语义分析　实习指导

许畅 陈嘉 朱晓瑞

> Whatever you say it is, it isn't. ——Alfred Korzybski

除了词法和语法分析之外，编译器前端所要进行的另外一项工作就是对输入程序进行语义分析（semantic analysis 或 semantic elaboration）。

为什么要进行语义分析？原因其实很简单：一段语法上正确的源代码中仍然可能包含严重的逻辑错误，这些逻辑错误可能会对编译器后面阶段的工作产生影响。首先，我们在语法分析阶段所借助的理论工具是上下文无关文法，而从名字上就可以猜出来上下文无关文法没有办法处理一些与输入程序上下文有关的机制（例如，变量在使用之前是否已经被定义或者声明过了？一个函数内部定义的变量在另一个函数中是否允许使用？等等）。这些上下文相关内容都会在语义分析阶段得到处理，因此也有人将这一阶段叫做上下文相关分析（context-sensitive analysis）。其次，现代程序设计语言一般都会引入类型系统，很多语言甚至是强类型的。引入类型系统对于程序设计语言来讲好处多多，例如它可以提高代码在运行时刻的安全性、增强语言的表达力，还可以使编译器为其生成更高效的目标代码。对于一个具有类型系统的源语言来说，编译器必须要有能力检查输入程序中的各种行为是否是类型安全的，因为类型不安全的代码出现逻辑错误的可能性相当高。最后，为了使之后的阶段能够顺利进行，编译器在面对一段输入代码时不得不从语法之外的角度对其进行理解。比如说，假设输入程序中有一个变量或者函数 x，那么编译器必须要提前确定：

- 如果 x 是一个变量，那么变量 x 中存储的是什么内容？是一个整数值、一个浮点数值、一组整数值或是其他的某种自定义的结构？
- 如果 x 是一个变量，那么变量 x 在内存中要占用多少个字节的空间？
- 如果 x 是一个变量，那么变量 x 的值在程序的运行过程中会保留多长时间？什么时候应当创建这个 x，它又应该在何时消亡？
- 如果 x 是一个变量，那么谁该负责为 x 分配存储空间？是用户显式进行空间分配，还是由编译器生成专门的代码隐式地完成这件事？
- 如果 x 是一个函数，那么这个函数要返回什么样的值？x 要接受多少个参数，这些参数又都是什么？

以上这些与 x 有关的信息中，几乎所有的信息都无法在词法和语法分析的过程中获取到。换句话说，输入程序为编译器所提供的信息量要远远大于词法和语法分析能从中挖掘出来的信息量。

从编程实现的角度上看，语义分析可以作为编译器里单独的一个模块，也可以并入语法分析模块或者并入中间代码生成模块中。不过，由于其中牵扯到的内容较多而且较为繁杂，因此我们还是将语法分析作为一次单独的实习任务。在接下来的内容里，我们会先对语义分析所要用到的理论工具——属性文法做一些简要的介绍，之后是对 C--中的符号表和类型表示两大重点内容进行讨论，最后是保证你顺利完成本次实习的一些建议。

属性文法

在词法分析过程中，我们借助了正则文法；在语法分析过程中，我们借助了上下文无关文法；现在到了语义分析部分，不知道你有没有想过：为什么我们不能在文法体系中更上一层，采用比上下文无关文法表达力更强的上下文相关文法呢？

之所以不继续采用更强的文法，原因有两个：其一，识别一个输入是否符合某一上下文相关文法这个问题本身是 P-Space Complete 的，也就是说如果使用上下文相关文法那么基本上就不要指望我们的编译器能跑出结果来了；其二，编译器需要获取的很多信息很难使用上下文相关文法进行编码。这就迫使我们需要为语义分析寻找其他更实用的理论工具。

