段或者说生成抽象语法树的阶段，到底是在语法分析后得到抽象语法树还是在语义分析阶段得到呢？我想无论是我还是谁，都会说，肯定是语法阶段。

有的说“这将抽象语法树与具体语法树区分开来，具体语法树通常被指定为解析树，解析树通常由解析器在源代码翻译和编译过程中构建。一旦建立，通过后续处理，例如上下文分析，将附加信息添加到AST中”，换而言之，这是在解析之后的后续处理，并且还需要根据上下文的分析，加入附加信息才构建的ast，那么，这就必须在语法分析阶段的后一个阶段，语义分析经过检查再附加信息了咯？

在mcc的实现中，在语法分析阶段，每个节点在完成语法分析之后还有一定的后处理，用于处理这些事情，这是我个人的理解。

但是，语法分析和语义分析，在某些文法的情况下，本身也是可以混在一起的，尤其是类似手工编码的情况，某些语义分析和错误处理和前一批次的语法分析很难区分开来。

我所看到的一个重要的例子，就是说，CFG的文法，有些文法细节是CFG无法表达的，比如自定义的类型，同样拿前面说起我的记忆化搜索所需要满足的文法要求里面，那个自定义类型的例子来说，这是需要在每个节点结束之后的parser后处理中来处理的。

最后，在mcc的实现里面，我只做一些在语义分析阶段不会存在实际意义的节点的删除，不仅仅是逗号，还比如说，“||”之类的，但并不包括一些有确切含义的情况，比如，我的沙雕文法设计（怪我）让多维度的数组变成了一堆横向的节点，如果不加方括号，那就没人分的清了，当然，引入新的非终结符（实际上是不删除）可以解决这个问题，只是这太麻烦了，所以并不是所有的括号都被删掉。

对于压缩表达式树深度的部分，我认为应当做，当然这意味着需要引入新的东西，修改一些原有的问题。但是，一个问题来了，很多的运算符都被归类到某个类型的节点中去，而提升层级本身是个好想法，可是对于“||”如果出现一堆并列的情况下，按照提升操作类型作为父节点，操作数作为其两个子节点这种做法，这反而会加深树的深度，并不是我不愿意做，只是，明明一个for循环就能解决，所以我选择缩减树，但是不会改变其节点类型。

因此，我想，我在这里经过这些后处理之后的语法，可以被视作一种抽象语法树，虽然他本身更加贴近于原生的具体语法树，可谁叫我找不到一个明晰的抽象语法树的说明呢，每个编译器还自己设计自己的，真讨厌。

**符号表管理**

另外还有一篇文档用于专门描述符号表的工作，首先让我说明一下这两部分的区别，对于另一篇文档的符号表管理，里面的内容主要是符号表和哈希表的数据结构和算法，而在这篇文档中的符号表管理，则是涉及到在遍历语法树的过程中，何时进行符号的插入，使用符号表进行判断当前遍历是否符合要求等等内容，侧重的是在语法分析层次与符号表的交互的问题。

关于这部分我其实遇到过相当多的错误和问题，甚至相关和符号表的代码全部进行了重构，比如说由于作用域的不同，使得本来应当在另一个子作用域中定义的符号，因为符号相同，被当前子作用域视作已经定义的符号，从而导致重复定义错误，以及函数定义和声明的重复性，函数名称的重复定义的判别等等问题。

建议先查看符号表的那篇文档了解相关内容。

另外，我认为一个非常重要的任务是，厘清语法分析和语义分析之间的任务差异，也就是说，哪些任务是语法分析应该做的，哪些任务是语义分析应该做的。

我们的任务是什么？——建立起一棵正确的语法树！

其他乱七八糟的要不要检查？没必要！

* **总的算法处理**

以下分为几个步骤来描述：

首先是作用域和命名空间的划分，相关内容下面会说，当访问到当前节点，为特定的几个类型时，建立一个子作用域和新开一个子命名空间。

其次，在遍历所有子节点之后回到当前节点之后，对子树的标识符进行一个处理，原因是很多类型的符号去进行判定，需要前后相关的查看，并且无法确定前后相关的长度，只有子树构建之后才能进行判断。并且必须在子树建立之后立即操作，否则比如先定义再使用这样的规则下，在某个定义之后需要立即进行符号表插入，不然后续没法使用。在这一步，需要进行作用域和命名空间的判断并进行符号表插入。

