本文档建议对照着C11标准看，相关概念什么的我就不多说了，主要是用于查找方便和便于理解（我加入了相当多的例子），这玩意实在太（TM）多了！

|  |  |  |  |
| --- | --- | --- | --- |
| 节点类型 |  | 相关描述和注释 | 完成情况 |
| 常量表达式 | 约束 | 1. 一个常量表达式不应该包含以下的部分：   Assignment操作符（assignment-expression中只能存在一个节点）  增加和减少操作符（没有特别指明，但是应该包含前缀式和后缀式两种吧就是++和--）  函数调用（在postfix-expression中）  逗号操作符（在expression中）   1. 每个常量表达式应当使用他们本身类型所能表示的值来表达 |  |
| 语义 | 1. 在编译环境中执行和运行环境中执行，应当有一样的结果，比如一个浮点数，就应该有一样的精度和大小范围（我不认为这件事在通常情况下有什么问题，但是在交叉编译情况下就不一样了，所以对于这件事情我认为有必要声明一句）。 2. integer constant expression：应当具有整数类型，并且只能具有作为整数常量的操作数、枚举常量、字符常量、结果为整数常量的sizeof表达式、\_Alignof表达式以及作为强制转换的直接操作数的浮点常量。整数常量表达式中的cast运算符应仅将算术类型转换为整数类型，但作为sizeof或\_Alignof运算符的一部分的情况除外。 3. 在初始化列表的情况下，常量表达式具有更大的自由度（注，是指designator里面的那个，而不应该是那个assignment），初始化列表的常量表达式可以是以下几种   arithmetic constant expression  a null pointer constant  an address constant（我觉得同上面一种没啥差别）  an address constant for a complete object type plus or minus an integer constantexpression.（就是地址的加减）   1. arithmetic constant expression算术常量表达式应具有算术类型，并且只能具有整数常量、**浮点常量**（多了这个）、枚举常量、字符常量、结果为整数常量的表达式的sizeof以及\_Alignof表达式的操作数。算术常量表达式中的强制转换运算符应仅将算术类型转换为**算术类型**（和int的不同），除非作为操作数的一部分转换为sizeof或 \_Alignof运算符。 2. 地址常量是空指针，指向指定静态存储持续时间对象的左值的指针，或指向函数指示符的指针;它应该使用一元和运算符或整数常量转换为指针类型显式创建，或者通过使用数组或函数类型的表达式隐式创建。数组下标 [] 和成员访问.和 ->运算符、地址符&和间接寻址 \* 一元运算符以及指针转换可用于创建地址常量，但不能使用这些运算符访问对象的值。 3. 实现中允许其他形式的常量表达式（肯定不做） |  |
| Declarations | 约束 | 1. 除了static\_assert之外一个declaration至少包含一个declarator（这个declarator不能是函数的参数或者struct union的成员），一个tag（如果一个匿名结构体中包含了一个tag的结构体，理论上应该是可以符合要求的，所以这里存在一个递归的问题，但是我这么做了，在gcc中仍然会有一个没有定义任何东西的warning，我认为这是执行了前面所描述的语义定义，把他也当成成员了，毕竟在struct中并不是一个declaration，而是被称作struct-declaration），或者一个枚举成员（所以一个空的enum是错误的） 2. 如果一个标识符链接类型为**no linkage（但是internal和external则不一定，所以需要记得检测linkage哦）**，那么相同作用域和命名空间中，不能多次在declarator或者type specifier中声明，但是有两种例外的情况。一个是typedefname可以多次定义以表示同一种类型，**但是不能是VM类型的**，另一个是struct union enum的tag可以多次重声明，这在后面会描述到。 3. 同一范围内引用同一对象或函数的所有声明都应指定兼容类型。（没懂啥意思） |  |
| 语义 | 1. 声明指定一组标识符的解释和属性。一个definition，对于一个对象，是导致他占用存储的情况，对于一个函数，包括了他的函数体，对于一个enum类型，只有那个标识符（我感觉是tag）的声明，对于一个typedef name，同样是只有那个标识符的声明。 2. （貌似没啥语义检查的用处） 3. （表示声明在何处结束，用处不大）如果没有链接类型，那么在declarator结束，或者有初始化的在初始化结束处结束，如果function，在参数处结束 |  |
| Storage-class specifiers | 约束 | 1. 除了\_Thread\_local关键字可以以static或者extern为关键字之外，其余的storage类型限定符必须最多出现一个。**（done）** 2. 对于\_Thread\_local关键字，如果出现在一个block scope作用域的对象中，那么他必须包括static或者extern。（但是反过来，对不是block作用域的情况，我个人认为，可以不包括吧）**（done）**如果一个对象有多个声明，任何一个对象声明中出现了\_Thread\_local关键字，那么所有声明中都必须出现这个关键字。 3. \_Thread\_local关键字不应该出现在一个函数对象的声明中。   （看上去就是对\_Thread\_local的约束了） |  |
| 语义 | 1. （没啥用，就是把typedef归类到这里就是为了方便语法）。 2. （建议有register类型的对象尽快访问，但是是由编译器实现的，而且只是个建议，换句话说，额，卵用没有，我不实现了有啥问题么？）对于register类型的变量，比如说用个register int i来做循环，那么直接用ecx来计数就好了——但是这应该取决于寄存器分配策略，万一声明太多不够用寄存器了，那还是该咋咋地，所以忽略就好，而很多编译器其实都是忽略这一点的。另外需要注意一点，**寄存器类型的变量不应当取地址**。（存疑，我试图在unary 操作的地址那边找到这条说明，但是貌似没有） 3. 对于定义在block scope中的函数identifier，不应当存在extern之外的stroage-class 限定符。（对此我说明一下，一个函数的identifier为什么会存在block scope，C中可以在函数内声明函数，但是不应当在函数内定义函数，就是说不应当存在函数体） 4. 如果使用一个typedef之外的存储类型说明符来声明聚合或者联合对象，则除了表示连接的属性（那不就是去除extern和static么）其他属性应当作用于对象的成员。 |  |