目前被广泛使用的用于进行语义分析的理论工具叫做属性文法（attribute grammar），它是由 Knuth 在 50 年代所提出的。别看“属性文法”这个名字好像很吓人，它的核心想法其实我们早已接触过了，那就是为上下文无关文法中的每一个终结符或者非终结符赋予一个或多个属性值。对于产生式 **A→ X1. . .Xn** 来说，在自底向上分析中 **X1. . .Xn** 的属性值是已知的，这样语义动作只会为 **A** 计算属性值；而在自顶向下分析中，**A** 的属性值是已知的，

在该产生式被应用之后才能够知道 **X1. . .Xn** 的属性值。终结符号的属性值通过词法可以得到，非终结符号的属性值通过产生式对应的语义动作来计算。

属性值可以分成不相交的两类：综合属性（synthesized attribute）和继承属性（inherited attribute）。在语法树中，一个结点的综合属性值是从其子结点的属性值计算出来的；而一个结点的继承属性值是由该结点兄弟结点和父结点的属性值计算出来的。如果对以一个文法 P，$\forall A \to X_1X_2 \dots X_n \in P$都有与之相关联的若干个属性定义规则，则称 P 为属性文法。如果属性文法 P 只包含综合属性而没有继承属性，则称 P 为 S-属性文法；如果每一个属性定义规则中的每一个属性要么是一个综合属性，要么是 X_j 的一个继承属性，并且该继承属性只依赖于 X1,X2…X_{j-1} 的属性和 A 的继承属性，则称 P 为 L-属性文法。

以属性文法为基础可以衍生出一种非常强大的翻译模式，称为语法制导翻译（Syntax-Directed Translation, SDT）。在 SDT 中，人们把属性文法中的属性定义规则用计算属性值的语义动作表示并用花括号{}括起来，它们可被插入到产生式右部的任何合适的位置上，这是一种语法分析和语义动作交错的表示法。事实上，我们在之前使用 Bison 时已经潜移默化地用到了属性文法和 SDT，因此这一部分内容接受起来应该是没有难度的。

符号表

符号表对于一个编译器的重要性无论如何强调都不过分。在编译过程中，编译器使用符号表来记录源程序中各种名字的特性信息。所谓“名字”包括: 程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等，所谓“特性信息”包括: 上述名字的种类、类型、维数、参数个数、数值及目标地址（存储单元地址）等。

符号表上的操作包括填表和查表两种。当分析到程序中的说明或定义语句时, 应将说明或定义的名字, 以及与之有关的特性信息填入符号表中，这便是填表操作。查表操作被使用得更为广泛，需要使用查表操作的情况有：填表前查表，检查在程序的同一作用域内名字是否重复定义；检查名字的种类是否与说明一致；对于那些类型要求更强的语言，要检查表达式中各变量的类型是否一致；生成目标指令时，要取得所需要的地址或者寄存器编号，等等。符号表的组织方式也有多种多样，你可以将程序中出现的所有符号组织成一张表，也可以将不同种类的符号组织成不同的表（例如，所有变量名组织成一张表，所有函数名组织成一张表，所有临时变量组织成一张表，所有结构体定义组织成一张表等等）；你可以针对每一个语句块、每一个结构体都新建一张表，也可以将所有语句块中出现的符号全部插入到同一张表；你的表可以仅支持插入不支持删除（此时如果要实现作用域的话需要将符号表组织成层次结构），也可以组织一张既可以插入又可以删除的、支持动态更新的表。不同的组织方式各有利弊，我们希望你在看完了这篇文章并经过自己的仔细思考之后自行权衡利弊，并做出你自己的决定。

符号表里应该填些什么？这个问题的答案取决于不同的程序设计语言的特性，更取决于编译器的设计者本身。换句话说：只要你觉得方便，随便你往符号表里塞任何内容！毕竟符号表归根结底就是为了我们编译器的书写方便而设置的。单就本次实习来看，对于变量你至少要记录变量名和变量类型，对于函数你至少要记录返回类型、参数个数以及参数类型。