进行后处理的另一个优势在于，此时插入符号，整棵树的形状一定是确定的。

第三步则是当新搜索到某个终结符，并且该终结符为identifier类型时，结合符号表和作用域命名空间，所做的判断算法，如果当前是一个定义，那么检查是否重复定义，是否为一个typedef name类型，如果当前是一个引用，那么检查是否已经定义过，或者是goto的label类型，无需检查（当然会在语义分析或者更之后的时候进行检查）。

* **作用域和name space**

这一段落描述的作用域的内容，主要是说明，当何时，需要创建一个子作用域，我会在下面一一举例说明。请注意，不能简单的进行存在左花括号，就新开一个子作用域这样的判定，举个例子，enum结构里面，花括号里面的符号，都会被当做上一层作用域的符号所判定。

（以下内容重构）

在一开始，我仅仅凭借着自己过去的经验，以及个别特殊的样例来处理作用域的问题，然而很快我意识到，这样的特殊样例非常多而且难以全部列举用于验证，后来，我仔细查看了文档和找了相关说明来了解标识符的作用域问题，重构了以下内容，并进行说明，包含两部分，一是C标准对于作用域和namespace的说明，以及网上找到的文档，加之自己的理解，二是对于这样的含义，如何使用算法进行描述。

（我尽量将内容描述的结构化，但是由于需要考虑的情况的细碎化，免不了东一榔头西一棍的，还请谅解）

* + C标准

C标准中对于符号的作用域的部分在6.2节。

* + - 标识符可以具有的类型（和后面的类型系统建立相关）

一个identifier可以具有如下几种类型：

对象名，函数名，tag或者是struct的成员，union的名称，枚举类型，typedef名称，标签label名称，宏名或者宏参数（预处理阶段）

* + - 作用域的类型

存在四种类型的作用域

Function，file，block以及function prototype类型

最后一种主要是函数声明的时候，括号内的参数的作用域，这个参数的名称和具体函数定义的时候的参数名称没啥关系，只要参数类型一样就可以了。

File的作用域类型很好理解，相当于translation unit表示的

Function主要是包括在某个函数里面的

Block则是语句块，比如复合语句就是一个块。

* + - 各种类型标识符作用域的范围

标签label名称的作用域，是function的，并且可以在该函数的任何地方使用goto语句进行访问。

其他的标识符的作用域由其定义的地方所决定，在一个declarator或者type specifier的地方进行定义（注：后者主要是struct、union这些）。

如果定义在任何block或者parameter list 外面，那么就是文件作用域的，其作用域从定义位置开始到translation unit结束。如果在任何block或者参数列表内定义，那么就是块作用域，从定义到相关的块结束为止。

如果是函数原型，那么就有function prototype的作用域。只在函数原型内部有用，确切的说吧，如果参数有相同的名称，会出错，但是参数可以和函数原型的名称相同。

另外，如果两个同名的标识符，指向两个不同范围作用域的对象，那么会有覆盖情况overlap。就是说，范围大的标识符，包含了范围小的标识符，但是小的范围中，该标识符指向的就是范围小的那个对象，而范围大的那个对象就被隐藏了，其他范围内，则是指向大的标识符。

对于struct union以及enum的tag，他们的作用域都在type spec声明了tag之后，但是每个enum的常量则在enumerator list中定义出现之后就开始了的，意思就是，enum的常数，如果在enumerator列表中重复定义，则会出错。

