**概述**

语义分析其实就是按照C标准的语义规定，将每个节点的东西都进行一番检查

我对于这个阶段的打算是，在语法分析之后，重新遍历一遍。关于这个事情本身到底应该放在那里，我其实也有过犹豫，按道理来说，这些东西，其实完完全全可以在语法分析的时候，直接嵌入进去的，没必要在下一个阶段再遍历一遍。

然而我在语法分析阶段的做法，决定了，我很难这么做，首先是EBNF写法的关系导致的大量非终结符的引入，如果在语法分析加入了这些内容，很多都是毫无意义的非终结符节点，这部分的操作完全属于性能损失。然后是由于大量回溯的问题，如果加入进去，当前节点的树出了错，结果兄弟节点有可能一棵很大的树的遍历，全错了，那么在这个阶段就进行语义分析和翻译，这对于性能而言是非常大的损失——这是确定性的预测分析法不会有的问题。

因此，出于这些考虑，我决定在语法分析之后再进行语义分析和翻译。说到底是一颗相比于语法分析遍历的节点少的多的树遍历，性能需求确实不大。

先把我整个语义分析工作的打算描述一遍，首先是，对于语法树的每个节点，按照分类，参照C11的描述去增加节点属性以及符号表属性（当然这不是在代码流程当中而是在头文件中的），随后，进行一个树遍历，遍历每个节点并查询其属性是否符合语义，并生成当前节点的属性值，另外，需要在每个节点进行一个代码的转换，根据子节点的情况，转换为中间表示——这也是为什么我把这两个部分放在一起的原因。

重定义的检查问题。努力良久，最后，我还是在语法分析过程中，删去了重定义和未定义符号的检查，这是源于树遍历所导致的问题，但是，当语法树确定下来之后，重定义和未定义符号的检查则是必须做的事情了，事实上这还需要涉及到linkage以及一堆语义检查的要求。

在语法分析阶段，表达式，声明和外部表示，我并未区分开来，但在这里，我必须加以区分，对于三者的处理是不一样的。

以类型系统为例，语句类型对于类型系统而言并不那么重要，其对应的节点无需使用类型系统，而声明的处理主要是对于类型系统的构建，表达式则是对于类型系统的使用，声明的语法树中，并不是每个节点都需要类型系统，只要标识符还有初始化等节点有类型就够了，这些部分存在于对应的符号表中，而表达式的子树每个节点都需要类型系统，不然无法进行相应的处理。

相关的内容由于仅仅涉及到对C11标准的翻译，而不涉及算法和代码细节，我放在另一个文档中（以便于我查看，并不是不能直接阅读英文，只是本人毕竟不是英语母语环境下的，抱歉），另外，有些部分涉及一些具体实现上面的讨论，在这篇文档中直接写了，所以并不全面的进行翻译，对此我表示抱歉，说到底，这是一篇开发文档，写给自己看的呀。

* **整体框架**

（先声明我是在完成了声明的类型构建之后开始头疼这个问题的）

原本而言并不应该陈述这个问题，因为理论上这非常清晰，一棵树的递归调用而已，前面也有所提及。然而在实际中，一方面，类似于declarator或者expression这样的节点会被多个不同的上层节点所调用，而且，表达式类型和语句类型的节点有其各自非常鲜明且相似的特征，这在语法分析中就可见一斑，这为模块的划分提供了一定可行性。

另一方面，比如声明这样的节点的分析，由于其复杂性，每个节点的代码长度都相当长（PS：一个月后回来考古，简直相当长，而且差点自己都不明白当初自己想弄些啥），如果，所有的节点的代码都写进一个文件中，就目前（我设计这些的时候）而言，可能会一个文件超过2k行代码，这不是一个优雅的选择。

第三，类型系统是semantic的一部分，但不是全部，我不可能将所有的语义检查的部分设计进类型系统中去，尽管有所涉及，但那不过是为了减少重复的ast遍历（既然遍历到了就顺便检查一下吧），语义检查和类型系统构建的划分仍然是必要的。加之，在语义分析的过程中，还涉及到语法翻译，这让整个框架设计变得重要——我不可能将这些全都写在一个文件中，尽管从编译原理课程上写的而言，这么做没什么毛病。这是我实际遇到的现状，也是为什么我需要开始对整个语义分析和语法翻译的框架进行描述的原因。

一个问题，模块间的互相调用。声明的时候需要初始化或者field位宽，这些很多时候是一个常量表达式，这是一个调用。表达式中，也有很多type name和initialize的地方，而这些被当做声明的一部分。

对于常量表达式，从原先的单独一个模块更改为和表达式模块合并的一个设计。

在这种情况下，需要返回一个常量表达式的值，这个值应当存储在节点中而不是作为函数返回的参数被调用，因为在语义分析之后的代码生成部分仍然需要用到这些，以及在上层能够检测，下层是否是一个常量表达式的bool类型（而不是像之前设计的那样，是否在上层就决定是否是一个常量表达式并检测）。

从上到下改为从下到上的原因是，编译优化的考虑，即使一个表达式不是常量表达式，但是该表达式的局部很有可能是一个常量表达式，如果在计算过程中重复进行相关的计算是不划算的。比如point+2\*3这样的指针计算很常见，显然，如果在2\*3表达式的计算过程中直接确定2和3都是常量表达式，满足常量表达式的计算规则时，直接用6来替代，所以尽管整个表达式不是，但子表达式是可能符合的，所以检测要从下到上。

另外，在每个节点的检测中，都必须包含对是否是常量表达式的判断，常量表达式限定了其所能包含的操作符的类型，只要不是该类型的表达式，都应当在表达式节点中，表明这不再是一个常量表达式。

这里面需要注意的是**和代码生成的差别**，只有局部或者全部都是常量表达式的时候，才直接计算。剩余情况则是对操作数的类型进行判断（或者转化）。因此对于表达式的节点，需要某些表达式或者全部表达式为常量表达式的时候，才能直接给出当前节点的值（或操作，就是只要有表达式为常量且为真就是真了呀）。这是将常量表达式模块嵌入进表达式模块的必然要求，很多时候，就把两者弄混淆了，并试图在其中做好代码生成该做的事情。

也因为这个原因，所以需要直接合并到表达式模块中去。（我去，得改设计！我跟你说，这玩意儿让我心累）

对于模块间互相调用的问题，我认为，最终，多个模块共用同一个头文件（当然实际操作上，会有模块同名头文件，去包含这个共同的头文件来避免一些问题），这样，所需要的函数都已经有了声明。

* **实现细节**
  + Linkage

（事实上，这是我在一个多月后的考古，我是在检查的时候，才发现，linkage会导致不应该重定义错的地方导致重定义报错）

对于linkage的使用，语法分析部分我想应该说的很清楚了。我就懒得讲了。主要描述一下数据结构和算法的设计。数据结构好说，linkage总共三个，而且绑定了identifier，塞里面就完了。

算法主要是在查找到某个identifier的时候，需要判断是否存在重定义的问题，这时候，需要结合当前identifier和已经定义过的identifier的linkage去判定是否存在重定义问题。

（总之我觉得自己有些迷惑，emmm好吧，我知道了这个迷惑的来源主要是不明白不同情况下的处理。所以还是描述一下吧）

一个是遇到某个identifier的时候对他的linkage的判定。

另一个是当判定了某个identifier之后对于他是否存在重定义的问题的检查，这时候是linkage的使用了。

上述两者，通常是放在一起的，并且对于某个identifier的linkage的判定也需要依赖于上一层的identifier的查找结果，如果是extern的话。但确实，这是两回事情。

第一个判定那里。在object和function

中的定义处（其实就是declarator里面的identifier）。

1、如果是file scope，加了static是internal，不加则都被当做external。同时存在internal和extern的某个identifier定义，行为是未定义的，这里我当做出错处理。

2、如果是在block scope，有extern，则向上层查找，如果没找到或者找到了但是是no，那么这个identifier应当被当做external，如果找到了，是internal或者external，那么和他相同。如果没有extern，function类型则相当于加了一个extern，其他的都是no linkage。

3、如果不是declarator那里的其他声明，或者是function参数的identifier（也就是prototype作用域），则统一被当做no linkage，至于function作用域，他除了管理label还能干嘛？label是按照no linkage处理的。（哪怕是存在extern的声明，比如函数参数，从语法而言，是可以存在extern的，当然后面会报错）

如上，是当前identifier的linkage的判定。

在实现中，则需要结合第二个部分的检查，由于当前的identifier和上一层的identifier，实际上指向的是同一个对象，而一个对象的linkage在声明之后就不会改变了。所以，尽管在第二种情形那里，存在其对应的语义，但是其实现应当加上查找而不是简单置位为external：

如果有extern，先在当前层查找，如果找到了并且是extern的，不报错，但是也不插入（在同一个block scope里面，有同样的extern的变量应该指的都是外面的东西），否则报错，随后向上层查找，没找到，那就是新插入一个identifier，并且是external，如果找到了，进行检查两个identifier的类型是否相同（检查的时候不用检查storage class spec），相同则插入到当前符号表，不同则报错，如果没有extern，function类型应当向上查找，如果存在相同参数类型和返回类型的则同样插入符号表，不同则报错，找不到则插入，并且为extern。其他情况，当成no linkage来处理，那就是在当前符号表查找是否存在重定义问题。

对于第一种情况，file scope，这是在同一个符号表中，按照internal和external的语义定义来考虑，那么应当是唯一的。所以，两个相同（函数当然参数类型相同即可）的声明都是static或者extern的时候，是可以的，一个static一个extern的时候，都报错吧，反正标准说，实现定义的。

至于no linkage，不可能的啦，file scope就两个。

第三种情况，就是除了declarator之外的那些，既然都是当成no linkage，那就按照no的规则，除了typedef和tag不可多次重复声明。

* + 类型系统

总体而言，类型系统是在声明中被定义好每个identifier以及常量的类型（这通常是终结符），同时，对于非终结符他们的类型是由什么综合属性啦，继承属性啦来决定的。我也是参考了ucc的做法，让其分为基本类型和衍生类型。基本类型直接就是type spec里面描述的类型。

我们必须要分清楚的一件事情是，如果不加signed或者unsigned，就比如说char吧，那么到底是有符号还是没符号的char呢？我查了一下，这还真的取决于编译器。。。内心一句mmp。Mcc怎么处理呢，既然使用了gcc作为最初始的编译器，自然是和gcc一样呗，不巧的是，x86和arm的gcc还据说不一样。那，既然开发平台是x86，那x86吧，char等同于signed char。顺带在特性里面提一下。

随后是衍生类型。衍生类型由于存在各自特别的记录，因此在数据结构上需要加上去。Struct或者union类型，需要记录一个其下所有定义的identifier的列表，enum类型，同样。指针类型，需要记录其代表的指针类型。Function类型，需要记录参数类型列表，array类型，由于语法树的设计问题，需要记录每个维度的长度（被自己坑了呗）。自定义的类型，需要指出他指向的类型，基本类型或者是其他类型——这也意味着在上一步做的typedefname的地方需要做一个。

**数据结构和算法的设计**

上面描述了大概的内容，接下来描述一下数据结构的设计以及在之前所确定的ast和符号表框架中的位置。

在这里，我有必要强调一点，那就是类型和实例的区别。举个例子吧，声明一个struct的类型和声明该struct的一个实例，两者是不同的，一个struct的类型数据结构，他会被所有的该struct的实例所共用，而且自声明到结束生命周期都是一样的，而某个实例，是一个模板，其内容随着计算所发生着变化。（最开始我非常迷惑于类型结构的数据结构的设计，就在于此，试图将两者一起完成，导致了数据结构的设计变得困难）。