符号表应该采用何种数据结构进行实现？这个问题同样没有一个统一的答案。不同的数据结构有不同的时间复杂度、空间复杂度以及编程复杂度，几种最常见的选择有：

- 线性链表

  符号表里所有的符号都用一条链表串起来,插入一个新的符号只需将这个符号放在链表的表头,时间效率为 O(1)；在链表里查找一个符号需要对其进行遍历，时间效率为 O(n)；删除一个符号只需要将这个符号从链表里摘下来，不过在摘之前由于我们必须要执行一次查找操作找到待删除的节点，因此时间效率也是 O(n)。

  链表的最大问题就在于它的查找和删除效率太低，一旦符号表中的符号数量增大之后，查表操作将变得十分耗时。不过，使用链表的好处也显而易见：它的结构简单、编程复杂度极低，可以被快速地、无错地实现。如果你事先能够确定表中的符号数目非常非常少(例如,在结构体的定义中或者在面向对象语言的一些短方法中)，那么链表也是一个非常不错的选择。

- 平衡二叉树

  相对于只能执行线性查找的链表而言，在平衡二叉树上进行的查找天生就是二分查找。在一个典型的平衡二叉树实现（例如 AVL 树、红黑树、伸展树等）上查找一个符号的时间效率为 O(logn)；插入一个符号相当于进行一次失败的查找找到待插入的位置，时间效率同样为 O(logn)；删除一个符号可能需要做更多的维护操作，但其时间效率仍然维持在 O(logn)级别。

平衡二叉树相对于其他数据结构而言具有很多优势，例如较高的搜索效率（在绝大多数应用中 O(logn)的搜索效率已经完全可以被接受了）以及较好的空间效率（它所占用的空间随树中节点的增长而增长，不像散列表那样每一张表都需要大量的空间占用）。平衡二叉树的缺点是编程复杂太高，成功写完并调试出一个能用的红黑树所需要的时间不下于你完成本次实习所需的时间。不过如果你真的想要使用红黑树，其实并不需要自己动手写，从其他的地方（例如，Linux 内核代码中）寻找一个别人写的红黑树一样是可行的。

- 散列表

散列表代表了一个数据结构可以达到的搜索效率的极致，一个好的散列表实现可以让插入、查找和删除的平均时间效率都达到 O(1)。同时，与红黑树等操作复杂的数据结构不同，散列表在代码实现上竟是如此令人惊讶地简单：申请一个大数组、计算一个散列函数的值、然后根据这个值将该符号放到数组相应下标的位置即可。对于符号表来说，一个最简单的 hash 函数只需要把符号名中的所有字符相加，然后对符号表的大小取模。你可以自己去寻找一些更好的 hash 函数，这里我们提供一个不错的选择，由 P.J.Weinberger 所提出：

```
unsignedinthash_pjw(char* name)
{
    unsignedint val = 0, i;
    for (; *name; ++name)
    {
        val = (val << 2) + *name;
        if (i = val & ~0x3fff) val = (val ^ (i >> 12)) & 0x3fff;
    }

    return val;
}
```

需要注意的是，该方法对符号表的大小有要求——必须是 2 的某个乘幂。这个乘幂到底是多少，留给需要采用这种方法的你在理解这段代码的含义之后自己思考。

如果出现冲突，则可以通过在相应数组元素下面挂一个链表的方式（称为 open hashing 或者 close addressing 方法，推荐使用）或者再次计算散列函数的值从而为当前符号寻找另一个槽的方式（称为 open addressing 或者 rehashing 方法）来解决。如果你还知道一些更加 fancy 的技术，例如 Multiplicative hash function 以及 Universal hash function，那将会使你的散列表的元素分布更加平均一些。