* + - Linkage

这部分其实涉及到link阶段，主要是extern关键字指明的对象可以被多个文件所引用，而实际的linkage包括三种，extern，internal，none，虽然感觉这个阶段没啥用，还是努力研究一下下吧。

具体描述如下：

1. 在由一堆translation units构成的program当中，有external的相同identifier的object表明是同一个对象（显然不同identifier是no linkage的），而在一个translation unit中，有internal的是一个对象。如果是no linkage， 那么每个identifier的声明都是一个独立的个体。
2. 如果一个**file scope（一定是file scope才是被认为internal的哦，所以其他作用域里面有个static会被当做no）**的object的identifier或者function包括了static，那么就是internal的linkage。
3. 如果一个identifier定义了extern，而且在scope里面，可见到的，存在一个先前（用的是prior，而不是previous，虽然我英文不好，但我认为应该翻译为，更优先或者上一层的含义）的定义。如果前一个定义明确指出了使用internal或者external，那么这个identifier的定义和前一个相同，（没错如果前一个使用了internal，后一个使用external，也是internal），如果前一个不存在，或者前一个认为是no linkage（对于这个我表示存疑，毕竟说的明白，对于no linkage，不应当存在namespace和scope相同的identifier定义，对于gcc，我在一个block scope中，前一个为no，后一个extern，会报错，但是file scope则可以过，但是后者，file scope则是按照规则4，变成了external吧），那么这个identifier存在external linkage。（对于我个人认为的符号表处理上面，emm，仅仅多加一个检测吧。）
4. 对于一个function，如果不加任何storage class，相当于加了一个extern。如果对于一个object，存在file scope 并且米有storage class specifier，那么也被当做external。
5. No linkage的情形如下：除了object和function之外的其他声明，作为function parameter的一个identifier，在block scope的object的identifier——如果这个identifier没有一个extern的话（gcc如果写了个static在block scope，试图把某个identifier当成internal的话，貌似没啥报错，但是还是当成no linkage）。
6. 如果在一个translation unit里面，同时存在internal和extern的某个identifier定义，行为是未定义的。我测试了gcc里面，我在一个file scope下面，先extern后static，会报错，先static后extern，会过。关于这个问题，我查到了一条C++标准里面的

|  |
| --- |
| //first example  static int a; //a has internal linkage  extern int a; //a still has internal linkage  //second example  extern int b; //b has external linkage  static int b; //error:inconsistant linkage |

（我觉得，emmm反正碰见，我都报错，甭管怎样。这是C，不是C++）

* + - Namespace

这个就让人头痛了，

首先同一命名空间且作用域相同，任何名字具有唯一性

每个struct和union内部具有一个独立的命名空间，内部成员在这个空间内

Struct，union，enum的tag共用一个命名空间

Goto的label在同一个函数内部唯一

其他情况共用一个命名空间。

综上所述，虽然复杂，但是对于指导写出正确的代码我想已经够了，比如在实际中遇到的struct中的内部成员，我一开始还以为是不同的作用域问题，实际上这是一个命名空间的问题。

* + 在代码中的实现

**Scope**

（scope加上namespace，我觉得我头发已经没了）

首先，无论是否进行符号的插入，都应当进行一个作用域的划分，在这里，作用域是通过层次化的符号表进行实现的，因此，首先需要在符号表中表明当前符号表所代表的是哪种类型的作用域，这是非常自然而然的想法。而实际的实现则在符号表中有所描述，在此不做多讲。

上述有四种类型的作用域，

file作用域对应着translation unit节点

function作用域对应着function\_definition节点，~~另外，对于函数定义的参数，我也放到了function的scope里面去，而不是另外开一个，否则由于定义的并列，会导致函数体内和参数列表分属于两个作用域。但是，对于函数名，他必须是在全局可访问到的，那么函数名反而要放到function上一层的作用域当中。~~

而block情况比较复杂，我总共找到了三个地方明确表明了这是一个block的：

* + - A compound statement is a block
    - A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement.
    - An iteration statement is a block whose scope is a strict subset of the scope of its enclosing block. The loop body is also a block whose scope is a strict subset of the scope of the iteration statement.