因此类型系统的数据结构如下，在开头有一个type common的部分，作为基类。后面针对性的扩充一些属性。

感谢ucc的这个设计，至少这部分我承认有所借鉴，当然由于ucc所针对的c89，其type设计的时候只有两个qual属性，而C11相比起来有很多的新的特性，因此type类型虽然有所借鉴，但是说抄肯定是不至于的。举个例子，在ucc中有一个对一些基础类型进行初始化的设计，然而我实际考虑了，如果按照qual仍然作为其中一个部分的情况，C11中的type的排列组合，发现这是一件恐怖的事情（即便可以用循环去解决），从性能的角度而言并不算优秀，因此在实践中，我重新设计了这些结构（下面举得例子也可以证明，你去看了就知道，显然这没有抄对吧，设计都不一样）。

（我不得不陈述这个数据结构设计的困难之处，以供后来者借鉴，或者设计更优的结构，对于类型链以及相关的东西我就不说了，常识慢慢去查吧）

首先，对于int或者struct等等也就是type spec的类型之一，作为类型的一个字段是毫无疑问的。

存在疑义的是其余字段，storage class、qualifier、function spec、alignment，后两者暂且不论吧，毕竟判定明确在semantic中会给出，只针对特定类型。

如果这些字段和type spec放在一起，一起组合成为type结构，考虑以下一段代码：

|  |
| --- |
| const struct a{      int c;  }b;  struct a z;  int xxx(void){      b.c=1;      z.c=2;      return 1;  } |

显然第一个结构成员c的赋值会报编译错，但第二个赋值会过编译。但这至少说明一件事，const这个type qual和struct这个type spec并不是绑定在一起的，否则如果像上面一样再声明一个同样的struct a z，那也一定含有const的情况，但是实际上z没有const，说明qual这些字段不能跟type spec放在一起，由上面一个可以看出，有可能是const->struct，这样一串type链，或者是由上面代码的b和z这两个identifier持有const。

那么这些字段能否作为声明的declarator的identifier的一个字段，也就是后一种情况呢，不行，这里我以C11标准所使用的一个例子来说明

|  |
| --- |
| const int \*ptr\_to\_constant;  int \*const constant\_ptr; |

一个是一个指向某个constant对象的ptr，另一个是constant的ptr，注意后者，这是对于ptr的修饰。而如果按照放在identifier中来设计，是无法完整的表达这种情况的。

（虽然实际上，这是在point里面定义的type\_qual\_list）

因此，这些结构，既不能和type spec放在一个结构体内，也不能作为identifier的某个属性，但是又必须在这个type链中进行修饰，和type spec共同表达某个类型。

（呼，为了上面这点，我已经试了一大堆错，最后还是得回来重新设计，呸！）

然后对此，我有两种设计方向，1、除了type spec之外的其他部分合起来组成某个结构，用于修饰type spec，2、其余所有结构每一种类都单独设计一个结构。

这里面的关键在于，其他类型是否像上面所述一样存在冲突。

我无法确定这一点，但是，简单问个问题，拿上面那个来举例，const可以用来修饰point或者int什么的类型，可是加一个static，是const修饰static还是static修饰const呢，两者一定是并列关系的对吧，所以1咯，不然怎么确定修饰的顺序呢。

当然，是否用指针来表达一串type链，我觉得用一个vector更为合适，前面的修饰后面的，或者相反的顺序，后面的修饰前面的，我个人倾向于后一种，那就这么做了吧，因为可以继续扩展（这跟一个栈没有差别吧，下面就当成栈了，栈底是vector的头部，而栈顶向后生长）。在插入顺序中，declaration spec的type spec指定的类型作为基础类型先插入栈中，然后其他的storage class或者qual之类的，作为修饰类型后插入栈中。（所谓的后面的修饰前面的嘛）

理论上，vector里面的每一个type结构都应当是独立的，以取代之前的指针构成的链表。

但是这里面有个例外，那就是typedef，这个类型，完全可以存在typedef const int xxx之类的，所以我在typedef这个结构其中内建了一个vec。而当使用typedef name来定义某个identifier的时候（也许会继续嵌套，比如这个例子，就static xxx\* x，完全可能对吧），我认为需要做的事情，不是继续用什么嵌套，这太恐怖了，而是，将该typedef name所指向的vec里面的type结构体全都复制过去。

仍然拿C11标准给出的例子来看

|  |
| --- |
| typedef int \*int\_ptr;  const int\_ptrconstant\_ptr; |

标准中说后面那个是const-qualified pointer to int。

一开始我也不李姐啊，这const并不是作为point之后的type qual来使用的，给出这个const结构修饰int结构这种分离设计之后，总算能够符合了，下面我会说明啊。

按照上面的设计，int\_ptr的类型是typedef name，然后他的vec的布局如下，typedef int是作为declaration spec的部分，一定是先插入的，而栈先插入基础类型，然后插入修饰类型，这里的typedef是个特殊类型，虽然是修饰类型，但是在标准中也说，把typedef归类到这里就是为了方便语法，不能当回事，所以是最后处理的，所以栈底是int类型，然后处理declarator部分\*int\_ptr，压入point类型。

当typedef的vec复制过去之后，int\_ptr作为后一个声明的基础类型，随后呢再压入const这个修饰类型，从栈顶到栈底依次是，const，point，int。就变成了const的point指向int类型了。

和直接定义成const int \*constant\_ptr作对比（这是把int \*直接复制过来哦）。Const int部分作为declaration spec，压栈顺序是先压int再压修饰类型const，然后再处理declarator里面的一个指针，这时候压栈的顺序是从栈顶到栈底point，const，int，这才是point指向一个const的int。

这也是我想到的唯一能够能够解释这种情况的结构了。

对于array类型的设计，原本由于文法设计的问题，是一个扁平的结构，就是比如a[2][2]，文法分析的时候，两个维度是在同一层的，但是绝大多数的array都是一层套一层。一开始我在考虑是否用一个vec来表示各个维度，使得用一个结构体就能完成多个维度的表示。但是当我看到typedef一个array，外面再套一个array的时候，我就知道，这只能一层套一层来表示多个维度了。

所以那个struct里面套一个union，里面再套两个struct的破破烂烂的结构体，就是这么设计出来的，希望能加以改进吧。（相关代码又得全部扔掉重来了）

对于类型的存储，我使用了hash表（这个hash表一开始我也是只针对于symbol 写死在里面去了，在这里改成了一个通用的结构，因为改写，所以非常可能存在bug），在具体使用这个hash应用到某个结构的时候，在结构开头需要包含hash common，以及需要为该结构设计一个比较两个结构是否相同的函数，以及一个根据结构体生成key值的函数。

在每个符号表中都对应一个类型表，用于表示每个类型都存在对应的scope。

每个符号都理当对应于某个类型，而对于struct union或者enum，应当为tag或者member的名字，确定对应的指针。

如何确定某个符号是一般的declarator还是tag或者成员，其实只要看命名空间即可。

（2022.6月补充设计）再考虑storage class和alignment function的情况，他们只会在declaration中出现。而这些属性理论上是和对象绑定在一起的，因此，在declaration的最后，应当将这几个属性放置到对象中去，并且清空（懒清空）原有的type\_vec的这几个属性，从而整条类型链，只剩下type specify和type qualifier两种属性需要考虑。

这里并不选择更改数据结构和算法——这相当困难不说，更重要的是，declaration下面的函数的调用，属性的查找，相当复杂，而在declaration结尾统一处理，会更简单一些。

这么做的原因很简单，我发现，这几项属性在声明中很多地方需要，但是由于其指定的其实是identifier的情况，而并非像qualifier的修饰是针对某个局部的类型的，所以在表达式中进行相关处理的时候，这些属性的处理非常的困难。

而直接清空的情况并不可取，比如linkage类型，很有可能出现后面的重复定义（并不是所有的重复定义都算报错的呀），直接清空会导致结果出错

所以，当expr中进行运算的时候，忽视modify中的storage class alignment function这三种类型。

* + 类型的确定。

在类型的确定部分，其实很多地方，尤其是声明类型的部分和semantic的部分是重合的，同样举那个struct的例子，约束和语义很多地方都是决定struct类型结构具体数值的，因此，在另一篇关于语义和约束的文档中，我会详细的标注好已经完成和支持的部分——否则谁弄得明白到底做了啥。

但是类型确定和语义检查并不是完全等同的，很多东西必须在类型确定之后才能去进行检查。

声明类型的确定。声明类型下面是一棵树，而其下的declarator的部分和类型部分是分开的，只有到了declaration节点才能确定declarator节点的类型是什么，而在declarator节点所能确定的只有array，function和point类型——当然还有最普通的

* + 序列点和副作用

其实我并不很确定该如何去表达这些翻译过来的事情，但是总之，一个序列点表明在这之前的副作用是确定的已经完成了，而在这之后的副作用确定还没开始。

我查找资料的时候相关内容中举了几个经典的例子，比如i++，得到的值是i原来的值，同时有个副作用就是i现在的值加了1 。这就是所谓的副作用。对于一个表达式的值，在生成一个值得时候，有些表达式会改变某个对象的原有的值。

由上面一个表达式i++的例子衍生出来的对于序列点的说明，的一个例子就是

|  |
| --- |
| i=0；  printf(“%d %d”,i,i++); |

类似的例子总是被使用，到底是执行后面的在先还是执行前面的在先是不确定的，是根据编译器来实现的，而这就是在上一个序列点到下一个序列点之间的，所以是不确定的。

这和语义检查关系不大，主要是代码生成的问题。

* + Define和declara

这个问题让我头疼，主要是redeclaration和undefined的检测。以及在不同scope可以重定义不影响的巴拉巴拉问题。

先说tag，tag只有struct union和enum三种。

这里面有三种，一种是defination，这个很好说，就是后面包括了括号和列表。

第二种是比如struct s ss；这种用某个tag表示的类型，表示相应的对象，这个对于struct union可以出现在前面后面都行。

第三种是只有struct s；这是在声明某个类型，他有可能出现在其具体的定义之前，也可能之后，也有可能不出现。这里需要注意标准中，对于enum类型，如果他的存储类型没确定是不能出现这种东西的，但是我认为，我在这里已经指定了存储类型是const signed int，那也没什么烦恼了。反正标准中说了这一点是由编译器指定的对吧。

那么，如果出现了某个tag，需要进行查找，如果找不到（条目不存在或者count==0），那自然是新建一个，如果找到了，对应于不同的情况，有不同的处理。

对于defination。查找不到自然新建一个，查找到了，此时的declaration必定为true，如果是之前只有比如struct s这种声明的情况，显然那个时候会生成一个结构体，但是其内容不会去填充进去，也就是declaration为true，但是definition是false，那么需要去填充。如果之前已经有了definition，那直接报错重定义。

对于后面的情况，如果之前没有，正如前面所说的，需要新建一个对象，但是不填充内容，并且declaration也需要设置为true。如果之前有definition，说明有了定义，那么直接引用，没什么毛病。如果之前同样是只有声明，那么也同样引用对象。但这有个例外，在结构体中不允许包含incomplete的类型或者function类型，所以如果声明自己或者声明一个没确定存储大小的结构体或者array什么的，啊，那得报错。