由于散列表无论在搜索效率还是在编程复杂度上的优异表现，它也成为了符号表的实现中最常被采用的数据结构。

- Multiset Discrimination

抱歉不知道应该怎么翻译这个算法名。虽然散列表的平均搜索效率很高，但在最坏情况下它会退化为 O(n)的线性查找，而且几乎任何确定的散列函数都存在某种最坏的输入。另外，散列表所要申请的内存空间往往要比输入程序中出现的所有符号的数量还要多，未免有些浪费——如果我们为输入程序中出现每一个符号都分配一个单独的编号和单独的空间，岂不是既省空间又不会出现冲突吗？

Multiset discrimination 所基于的就是这种想法。在词法分析部分，我们统计输入程序中出现的所有符号（包括变量名、函数名等等），然后把这些符号按照名字进行排序，最后开一张与符号总数一样大的散列表，某个符号的散列函数即为该符号在我们之前的排序里的序号。

支持多层作用域的符号表（选读）

如果你的编译器不打算对局部作用域提供支持，即所有变量的声明都是全局的，那么你可以跳过本节内容。否则，请考虑下面这段代码：

```
…
int f()
{
    int a, b, c;
    …
    a = a + b;
    if (b > 0)
    {
        int a = c * 2;
        b = b - a;
    }
    …
}
…
```

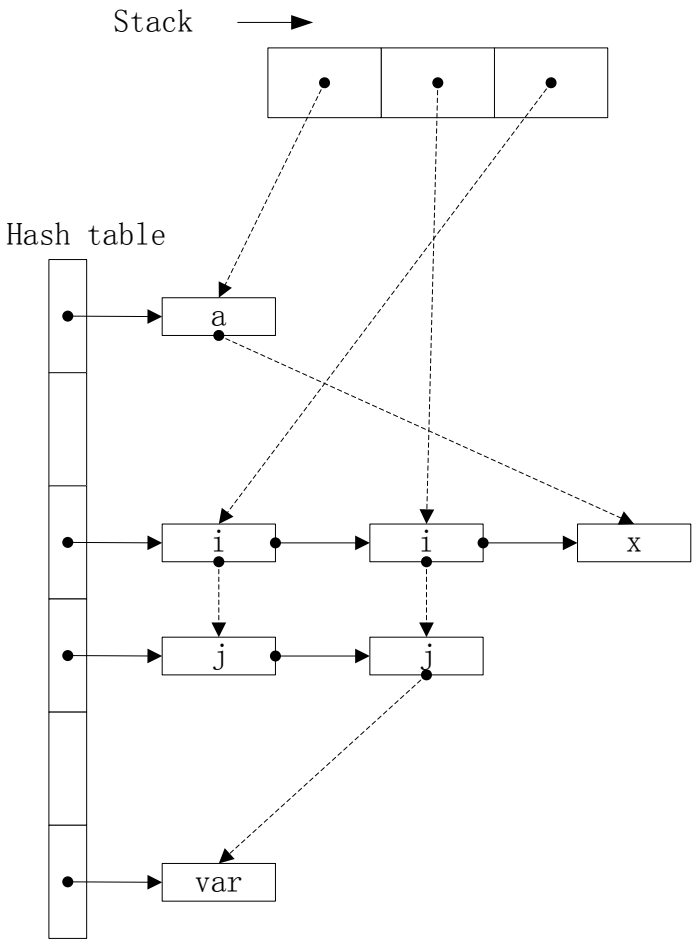
我们在函数 f 中定义了变量 a，在 if 语句中也定义了一个变量 a。如果要支持层次作用域，那么第一，编译器不能在 int a=c*2 这个地方报错；第二，语句 a=a+b 中的 a 的值应该取外层定义的 a 的值，语句 b=b-a 中的 a 的值应该是 if 语句内部定义的 a 的值，而这两个语句中 b 的值都应该取外层定义的 b 的值[1]。如何使得我们的符号表支持这样的行为呢？