在这几个地方，对于block作用域的判定，需要仔细。

对于最后一个function prototype作用域，他本身是在declaration中的，他实际产生作用的地方，是在parameter-type-list中所产生的，因此，可以将parameter-type-list中，开一个定义域，但是这里有个问题，在函数定义的地方也是有这个参数列表，但是那个参数列表，我在这里用的是function的scope，所以，办法有一个，查看trace，如果trace里面有function definition，那么就不开，相反的，就开一个。

（若干天后添加，我不知道发生了什么word保存不成功，如果发现了重复的，请不要意外，关于上面划掉的部分，我在语义分析的function definition那里，让参数和compound stmt部分指向了同一个符号表，来消除这个问题，并同样将函数名称插入到上一层的符号表中去。）

**Namespace**

对于namespace。由于其本身并没有严格的层次化情况，但是类似于struct里面嵌套struct的情况，又有层次化的形式，所以我做了个测试：

|  |
| --- |
| struct test{  int a;  struct b{  int c;  int d;  }tmp;  };  struct b tmpt; |

上面这段代码是可以过GCC的编译的，说明一件事，那就是namespace并没有所谓的树状层次结构，struct没有新开作用域，而是开了个子命名空间，但是这里的b属于tag，不属于子空间，而tmp作为一个子成员则属于子空间。因此，namespace的数据结构设计也就可以清晰了。

没啥好多说的，就是一个unsigned int类型即可。同样类似上面所描述的作用域，在每次访问到规则节点时都进行判定。

命名空间的数字id含义如下：0表示unknown，1表示默认空间，2表示tag，3表示error，更高的则是需要进行申请，申请到的是谁就是谁，另外，namespace仅仅对于identifier有效，其他的关键字什么的，压根儿一点关系都没有。

需要涉及到命名空间的情形如下：

一个是符号表的插入，通常这在一个新定义中发生，这时候需要决定某个符号的命名空间，另一个是搜索，这件事在遇到一个新的identifier的时候发生，遇到相同作用域且命名空间相同的符号时，就判断为相等，否则就是一个未定义错误出来。第三个是某些情况需要开一个命名空间，比如function定义，以及struct定义的时候，下面会描述。

普通的id自不用说，在默认空间里面。

Struct，union，enum的tag，需要指向相应的空间，无论搜索还是插入，无需做的很复杂，只要前一个符号是struct union或者enum，那么必然是相应的命名空间

对于struct union的成员，插入符号表和搜索符号表并不相同，插入的时候，就是在struct类内定义的时候，需要新开一个命名空间，对应的节点应该是struct-declaration-list。

而搜索的时候，前一个符号必定是一个点或者指针，但是，如何知道是在哪个命名空间（也就是对应的struct的命名空间去搜索呢）呢？我可以知道前一个符号，但是前一个符号再之前的一个符号可能是一个表达式，可能是其他，记录其类型，需要首先构建类型系统，这样会耦合太多无法处理。

因此，对应搜索这件事，我们不处理，因为如果使用了点或者指针，必然是一个引用，而且前面的类型必然存在。搜索到这种符号不插入符号表也不进行查找，对应token的命名空间为0。

对于label类型，无需多言，这必然和某个function\_definition的作用域相绑定的，因此，在遇到function的时候，直接新开一个命名空间专门用于label，而插入的时候，如果前一个符号是goto，则必定是label的类型，其命名空间，从当前作用域向上查找，直到找到function\_definition，然后定为对应的命名空间。如果是label statement，那么trace 里面，一定会紧邻这一个label\_statement（实际上由于我文法的写法，所以是间隔了1个，但是这不影响理解。）随后插入的符号表也是对应着的function scope的那个符号表。