* + 匿名结构体问题

匿名结构体的使用需要满足以下几个条件，首先，只有在一个struct内部使用才算合法，否则在最顶层的话会报没有声明任何东西的warning（也许是error？）其次，tag肯定没有，这没的说，第三，后面的struct-declarator-list opt也必须不存在，也就是未命名这一个条件。这时候，外面的结构体可以直接访问该结构体内部的成员（可以递归）。理论上，这部分可以递归，只有struct的情况下，这毫无问题，有问题的是，加上了union怎么办呢，union和struct混用的情况怎么办呢？就拿我用的那个m\_type的结构体，如果按照这么去缩减树的规模，那么就无法同时表示某些部分存储顺序的并列和串行。不明白意思的就去看m\_type.h里面我前面所设计的那个存储type的沙雕数据结构，他混用了struct和union，如果直接缩减树，上层当然能够直接访问到孩子了，但是union的功能就没了。

这件事情我认为可行的方案，就是在语法树中，该节点仍然是上层节点的子节点而非孙子节点，但是通过在每个field里面存储的偏移值来确定，我相信实际中别的编译器也是这么做的。但是对于解析这个偏移的算法而言，就不是那么美妙的一件事情了，因为当多层的时候，就是递归了。

一个很自然的想法就是，匿名结构体专门写一个递归型的解析。但是这后来被我否决了。

我想到的办法如下，首先，最顶层的结构体本身，仍然按照原本的设计去遍历所有的member，在每个member的类型部分，递归的调用type spec的分析函数，碰到匿名结构体也一样正常分析，最后会生成一个member的vec（详见设计的数据结构），对吧，每个member里面都会有一个对应的type vec，而匿名结构体的member的特征是，没有对应的member name，而且typevec里面第一个是struct/union（这个vec有可能有其他的修饰类型在后面，我不确定），而这个struct/union的tag也是没有的。

关于这么做，我有个gcc编译器里面过了编译的特别的样例来分享一下，从而侧面来说明，可能的处理逻辑顺序（我也觉得这个样例非常的神奇，gcc怎么能让他过编译呢，估计是不是我看漏了？猜不出来原因）：

|  |
| --- |
| struct a{  struct {      int c;      int d[];  };  int e;  }; |

首先这个结构体里面嵌套了一个匿名结构体，按照标准的说明，他应当等同于去掉那个乱七八糟的结构体，里面的成员放到外面，但是，这样一来，那个柔性数组成员，就不再是最后一个成员了，直接报错好吧。

可是gcc让他过编译了，而且连warning都没有。也许是我版本问题？？？

我在struct里面另外找到一处可能可以解释gcc这么做的原因，但是我不认为他应当是这个意思：

|  |
| --- |
| A structure or union shall not contain a member with incomplete or function type (hence,  a structure shall not contain an instance of itself, but may contain a pointer to an instance  of itself), except that the last member of a structure with more than one named member  may have incomplete array type; such a structure (and any union containing, possibly  recursively, a member that is such a structure) shall not be a member of a structure or an  element of an array. |

最后说，这样的一个structure不应当以包括递归的方式成为一个struct的成员，换句话说，如果这半句话前面的不是分号的话，我认为他指代的应当是有flexible array type的一个struct 不能成为上层的成员，也就是能解释这种情况。但是，既然是分号，那我觉得这个such struct应当指代incomplete或者function类型的成员的结构体，自己品，不多解释。（非常迷惑）

这从侧面说明一个事情，首先处理的是匿名结构体和后面的int e，然后再让匿名结构体的成员提升一个层次，是这样一个顺序（假如你确实能够弄懂的话，我相信会对你写struct的分析函数有一定帮助，尽管我认为这样是一个错误）。

至于sizeof(struct a)的结果，是8，也就是两个int的大小。

当然，我认为这样的样例是不符合C标准中对于柔性数组成员这件事情的本意的，因为柔性数组成员本身就是可能发生的一个可扩充的存储空间的情况，这也是为什么将这样可变大小的成员放在最后的原因。反正我的mcc编译器是绝对不给这个样例过的。

前面说的是检测匿名结构体的方式——而不是通过直接查看ast了。随后，在第一次生成之后，再次遍历，依靠上面的方法，当检测到匿名结构体的时候，将匿名结构体的所有member取出来插入到上层的member vec中去。由于本身就是递归的去做type的读取，所以无需担心多重嵌套情况下的struct。

这里这样做存在一个问题就是name space的问题，对于匿名结构体，显然按照上面的设计，是已经生成了的新的类型，而按照原有parser的设计，一定会分配一个新的namespace给他的，现在，却告诉我，这些节点的namespace和上层结构体的namespace是相同的，这意味着可能需要这样一个转换。问，有没有必要改namespace？我认为可以不改，因为这会引起hash表中插入删除的很大麻烦。而这样做的目的本身就只有一个，用于同一命名空间的重定义的错误检测以及在不同命名空间中重定义同一个符号是可以允许的，因此，对于子匿名结构体的向上插入过程，还需要多一个步骤，查看父结构体中，使用父结构体的namespace的同名对象是否存在，但不对其具体的namespace进行更改，因为毫无必要。

（我考虑过是否将在parser阶段生成namespace号的过程移动到type类型系统的建立过程，因为parser本身用不到这些，而将这一过程移动到这里，可能避免namespace的重新检测，我上面提到这么做，更像是一个补丁，这不优雅，然而后来我发现这并不容易，遂废弃该考虑。）

在这第二次遍历的过程中，很自然的，对incomplete的类型进行检测，如果发现incomplete或者function的类型，那么就去报错。

然而这里又存在一个问题，对于子结构体和父结构体的tag相同的情况怎么办。这里又分为两种情况，一种是在子结构体里面重新定义了，另一种是，在子结构体处只是一个引用。前者，让插入tag并检查重复的代码放在结构体内部的member以及相应的第二遍遍历的检查都完成之后再进行检查，这会产生一个重定义错误。而对于后者，在子结构体处引用了，由于这时候还没有执行上述插入tag的代码，会生成一个空的struct，并且tag为这一相同的tag，可以让complete设置为true的步骤放在最后，从而在子结构体引用时，complete仍然为false。

对于成员的declarator的type vec，emmm这个问题我还是举个例子来说明吧，比如说结构体里面有一个成员声明如下：

|  |
| --- |
| Int \*a,b; |

这是很常见的事情对吧，但是由于我在declarator中的设计，他会直接改变插入的vec的内容，所以如果使用一个vec来表示多个declarator的类型，那么后面的声明符的类型会被前面的声明符的类型所“污染”，所以这里遍历declarator list的时候，每次都需要新开一个vec并将原来的vec复制过去。

同样的，对于计数declaration里面declarator的值，也需要还原

对于偏移值的确定部分，对于子匿名结构体或者匿名联合里面，相对于结构体或联合的头部的偏移是可以确定的，然后根据父结构体或者联合的类型以及之前的偏移值来确定子匿名结构体插入到上层的偏移是多少。

对于柔性数组成员，我想这部分的检测是必须去做的。包括两部分，一个是在非最后一个成员定义的柔性数组成员的报错，以及一个在最后一个柔性数组成员的类型建立。

总而言之，步骤如下：

检查是否存在重复的tag并且已经定义过了。

第一遍遍历，生成结构体内部的members及其类型，而不做任何检查

第二遍遍历，检查上述提到的东西，包括匿名结构体啦，incomplete和function的成员啦等等，在这一步遍历的时候会确定偏移值。

然后，对结构体的tag所表示的结构，确定其各个参数值并插入各个vec。

最后，完成complete。

我想对于这个算法的描述应该已经是非常详细了。

（最后的最后，我必须深切的鄙视一下VSCode的C插件，虽然很多时候他会给出一定中肯的建议，但是，很多明明符合C11标准，以及gcc也能过编译的东西，他都会毫不留情的报错，尽管正常人没人会去这么用。可是我写的是一个编译器啊，你能不能不要再在这些测试样例里面乱报错了，大哥我求你了。）

* + 联合和结构体的内存对齐和偏移问题

这里并不需要实际填入数据，但是需要计算每个成员的偏移情况。

考虑这个问题的时候，首先先不考虑位域字段问题。

（容我先为了对齐的事情，去考古修改一下内存池那个分配器，相关内容可以去看内存分配那篇文档的记录）

OK现在，保证内存池或者直接使用m\_alloc的情况，都是保证了首地址一定能够做到按照给定的系统默认对齐系数来对齐。

对于这个问题，先来看看C标准的约束（这些部分也放在另一篇描述约束的文档中）

1. Struct是存储按顺序分配的，而union是存储重叠分配的
2. 应当保证分配足够大的可寻址单元来存储位字段，如果当前单元空间足够，下一个成员的位字段内容应当紧跟着，**如果空间不够了，到底是放到下一个存储单元还是跨单元存储，这是由实现定义的。**单元内的大端或者小端分配是实现定义的，对齐方式并不指定。
3. **对于非位域的成员，根据其类型决定合适的对齐方式（这句话意思就是，显然标准没有给出所谓的合适的方式，是编译器决定的）**
4. 对于成员，地址是按顺序增加的，指向该结构体的指针（经过一定转换）指向的是第一个成员的地址单元（哪怕是位字段）。结构体中可能存在未命名的填充，但是开头没有填充
5. 对于union，其地址大小足以容纳最大的成员，在同一时刻，最多只有一个成员被存储其中，指向union的指针，指向每一个成员，位字段则指向其存储的单元。
6. 在struct或者union的结尾可能存在未命名的填充

然后是通用的实现（这是查的资料，非我原创，未必准确）

反正我看的头大了。说说我的看法吧。

我总结之后，每个成员应当有一个size和align，对于一些基础类型，char和int这种，这两者总是相等的，但是当遇到特别的类型的时候，则不同，function不说了，struct中不允许出现function，对于array，比如char a[5]，他的size是5\*1=5，而align的是按照基础类型char的大小来的，对于struct/union，size是包括后面填充的大小的总体大小，而align值则是内部成员最大长度。

除此之外还有一个系统的sys\_align.