第一种方法是维护一个符号表栈。假设当前函数 f 有一个符号表，表里有 a,b,c 这三个变量的定义。当编译器发现函数中出现了一个被{}包含的语句块（在 C--里就相当于发现了 CompSt 语法单元）时，它会将 f 的符号表压栈，然后新建一个符号表，这个符号表里只有变量 a 的定义。当语句块中出现的任何表达式中使用到某个变量时，编译器可以先查找当前的符号表，如果找到就使用这个符号表里的该变量，如果找不到则顺着符号表栈向下逐个符号表进行查找，使用第一个查找成功的符号表里的相应变量。如果查遍了所有符号表都找不到这个变量，则报告当前语句出现了变量未定义的错误。每当编译器离开某一个语句块时，会先销毁当前的符号表，然后从栈中弹一个符号表出来作为当前的符号表。这种符号表的维护风格被称为 functional style。

functional style 的维护风格最多会申请 d 个符号表，其中 d 为语句块的最大嵌套层数。这种风格比较适合于采用链表、红黑树的符号表实现。假如你的符号表采用散列表进行实现的话，申请多个符号表无疑会占用大量的空间。另一种维护风格称作 imperative style，它不会去申请多个符号表，而是自始至终在单个符号表为基础上进行动态维护。假设编译器在处理到当前函数 f 时符号表里有 a,b,c 这三个变量的定义。当编译器发现函数中出现了一个被{}包含的语句块，而在这个语句块中又有新的变量定义时，它会将这个变量插入 f 的符号表里。当语句块中出现的任何表达式使用到某个变量时，编译器查找 f 的符号表。如果查找失败，则报告一个变量未定义的错误；如果查表成功，则返回查到的变量定义；如果出现了变量既在外层又在内层被定义的情况，则要求符号表此时能够返回最近的那个定义。每当编译器离开某一个语句块时，会将这个语句块中定义的变量全部从表中删除。

imperative style 对符号表的数据结构有一定要求。下图是一个满足要求的基于十字链表和 open hashing 散列表的 imperative-style symbol table 设计：

[1]我们通常使用的程序设计语言（包括 C、C++以及 Java）其作用域规则都来源于 Algol，即内层的变量定义总会覆盖外层的变量定义。



这种设计的初衷很简单：除了散列表本身为了解决冲突问题所引入的链表之外，它从另一个维度引入链表将符号表中属于同一层作用域的所有变量都串起来。如上图，a、x 同属最外层定义的变量；i、j、var 同属中间一层定义的变量；i、j 同属最内层定义的变量，这两个变量和中间一层的 i、j 同名，因此被分配到了散列表的同一个槽里。每次向散列表中插入元素时，总是将新插入的元素放到该槽下挂的链表以及该层所对应的链表的表头。每次查表时如果定位到某个槽，则按顺序遍历这个槽下挂的链表并返回这个槽中符合条件的第一个变量，如此一来便保证了如果出现了变量既在外层又在内层被定义的情况，符号表能够返回最内层的那个定义（当然最内层的定义不一定在当前这一层，因此我们还需要符号表能够为每一个变量记录一个深度信息）。每次进入一个语句块，需要为这一层语句块新建一个链表用来串联这层中新定义的变量；每次离开一个语句块，需要顺着代表该层语句块的链表将所有本层定义的变量全部删除。

如何处理作用域的问题是语义分析的一大重点和难点。考虑到实现难度的缘故，我们的实习中并没有对作用域作过多的要求。不过现实世界中可不是这样的：想想 Perl 和 Common LISP 中的动态作用域应该如何实现？引入面向对象机制（尤其是单继承/多继承机制）以后每个类的成员变量的作用域应该如何实现？仔细思考你甚至会发现某些与作用域相关的问题甚至牵扯到代码生成与运行时刻环境！

类型表示

所谓“类型”需要包含两个要素：一组值，以及在这组值上的一系列操作。当我们在某组值上尝试去执行它所不支持的操作时，类型错误就产生了。一个典型程序设计语言的类型系统应该包含如下四个部分：