| Type specifiers | 约束 | 1. 一个声明**（done）**，以及struct和type\_name的spec\_qual\_list中，最少需要使用一个Type specifiers。然后spec使用char、uschar等等那么一大堆里面的一个，以及顺序随意，还可以混用其他的（总之就是折腾编译器玩家呗） 2. \_Complex 在实现不支持的情况下不应当出现。（虽然我也懒得做，但是还是支持一下下吧，当然，这一定给选项）**（done）** |  |
| 语义 | 1. （没啥用，指明一下相关说明位置）**（done）** 2. 那一大堆类型限定符，每一行表示的是同一个意思。以及Int到底是signed int还是us int的随编译器定**（done）** |  |
| Structure and union specifiers | 约束 | 1. 如果一个struct或者union的声明，不是匿名结构，那么需要包含一个struct-declaration-list（然而对于匿名结构体呢？目前我不太清楚，按照语法，即便是匿名结构，仍然需要包含，但是gcc并不这么做，如果直接声明一个没成员的匿名结构体，会有一个warning但是如果包含在某个结构体中——这也是匿名结构体通常做的，而该匿名结构中并不包含任何member，那么连warning都没有） 2. Struct或者union不应该包含一个不完整或者function类型的成员，换句话说，结构体不应该包含他自己，但是可以包含一个指向该结构体的指针。但这里有个例外（有一说一，这每个字我都认识，但是确实不明白这个例外是什么意思），除了具有多个命名成员的结构中的最后一个成员可能具有不完整的数组类型;这样的结构（以及任何可能以递归方式包含作为这种结构的成员的并集）不应是结构的成员或数组的元素。 3. 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 4. 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，并且atomic是否支持是由编译器决定的（那我不支持了） |  |
| 语义 | 1. Struct是存储按顺序分配的，而union是存储重叠分配的 2. Struct或者union关键字是用来指明是struct还是union的 3. Struct-declaration-list是在translation unit中实际声明了一个新的类型（而不是通常以为的带前面的tag，tag只是给予了这个新类型一个名字）。如果一个Struct-declaration-list不包含任何有名成员（通过直接定义或者通过子匿名结构体或联合定义的），那么行为未定义（好吧前面约束的1我补充的说的就是这件事，gcc是允许他过编译，但是为了语法的严谨，我拒绝这样）。这个类型的定义是到右大括号结束的。 4. 结构或者联合的成员可以具有除了可修改类型（variably modified type）以外的任何完整对象类型，或者声明为位域类型 5. （同样跟位域有关，感觉没啥用） 6. （针对具体实现的）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。 7. 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。 8. **未命名**的成员存在Structure specifier并且之后**没有tag**被称作匿名结构，匿名结构或者union的成员被看做上一层的结构或者union 的成员。如果上一层也是匿名的，那么用递归的方式去实现 9. 对于非位域的成员，根据其类型决定合适的对齐方式 10. 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 11. 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。 12. 在struct或者union的结尾可能存在未命名的填充 13. 就是前面约束里面的2的问题，这个就是柔性数组成员（在GCC中，这里面用一个长度为0的数组来代替）。具体内容如下，首先，这必须是最后一个成员，并且，这个struct必须存在大于一个的成员（就是柔性数组成员不能作为唯一一个成员），是一个不完整的数组定义，比如int a[]。其次，结构体的大小就是去掉了柔性数组成员的大小，所以init的时候初始化这个struct是不行的（应当使用malloc），否则会越界。然而当他试图访问这最后一个元素的时候，那么就和这个成员是个真实存在的成员一样。最后，如果这个元素中没有任何成员，那么试图访问或者获取地址，比如&或者a[0]这个行为是未定义的。   （关于这一点，建议详细看后面的examples） |  |
| Enumeration specifiers | 约束 | 1. 如果将某个枚举类型定义为某个值，那么必须是整数值以及可以是表达为int的常量。 |  |
| 语义 | 1. Enumerator的标识符必须是一个int，并且可以在任何允许出现的地方出现。如果有等于号，那么其值就是对应的int值，如果第一个没有等于号，那么从0开始，之后的没有等于号的，一直加1，数字可以重复，enumerator是作为enumeration的一个成员出现的。 2. 具体实现中，可以用char 有符号或者无符号的int来表示，并且可以推迟到在所有的枚举都枚举完了之后来决定，唯一的要求是，这个枚举的类型必须包含所有的值。（在实现中，我觉得char就算了，如果这样的话，这会做相当多的隐式类型转换，而既然在前面全都被设计为int型，那么何必麻烦的去实现别的呢？我认为设计为char确实可行，因为很多时候enum类型并不超过256，甚至128都少，但这通常可能被设计为一些单片机等需要一定性能约束的情形。显然mcc不是。对于是否设计为const类型，我认为是必要的，谁会改enum呢。） |  |
| Tags | 约束 | 1. 指定类型最多define一次 2. 如果声明了多次（当然是用同一个tag了），那么struct union和enum必须选择同一个类型 3. Enum加上identifier后面啥也不加，必须在他指明的类型决定了之后才可以（我的理解是，既然前面说了，枚举类型的右括号出来之后才能决定type到底是使用char还是int什么的，那么这句话也就是说，只有确定了存储长度之后才能确定这一点，但是先写enum identifier再声明该enum类型中有哪些，gcc中是能过编译的，但是不声明，则报错的是没定义存储类型，说明，是先做类型系统建立，然后再做的semantic，两遍遍历） |  |
| 语义 | 1. 同一个scope和tag声明表明的是同一个类型 2. 不同标记或者同一个标记在不同scope声明的是不同类型，任何没有tag的struct union，enum表示不同类型 3. （一堆有的没的） 4. 同上 5. 如果只有 struct identifier的声明，那么是一个不完全类型 6. 如果只有struct identifier或者enum identifier 后面没有列表什么的那么不算做重定义   （对于下面的第二个例子，我想了半天，并不明白这怎么个弄法，我实际代码试验了半天，没法在gcc中得到相同的结果，另外，tag都是同一个scope，何来的enclosing scope和inner scope的描述呢？） |  |