随后是有效对齐值，valid\_align，这个值等同于min(sys\_align，align)。换句话说，加入是64位的longlong int，8个字节，但是系统对齐系数为4，那么就需要按照4字节来计算。

每个成员的首地址必须整除于该成员的valid\_align。

并且结构体的话是地址递增的，中间如果有空的，就填充。

最后，结构体的size还需要满足整除valid\_align。

应该就是这些了（这些变量可能在不同的文档里面叫法不一样，但是大体肯定是不变的）。

我按照自己的理解写了以下的伪代码：

|  |
| --- |
| #define ALIGN（addr，align） /\*地址向上按照align对齐的宏\*/  #define SYS\_ALIGN /\*默认的对齐模数，相当于#pragma pack给定的\*/  Size\_ttype\_size（type）； /\*获取某个成员总占用大小的函数\*/  Size\_ttype\_align（type）； /\*获取某个成员对齐的大小的函数\*/  #define VALID\_ALIGN（member） min（SYS\_ALIGN，type\_align（member->type））  Struct\_total\_size=0; /\*注：对于一个空结构体C编译器是规定多少大小的问题。\*/  Struct\_align\_size=0  For member in struct  If struct->type==union  Member->offset=0；  Struct\_total\_size=max（Struct\_total\_size，type\_size（member->type ））；  Else if struct->type==struct  Member->offset=ALIGN(Struct\_total\_size, VALID\_ALIGN（member）)；  Struct\_total\_size= Member->offset+type\_size（member->type ）；  End if  Struct\_align\_size=max （Struct\_align\_size，type\_align(member->type）；  End for  Struct\_align\_size=min（SYS\_ALIGN,struct\_align\_size）;  Struct\_total\_size=ALIGN（Struct\_total\_size，Struct\_align\_size）； |

但是，当结构体是匿名结构体的情况会怎么样呢？显然，直接从含义出发，匿名结构体虽然作为上一层结构体的子对象来进行访问，但是并不应该改变其offset，考虑一下union中嵌套一个匿名struct，该struct存在多个子成员的情况。从语法含义出发，这个struct的多个子成员应当按照顺序存储，而不是看似作为上一层union的子成员，而改变其实质是一个struct的情况，否则就没必要这么做了，实机测试的情况也是如此。

~~不过，如果是在specify里面指定了align的情况下，那么另当别论，这时候成员的offset不再是按照valid\_align，而是按照指定的align来设计。~~（这种情况不会发生，因为结构体的那个里面是不允许指定alignment specify的）

当然，由于前面所描述的两趟遍历来解决匿名结构体的情况，在第一趟中，仍然作为一个struct来处理，第二趟再提升层级，那么，在提升该层级的过程中，子匿名结构体成员的offset还需要加上该子匿名结构体的偏移值。

然后再来说说加入了位域字段的情况。

同样先说C标准的约束

1. 对于表示位域的形式，后面必须跟着一个常量表达式，并且大于等于零，如果为0，那么这个冒号前面的declarator必须不存在
2. 位域字段的类型应当为（qualified或者不带qualified）的\_Bool，signed int，us int或者其他由编译器实现定义的类型（按理来说，应当是int这种整数而不是float或者其他乱七八糟的类型，typedef当然可以，实际算类型的时候，是直接把类型复制过去的），并且atomic是否支持是由编译器决定的（那我不支持了）
3. 应当保证分配足够大的可寻址单元来存储位字段，如果当前单元空间足够，下一个成员的位字段内容应当紧跟着，如果空间不够了，到底是放到下一个存储单元还是跨单元存储，这是由实现定义的。单元内的大端或者小端分配是实现定义的，对齐方式并不指定。
4. 没有declarator但只有冒号和宽度的位字段声明表示未命名的位字段。作为一种特殊情况，宽度为 0 的位字段结构成员表示不再将位字段打包到放置前一个位字段（如果有）的单元中。

（这个翻译并不算完整）

对于通用的实现，我只能说，C11标准同样没有规定具体的实现。对于位域的处理，也存在编译器之间的差别。存在压缩和不压缩的区别（但这指的是，相邻位域类型不同的情况）。

大概做法如下：

1. 相邻都是位域类型，且基本类型相同的，如果按照该基本类型给出的类型大小，可以放在同一个类型大小的区间（也就是在该单元内），那么紧跟，否则，offset就到下一个（该基本类型给出的大小）单元内。新单元的offset按照前面没有位域类型的给出来。
2. 当都是位域类型，但基本类型不同，这里存在压缩和不压缩的方式。还是举个例子吧，假定一个struct里面，先放一个char的位域，实际占用宽度2位，再放一个int的位域，实际宽度为3位。Char从0开始，而按照没有位域类型的情况，会给出下一个单元，也就是int的offset，应当为4，和第一个位域类型中间一定会存在多余的空间（如果空间不够，那肯定放到下一个单元了），这种情况下，压缩就是放得下的放到前面一个空间去，换句话说，那3位直接放到offset为0的地方，不压缩放到下一个单元，也就是偏移为4的地方去。

（由于gcc使用压缩方式，所以默认我一定是使用压缩方式的，没有办法呀，屁股决定脑袋，而且从位域本身的使用来看，目的就是为了压缩存储空间，当然，我会按照宏定义给出不按照不压缩方式给出合适的代码）

不过这样一来，本来，1和2按照压缩方式，其实两种情况可以并列起来考虑，但是加了这个宏定义之后，emmm那就只能检查前一个位域字段的类型了。

1. 如果位域字段之间穿插着非位域字段，则不进行压缩（相当于查看前一个，不是位域类型就行了。）

那么我具体的实现如下（主要是基于上面的算法的，这就不用伪代码了吧，描述一下）

一个非常关键的问题在于，原先的offset都是按照byte来算的，但是在这种情况下，则是按照bit来算，怎么办？我的做法是，在member中加一个字段表示bit相对于单元offset的偏移bit数，而offset还是原来那个。这里面就涉及到小端大端存储问题。当然，这有办法规避不去涉及。

同时还需要记录位宽度是多少。

每次都去访问前一个member，按照上面三种情况区别处理，当然，还要考虑无名位域的情况（网上说的什么压缩，语焉不详，我看着也头痛）

再来说说加上了\_Alignas关键字的情况，先看标准怎么说的吧。（这里面依然存在和C++标准的区分），最为关键的问题是，这是否影响struct的内存分布，在C++中，struct alignas(8) struct\_test{}这样的结构是存在的，但是在C标准中，这连parser都过不去的。因为struct后面不能跟这个东西，我在网上看到的翻译是，C11标准，正如我看到的文本一样是不支持struct的这个所谓align的，但是什么DR 444之后就让他成立了（天知道这个乱翻译的英文到底是什么个意思），所以，我想，在整个结构体的布局上面，我没必要去考虑特殊的对齐，而是按照之前的设计即可。

当然，结构体的成员的对齐则并非如此。这个align是可以修改每个局部的结构体成员的类型对齐要求的。

不过既然这里提到了，那么我也就不想在别处再进行一通分析，这太乱来了。干脆就这儿写完吧。

1. alignas不能修饰一个declaration of typedef，bit-field，function，或者一个parameter或者有register存储类型的对象（这个的难点在于，何处做这样的检验，bit-field的检验肯定是在struct对象那里**DONE**，parameter可以在function那里检验（但是我认为旧式风格的parameter也需要，这就是两处地方）**DONE**，而typedef可以在alignas后面统一做**DONE**，function则是在function的identifier那里，有register的对象则在declaration那里做。到处都是，真一句mmp）**DONE**
2. 常量表达式是一个常量，且是int类型，它应当为有效的对齐类型，或者是其他的扩展（我个人对后面半句话的理解是，不能任意指定一个对齐类型，而是一定要4、8、16这种常见的对齐。）**DONE**
3. 对于alignment，不能比修饰的类型本身更不严格，就是某个类型，本身是align为4，你一定要给他2，那么就忽略这个alignment（在我的实现中，会记录下这个align，但是在计算alignof的时候，进行比较去忽略这个更不严格的修饰）**DONE**
4. 如果是一个typename，那么类似于alignas（alignof（type name））**DONE**
5. 对象按照alignment来对齐，如果指定为0，那么没啥用，多个声明alignas的时候，使用更加严格的那个**DONE**
6. 如果一个对象的definition有alignas，那么其他的声明要么相同alignas，要么没有alignas，不能不一样，如果一个definition没有alignas，那么其他声明不能有，如果两个不同translate unit有不一样的alignas，行为未定义。（这个可以在declarator那里看identifier）**DONE**

* flexible array的情况。

本来作为一种特殊情况，并不需要太多的检测。但是可惜这里面各种乱七八糟的规定太麻烦。而且还跟array有关系，我干脆放到array里面去讲，太烦了。

* 类型入栈顺序

这里的关键之处就在于如何处理这些类型的压栈。从直观的语义上而言，array类型最顶层的axis应该是放在最前面，前面也说过我的类型栈是从前向后，因此，按照这样的想法，最顶层的axis应当在最后压栈，对于(\*r)，这当然不是point direct-declarator了，而是作为direct-declarator的(declarator)来处理，而越往后的应当越先压栈，因此，这第一条规则就是在direct-declarator节点，最后面的最先处理并压栈。

当然，对于指针而言这顺序是恰恰相反的，如果一个指针的指针，他们的压栈顺序则是从前往后压栈，这个事情很好证明

|  |
| --- |
| Int \*\*const a;  Int \*const\* a; |

前者是一个const的指针类型，指向一个int指针，所以对其赋值一定出错，后者则是一个指向const指针的指针，所以没啥问题，前面说过修饰类型，他们修饰的一定是前面一个\*所表示的指针，因此，其压栈顺序一定是从前向后，而不是像数组那样从后向前。

总而言之，压栈顺序，在direct-declarator节点，一定是从后向前压栈的，在前面的pointer节点是从前向后，而spec或者qual list等等都是从前向后（请结合文法去看这几个节点），并且都是先压基本类型，再压修饰类型的，所以再举个几个例子，我自己设计的：

|  |
| --- |
| Int a[2](void) //不合法，表达的是一个数组，元素是返回值为int的函数  int a(void)[2] //不合法，是一个函数，返回值为数组，该数组成员是int型的。  Int a(void)(void) //不合法，返回值是一个返回值为int的函数。  int (\*a(void))(void){…} //合法，表示的是一个a函数，返回一个指针，该指针指向一个返回值为int的函数，我估计没人这么做，会被打的，但是是可以做成这样的吧 |

这里面可以深切的理解这个type的压栈顺序了吧？复杂声明的解析基本上就是这样了。

当然，我必须指出，这是我选择的实现做法，并不是实际最优的做法，例如，在栈中，storage class只会出现在最底下的一个modify节点中，而上面的不会出现这个东西，同理，pointer，array和function也不会出现在最底下的那个base type节点中去，所以，数据结构的设计，完全可以有更多的考量。我这么做还浪费空间，只是，没必要太细罢了。

* + Array类型

对于array类型，他的难点并不在那一大堆乱七八糟的，而在于一个简简单单的指针，源于其本身能够用指针来表示某个array。

这里我同样直接拿标准中的例子来进行说明，推测这个东西应该怎么做。

|  |
| --- |
| extern int n;  extern int m;  void fcompat(void)  {  int a[n][6][m];  int (\*p)[4][n+1];  int c[n][n][6][m];  int (\*r)[n][n][n+1];  p = a; // invalid: not compatible because 4 != 6  r = c; // compatible, but defined behavior only if  // n == 6 and m == n+1  } |

这里面当n和m满足条件的时候，r和c是能够进行比较的，说明他们的类型是相同的。

（复杂的规则让我头疼，我觉得有必要让这些定义说人话。）

还是回归语法要求吧。包括function类型在内，一个是括号内就可以判定的东西，而另一个则是在上层declarator中进行的一些判定。

Array类型里面，首先需要弄清楚的是VLA和VM——VLA（variable length array）、 VM（variably modified type）。

对于前者，这很好定义，就是一个array，他的size在编译期是不能确定下来的，就被叫做VLA。对于后者，我找来找去，只找到这么一个说法（我确定，这应该是VM的定义之处）：

|  |
| --- |
| If, in the nested sequence of declarators in a fulldeclarator, there is a declarator specifying a variable length array type, the type specifiedby the full declarator is said to be variably modified. Furthermore, any type derived bydeclarator type derivation from a variably modified type is itself variably modified. |

翻译过来就是，对于一个嵌套在fulldeclarator中（full declarator就是嵌套里面最上层的那个declarator）的declarator序列，如果其中某个declarator是VLA，那么最上层的那个就是VM，这个还好理解。后面那句应该是说，由VM衍生出的类型都是VM。Emmm因为肯定有array，所以直接连着array和point是可以的，但是直接连着function是没可能的。所以检测这件事情就很简单，查看type，然后所有的type向量里面，是否存在VLA就完了。