- 一组基本类型。在 C--语言中，基本类型包括 int 和 float。
- 从一组类型构造新类型的规则。在 C--语言中，可以通过定义数组和结构体构造新的类型。
- 判断两个类型是否等价的机制。在 C--语言中，我们默认要求你实现名等价，但是需要完成分组 2.3 的同学则需实现结构等价。
- 从变量的类型推断表达式类型的规则。

目前程序设计语言的类型系统分为两种：强类型系统（strongly typed system）和弱类型系统（weakly typed system）。前者在任何时候都不允许出现任何的类型错误，而后者可以允许某些类型错误出现在运行时刻。强类型系统的语言

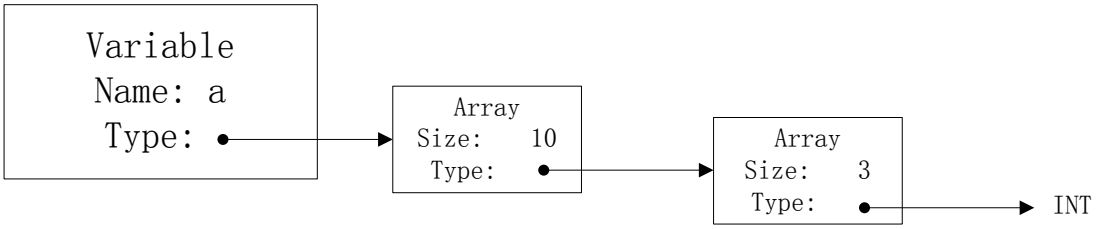
包括 Java、Python、LISP、Haskell 等，而弱类型系统的语言最典型的代表就是 C 和 C++语言[2]。

编译器尝试去发现输入程序中的类型错误的过程称作类型检查。根据进行检查的时刻之不同类型检查同样也可以被划分为两类：静态类型检查（static type checking）和动态类型检查（dynamic type checking）。前者仅在编译时刻进行类型检查，不会生成与类型检查有关的任何目标代码，而后者则需要生成额外的代码在运行时刻检查每一次操作的合法性。静态类型检查的好处是生成的目标代码效率高，缺点是粒度比较粗，某些运行时刻类型错误可能检查不出来；动态类型检查的好处是更加精确与全面，但由于在运行时执行了过多的检查和维护工作故目标代码的运行效率往往比不上静态类型检查。

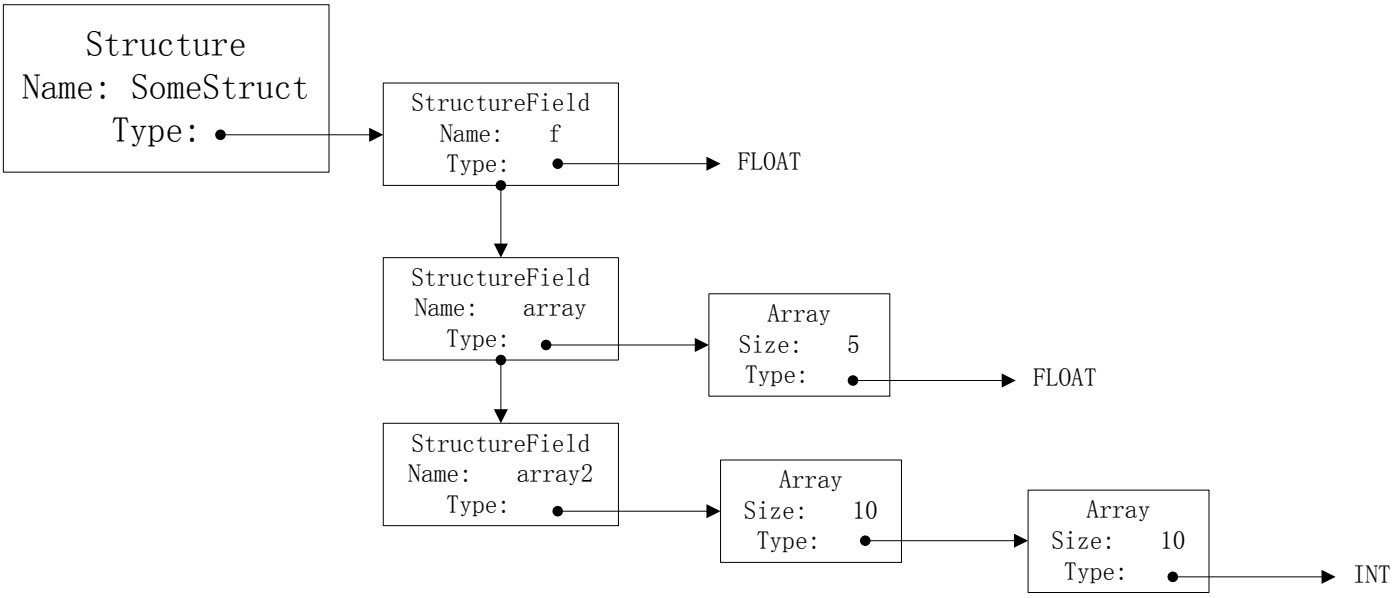
关于什么样的类型系统是好的，人们进行了长期的、激烈的而又没有结果的争论（这一争论被称作“Type War”）。动态类型检查语言更适合快速开发、构建程序原型（因为这类语言往往不需要指定变量的类型[3]），而使用静态类型检查语言写出来的程序常拥有更少的 bug（因为这类语言往往不允许多态）。强类型系统语言更加健壮，而弱类型系统语言更加高效。总之，不同的类型系统特点不一，目前还没有哪一种选择在所有情况下都比其它选择要来得更好。