| Atomic type specifiers | 约束 | 1. 如果实现不支持，那么可以不做。（不做了） 2. 在atomic后面的type name不可以是array type, a function type,an atomic type, or a qualified type. |  |
| 语义 | 与原子类型关联的属性仅对作为左值的表达式有意义。如果\_Atomic关键字紧跟左括号，则将其解释为类型说明符（具有类型名称），而不是类型限定符。（这在语法分析阶段就由于不同的语法表示所区分开来了） |  |
| Type qualifiers | 约束 |  |  |
| 语义 |  |  |
| Formal definition of restrict | 约束 |  |  |
| 语义 |  |  |
| Function specifiers | 约束 |  |  |
| 语义 |  |  |
| Alignment specifier | 约束 |  |  |
| 语义 |  |  |
| Declarators | 约束 | 1. 每个declarator声明一个identifier，如果表达式中出现相同的函数或者对象，那么具有该声明相同的作用域，存储持续时间，以及类型 2. Full declarator，这是不属于另一个declarator的声明，一个Full declarator末位就是一个序列点。另外，定义了variably modified 类型，VLA（variably length array）就是该类型的，以及，派生出来的类型，都算作该类型 3. 下面都用T D1这种方式描述某个声明。 4. 如果D1是个identifier的形式，那么描述的T就是D1的类型 5. 如果是用(D)的方式进行的描述，那么就是D。但是如果是个复杂声明符，可能由于括号的结合作用发生改变 |  |
| 实现约束 | 正如5.2.4.1中所描述的那样，对于pointer，array和function声明符都有数量的要求 |  |
| Pointer declarators | 约束 | 无 |  |
| 语义 | 1. 形式都是一个\* type-qualifier-list opt D的形式。表达的意思是指针的类型是type-qualifier-list，这样一个类型的指针指向前面的T 2. 如果两个指针是可以兼容的，那么首先他们的qual是相同的，并且指向的类型是兼容的 3. 例子，就是const指针和指向const类型的指针 4. 例子，同样是在另一份文档中有所设计说明的。 |  |
| Array declarators | 约束 | （我必须先说两句，这个约束我并不是特别看得懂，如果你也看不懂，建议先看array类型的语义，这会给你更多例子和建议让你明白为什么是这样）   1. 方括号当中，有表达式或者\*或者qual类型以及static，如果分隔表达式，那么应当具有整数类型，如果是常量表达式，应当大于零，并且元素的类型不能是不完整或者函数类型（对于后者的函数类型，我必须说明一下，就是说，a[n](parameter-list),由于压栈的顺序——这个顺序在语义分析文档里面有所描述，显然，是以函数类型(parameter-list)为基本元素的一个array）。qual类型以及static应当仅仅出现在具有数组类型的函数参数的声明中，并且是在最外层的数组类型（这应当在function中被检查而不是array处） 2. 如果identifier声明为VM，那么这个标识符应当为普通标识符（就是不能是label name，tag，member），链接类型为no linkage，并且要么block scope要么function prototype scope。如果是static或者thread的存储类型则必定不能为一个VLA |  |
| 语义 | 1. 一个array类型是那四种，大概就是用方括号括起来的就算。 2. 如果array的长度未定义，那么这是一个incomplete类型。如果长度是个\*，而不是一个expression，那么这是一个VLA并且未定义长度，这种方式只能被用在声明或者函数参数作用域的类型名，这种array的类型是complete的。如果array的size是确定的，并且元素类型的大小也是确定的，那么就不是一个VLA，否则就是（VLA未必需要实现的支持） 3. 如果size是一个非const int的表达式，如果这件事情发生在function prototype scope，那么就好像这个size被用\*指针替代了一样，否则（这个非const int但不在函数参数作用域的情况），这个expression必须每次被计算的的时候都是得到一个大于零的值。每个实例，即便是VLA，也不会在他的生命周期内变更大小。如果size表达式是sizeof的一部分，并且更改size表达式并不会影响运算符结果，那么无需指定是否计算表达式大小。 4. 如果两个array类型是可比较的，那么他们的基础类型之间都是可以比较的，并且如果他们的spec的大小都存在且为常数是，他们的大小说明符的大小应该相同，如果不相同，并且仍然需要比较，那么是未定义的行为（那就判错吧） 5. 几个例子，比如int \*a[n]就是指向一个指针类型的array，该指针指向一个int。如果是int (\*a)[n]又是另一个故事了，这里的\*应用的是上面那个规则，表示的是array的一个维度。 6. 所有的variably modified type必须要么块作用域，要么function prototype scope，所以file作用域下的VLA是有错的。含有thread\_local static extern或者static的存储限定符不能指定VLA类型，但是static可以指定VM类型（也就是对一个VLA类型的指针）。VM类型不能是在struct union的成员 7. （我必须在这里说明，VM包含VLA和指向VLA的指针） |  |
| Function declarators (including prototypes) | 约束 | 1. 一个functiondeclaratory不应该返回一个function或者array类型的返回类型。（对于这两个返回值的说明，第一个function，就是说，int a(para)(para)，这样连着两个括号表示的参数列表是不允许的，第二个array，则是说，int a(para)[n]这样是不被允许的，至于为什么返回类型是function和array，请参照类型的压栈顺序。） 2. 在参数列表中，只有register可以是唯一能够使用的storage class，就是其他什么extern static typedef都不能存在。 3. 函数声明器中不属于该函数定义的标识符列表应为空（没懂）。 4. 调整后，作为函数定义一部分的函数声明程序中参数类型列表中的参数不应具有不完整的类型。 |  |