另外，对于array这个数据结构的设计，就有所影响，如果不是一个constant的结果，那么必须指出其对应的ast node，用于运行期的计算。

对于语义分析中的array的部分，由于方括号内的东西其实没啥啥，有关系的检测，都在上层调用到这个array的地方。

我对此进行各个需要检查的地方的归纳总结（代码实际上散落各处，真头痛）

1. 如果表达式存在，那么必须是整数类型，不存在则是incomplete类型，并且也应该被算作vla。**DONE**
2. 常量表达式必须大于0（当然，gcc是可以0长度数组的，但是一般都用在创建flexible array type上面）**DONE**
3. （在上层）检验每个元素（也就是他所修饰的类型）的类型，不能为不完整或者函数**DONE**
4. 如果出现了qual或者static，检验其是否在函数的参数列表中（这里面并没有要求是在function prototype scope，而是只要是参数列表就行了，所以检验方法是检查declarator的父节点是否为parameter\_declaration**DONE**），并且是在最顶层的数组（换而言之，当插入一个数组的时候，检查下一层的数组，那么他的qual或者static必须为不存在，关于这一点，我测试了gcc，发现其貌似并未遵守，**DONE**）
5. 当识别一个标识符时，如果为VM，检查是否为普通标识符（因为label name和tag基本不会出问题，这是由文法保证的，主要是在结构体中的member检查是否为VM类型，而不是在识别identifier的时候，当然，这里面的例外就是flexible array type，这部分请结合后面去看）。同时，检查是否为no linkage，以及在block和prototype 的scope中。如果都是，则没问题，否则报错。**DONE**（同理，在struct中的flexible array type是不一样的）
6. 识别标识符的时候，如果为VLA，检查static或者thread的存储类型，不应当为static或者thread（注意是VLA不是VM，如果是一个指向VM类型的指针，是可以存在static的，另外，按照我的设计，如果没有存储类型不会生成节点的）。（**DONE**）
7. 如果长度是一个star而不是expr，这是一个未定义长度的VLA，只能用在函数参数作用域中的声明或者type name（**DONE**）（测试了一下，gcc也还是function prototype scope），但是，这是complete的类型（**DONE**）。
8. 如果size不是一个const int的表达式，在function prototype scope的时候像\*的情况被对待，否则必须保证每次运行都得到大于零的值（我对此表示非常疑惑，我不明白这应该怎么被检查，毕竟这是在运行期决定的参数，不能在编译期所确定，而运行期不可能插入过多的符号，不过，总之我决定，这不在考虑范围内，因为我使用了gcc做了测试之后，发现如果声明一个变量t为-1，而声明一个arr[t]根本不会报任何编译错）**DONE**
9. 对于括号内为空的情况。事实上，这个地方让我疑惑。我描述一下，主要是在各个作用域内判定的情况。（**TODO**）

把这几个检查都做了，就好了（依然头痛，哪里好了？）。

关于array最长的长度。在limit那里，我记得我弄过一个object最少得65535。但那是最少，关于最多那个问题，啊我查资料快吐了。就gcc编译而言，uint64 max那个大小放进去肯定挂，可是问题来了，他之所以挂了是在检查整个object的最大长度的时候。那么array的一个轴最长能有多少呢？我估计做的时候肯定没检查。

由于这边涉及到了flexible array type，而且前面遇到这个的时候，我也没搞他，所以这边弄一下吧（前面有说那个gcc嵌套匿名结构体里面flexible array的问题吧，直到我开始做这个，算是知道为啥他那么设计了，如果不这样做，要多好多代码，真心累）。

1. 检查是否是incomplete array type，检查去掉这玩意之后是不是还有别的成员在，检查是否是最后一个成员。
2. Struct的大小要去掉这个member（事实上，对于VLA的成员，size设为0，而且如果不是最后一个出现了，会报错，所以，对于size没有影响，但是，有影响的事情在于align。想象一下，一个char后面跟着一个int的flexible数组，后面的flexible当做不存在，那么到底是按照char还是按照int的大小来算align呢。我在gcc中实测了以下，他的sizeof设为了4，也就是按照int存在的情况来算align和实际结构体大小的，所以无需更改）
3. 对于不是flexible array type的情况，需要按照上面的5进行检查。（但是貌似我无法触发这个错误，因为他在之前就会被检查出来）

别的废话我也不多讲了，没力气讲了。

* + Function类型

然后接着，弄function的东西，这里面规则不少，我先翻译一波，去看另一篇文档吧。

Function参数需要有个adjustment去调整。

然后正如array里面一样，我总结一下需要检查和设计的部分，这里需要加入function的类型构建部分，而不仅仅是检查，类型构建需要考虑的东西，尤其是旧式风格的情况，这些都放在具体的约束和语义分析描述那篇文档中去了。在此只做实现和语义检查上面的设计说明。

**实现方面**

我设计了一个函数，像array里面那样，只对括号内的东西检查，然后提取出他的参数等等，关于旧式风格，如果括号内啥也没有，则是旧式风格的，如果括号内是identifier列表，同样当成旧式风格来处理。**DONE**

他的上一层应当而且必定是一个declarator。在这里去检查参数列表中无法检查的问题，这在下面列举出来了。同时检查重定义的时候，如果是个function类型，重定义还需要考虑兼容类型以及composite type的情况。当两个兼容的function声明的时候，需要将他们转化为一个公共的类型。（这个部分应当在function definition中完成）

按照文法分析的我不多说了，一个个节点弄过去，主要是adjustment方面，需要调整节点的类型。

对于array的调整，应当调整为指针。指针的qualified类型是在括号里面所指定的，但是对于static的情况下不做检查（感觉没法检查呀）。**DONE**

对于function的调整，调整为pointer to function return type，简而言之，加个pointer。**DONE**

Void类型，不应当插入该参数，而应该转而为无参数。**DONE**

**检查方面**

1. 在上层检查function的返回类型，不能是function或者array的**DONE**
2. 检查参数列表中的type的storage class，只有register是被允许的。**DONE**
3. 如果在函数声明的地方，出现了identifier list，应当判定错误。**DONE**
4. 经过调整后，一个函数定义中的declarator部分（换而言之，声明可以）不应当存在不完整的参数类型**DONE**
5. 如果存在void，那么必须只有这一个参数，就不能是（void,int a）这种**DONE**
   * function definition

对于function definition的地方，这里处理了下层的function部分所无法处理（或者看不到的）那些节点。并且，对于旧式风格的一些检查也在此完成。

从理论上来说，这应当放在external部分来进行，但是出于完整性的考虑，我选择将它放在这里来陈述。

另外，在mcc的具体实现中，我必须在此细致的描述我在符号表中对这部分的处理，函数的名称应当被视作file scope的，而函数的参数，则是应当被视作和下面的compound statement同一个block scope。不巧的是，在这两层符号表中间，还需要夹杂着一层function scope，这一层符号表从逻辑上而言，应当是在function definition节点生成的——但这样，由于语法的问题，函数名称会被当做function scope，而不是file scope，所以，需要插入到上一层的file scope并在这一层删去。

对于下层的参数和compound stmt部分，本来他们会分属不同的符号表，从大部分编译器所需要的功能来看，我应该做个符号表合并的工作，但是我想这不太乐观，因为，这意味着可能需要在符号表中，重复在declarator中就有的检查，而且这样的合并可能会引起一些奇奇怪怪的内存错误问题（也许我应该将检查放在符号表插入中？但是这样并不能看到语法树。）所以最后我选择将它指向同一个符号表，完事儿！

总之，当所有的类型都构建完成后，这样一个基础的pipeline已经完成了。

* + 符号表搜索

在这里，有几个实现中的问题，需要阐明。对于符号表的搜索，有两种，一种是判定重复，对于最普通的标识符而言，是只在当前的符号表查找的。另一种则是需要遍历当前符号表和上层的所有符号表，比如在文件作用域定义了一个符号a，在块作用域引用了他，那么需要向上搜索才能确定。

如果仅仅如此倒也罢了，不巧的是，还有一个typedef，这玩意儿可以overlap，也可以重复定义相同的类型，只要定义的类型相同，重定义是完全可行的。

因此在此处，我打算专门描述一下在声明以及语句之中引用的时候，当遇到某个符号的时候所做的处理。

emmm继续举个例子来说明吧

|  |
| --- |
| Typedef struct s{ int a; } str;  Int main(void){  Str tmps;  Int str;  } |

对于上面这个例子，实际中是允许的，只是在int str之后，所有有str的地方都指向了这个int。而前面那个str为什么能够作为一个struct呢，他用的并不是本作用域的str，而是上一层file scope的str符号。

如果将函数中的这两行代码换一下，那么会报错，在声明了int str之后，在这个作用域剩下的地方str都指向了这个int的对象。

这就是overlap（的某个样例。）

而这也体现了向上一层递归的搜索匹配和在当前一层进行匹配来确定redefine的不同。这里面的区别在于，在作为typedef name被引用的时候，该名称有可能是在当前层，也有可能是在上层，而当作为被命名的identifier的时候，则一定是在当前层

* + 表达式的类型系统

表达式本身类型系统并不算那么复杂，但是，有问题的地方在于相容类型和强制类型转换。另外，表达式本身除了使用声明中所使用的identifier之外，还涉及到了type-name以及initialize等部分。

老实说我已经受够了这在一个文件里面堆叠大量代码的事情，然而他们本身的互相调用让我非常烦恼

* + Compatible type、implicit conversion、lvalue

这个部分是无论如何也绕不过去的。

先按照惯例，翻译一波先。

* + 类型转换

按理说，类型的介绍应该放在这部分前面，放在后面自己看吧。

每个int类型都有一个integer conversion rank

即使由于两种类型的表示相同（比如某些环境下long和long long都是64位），但是他们的rank仍然是不同的

Long long>long>int>short>char,unsigned和signed具有相同的rank。

标准的int大于同样存储大小的扩展int（我可没做扩展int型）

Char的rank应当等于unsigned和signed的char的rank

Bool应当小于任何其他int的rank。

枚举类型应当等于compatible类型的int

**integer promotions整型提升**：如果一个对象或者表达式的integer conversion rank小于等于int或者unsigned int的rank，或者是一个位域成员具有\_Bool, int, signed int, 或者unsigned int，如果能够完整表达所有的值，则转化为int，如果不能则转化为unsigned int，这被称作整值提升。

（所以如果用来在表达式中做比较或者做运算，需要提升为int或者unsigned int，这里存在一个隐式类型转换）

如果scalar 类型被转化为bool类型，则要么值为0，则为0，否则为1。

对于int类型间的转换，如果新类型能够表示，那么仍然为原来的值，如果不能，如果是unsigned的新类型，那么重复增加或者减少新类型的最大值直到能表示，如果为signed的类型，那么实现定义。

对于int和float之间的转换，如果是float转换为int，小数部分直接被丢弃，如果不能用int类型来表达，那么结果是未定义的（不是实现定义的）。

如果int转换为float，如果能完全表示，那么依然不变，如果不能完全表示，那么根据实现，转化为高于或者低于该值的最接近的值。如果超过了可以转换的类型大小，那么未定义。

对于float类型之间的转换，同上，貌似两段话完全没变。

Complex类型同上面的一样

Float和complex类型之间的转换。

Float转换为complex，实数部分，按照上面的float转换规则，虚数部分则设置为0，相反的complex转换为float，实数部分，仍然按照上面的转换规则，虚数部分丢弃。