之所以用scope而不是ast节点的记录用于查找是因为，这时候还没有建立起来上面的那个节点啊。

至于查找，这一阶段就不用考虑查了吧，goto向前向后都可以跳转的，先定义再使用就是放屁。

同样，符号表中哈希表的表项，也需要相应的记录起命名空间。

（我觉得我讲的太复杂了，算了，看代码也一样的，顺带一提，代码中，慎用else if，这个else的情况仔细考虑，我这里namespace就为此出了不少错误。）

**Typedef name**

对于typedef name，我也是头疼的紧，因为这个东西，重构了很多次代码了，文档也重构过很多次了。

先来总结一下标准当中怎么说的，只有一句描述

|  |
| --- |
| If a typedef name specifies a variably modified type then it shall have block scope. |

就是typedef之后，如果对象的类型是有block scope的，那么同样受到相同的约束。

但是我没弄明白的是这个scope是根据原来的对象名的scope呢，还是根据typedef所在的位置呢

所以我做了以下的测试：

|  |
| --- |
| struct tst{      int x;  };  int a(void){      typedef struct tst b;  }  b c; |

如果是根据原来对象名称的scope，那么tst有file的scope，而实际上，这是根据typedef所在的位置来设置的。

由于在语法分析阶段，不需要全部做，那么这里有以下几个问题：

如何指定typedef的identifier(s)是哪几个，因为可能定义多个符号。

如何插入符号表或者其他数据结构来进行记录。

标准中提到的作用域问题如何解决。

第一个问题必然是在某条定义结束之后才进行判定的。

我在此截取下我所生成的语法树的一部分进行说明

|  |
| --- |
| -<external\_decl>  --<declaration>  ---<&declaration\_spec>  ----<&storage\_class\_spec>  -----<KW\_typedef:typedef sym level: 1 namespace:0>  ----<&type\_spec>  -----<KW\_int:int sym level: 1 namespace:0>  ---<&init\_decl\_list>  ----<&init\_decl>  -----<&declarator>  ------<&direct\_declarator>  -------<identifier:ccc sym level: 1 namespace:0>  ---<;:; sym level: 1 namespace:0> |

上面的语法树对应着如下一条语句

|  |
| --- |
| Typedef int ccc; |

虽然正常的typedef写法我想绝大多数人都写得出来，但是，可能的阴间的写法也必须覆盖到，因此我们做一下typedef的分析

对应的语句是一句declaration，其最左节点一定是KW\_typedef这个符号，所以可以据此进行判定，其次，declaration的第一个子节点表示的是specify 而第二个子节点才是对应的declarator所在的部分，上述的语句只有一个declarator，如果有多的，应该可以更多个子节点，而这些节点应该在init\_decl的child向量中。

前面描述符号的作用域的时候提到过，一个identifier的符号应当在一个declarator或者type specifier中被定义，而后者主要是struct union enum。

因此，可以专门对两种声明identifier的情况做一个函数在其中处理符号表的插入问题。当然考虑到实际的情况，declarator作为子节点一定是先完成搜索的，而这时候无法确切的知道是否为一个typedef类型，因此，不推荐在后处理过程中declarator直接使用该函数。

因此，如果在一个declaration中有typedef，则将identifier插入到类型列表中，否则，插入到符号表的哈希表中

第二个问题需要在插入符号表之余，另外设计一个数据结构用于表示，考虑到第三个问题，也就是作用域问题，同样将这个数据结构挂在某层符号表上面，表示在这个作用域当中定义的类型。