| 语义 | 1. 照例是形式说明。 2. Parameter type list指定了参数的类型，也许会声明参数的identifier 3. Array的调整：array of type应当调整为qualified pointer to type，其中的qualifier是在array的括号里面指定的，如果static存在，那么每次call这个function，实参应当提供对数组中第一个元素的访问并且，该数组元素至少应当与size表达式指定的元素数目相同或者更多。（代码生成阶段，但是我感觉没法检查） 4. Function的调整：如果一个参数的返回值是function return type，应当被调整为pointer to function return type 5. 三个点的情况说明 6. 只有一个void，当成没参数。 7. 如果参数名字既能够被当做typedef name，也能当做参数名，那么当做typedef name。 8. 如果function declarator不是那个函数定义的一部分，（就是只声明没有函数体咯）。参数可以有incomplete类型并且可以用【\*】来指定VLA类型。 9. Storage class specify在参数的specify中会被忽略（按照约束，register可以有），除非声明的参数是函数定义的参数类型列表的成员之一。（没懂后面半句话） 10. 对于identifier list的情况，（我从没有遇到过这样的情况，我也不敢说这么翻译准确）。空列表的情况，这翻译让我恼火，下面有对于旧式风格的具体描述自己去看吧。 11. Function 类型compatible的情况。（但是实际上，两个function类型的compatible，是在definition那边被比较并进行合并的） |  |
| Initialization | 约束 | 这个确实老长老长了。  1 任何初始化器都不应尝试为未包含在被初始化实体中的对象提供值。  2 待初始化实体的类型应为大小未知的数组或非可变长度数组类型的完整对象类型。  3 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。  4 如果标识符的声明具有块范围，并且标识符具有外部或内部链接，则声明不应具有标识符的初始值设定项。  5 如果designator具有以下形式  [ constant-expression ]  那么当前对象（定义如下）应具有数组类型，并且表达式应为整数常量表达式。如果数组的大小未知，则任何非负值都是有效的。  6 如果designator具有以下形式  . identifier  那么当前对象（定义在下面）应具有结构或联合类型，标识符应为该类型成员的名称。 |  |
| 语义 | Initialization有两个，一个是assignment-expression ，另一个是括号形式的。  结构体或者union的未命名成员不参与初始化。  如果具有auto存储持续时间的对象未显式初始化，则其值是不确定的。  如果具有static或thread存储持续时间的对象未显式初始化，则：  — 如果它具有指针类型，则将其初始化为空指针；  — 如果它具有算术类型，则将其初始化为（正或无符号）零；  — 如果是聚合，则每个成员都根据这些规则（递归地）初始化，并且任何填充都被初始化为零位；  — 如果是联合，则根据这些规则（递归）初始化第一个命名成员，并将任何填充初始化为零位；  （没必要考虑那么多，其实全都设为0就行了）  对于scalar类型的初始化，应该是一个单独的表达式，可以被括号括起来，对象的初始值是表达式的值（在转换之后，我的理解是，需要从表达式的值进行一定的转换），使用simple assignment转换相同的约束。——这部分去看后面表达式里面的simple assignment。将scalar的类型变为他声明的类型的unqualified版本  剩下的子条目用于处理拥有聚合类型或者union类型的情况。  对于具有automatic storage 持续时间的structure或者union对象的初始化要么是一个下面描述的一样的initializer list，要么是一个与该structure或者union相兼容的single expression类型。在后一种情况中，该对象，包括未命名的成员，的初始值是该表达式的值。  一个字符类型的array可以被初始化为一个character string literal或者utf-8 string literal，可以在一个括号中。包括终结的那个null，或者array长度未知的情况。连续的字符用于初始化这个array。  使用wchar\_t,char16\_t,char32\_t类型的元素被初始化为宽字符，其他和上面的描述差不多。  否则，对于一个聚合或者union对象的初始化应当放在一对括号包含起来的对元素或者成员的初始化列表中。  （我的理解是，用于区分带不带括号的array，在实现中，这两者需要都写一遍，因为文法树解析出来不一样，然后我又选择了分开处理。）  每个由括号括起来的初始化列表含有一个关联的Current object，如果没有用designation特别指定，那么这个当前对象的子对象按照以下规则顺序排列：array按照递增顺序，structure按照声明顺序，union中按照第一个有名成员（注意这一点）。相反的如果用一个designation指定某个子对象，那么接下来从下一个对象开始。  （如果一个子聚合没有相应的左右括号，那么子对象像通常那样进行初始化，但是并不是变成current object，current object仅仅指代由括号括起来的initializer list）  （对于union，下一个object 并不是union的下一个成员，而是包含union的下一个子对象）  对于designatior list，每个list都是从最近的括号对开始的那个current object，每个desiganator则对应的去修改当前object的内容。  初始化应当按顺序进行，如果对同一对象多次初始化，则覆盖前面的，被覆盖的那个就没有用了，所有没有显式初始化的都应当等同于那个对象含有static存储持续时间的隐式初始化。  如果聚合或者联合含有聚合或者联合的成员，那么需要递归的进行。如果子聚合类型的初始化是以左大括号开头，那么用对应的右括号括起来的部分进行初始化，否则，如果是子聚合类型，给他找足够的元素即可，如果是union，则用于第一个元素。剩下的元素用于初始化后续的部分。  如果大括号的部分，少于初始化需要的成员，那么剩下的部分应当隐式初始化，同样如同static存储持续 类型。  如果对于未定义长度的array进行初始化，它的长度和显式初始化的最大长度相同，并且在这个initializer list 的结尾变成completed  初始化列表的表达式是不确定的序列，因此它的副作用也是不确定的。（从我的实现角度，应当采用从前向后的方式）  （我一直没动这一块的原因在于初始化的结果该如何使用，我现在的想法是，对于数组和结构体之类的，分配一个对应大小的区域，按照4字节或者8字节（看平台位数）读取，如果不为零则生成代码填充。这样子，而这样做不影响初始化的内容，就其他编译器的实现来说，也是差不多的方法，在栈上分配对应的空间，直接push xxx子节，如果有初始化，那么再在某几个需要初始化的字节处填上位置，而非初始化的部分，比如auto，其本身只是移动了SP指针，不影响其原来的值，所以不能轻易的判断就是0） |  |