**usual arithmetic conversions：**

这种转换，首先并不会改变操作数的类型，只不过是为了计算的方便。

如果一个操作数是long double，另一个会被转换为long double之后计算

Otherwise（这里我认为相当于elseif，也就是除了上述的情况之外），如果是double，转换为double。

Elseif，float，转换为float。

Elseif，会进行int promotion。

在这里，如果类型已经相同，则不做任何操作。

如果都是signed或者unsigned类型，那么rank低的转换为高的。

如果类型不相同，如果是unsigned的类型具有更高或者相同的rank，那么int类型的会转化为unsigned。

否则，如果signed的类型，能够表示unsigned的类型，那么进行转换为signed的类型。

否则，则两个数都将转换为与具有有符号整数类型的操作数的类型相对应的无符号整数类型。

（哇，总而言之就是一个又臭又长的类型转换规则）

（对于最后signed类型能否表示unsigned类型的情况，就是倒数第二条，我也看了不少资料但我没找到明确的说法，由于存在两个不同rank的int具有相同长度的情况，所以有可能无法表示的。如果，对于这句话的理解是，变量中的**值**是否能够转换为signed，这种情况下，需要查看具体的值才能判定是否转换。可是这一点在编译期是无法确定的，也就是说，这必须运行期才能确定是否进行这样的转换，显然这不符合实际情况。

因此，我对于这段话的理解是（我不确认这样正确），举个例子，一个signed的64位类型，能够包含一个unsigned的32位类型的所有值，因此后者会被转变为前者，而如果是一个64位的signedlonglong和64位的unsignedlong，前者的rank要更大，但是前者不能包含所有的unsignedlong，哪怕后者的实际值能够被前者所表达，仍然不按照这条规则来使用）

上面是int类型的部分。

浮点操作数和浮点表达式可以用更精确的类型来表达，但是他们本身的类型并不会被改变。

指针和int型的转换。

* + Lvalue

虽然被翻译成左值，但是显然这是历史原因，包括标准中也说，lvalue更加像是，locator value，大意就是可寻址，而rvalue，也不再是右值，则是有了read value的含义。不过，还是按照这么翻译呗，不然无论是理解还是查资料都不太方便。前辈们的辛勤劳动我怎么忍心忽略呢。

左值的定义，左值是一个expression，没错，不是一个object，但是他可能指明一个对象，这个对象包括符号表里面的，也包括符号表没有的或者经过一些表达式操作转换出来的。下面提到的左值，都会被当做一个expression。

可修改的左值，不能是array类型，不能是不完整类型，不能有const的限定。如果是一个struct或者union，不能以递归等形式存在const的成员。

**左值转换**：除了几种操作符之外，其他没有array的左值被转换为存储在其中的值，这几种操作符包括，sizeof， \_Alignof，unary中的&，++，-- operator, the left operand of the . operator，assignment operator

当发生了左值转换的时候，如果原先的左值是一个qualified的版本，那么转换后的结果具有非qualified的版本，如果左值是一个原子性的，那么转换后不具有原子性。如果左值具有不完全类型，并且不是array（额，那不就是说，void或者struct嘛），那么行为未定义，（直接判定错误就好了）另外，如果lvalue指定了一个自动存储持续时间的对象，该对象本可以使用寄存器存储类声明（从未获取其地址），并且该对象未初始化（未使用初始值设定项声明，并且在使用之前未对其执行任何赋值），则行为未定义。（说了一大堆，就是说，这玩意儿没定义也可以用寄存器也没存任何东西进去）

（就我个人对这个概念的理解，他应当是说，取其值（或者地址之类的）进行运算，但是并不加以改变object内的存储值，所以，一个const的对象里面的内容完全可以作为一个运算的部分而不是一个左值，更简单一点，读和写的差别，读不会改变他的值，所以可以做一些隐式转换，但是写的情况，需要保证类型，如果你不了解，要理解这些可能需要你自行查阅相关资料）

对于array转换为指针的情况：除了几种情况之外，有array类型的**表达式**被转换为指针，指向array开头的元素，并且**不被当做左值**。如果是寄存器类型，则未定义。例外包括，sizeof， \_Alignof，unary中的&，以及string literal

对于function designator的情况：包含function类型的一个**表达式**，除了sizeof， \_Alignof，unary中的&之外，一个对象有function的返回类型的，那么被转换为指向该function返回类型的指针。

其他：我在这里遇到了一个emmm让我有一点点头痛的事情，源于之前简单粗暴的对specify的处理，主要是\_Atomic这部分，当左值转换的时候，特意提了一句关于这个atomic是因为这不是在qualified部分里面，而是作为specify的一部分。这两者的区分在于atomic关键字之后是否紧跟着左括号。

另外，在atomic之后跟的type仍然需要去判断好。关于他实际的含义，我想对此加以补充：

甚至于会出现例如\_Atomic \_Atomic(type name)这种能过编译的奇奇怪怪不知道为什么要写两遍还没啥问题的写法。

有的时候我也非常疑惑，既然atomic类型说明符作为只有在lvalue处才有意义的说明符，并且，lvalue进行转换的时候，还会把他转换成non atomic的，那么他存在的意义是什么？听上去很神奇。。。

* + Compatible type and composite type

先是compatible type，如果1、两个类型完全一样，则是compatible的。

2、（他说别的规则在后面的部分，所以我去找吧。）对于两个qualified的类型，他们应该具有可以兼容的类型的相同qualified版本（就是说，去掉qualifier符之后的版本应当兼容，以及qualifier应当相同）。

3、对于两个指针类型是compatible，那么他们应当具有同样的qual限定符，并且指向compatible的类型。

4、对于两个函数类型，如果是compatible的话，他们的返回类型应当compatible，并且，如果都存在函数参数列表，那么应当在省略号和参数数量上面维持一致。如果一个存在，而另一个类型由不属于函数定义的函数声明符指定，并且包含空标识符列表，则参数列表不应具有省略号终止符。如果一个类型具有参数类型列表，而另一个类型由包含（可能为空的）标识符列表的函数定义指定，则两者应在参数数量上达成一致，并且每个原型参数的类型应与将默认参数提升应用于相应标识符的类型所产生的类型兼容。（在确定类型兼容性和复合类型时，使用函数或数组类型声明的每个参数都被视为具有调整后的类型，并且使用限定类型声明的每个参数都被视为具有其声明类型的非限定版本。）（我可耻的靠翻译软件复制过来了，压根不想自己翻译）

5、对于struct union enum，tag和成员需要满足以下条件：如果一个用标签声明，另一个应用相同的标签声明。如果两者都在各自翻译单元内的任何地方完成，则适用以下附加要求：其成员之间应有一对一的对应关系，以便每对相应的成员都以兼容的类型声明;如果对的一个成员使用对齐说明符声明，则另一个成员使用等效的对齐说明符声明;如果对中的一个成员使用名称声明，则另一个成员使用相同的名称声明。对于两个结构，相应的成员应按相同的顺序声明。对于两个结构或并集，相应的位字段应具有相同的宽度。对于两个枚举，相应的成员应具有相同的值。

* + 表达式和常量表达式模块的设计

对于表达式模块，首先需要考虑的是，如何表示他的返回类型。

每个表达式节点都应该存储类型，而且在每个表达式结束之后都需要考虑是否通过lvalue那部分里面描述的方式进行左值转换。

再说常量表达式吧，这算是表达式的一个子模块。

我在此对于C的type进行一些翻译性的工作，主要是用于理顺各个名词之间的包含关系。按理说这应该在类型系统建立之初就开始写，可惜C的声明的类型系统还包括一大堆限定符，包括结构体在内的复杂语法才是类型系统本身的重中之重。

standard signed integer types：包括五个：signed char, shortint, int, long int, and long long int，他们的存储长度我就不多说了。

extended signed integer types：由实现来完成的一些扩展。

signed integer types：包含上面两种signed int

standard unsigned integer types：上述标准有符号数的无符号版本，各有五个，加上\_Bool类型，总共六个。

extended unsigned integer types：同上，扩展

unsigned integertypes：包含上述两种unsigned int

standard integer types：由standard signed integer types和standard unsigned integertypes组成

extended integer types：由extended signed integer types和extendedunsigned integer types组成

real floating types：三种，分别是float, double和longdouble

complex types：float \_Complex, double\_Complex和long double \_Complex

floating types：包括real floating types和complex types

corresponding real type：（我为这个概念而感到迷惑）他的说法是，对于real floating types，形式完全相同，这没问题，但是对于complex类型，就是去掉了complex关键字一样。

当然这在下面有解释，complex类型相当于两个同样的浮点值排列成一个宽度为2的向量一样，第一个表示实数部分，第二个表示虚数部分。

basic types：包括 char和有无符号的integer types以及float type

（这里有两句话非常关键，一个是basic types是complete的，另外，即使两个声明使用了相同的basic types的表示方法，仍然是不同的类型，这句话我的理解是，比如double和long double在某些实现中都是64位，但仍然是两种类型）

character types：char, signed char和unsigned char，但char和哪个相同是由实现定义的

enumerated type：不解释

integer types：char, the signed and unsigned integer types以及enumerated types

real types：integer and real floating types

arithmetic types：Integer and floating types

scalar types：arithmetic types和pointer types

aggregate types：array和struct类型

另外，void类型是不完整的，并且不能被补全。

（记住这些关键字，当然实际上整个标准值得留意的关键字还很多）

然后来聊聊具体的存储和实现。

**·一个简单的想法**

对于地址类型，这当然很简单，一个指针即可，而对于其他类型则不然，int类型，虽然类型众多，都可以用最大的存储也就是8个字节来存储，因此差别不大。

然而，字符串常量怎么办呢？由于其长度未定，并且在这之前还能有L或者U之类的限定来表示宽字符，所以只能给个地址，由具体计算的时候，再进行更改。

对于算数常量表达式，这里需要的空间就不是八个字节就行了，算数类型包含整数类型和浮点类型，而浮点类型包括实浮点数和复数类型，对于复数需要的空间最大应该为两个double，也就是16个字节。

由于可以确定的类型中，最小的类型是1个字节，而最大的多达16个字节，如果统一标准，会引起巨大的浪费，更不用说所谓的字符串了。最简单的想法，统一拿个指针去指来指去得了。

对于常量类型的设计，一种方案是直接使用原有的类型，这并没有什么冲突的地方，但是缺少字符串常量的表示。事实上，这在具体实现中，应当表达为一个char []。如果看过指针和数组互转，那么应该知道这可以表达为一个char指针，而这边，本来就是拿一个指针指来指去，所以，指向一个字符串常量也是没有问题的。

对于long double和long double \_complex类型，其实说到底就是long double到底应该多少位。在我的gcc上打印出来的分别是16和32个字节。

另外，对于char和string的区别我也必须在后面进行说明。

**·实际遇到的困难之处**

拿指针本来没啥问题，有问题的地方在于，当这些常量表达式需要被用到的时候，由于去掉了所有的类型信息，就看上去没那么美好了。

这么说吧，比如一个array需要一个常量的长度，当因为层层expr递归，首先，由于类型的转换，子表达式的类型长度和实际的数据长度之间未必相等，这时候，数据长度本身是真实的，但你无法再得知这一段信息的编码方式，因为类型已经丢失，这可以在。