对于typedef类型的数据结构设计，使用一个vec记录每一层scope的定义的类型即可。

**Declaration spec**

另外一个非常实际的问题在于限定符。

仍然举个例子：

|  |
| --- |
| typedef int ccc;  int test(int ccc){      return ccc; } |

这个例子来源于ucc的代码测试当中。

在下一层的作用域当中，一个declarator中，可以覆盖上一层的typedef的同名的类型。

来考虑一下C标准和代码。如果不加以任何限制，仅仅按照文法规则（不考虑C标准中文法下面的一堆规则的话），那么没有任何规则能够阻止将参数列表的ccc识别为某种typedef name，而不是一个新的token，原因是，先判定type spec（int和typedef name都是作为一种type spec的），并且，可以重复多次，而参数列表更有一个可选的abstract declarator的情况，从而，造成识别错误。更重要的是，在C标准中，有这么一段话：

|  |
| --- |
| If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name. |

实际跑出来的代码以及这些说明，让我相当的迷惑。按道理说，参数列表应当可以重载和某个typedef name同名的变量。

所以我认真看了一下这几个限定符啊巴拉巴拉下面的说明。

我认为可能的原因是，不同类型的修饰符有不同的数量要求。Declaration声明的前缀符号storage-class-specifier，type-specifier，type-qualifier，function-specifier，alignment-specifier这五种。其中：

storage-class-specifier除了\_Thread\_local可能有重复出现storage-class-specifier中的static或者extern之外，只能出现一次。

而上述出现的问题type-spec里面，尽管各种符号可以出现多次，但是atomic-type-specifier，struct-or-union-specifier，enum-specifier，typedef-name是只允许出现一次的，而且即使是能够出现多次的，同样有其数量限制，比如我写几十个long最后来个int，那就是报错，同样的，int加一个typedef name，那么也会报错，但是非常讨厌的是，他们和其他的spec可能以任何顺序出现。

对于这件事，我认为，只需要在declaration spec节点的后处理当中，加入一个检查，检查这个限定符是否合法即可。

此处，我们并不做过分多的检查，毕竟回溯意味着很大的成本，只做以下处理：

Storage类型，如果出现3次及以上，直接判错，但不检查是否出现两次的时候，具体的孩子的类型。

Type spec，大于等于5次，则判错，因为最多4次，如果出现前面所说的那4种spec之一，并且大于1次，则判错。

Type qual，不做处理，由标准注明的

|  |
| --- |
| If the same qualifier appears more than once in the same specifier-qualifier-list, either directly or via one or more typedefs |

可知，其实哪怕相同的type qual多次出现都是可能发生的。

Function spec，可以出现多次

|  |
| --- |
| A function specifier may appear more than once; the behavior is the same as if it  appeared only once. |

Align

|  |
| --- |
| When multiple alignment specifiers occur in a declaration, the effective alignment requirement is the strictest specified alignment. |

表明，按照最严格的对齐来处理多个align的情况，因此，同样可以出现多个。

另外，顺序还当真不用管，我曾经天真的以为，long static long int或者long int long可能过不了编译，但是，gcc竟然让他过了。所以，无需管顺序问题。

对于发现这个地方出错之后的处理，一开始我以为直接返回出错返回null，但是，发现无法成功，原因是，以上面那个int ccc的例子为例，int和ccc都识别为type spec的情况出错了，但是int是对的，直接返回出错的意思就是int也不对了，从而造成parser失败。

对这种情况，我思考了很多，一开始想着在遍历的时候进行判定，但是遍历时候的文法规则是更改过的，相比于在后处理之前缩减后的树，更加复杂，难以直接判定，更何况，即使是后处理缩减过的树，spec的列表下，也并非是identifier终结符，而是一个非终结符——这意味着需要在通用处理中侵入代码，我表示拒绝。

总之，最后，我想了个办法，也就是说，仍然放在后处理中去，但是，做法改成，哪里出错，哪里直接删掉那个节点以及后续的节点，而前面节点的正确性则由上层规则进行判定正确与否，比如int ccc，int和ccc是两个子节点，并且都是type spec，然后查看ccc发现是个多的特殊节点，ok，出毛病了，在spec的树中删掉ccc和后面的节点的子树，最后，剩下一个int。换成long long int ccc也一样的。