Attention:为了消除文法中的左递归问题，所以这里面很多的表达式节点都是一串一串的，而不是如同C文法那样只有两个操作数，请务必仔细检查。

|  |  |
| --- | --- |
| 表达式语法节点 | Lvalue，type等等属性约束和要求 |
| Primary expressions | Identifier：声明一个对象或者函数名称，前者是lvalue，后者是function designator  Constant：根据实际情况决定的（TODO）  string literal：是一个lvalue（TODO）  括号表达式：根据下一层的expression节点决定的  Genetic：其类型根据实际的表达式决定的（TODO） |
| Array subscripting | 两个参数一个要求pointer to complete object type（也就是array经过转换之后），另一个要求integer type（显然是整型提升之后）  附：一些实现和描述在另一篇文档中（是我考虑不周） |
| Function calls | 如果是function，被转换为point to returning type  如果存在函数原型，那么参数数目应当相同，并且每个参数都应当可以被转化为unqualified的原型参数类型  括号里面可以是empty，或者是由逗号分隔的表达式表明函数的参数  在准备调用函数阶段，该参数被计算，并且对应的参数被填充  对于函数参数表达式的值，如果有函数的返回值指针，则该值和类型为那个返回值，如果没有，则是void（我的实现中无需考虑这个，因为void也作为类型会被插入到类型链中）  如果没有原型函数，（就是那个激动人心的旧式风格的函数隐式声明）那么这个函数调用对所有的参数都实行了整型提升，float被提升为double。这被称作default argumentpromotions，如果参数数量不符合，在这种情况下未定义  如果有原型函数，但是有ellipsis，就是后面跟着三个点，或者参数在提升之后并不compatible，那么行为未定义，但是有例外情况。  例外情况如下，一个是参数提升后变成了signed integer类型，而要求的是unsigned的，并且是对应的类型，（显然他们的表示在两边都是有意义的）  另一个是都是对于qualified或者unqualified的指针，指向character类型（就是char和unsigned和signed char）或者void类型  没了  如果有原型，那么会自动进行类型的转换，转换成unqualified版本的对应类型（所以在函数声明中的那个qualified其实不需要考虑就是这个原因）  如果原型中有三个点，也就是ellipsis，那么这个参数转换会直到最后一个，并且，执行的是默认的参数转换（针对省略号多出来的那些参数，前面的还是一样）  其他隐式转换不被执行，特别的是，如果是旧式风格的情况（也就是没有原型啦）就不做参数数量和类型的比较。  函数定义的类型和表达式调用的类型不兼容，那么行为未定义（肯定报错咯，不然强转么？）  实际调用之前对function designator和参数计算的之后，这中间存在一个序列点。  参数的计算顺序应当是不确定的，当然实际上实现中按照传统cdecl的调用约定，那么应该是从右到左的入栈顺序，也应该按照这个顺序去计算。  C标准中的例子：  (\*pf[f1()]) (f2(), f3() + f4())  其中1234的function是任意顺序被计算的。  允许函数调用的递归。  附注：考虑到旧式风格的实际情况，需要在遇到没有函数原型的情况下，自己生成一个函数对象并插入到最上层的filescope中去（但是gcc应该是插入当前scope，但是一旦在后面遇到了定义，则需要被取代。通常返回值设为int，而参数则按照f（）这样无参数的情况进行设置。 |
| Structure and union members | Dot:第一个操作数需要是atomic，qualified或者无qualified的struct或者union，第二个操作数需要是成员名称。如果前者的表达式是一个lvalue，那么dot操作之后，仍然是一个lvalue。如果前者有qualified，那么整个表达式仍然具有相同的qualified类型。  ->：第一个操作数是上面提到的类型的指针，第二个一样。是一个lvalue，qualified同上。  操作一个atomic的struct或者union是未定义的行为（并非编译器定义，所以需要报warning）  一个特殊情况：common initial sequence（well，去Stack Overflow上面查资料看到了个非常有意思的问题，总之，C语言真神奇）emmm我看了一眼，总之，这应该跟struct的内存分布有关，而且和我现行的设计并不冲突，但是我不太明白下面两个例子。我不对此做更多说明是因为，虽然他提出了需要compatible的类型，但是貌似并没有需要检查的部分而纯粹是定义。 |
| Postfix increment and decrement operators | 操作数应当为atomic, qualified,和unqualified的real或者point类型，并且应当具有可修改的左值  该操作符表达式的结果是操作数的值，有个副作用是操作数中存放的值进行了相应的运算，这么说很讨厌对吧，b=1，a=b++，那么a是1，b是1+1=2。稍微对此有所了解的应该都清楚讲的是这件事情。  并且，表达式的值是在进行更新存储对象中的数值之前，就是所谓的先取值再加加。对于不确定顺序的函数调用，（我的理解是函数调用的参数中，使用了++操作符）++操作符视作单独的计算（我不知道怎么翻译，但是我的理解是在参数中的这个东西，++之后是一个序列点，而不同参数之间的顺序，似乎是编译器或者说压栈顺序所确定的。）  如果是用在一个atomic类型的对象上面的++，等同于memory\_order\_seq\_cst函数中的语义。  --操作符一样 |
| Compound literals | 在postfix 操作中的（type name）开头的就是compound literals，这个type name指定的类型应当为complete或者不确定大小的array，但是不能是vla。  对于initializer list的部分的约束也适用于这里。（反正我都用一个函数调用过去了）  前面部分给定了type，后面给定了值。最后变成了个未命名的对象  如果是一个未定义长度的array，那么他的长度由后面的init部分确定，并且当做确定长度的complete的array类型来对待。  所有情况下，都是一个lvalue。  对于这个未命名的对象所具有的作用域（因为没名字肯定没有linkage啊）在函数外则具有static的生存时间，否则具有和block一样的自动生存时间。  String literal和compound literal如果具有const-qualified types，那么不需要指定不同的存储对象，或者说，实现中完全可以让存储内容相同的部分共用存储空间。 |
| Prefix increment and decrement operators | 类型要求和上面的后缀的++ --相同  并且，这个的含义，我想不必多说，先加加再取值。  PS:是我太天真了，unary的++ --，在左值转换那边单独提取出来，我想并不是毫无道理的。自己琢磨，我也不知道这个左右值，反正我已经晕了。 |
| Address and indirection operators | \*的操作数应当具有指针类型  &的操作数有几种选择：function designator，经过[]或者\*之后的array类型，不能是bit field或者具有register的lvalue类型  &操作符相当于取地址，如果对一个type类型使用&操作符，那么结果相当于point to type，如果这是一个\*的结果，那么&和\*相当于相互抵消掉了。除非由于约束导致的\*操作符的结果不是一个lvalue，相似的，如果是[]那么也一样，相当于括号和&抵消，变成了arr首地址加常数。  \*相当于去掉了某个类型的pointer |
| Unary arithmetic operators | 对于unary的+和-，应当具有arithmetic的类型，如果是~，应当具有integer类型，！，具有scalar类型。  （这些是需要考虑**整型提升**的，不是别的promotion）  +：就是提升后的结果  -：负号的提升后的结果  ~：对于提升后的类型，每个bit都取反。  ！：如果为0，取不为零，不为零，取零，结果具有的类型是int。  这几个的结果有可能是一个const的 |
| The sizeof and \_Alignof operators | Sizeof不应当应用到一个具有function或者不完整类型的表达式上面，包括加了括号的时候，或者应用到一个bit-field的成员（关于这一点的实现上面，由于sizeof存在两条规则，需要针对性的进行修改。一条是针对unary的，另一条是针对type\_name的，对于后者，由于type\_name里面不可能存在点操作，所以不会发生bit-field的情况。对于function的判定很简单,对于complete的判定则调用函数就好了）  Alignof不应当应用到function或者不完整类型  Sizeof结果是一个integer，如果是一个vla，那么**需要加以计算**，此时不是一个常数，否则必须是一个const的integer  Alignof表示对齐因素，结果是一个integer const，如果应用到一个array，那么结果是成员类型的对齐  对于struct的sizeof，是考虑了尾部的padding的  Alloc函数分配的时候，需要保证返回的指针是和需要分配的类型的align是对齐的,但这通常已经被考虑进去了，那是memory模块的事情。 |
| Cast operators | 显式类型转换前面的type\_name应当指明一个void，或者atomic，qualified，unqualified的scalar类型，操作数应当具有scalar类型  Point的转换除了在后面6.5.16.1有个约束的地方允许之外（letmesee），其他应当显式转换（我个人觉得这条不应该在这里存在任何操作，换句话说，如果没有显式转换，那么point除了后面的assign之外，不应当有任何其他的类型上面的改动。）  Pointer类型不应当被转换为任何floating类型，反过来也一样不行 |