其次，由于去掉了类型信息，而只是对于数据的一个指针。那么一个32位的到底是一个float还是一个int，这里面需要做一定的cast。

假设你听明白了我说的这些，应该明白这里面的困难之处。

最后，在代码生成部分，某些地方需要将这些常量表达式作为一个常量直接使用到代码中去。

因此，相比于代码本身的困难之处，浪费的存储空间本身也变得无关紧要了。甚至，如果能用一定的空间浪费来换取代码的简单，其实更好一些。

最开始的想法是，所有的类型都弄一遍，不管怎么转换，反正直接根据类型取值。如果发生类型的隐式转换，直接在对应的数据上面改，而无需考虑怎么简单怎么来。

我有考虑过某些类型可以发生合并的情况，但是并没有太大的意义。

**char const和string literal的区别**

任何一个脑子正常的C程序员并不会特意的去区分字符常量和string literal，事实上这样的混用，我也经常这么做，谁在乎呢，有编译器。然而对于编译器程序员则不是如此。

翻译的事情我也懒得做了

|  |
| --- |
| struct {      int b:' ';  }a; |

就举这个例子吧。后面这个单引号中间是个空格，对应的ascii码是32，这么写，绝对不会报任何乱七八糟的错，但是，如果，我改成了双引号，gcc会报错。

就一句话就能解释这两者的差别，单引号的是一个int型，他就是一个数，对应的ascii码表示的数（当然如果加上前面的LuU的限定符，可能会对存储位宽有一定影响），而双引号，则是表示的字符，而不是数字。

也因此，在常量表达式中，可以将这个作为一个int，而不是按照string一样的字面值，学废了吗？

对于多个字符存在的情况，比如‘ab’这样的，或者说，按照预处理第五步，那些转义字符或者扩展字符将其处理为多个char字符的情况，标准中认为是一个实现定义的行为。

但是我想，绝大多数的实现，都会连续起来，当成一个char的吧，这么说太抽象了，举个例子，那个’ab’，a是97，也就是0x61，b是98，0x62，那么连起来就是0x6162，往前扩展添加0，也就是0x00006162，就是‘ab’的int值。

随后是宽字符的问题。这个事情，本来没必要太复杂，据说gcc的wchar\_t是32位的，换句话说，宽字符他只允许出现一个字符转换为int，多余的都会报个warning。

毕竟，这也算是，实现定义的嘛。

（顺带科普一下，多字符和宽字符的区别，对于ascii码，他被映射到0-127，而大于等于128的，由于开头的位数是1，因此通常两个或者更多个第一位为1的char，才能表示一个字符，如果直接输入，那么每个字符的长度都是不同的，有的一个字节，有的两个。而宽字符则让他统一到2个（short）或者4个（int））

在这里呢，我仍然要做一个mbtowc, mbrtoc16或者mbrtoc32来进行这个转换，从而使他顺利成为宽字符，按照上面的描述，我不能简简单单的将第一个char转换为int就完了，会出问题的。

* + Postfix expression的语义分析

其他表达式没什么需要多做的步骤，但是postfix需要做不少的分析，其他一元或者二元表达式讲真也没啥好分析的，在这里，需要结合前面所描述的类型转换和左值的部分，将其进行一定的转换，从而去做一些语义分析，这个转换的结果类型应当被存储在语法树节点中。

确切一点的讲，表达式的语法树节点，是一个<值，类型>来共同确定的，值的部分，对于常量表达式而言，需要在这里就进行计算，而类型的部分则是语义分析的主要内容。

不巧的是，这个乱七八糟的文法，为了去掉左递归，也许是我写的不好，但最后postfix的表达式变得扁平，这带来了很多问题，甚至于在语义分析中，出现不怎么符合C的标准的事情，比如，下面的数组里面要求数组名称被转换为指针，但是如果二维数组要怎么办呢？由于其处在一个节点下面而不是组成一个树状，这种转换变得存在问题，指针的指针和指向数组的指针并取其首地址是不同的。

为了解决这个问题，在这一段的分析过程中，不得不加入临时的类型，并且按照顺序从前到后进行遍历，在这个过程中，原本每个节点中需要进行一些自动的转换过程，需要在每个临时类型中进行相应的转换。

由于其类型不少，这分析的复杂性，加之函数调用还必须考虑到旧式风格的函数，这代码写的让人头秃。

1. 数组

一个比较奇怪的地方就是他的约束，只要求其中一种存在array类型的指针，另一种有int类型即可，所以arr[1]可以，而1[arr]同样表示一个意思。当然，多维数组这就不太行了，我是说，并不能把这个arr放到多维数组最后。——由此可见，其读取顺序是从左往右哦。

另外就是\*arr和arr[]，这里面arr都被转换成了一个指向下一层arr的指针，那么显式的使用\*加上arr，和arr后面加上[]来隐式的表达，其结果都是表达了去掉了之前的指针，也就是下一层的arr的情况。

Arr的存储默认是按照行优先来进行的，或者说，最后面的最低级，而最低级的则聚集在一起。

1. 函数调用

函数调用一个非常让人讨厌的地方在于，类型的比较和转换，这里直接来了一句，the arguments are implicitly converted, as if by assignment——至于assignment里面到底有多麻烦，我就不想多讲。假如没有所谓的nullpointerconstant，这一切说不定还好，只需要考虑type\_vec，可是加了这个东西之后，还需要查看是否为NULL。

办法呢，函数复用呗，就是两个类型比较的进行重复利用代码。在此需要更改vec的这个数据结构，增加nullpointerconstant这种类型就好了，但这种类型本身不应当有实际含义，只不过是临时助记罢了，所以在其他数据结构中，比如类型的size那个向量啊之类的，不应当占据实际类型，另外，在expr节点之间传递的时候，也应该转换为sint来使用。

除此之外，假设需要在这里塞进去什么所谓的代码生成模块，那么将typevec进行比较的部分，进行函数复用都有问题了，一个是~~阆苑仙葩~~对着某个东西赋值，另一个是压栈（当然，你要是把栈指针的对象当成一个对象，也不是不行，然而这意味着更多的代码）。

这里需要考虑的一个事情是，在function模块中，已经做了一定的类型转换，所以在function结构体中的parameter已经是最后结果了，无需再进行转换。

我必须举一个非常非常有意思的例子，来说明这种隐式的转换，这是我在查资料的时候找到的，名为” C/C++左值性精髓”的文章里面，但原谅我没有找到原作者，另外推荐如果想要做的人，去找一下每个操作符结果是否为左值，网上也有详细的表格，请不要完全相信我另一篇文档中的记录。

对于某个函数p。

|  |
| --- |
| P（）;  （\*p）（）；  （\*\*\*\*\*\*\*\*\*\*\*p）（）； |

这样写都是等价且正确的引用了p这个函数调用的。如果你不明白那么让我来分析一下。

P在语法树中是在一个primary节点下的，他的类型是function，但是当在表达式中被引用的时候，他被转换为<point，function return type>。在这之前的\*会对这个指向function的指针解引用，从而变成了function。如此循环。

而当最后给出函数括号的时候，前面的是一个function，然后被自动的转换成<point，function return type >，然后用括号再解引用，去掉指针。（当然我无比确信如果实践中这么写，估计会被打一顿。）

所以有个问题，这个自动的转换，是在什么时候做的呢？在读取符号表得到符号的时候，还是在参与运算之前的时候呢？

以arr[0]这个举例子，如果不清楚，去写一个看看语法树吧，arr是在primary expr里面从符号表中得到的，他的类型是一个一维的数组，数组的类型是type（随便是什么）。然后他的类型会放到primary节点中，primary节点和他的兄弟节点[0]，共同确定了最后的postfix节点处应当是type类型。

以这个例子为例，我想问的问题是，这个转换应当在primary节点的类型确定之前做，还是在primary这个节点的类型确定之后，在上层的postfix节点中发生运算之前做呢？

认真查看资料之后，我认为是在上层的postfix节点去做的这个转换。而在primary节点则存储得到的真实类型。也就是说，总是在他参与某个运算之前，才决定是否做这个左值的转换。这点从左值转换里面的一堆运算符的例外就可以看出来。

既然考虑到了顺序，那么也顺便来说说，整型提升和左值转换之间的关系，他们同样被放在C标准中转换这个章节的部分。而且看上去都是必须要做的转换，那么他们之间谁先谁后，以及相互之间的关系呢？如果仅仅从语义来说，这貌似是个多余的问题，然而从实践的角度来说，代码中总要找个地方放这些东西，只要放了，就有先后顺序。

另外，如果对array和function进行pointer的转化，相应的就失去了function和array那个节点的信息，而array的长度也好，function的参数列表信息也好，都是非常重要的信息。

综合上述这些考虑，以及到处查找资料来推断其应当的语义，我在相关隐式转换上的做法如下。（顺便强烈推荐完完全全看了lvalue和相关的节点以及东西之后再来思考，这在C文法中表达式的语义分析中是个非常重要的东西——假如你想跟我一样做编译器，额目前是前端的话。）

在此之前，先假定一个表达式节点A，他存在两个孩子表达式节点B和C，并持有一个操作符O。

1. 左值转换也好，整型提升也好，他们都是在遇到某个操作符之后才进行的转换，这是在操作符上面具体规定的，换而言之，在B和C处存储的是B和C原本的节点和左值情况，不做改变。只有当遇到操作符O（或者说明确了目前是什么操作的时候，才对表达式节点进行转换）由于操作符会明确需要的类型，比如要求B是array，C是int，那么对int去进行默认的整型提升，而array则作左值转换，显然在这种情况下没有先后之分
2. 对于表达式，不应该更改他们在符号表中的本来的类型，因此在每个表达式节点都需要临时存储一个类型tmp type
3. 由于array和function的类型节点中存储重要的数据，所以虽然临时的tmp type中会将实际的节点表示为point to type和point to function return type，但是array和function的类型节点仍然需要在该操作中进行保留。
4. 表达式节点A具有经过BOC之后的类型而不做任何转换，正如1中描述的那样（这是递归的）。
5. Lvalue和具体的值类型是不同的，所以当某个操作符的返回类型要求是int或者其他算数类型等等，都是需要将lvalue转换为其具体的算数类型（当然实际上只要不要考虑lvalue标志就可以了）
6. 整型提升（其他类型的提升也一样）常见于二元操作符，当然也有一些一元操作符之类的会需要提升，具体见另一篇文档的表格

这样一来，转换都依托于操作符。对于操作符的这些转换需求我在另一个文档的表格中有所描述。希望能有所帮助。

* + 对表达式语义分析中的一些工具函数的设计以及类型，lvalue等等相关实现

由于在表达式中的这些检查，对于type向量中的那些检查的工具函数变得迫不及待。

并且，由于之前面对这些表示type的vector的处理非常简单粗暴，因此，有必要对之前的设计进行改写和梳理，在此我先列出来需要对type vector所支持的操作。

首先，type vector并不同于vector，虽然他们是一样的，没有必要增加特别的位或者结构体成员来说明这些（如果有C++，那么会出现继承）。但是可以保证的一些地方如下：

1、vector中的每一个成员都是指向M\_TYPE类型的指针。

2、某个type的向量，本质上是一个栈，按照类型的顺序压栈

3、由于type本身，被设计为基本类型和修饰类型，修饰类型用于修饰，栈里面该类型下面的那个基本类型。而基本类型是通常会影响存储大小的类型（我知道这个分类和设计比较奇怪，但是可以去看前面对于类型的设计），所以这个栈里面两种类型都会混用，基本类型一定会有，但是修饰类型可以不存在。

4、对于storage class类型只会出现在栈最底下的那个修饰类型中（如果该类型存在修饰类型的话）。对于alignment和function这两个地方，目前仍未实现，所以之后可能会考虑进来。