好了，这部分内容肯定还会出现在语义分析当中的。因为按照C标准，仍然需要做非常多的其他检查，之所以放到这里来，是因为这个问题影响到了正确语法树的构建，更缺切点的说，是作为typedef name的identifier被错误识别的问题，因为identifier是C当中，唯一一个不确定到底是啥，请况及其复杂的文法符号。而在语义分析中的那些错误，最起码，并不会被识别错这是啥类型。

* **后处理**

~~正如前文所述，在每个节点建立之后需要对那些定义的子树进行一个后处理，进行符号表的插入和typedef\_name向量的插入。以下列举了需要进行的操作（其中有些操作在上面就已经描述过了）~~

~~在后处理之前，由于我的EBNF文法为了编程方便而引入了大量的新的非终结符，如果不针对特定节点的通用处理，那么无所谓，但是如果针对特定节点，那么这些新的非终结符的处理将非常麻烦，并且，在之后，也用不到这些节点，因此，首先要做的就是树的缩减，如果子树的节点是新引入的符号，那么直接去掉这个节点，将子树的所有孩子插入到当前节点的孩子节点列表中，注意这个过程可以递归。因为子树的孩子也有可能是新节点。~~

~~首先是declarator节点和type\_spec节点，这两个作为被调用的子函数。这两个函数里面，type\_spec里面，只返回struct union或者enum的tag，而declarator情况复杂，但是总体而言，在开头要么是identifier，要么是（declarator），这是确定的，而这里面开头只会有一个identifier，后续的列表都在方括号小括号里面，那么，这在子节点遍历到参数列表或者其他列表的时候，都已经插入过了，因此不用考虑。~~

~~对于命名空间的处理则就是当前identifier的节点的namespace值，这在前面设置namespace的时候就做好了。~~

~~然后是对于当前节点的分类，在几种类型的当前节点的情况下进行一个处理。~~

~~Function definition节点，在函数定义当中有declarator，进行调用，需要将函数名插入上一层的符号表，而参数列表插入该function中的func scope的符号表中。~~

~~Declaration节点，一个是对于storage里面typedef类型的处理，前面提到过了，对于type\_spec里面的处理，包括了 struct union和enum，对于typedef name，前面有提到过如何插入typedef列表里面，在上层就进行了插入操作，而在下层的type\_spec节点中不应当采取任何动作，其中tag要插入到当前作用域的符号表中去。而其下的成员的identifier也应当在当前作用域的符号表中，但是，namespace不同。还有就是函数声明，函数声明中的函数名也应当插入上一层的符号表，而其参数则使用function prototype作用域，并且命名空间为默认空间。~~

~~Goto和label的statement里面的新identifier，向上查找到对应的function scope，并插入该符号表。~~

~~插入符号表的部分，应该就这样了~~

（以下内容重构）

由于在遍历形成语法树时，我无法确定整颗语法树的样子，因此对identifier做一些判断是非常困难的，什么重定义错误之类的，我也是碰到问题之后无法解决，才来这边重构设计。

这里，后处理当然要做，但是主要是针对于declaration中的typedef一种类型，不再试图插入符号表，因为压根儿没做比较。插入符号表的事情放到了语义分析中进行，在那里，构建了完整的类型和符号表符号，并进行了redefine之类的检查。

Typedef的检查，一个是创建typedef的时候，在declaration节点进行查找访问，另一个是在引用的时候，由于overlap现象，所以为了避免下一层的同名符号，被识别为typedef name，所做的检查。

后处理的另一部分，则是语法树的裁剪，一些无关紧要的节点，或者expr里面非常深但是却毫无任何用处的节点，都将被裁减掉

* **在遇到一个identifier时的判断算法**

~~正如前文描述，一个identifier可以具有如下几种类型：~~

~~对象名，函数名，tag或者是struct的成员，union的名称，枚举类型，typedef名称，标签label名称，宏名或者宏参数（预处理阶段）~~

~~当碰见一个identifier时，先判断所在的namespace，否则没法搜符号表的，首先先查看前一个符号，如果是点或者指针类型，那么显然是对于某个struct成员的引用，前文描述了需要做的处理，对应的namespace为0。~~