| Multiplicative operators | 每个操作都应该有arithmetic type，%应当有integer 类型  操作数使用通用的arithmetic conversion |
| Additive operators | （貌似传说中的难点来了）  对于加法：两个操作数都应当具有arithmetic 类型或者一个具有pointcomplete object type，另一个具有integer类型  对于减法，下面的规则应当遵守  1、两个操作数都应当具有arithmetic 类型  2、两个操作数应当具有point to qualified或者unqualified版本的compatible complete object type（真复杂）  3、左操作数是一个指向完整对象类型的指针，右边是一个integer（所以其实加法是可以无所谓左右的，但是减法具有左右之分）  使用usual arithmetic conversions  如果**指针进行加减**一个integer，**结果具有指针类型**  （下面一大段指针加减的说明，但是我觉得没啥用，因为对于vla而言，运行时去检查是否溢出，对于编译阶段是困难的，所以他们应当不会被实现）  两个指针的减法的size是实现定义的（但我个人认为应当和数组加法的int操作数相同，我是说，这应当是一个逆运算），减法的结果应当具有ptrdiff\_t |
| Bitwise shift operators | 应当具有integer类型的操作数  需要做整型提升，**结果类型**和提升后的左操作数相同，对于右操作数为负数或者超过了提升后的类型宽度，行为未定义（我个人认为前者不合法，后者警告——PS：来考古，我觉得不合适，因为在编译期只能确定是一个int，却无法确定是否超过范围。） |
| Relational operators | 两个操作数都应当有real类型  或者都是point to qualified或者unqualified版本的compatible complete object type  如果两个操作数都是arithmetic type，那么通用的转换一样需要  当两个指针被比较的时候的一些equal和大小的判定，但是我觉得这和我们常用的指针比较没有什么太冲突或者罕见的情形，我就不说了  **返回类型**具有int类型，尽管他们是0或者1的结果 |
| Equality operators | 类型要求  都是arithmetic type  都是point to qualified或者unqualified版本的compatible complete object type  一个是一个对象类型的指针，另一个是void的qualified或者unqualified版本  一个是一个point，另一个是null pointer constant  （关于这个null pointer constant,要么是为0的常量，要么是这样的常量被转换到void\*去了，后者可以合并到前面的void里面去，所以直接检测另一个是不是一个常量并且是否为0即可，这里和其他地方的查看一个常量的情况一样，直接强制转换为longlong类型就好了）  **结果具有int类型**。  同样会实行通用的arithmetic conversions  复数的等于，只有实数和虚数部分都相同才可以等于  两个类型等同都要求在转换之后等同才行  如果至少一个是pointer，如果其中一个是一个pointer，而另一个是null pointer constant，那么null pointer constant会被转化为那个pointer所指向的类型。  如果一个是对象类型的指针，另一个是void的两个版本。那么前者会被转化为后者的指针类型（也就是带或者不带qualified的void指针）  关于指针相同，我看了那么一大段废话，归根结底，判等就是看地址就完了，哪怕两个指针在逻辑上没啥关系，比如一个访问array的指针，指向了越界之后的那个位置，正好另一个指针指向另一个对象的开头，而那个对象就接续在这个array的地址后面，这种情况下仍然会判定相等，加上附注的那个undefined行为，其实说到底，就是看指针的那个地址相等呗。  而考虑到结果的类型是确定为int的，那么他们的类型是否仍然需要被转换，我认为实现中没有必要  另外我必须说对于float转换之后的问题，这么说吧，我在gcc中测试  if(((float)2.2)==((double)2.2))  printf("true\n");  这一段代码，一定打印不出来，当然如果都是float或者double，则可以打印出来。Int的转换我不想说，但是浮点数上面，如果两个类型并不相同， |
| Bitwise AND operator | 操作数应当具有integer类型  通用类型转换依然需要进行  返回值我推定应当是integer类型（具体的应当是转换后的integer类型,有可能是long什么的） |
| Bitwise exclusive OR operator | 同上 |
| Bitwise inclusive OR operator | 同上 |
| Logical AND operator | 操作数应当具有scalar类型  返回类型应当具有int类型  从左至右计算，并且，在每个&&之间具有一个序列点，如果前面的不符合，后面的就不做计算了 |
| Logical OR operator | 同上，只不过序列点不同，只要有一个符合，后面的就不计算了 |
| Conditional operator | 这里的类型要求比较复杂，第一个操作数应当具有scalar type  第二个第三个操作数的要求如下，需要遵守其中一个  1都是arithmetic type  2都是same structure or union type  3都有void类型  4都是pointers to qualified or unqualified versions of compatible types  5一个是point另一个是null pointer constant  6一个是pointer to an object type，另一个是pointer tovoid的任意qualified版本  在第一个表达式和第二第三个之间具有序列点  表达式的结果是根据第一个表达式的结果选择的分支的结果值，对于类型的表述如下  如果两者都有arithmetic type，那么结果类型应当由usual arithmetic conversions决定，如果两个都有struct或者union，那么结果就是那个，如果都是void类型，那么就是void  如果都是pointer，或者其中一个是null pointer constant另一个是pointer，那么结果是一个指针，该指针具有两者的qualifiers。  另外，如果是指向compatible 类型，或者是具有不同qualified的compatible类型，那么结果是一个复合类型，如果一个是null pointer constant，结果类型就是另一个操作数的类型，如果是指向void的，那么结果是对应的qualified的void。 |
| Assignment operators | Assignment的左操作数一定是一个modifiable lvalue。  该表达式的结果就是在assign之后的左操作数的值，但这个结果不是一个lvalue，该表达式的结果类型是左操作数在lvalue转换之后的类型。更新左操作数存储的值的副作用，是在左和右操作数都计算完成之后的。但是左右操作数的计算是没有序列顺序要求的（我个人的倾向应当是从右往左）还是举个例子吧，我网上看到的  int a=0；int b=（a=a++）  这里面从右往左，先算a++表达式，a++表达式的值是a的初始值也就是0，然后副作用是a变成了1，随后a++的表达式结果赋值给a，也就是0赋值给a，最后赋值表达式的值是赋值后的结果0，所以b为0  我对这部分的看法表示一定程度的存疑，因为很多时候C标准和C编译器标准，让人疑惑，尽管从某种角度来说，这有一些通用的约定和现行的实现。  因此，绝大多数情况下，我希望在这个编译器的代码中不要搞这些稀奇古怪的东西，会被打 |
| Simple assignment | 这个简单应该就是指一个等号的情况，没有附加任何别的运算。  对于两边的类型要求   1. 左边是atomic qualified或者unqualified的arithmetic type，右边是arithmetic 类型 2. 左右是相容的atomic qualified，unqualified的struct或者union 3. 左边具有atomic qualified或者unqualified指针类型，经过左值转化后，两个操作数都是指向atomic qualified或者unqualified的相容类型，并且指向左边的具有右边所有的qualifiers 4. 左边具有atomic qualified或者unqualified指针类型，其中一个操作数具有指向一个对象的指针而另一个是各种版本的void指针，对于qualified的要求仍然同上 5. 左边具有atomic qualified或者unqualified指针类型，右边是一个null pointer constant类型。 6. 左边具有atomic qualified或者unqualified的\_Bool类型，右边是一个指针。 |
| Compound assignment | 对于+=和-=，左操作数要么是atomic, qualified, 或者unqualified pointer to a complete object type，右边是个integer类型，要么左边是atomic, qualified, or unqualified arithmetic type，右边具有arithmetic type  对于其他操作数，左边必须是atomic, qualified, 或者unqualified pointer to a complete object type，右边和对应的二元操作符相同。  或者更确切的按照语义的规定，对于E1 op= E2，直接调用E1 = E1 op (E2)，这看上去也是个不错的选择。  如果左操作数具有atomic类型，那么复合操作应当和前面说的memory\_order\_seq\_cst语义相同  （对于这部分，我直接在parser阶段就将语法树转换成为E1 = E1 op (E2)这种形式 |
| Comma operator | 左操作符应当被当做void表达式（不太明白），两个表达式中间具有序列点。  逗号运算符不会产生左值。  整个表达式的值是后面的表达式的值。 |