5、一个类型的实际类型应当是栈顶最上层的除了\_Atomic之外的基本类型。之所以\_Atomic节点不能算，是由于这个别扭的语法导致的，毕竟\_Atomic既有说明符也有限定符，并且在语法中，显然说明符不被都归类到qualifier，限定符版本的\_Atomic被当做某个修饰类型节点，而说明符的话，则是当做一个基本类型节点，比如\_Atomic(int \*)被转变为<\_Atomic,point,int>，这里左边是栈顶。

然而这个atomic在确定其基本类型的时候，并没有什么卵用，所以需要撇除掉这个atomic。

考虑到表达式的语义分析，以及经常需要的一些操作，才做了以下这些操作的设计。

对于这个类型应当是**硬拷贝还是软拷贝**的问题。由于在表达式节点中，这些操作不管怎么操作，都不应当影响其下层子节点的类型，所以不管怎样改变，都会优先推荐硬拷贝（也就是vector类里面的那个copy函数）。这么做，额好像确实多了很多内存拷贝的工作，但是，没事儿，安全要紧，如果不小心修改了某些不能修改的东西，到时候头痛起来是真的没救了。

需要注意的是，该函数对于原来的目标向量，并不关心，而是直接新建了一个向量且不试图删除原有的向量，如果原来的目标向量经过这么一个copy之后不再使用了，需要上层函数自己删除，举个例子，某个表达式节点中，传入了一个向量A的指针，然后一堆操作，得到向量B，需要将向量B赋值给原来的A指针，那么A指针原先指向的内容在这个函数中是不做处理的，否则如果源操作向量和目标操作向量指向相同的内容（尽管他们经过各种赋值之后可能是完全不同的指针），容易搞出事情来。

1. 得到storage class的那个M\_TYPE节点，上面描述了，就实际情况而言，这个节点也会是唯一存储align和function specify的节点，如果有的话。如果有base类型，那么在栈中的顺序应该是从栈底向上数第二个，实际上序号是1，因为从0开始计数，有些情况下只需要一个qualifier什么的，那么就是第一个（虽然好像根本用不到大概）。所以查找栈低向上的第二个节点（如果存在，否则返回NULL），如果为modify并且提到的三个类型不都为0，那么返回该节点，否则只有qualifier，那一样返回NULL（这个函数并不应该在expr部分使用，因为使用了懒删除的策略，这个节点一定可以得到，但是在expr里面被认为删除掉了）**DONE**
2. 得到qualified或者unqualified，如果先遇到一个base type，就返回unqualified（实际上是返回NULL，即使碰到atomic的节点也一样，因为保证了atomic（type name）不应当包含存在qualified的类型，但是按照文法atomic可以修饰一个存在storage class的类型，所以，<atomic, modify type, base type>这样的栈顺序是可行的。但在这种情况下，栈中base type之上不会存在qualified不为0的节点），否则先遇上一个modify，如果那个修饰类型的qualified部分值不为0，就返回那个修饰类型，否则也返回NULL**DONE**
3. 得到atomic节点，下面既然在得到实际类型的时候，认为atomic虽然算在base type里面，但是实际上是修饰，所以，需要得到这个可能存在的用于修饰的atomic base type 节点，在这个节点之上，只有可能存在modify type，也许有，如果是另一个base type，比如pointer，那么表达的是一个指向某个atomic的类型的指针而不是一个atomic的pointer指针，所以会被当做pointer处理掉，其他类型也一样。**DONE**
4. 得到实际类型，他有可能被atomic的base type节点，或者qualified或者unqualified（也就是没有）的modify type节点所修饰**DONE**
5. 左值转换**DONE**
6. 是否是可修改的左值**DONE**
7. 对于单一操作数的整型提升**DONE**
8. Float操作数提升为double，这是由于default argumentpromotions中需要整型提升和这个提升都要。**DONE**
9. 二元操作数情况下的提升arithmetic conversions**DONE**
10. 两个类型是否compatible
11. 两个类型如果compatible（在调用这个之前一定要调用上面的函数），那么他们的composite 类型。为了防止没有调用上面的函数，需要设计的时候就设计好给出一个传入参数表明是否compatible以作为提醒。
12. 对于各种类型的判定——这个可以用宏（可以参考上面的类型部分）**DONE**
13. 如果是一个point，需要得到其所指向的类型（当然需要先通过得到实际基本类型，并判断是不是pointer）**DONE**
14. 如果是一个array，需要判断其子类型（同上判断）**DONE**
15. 如果是一个function，需要得到其返回类型（判断同上）**DONE**
16. 去掉atomic之后的类型（需要判断是不是atomic）**DONE**
17. 去掉qualifier之后的类型，（同上判断）

（在这几个操作中，理论上需要考虑一下modify的情况，如果在该节点之上还有修饰类型，比如pointer的修饰。但是考虑到实际的困难，并且由于这些操作可以在上层通过各种函数的组合确定，因此这边不写在函数内。也就是说，假如<qualifier, pointer, base type>这样，直接返回base type，而不考虑是不是变成<qualifier, base type>这样，虽然印象中貌似各种情况下都不会有，但我记不太全）

这里同样存在不少的重复代码，但是，我觉得吧，相比于用某种方式合并这些代码而言，这些函数本身由于其会经常被用到，如果合并的话，会增加我去查找对应类型的复杂度，所以还是分开算了。

1. 以上这些类型转换对于常量表达式的情况，也需要考虑进去，但是不会放在同一份文档中——在常量表达式中，比如两个数的比较，等等，同样需要整型提升等等类型操作，但是前面描述的提升需要类型的转换是针对类型链的，这里面的类型转换则是实实在在的值（可能需要扩展位数或者截断）。不过，这虽然放在这里描述，却不涉及到type vec向量的改变，而是const value的值的问题，所以，不会放在一起描述，至于type vec的改变，与上面没有太大差别。**DONE**
2. ~~判断一个类型是否位complete的——对于每一个type节点都有complete属性，但是整个类型链需要继续判定。这里面不应当包含function的判定，以及array本身有规定就是多层array不能是包含不完整类型。所以整体而言查看栈中最上层的节点即可。放弃了这个打算~~

（先完成这些工具函数，再说其他吧）

1. Null pointer constant的判定，有点麻烦了。。。来看一下这个东西的定义：An integer constant expression with the value 0, or such an expression cast to type void \*, is called a null pointer constant。
   * Statement的语义分析

理论上而言，expr算是最难的，而stmt这部分，基本上就是白给的，不过，即便如此，label的跨多个作用域的情况依然需要考虑。label并不是只有普通的标号需要考虑跨作用域的，举个例子：

|  |
| --- |
| switch(abcdefg)  {  case 0:  {  default:  i++;  break;  }  default:  break;  } |

这里面第一个default被嵌套在某个case里面了，但是他仍然是能被发现识别，而第二个则会因为和这个default冲突，所以报一个重复的default的错，也就是说，即便是嵌套，仍然可以跨作用域被找到。

另一个非常有意思的地方在于loop和switch同时存在，里面如果出现一个break，那么会怎么样呢？一种是，loop里面嵌套switch，另一种则相反。

对于这样一种情况，我采用的设计是，由于switch，loop或者function这些都会新开一个作用域，因此，将这些信息绑定在作用域当中去。考虑到某条语句可能同时在switch，loop，function这里面，所以，采用位域的方式，如果有嵌套，那么做and就好了。

同时，如果是function，需要增加一个指针指向一个结构体，该结构体应该记录好function的那个symbol结构体，同时记录在其中的label标号。对于switch，需要记录好对应的值，以及default的情况，防止重复。对于loop，倒是没那么多讲究。

另外，由于存在嵌套，所以要找的symboltable是当前某个位置位，并且父亲symboltable不存在或者未置位。

这只是做语义检查，但是在代码生成的部分，需要对此做更多的处理。

* + 初始化模块的语义分析

理论上，初始化不应该放在现在才做，但是其涉及到内存中数值的分布，跟代码生成已经有非常紧密的关系了，因此我不得不放在这里写，而且其本身也非常复杂。

这个复杂还包括了因为初始化还可以在compound literal里面使用。而那个地方并不会在符号表中申请一个变量，因为没有变量名字。而这会导致没有将结果进行存储。

对此，我并不再做更多的设计，在semantic的时候做一次检查，而后续中间代码生成的时候，再做一次init的操作。这确实会浪费一些，不过semantic和中间代码生成相分离，我认为是非常必要的。

这当然在一定程度上不符合简洁的原则，因为初始化模块的设计，需要考虑聚合类型以及其递归形式，其访问顺序和后续的init的访问是基本一致的，这相当于某种程度上的重复代码，不过好在只是多用了一次，而且合并的困难远比分开的要多，因此可以接受。

（注意：需要考虑初始化的元素为非常量表达式的情况，虽然在static等某些变量的初始化中明确说了不允许，而考虑到这个，就只能在代码生成的时候去做数值了。）

但是，在检查中并不是不生成任何数据，比如说，对于不确定长度的array，需要确定对应的长度等等。

在semantic过程中，需要检查的部分如下：

1/被初始化的类型，应当为大小未知的数组，或者非可变长度的完整对象类型。**DONE**

2/具有静态或线程存储持续时间的对象的初始化程序中的所有表达式应为常量表达式或字符串文字。

3/如果一个标识符的声明具有块范围，并且具有internal或者external linkage，那么不应当初始化。（显然这不会包括type-name所表示的类型，因为那个根本就没有标识符）

4/如果出现了designator，那么需要检查其当前对象，如果是[constant-expression]，需要数组类型，且里面的表达式为整形常量表达式，如果对象大小未知，那么任何有效的非负值都可以（顺带一提，在确定对象大小未知的情况时，需要考虑这个值，来确定数组大小），如果是具有.identifier，类型，那么具有structure或者union，并且identifier必须是该成员。

5/类型检查，初始化的很多地方，需要进行类型的转换和检查。

没显式初始化和static或者thread没显式初始化的，不需要检查，这对值有影响，但是不影响类型。

如果是scalar类型的初始化，可以被括号括起来，并且必须是单一的一个表达式，需要将原本的类型和unqualified的原本的类型用simple assignment的规则进行判断。如果是一个structure 或者union，那么需要是相兼容的表达式类型，这个意思，我举个例子，比如说 struct xxx a=(struct xxx){…}，这样的，显然右边是个postfix表达式，两边可以是不同的structure，但是要兼容就行了，当然另一种方式，那肯定就是右边不加type name直接初始化咯。

关于后者能不能用一个括号括起来，我测试了一下gcc，结果是不能，也就是说，右边是个assignment的表达式节点不带括号的，都可以直接赋值，相反的，如果是用括号括起来的，只有是scalar类型的，才需要simple assignment来检查，而structure union貌似直接不合法了。比如下面第一个就是合法的，但是第二个则不是。

至于这后一种为什么不合法，emmm，那就是跟后面的那个括号里的第一个表达式看作一个struct tmpa的对象，而试图赋值给tmpa t的第一个int对象，匹配不上才导致的。

也就是说，对于目标对象是一个scalar的情况，带不带括号都需要判断，否则，按照后面的初始化顺序来考虑即可。

（烧脑。。。）

|  |
| --- |
| struct tmpa{  int a;  int b;  };  struct tmpa tmp=(struct tmpa){1,2};  struct tmpa t={(struct tmpa){1,2}}; |

以及在检查的过程中，访问的方式和顺序如下（部分伪代码描述）：