~~如果前一个符号是struct或者enum或者union，那么就是namespace为2，这时候显然需要插入符号表（当然这在后处理当中的），另外也需要判断是否重定义错误。~~

~~对于enum constant，其namespace在默认命名空间1，并且由于其作用域描述，所以如果上层节点是enum const，需要搜索是否重定义。~~

~~对于label类型，上面有所描述，查看前一个符号是不是goto或者查找trace stack，但是，需要注意的是，无需进行符号表的查找和比较，因为，如果label stmt定义在后面，可能压根找不到，也无需插入操作，因为这个操作在后处理当中。~~

~~其余情况，判定的namespace均为1。~~

~~（其实简而言之，和前面描述的判定namespace的内容是一样的，但是有一点不同，就是增加了需要判断tag是否重定义，在实际代码实现中，当然要杜绝重复代码咯，直接写进去就好了）。~~

~~然后，对于namespace等于0或者3的情况，无需进行后续判断，0是成功的，这是一个成员，但是不知道是谁的成员，3是失败的，其余的情况才需要后续的符号表查找算法。~~

之后，则是对于typedef name类型的判定。这个判定是在父节点为对应的typedef name的时候才做的判定。

~~然后是goto和label stmt的判定，无论何种情况都当成一个新的符号来对待。只需要判定目前节点的namespace，是否和当前的function scope的namespace相同即可。~~

~~随后，是对于declarator以及type spec的判定（确切的说应该是struct union enum），如果这是在一个定义中，那么显然当前符号表中一定不存在相同namespace这个符号才是符合实际的情况，需要做一个查找，如果符号表中存在相同作用域和namespace的符号，那么报重定义错——需要注意的是，不要往上一层查找了，因为有overlap现象。~~

~~否则则是对于已有符号的引用，如果已有符号在符号表中查找不到，那么报未定义错误，这个查找则是从当前符号表向上一层的父节点不断查找的。~~

（由此可见，C的这个渣渣语法对于代码的侵入性到底有多大。哎，我尽量的将在语法树中遍历的通用代码在一起，而针对于某些特定节点的代码放在另一边，让代码能够解耦合，从而能够通用的去进行语法分析，又能够在发现某些特定节点的规则时，尽快的将规则加入进去。）

**错误处理**

错误处理真的让我脑袋一紧，这玩意儿，来让我描述一下吧，错误处理呢，理论上碰见错误，就应该报，但是，但是麻烦的是，这是一个树遍历，只有树遍历完了才知道正确不正确，而且，还是盲人摸瞎马的去LL0式的匹配，换句话说，如果有错的，说不定是压根没匹配到正确的呢？确定性的分析方法是没有这种烦恼的，换句话说，我这错误处理，能保证出错了肯定知道存在问题，但是绝对没法知道错的地方在哪里。

对于错误处理的算法，我个人的想法是，只记录最长匹配的地方出的错，剩下的就算了，另外，只在link类型中，第一个匹配上了，后面的没匹配上，才报错。

那么另一个问题是，对于中间引入的新非终结符怎么办？

显然这里面的报错应该是后续符号不匹配，也就是，“期待某几个符号”，这个报错。这不就是让我去整first集合么？就是不想做LL才搞成这样，哎，但是吧，我好像做了记忆化搜索的记忆，也就是说，去某个非终结符中的记忆，查看存在哪些终结符的方向是存在的即可。但是这样一来，万一吧，在非常接近root的节点出了错，那么这个记忆的可选集合非常长。怎么办？所以，如果超过若干个可选的符号，就直接报unexpected token吧。