语句的语义分析我不想写，原文就那么一点点。（虽然变成代码数量也不少，基本都是test）

Externaldefinition部分

* Functiondefinition

这部分需要和后面的旧式风格结合起来一起看。如果只是新式风格的C编译器，这检查并不算特别麻烦。

1/声明的identifier需要有function类型 **（done）**

2/返回类型可以是void或者完整的类型，但不能是array **（done）**

3/存储限定符，要么static要么external **（done）**

4/如果有参数列表，那么除非是void，否则都需要一个identifier **（done）**

5/如果是identifier列表，也就是旧式风格的情况，每个declaration都必须有一个declarator，并且只能声明属于identifier列表里面的identifier，而且每个identifier都应当被声明一遍（换句话说不重不漏），declaration列表里面的storage要么是no要么是register，并且不应该存在初始化（无初始化的部分在parser里面做了）**（done）**

6/对于旧式风格的参数，应当像函数声明里面那样进行一定的参数类型转化——这个请看前面的部分**（done）**

7/对于三个点的情况，没做规定**（done）**

8/每个参数具有auto的存储类型，并且identifier是一个lvalue，但是未指定相关的内存布局什么的**（done）**

9/在进入函数时，每个可变修改参数的大小表达式都被计算，每个参数表达式的值都被转换为相应参数的类型，就像通过赋值一样。（作为参数的数组表达式和函数指示符在调用之前被转换为指针。）**（done）**