介绍完了这么多基本概念，接下来我们就来考察实现上的问题。如果整个语言中只有基本类型，那么类型的表示将会极其简单：我们只需用不同的常数代表不同的类型即可。但是，在引入了数组（尤其是多维数组）以及结构体之后，类型的表示便不那么简单了。想像一下如果某个数组的每一个元素都是结构体类型，而这个结构体里又有某个域是多维数组，你该如何去表示它呢？

最简单的表示方法还是链表。多维数组的每一维都可以作为一个链表节点，每个链表节点存两个内容：数组元素的类型，以及数组的大小。例如，int a[10][3]可以表示为：



结构体同样也可以作为链表保存下来。例如，structSomeStruct { float f; float array[5]; int array2[10][10]; } 可以表示为：



具体地，在代码实现上，你可以如下定义 Type 结构来表示 C--语言中的类型（代码风格取自老虎书）：

[2]有关类型系统强弱的定义在不同的文献中似乎也不尽相同，例如你会听到另一种说法是强类型系统要求每一个变量在定义时都必须赋予一个类型，并且语言本身很少帮我们去做隐式类型转换。按照这种标准，C、C++语言就应该算是强类型语言，而那些类型系统比 C 还弱的像 Basic、JavaScript 才算是弱类型语言。

[3]对于那些对变量没有类型限制的语言，有一种生动形象的说法是这类语言采用了“鸭子类型系统”（duck typing）：如果一个东西走起来像一只鸭子、叫起来也像一只鸭子，那么它就是一只鸭子（if it walks like a duck and quacks like a duck, it’s a duck）。

```
typedef struct Type_* Type;
typedef struct FieldList_* FieldList;

struct Type_
{
    enum { BASIC, ARRAY, STRUCTURE } kind;
    union
    {
        // 基本类型
        int basic;
        // 数组类型信息包括元素类型与数组大小构成
        struct { Type elem; int size; } array;
        // 结构体类型信息是一个链表
        FieldList structure;
    } u;
};

struct FieldList_
{
    char* name;      // 域的名字
    Type type;       // 域的类型
    FieldList tail;  // 下一个域
};
```