10/当上述转变完成后，开始执行compoundstatement部分，——但是我觉得这和前面函数调用有关系**（done）**

11/声明和定义的类型检查以及复合类型的构建，如果有函数原型，那么需要进行类型的检查——原本这应该是在declarator就完成的部分，当识别了这个函数identifier的时候，就完成了重复的检查和相容function类型的composite的。然而，由于函数作用域的处理问题，这又变得复杂起来。

如果在同一个作用域中，检查到一个函数的identifier，那么必然会调用declarator里面对于compatible的检查（虽然我还没写）但是现在是在下一层的作用域先完成了这个函数declarator的工作，然后再插入到上一层去，那么这里面需要重新完成declarator里面的一些检查工作，来查看是否和之前的声明或者定义存在冲突。

去语义分析里面查看一下，function scope 里面啥也不应该有（实际上应该会有label什么的），参数应当放在下面一层的block scope，函数名称应当在这里被特殊处理到上一层去了。

（颇为头痛，但是不这么设计不行啊，咱也不知道，咱也不敢问，拉倒吧。我个人感觉另一种方式就是在处理函数那个括号内的时候就决定插入到哪个表里面去，上面的归上面，下面的归下面）**（done）**

* External object definition

其他

* 默认顺序

除了明确表明了运算顺序的运算符之外，对于没有明确表明顺序的情况，其他，我更加倾向于选择一个从左到右的运算顺序。

* 旧式风格function函数相关：

1. 旧式风格的function定义，这里面涉及到的两个地方，一个是在function declarator这里，括号里面只有identifier，另一个是在function definition这里，有个可选的declaration。一种就是下面的f(a,b)另一种就是括号内啥也不加也是旧式风格。

例如

|  |
| --- |
| Int f();  int f(a,b)  int a,b;  {  return a+b;  } |

注意不加任何东西的情况，（吓得我连夜跑路，啊不是，去检查了所有函数接口，不能直接一个括号，要加void），不加任何东西调用的时候不会检查数量报错，但加了void还传递参数进去会直接给个报错。（当然这都是在函数调用的时候的工作）

由于C11标准的文法中仍然保留了这两个节点的这种情况，因此，编译器应当而且必须支持这些，这让我非常的头痛——这种困难并不仅仅来自于标准C，而是来源于他没说的东西，尤其是旧式风格的adjustment这些调用约定。本来我不想做的，但是，emmm啊这，谁让他写在标准里面了呢？做不做的到是能力问题，做不做是态度问题（可是这种历史包袱让我简直不想活着了）。

对此，我先整理了我所能收集到的跟旧式风格相关的东西（我不保证全面）：

C 编译器对旧式风格的函数甚至连实参的个数都不进行检查。

在**试图调用**一个**尚未声明**的函数的时候，会被当做旧式风格的函数调用（实际上生成了一个默认的函数模板int f()给他）

实参提升，旧式风格的函数调用的时候，float传入会提升为double，低于int的int类型比如char什么的会提升为int。

返回值会被当成一个int类型（其实还是那个默认的函数模板问题）

抽象声明符不允许旧式风格的定义，当然这无关紧要，因为mcc的语法分析器会保证这一点

对于函数内定义function的linkage为extern的情况，在ucc的说明中我看到了这个奇怪的问题，由于其为extern，导致了不同编译器的不同行为，我在后面会说明对此的取舍问题，当然我选择支持gcc的做法。

另外就是函数声明和函数定义的不同，如果混用旧式风格和新式风格存在的问题。比如在a文件中用新式风格声明了接口，而在文件b中用旧式风格定义，显然a中对该函数的压栈会按照新式风格压栈，而b中执行则是按照旧式风格执行，这会存在问题。当然，反过来也一样。归根结底，这是在代码生成阶段，对于接受的数据的偏移值的问题。

兼容类型的情况。如果都是旧式风格，因为不检查参数，所以直接看返回类型，如果都是新式风格，这很好比较。如果混搭，情况复杂，但主要的原理就是，看看是否会因为实参提升，导致新旧类型混搭的情况下，所期望接受的类型长短不同，如果存在这样的问题，则不算做兼容类型

* 应该按照什么情况去实现呢？

问题来了，我在gcc上面实现了几个因为旧式函数隐式声明的问题导致的奇奇怪怪的错误，可是，我是应该给出算出来是错误的答案呢，还是不应该给出来呢？

从实现的方法而言，前者，意味着按照顺序，直接就生成代码或者中间表示了，所以按照顺序弄到正式的函数声明的地方的时候，栈已经压好了，那就报错了咯。后者，按照我这个设计，进行多趟处理，在隐式声明被正式的声明所顶替掉之后，再进行代码生成，所以不会给出错误的问题。因为虽然试图调用尚未声明的函数，但是代码生成的时候已经看到了他的声明。两者都是可以实现的。

我认为，就C标准而言，他应当希望的是按照旧式风格写的代码能够被兼容，C编译器应当支持一些历史包袱存在的情况，而不是为了正确的答案而忽略了这里面存在的约定，否则会出现一大堆的问题，因此，我认为，我也应该算出来是错的。

另外，我必须指出的是，非常可惜的是，对于ucc里面对于旧式风格的那个例子（有一说一，我只能发现很少的网上存在的关于这个东西的中文资料，其中甚至还有不少是从这里抄过来的），我却始终无法在我的gcc上面复现出来同样的答案就是3.0+3.0因为提升为double变成了2.125那个，结果变成了0.0000。我研究了之后，认为，大概出了一些变故。但这个例子本身应当算作前面所说的函数声明和定义的新旧风格混搭的情况，隐式声明了一个旧式风格的函数，而定义被当做新式风格去获取参数。

顺带一提，如果我在同一个文档里面在后面定义了function，虽然会有warning，但是并不影响结果的正确性，只有在不同文档里面定义了函数体，才会产生这个错误，我个人推测，这是由于后面的函数体进行了覆盖的结果