最后再多说两句：同作用域一样，类型系统也是语义分析的一个很重要的组成部分。我们的 C--语言属于强类型系统（当然，之所以强还是因为我们的类型系统比较 naive，只有 2 种类型……），并且进行静态类型检查。当我们尝试着向 C--语言中添加更多的性质，例如引入指针、面向对象机制、显式/隐式类型转换、类型推断等时，你会发现编程的复杂程度会陡然上升。一个严谨的类型检查机制绝对不会像我们这样 ad hoc 地进行实现，而会通过将类型规则转化为形式系统，并在这个形式系统上面进行逻辑推理。同样，为了控制实习的难度我们大可以不必这样费事，但大家至少应该清楚真正比较实用的编译器内部类型检查决没有我们想象中的那么简单。

如何完成本次实习

本次实习需要你在上一次实习的基础上完成。更具体地说，需要在你上一次实习所构建出来的那棵语法树的基础上完成。这次你仍然需要对语法树进行遍历，但目的不是打印节点的名称，而是进行符号表的相关操作以及类型的构造与检查。你可以模仿 SDT 那样，在 Bison 代码中直接插入语义分析的代码，但我们更推荐的做法是 Bison 代码只用于构造语法树，把和语义分析相关的代码都放到一个单独的文件中去。另外，如果采用前一种做法，那么所有语法节点的属性值请尽量使用综合属性（尽管 Bison 提供了计算继承属性的机制，但不建议使用）；而如果采用后一种做法，就不会有这么多限制了。

每当遇到语法单元 ExtDef 或者 Def，就说明该节点的子节点们包含了变量或者函数的定义信息，这个时候应当将这些信息通过对子节点们的遍历提炼出来并插入到符号表里；每当遇到语法单元 Exp，说明该节点及其子节点会对变量或者函数进行使用，这个时候应当查表确认这些变量或者函数是否存在以及它们的类型都是什么。具体如何进行插入与查表，取决于你的符号表和类型系统的实现。本次实习要求检查的错误类型比较多，因此你的代码需要处理的内容也比较复杂，处理起来也比较繁琐，请一定仔细。另外还有一点值得注意：在发现一个语义错误之后一定不要立即退出程序，因为实习要求中有明确说明需要你的程序有能力查出输入程序中的多个错误。

必做部分总共有 17 种错误需要你检查出来，其中大部分也只涉及到查表与类型操作，不过有一个错误例外，那就是有关左值的错误。简单地说，左值代表地址，它可以出现在赋值号的左边或者右边；右值代表数值，它只能

出现在赋值号的右边。变量、数组访问以及结构体访问一般既有左值又有右值，但常数、表达式和函数调用一般只有右值没有左值。例如，赋值表达式 x = 3 是合法的，但表达式 3 = x 就是不合法的；y = x + 3 是合法的，但 x + 3 = y 是不合法的。简单起见，你可以只从语法的层面上来检查左值错误：赋值号左边能出现的只有 ID、Exp LB Exp RB 以及 Exp DOT ID，而不能是其它形式的语法单元组合。最后 5 种错误与结构体相关，结构体我们前面提到过，可以使用链表进行表示。

选做内容中分组 2.1 与函数声明相关，函数声明可能需要你在语法中添加一条产生式，并在符号表中记录每个函数当前的状态：是被实现了，还是只被声明未被实现？分组 2.2 涉及嵌套作用域，嵌套作用域的实现方法前文已经谈过了。分组 3.3 为实现结构等价，对于结构等价来说，你只需要在判断两个类型是否相等时不是直接去比较类型名而是针对结构体中的每个域逐个进行类型比较即可。

当然，如果你有其他的方法能够完成本次实习而且你的方法也能得到正确的结果，你大可以采用你自己的方法而不必遵循上面所介绍的思路。另外，实习时你也难免会因为碰到了一些解决不了的问题、查不出来的错误而心生惧意。不要怕！这是你应当经历的训练的一部分，请一定坚持下来。

祝你好运！

> If a program manipulates a large amount of data, it does so in a small number of ways.
>
> --Alan Perlis