3.2 实验指导

除了词法和语法分析之外，编译器前端所要进行的另一项工作就是对输入程序进行语义分析。进行语义分析的原因很简单：一段语法上正确的源代码仍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，从名字上就可以看出上下文无关文法没有办法处理一些与输入程序上下文相关的内容（例如变量在使用之前是否已经被定义过，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）。这些与上下文相关的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。引入类型系统可以为程序设计语言带来很多好处，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都是类型安全的，因为类型不安全的代码出现逻辑错误的可能性很高。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程序时不得不从语法之外的角度进行理解。比如，假设输入程序中有一个变量或函数x，那么编译器必须要提前确定：

1) 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、浮点数值，还是一组整数值或其它自定义结构的值?

2) 如果x是一个变量，那么变量x在内存中需要占用多少字节的空间?

3) 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x，而什么时候它又应该消亡?

4) 如果x是一个变量，那么谁该负责为x分配存储空间？是用户显式地进行空间分配，还是由编译器生成专门的代码来隐式地完成这件事?

5) 如果x是一个函数，那么这个函数要返回什么类型的值？它需要接受多少个参数，这些参数又都是什么类型?

以上这些与变量或函数x有关的信息中，几乎所有都无法在词法或语法分析过程中获得，即输入程序能为编译器提供的信息要远超过词法和语法分析能从中挖掘出的信息。

从编程实现的角度看，语义分析可以作为编译器里单独的一个模块，也可以并入前面的语法分析模块或者并入后面的中间代码生成模块。不过，由于其牵扯到的内容较多而且较为繁杂，我们还是将语义分析单独作为一块内容。我们下面先对语义分析所要用到的属性文法做简要介绍，然后对C−−语言编译中的符号表和类型表示这两大重点内容进行讨论，最后提出帮助

顺利完成实验二的一些建议。

3.2.1 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，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[1]的，也就是说，如果使用上下文相关文法那么编译器的复杂度会很高；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它更实用的理论工具。

目前被广泛使用的用于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。属性文法的核心思想是，为上下文无关文法中的每一个终结符或非终结符赋予一个或多个属性值。对于产生式$A \rightarrow X_1 \ldots X_n$来说，在自底向上分析中$X_1 \ldots X_n$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知道$X_1 \ldots X_n$的属性值。终结符号的属性值通过词法分析可以得到，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而来的，而一个结点的继承属性值则是由该结点的父结点和兄弟结点的属性值计算而来的。如果对一个文法P，$\forall A \rightarrow X_1 \ldots X_n \in P$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**。如果每个属性定义规则中的每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$X_1 \ldots X_{j-1}$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衍生出一种非常强大的翻译模式，我们称之为**语法制导翻译（Syntax-Directed Translation或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来表示，并用花括号“{”和“}”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时已经用到了属性文法和SDT。

[1] https://www.princeton.edu/~achaney/tmve/wiki100k/docs/PSPACE-complete.html。

3.2.2 符号表

符号表对于编译器至关重要。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、具体类型、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。查表操作则使用得更为广泛，需要使用查表操作的情况有：填表前查表，包括检查在输入程序的同一作用域内名字是否被重复定义，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，则要检查表达式中各变量的类型是否一致等；此外生成目标指令时，也需要查表以取得所需要的地址或者寄存器编号等。符号表的组织方式也有多种，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）。你可以针对每个语句块、每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中。符号表可以仅支持插入操作而不支持删除操作（此时如果要实现作用域则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，你可仔细思考并为实验二做出决定。

至于在符号表里应该填些什么，这与不同程序设计语言的特性相关，更取决于编译器的设计者本身。只要觉得方便，可以向符号表里填任何内容！毕竟符号表就是为了支持编写编译器而设置的。就实验二而言，对于变量至少要记录变量名及其类型，对于函数至少要记录其返回类型、参数个数以及参数类型。

至于符号表应该采用何种数据结构实现，这个问题同样没有统一的答案。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难度，我们下面讨论几种最常见的选择。

线性链表：

符号表里所有的符号（假设有n个，下同）都用一条链表串起来，插入一个新的符号只需将该符号放在链表的表头，其时间复杂度是$O(1)$。在链表中查找一个符号需要对其进行遍历，时间复杂度是$O(n)$。删除一个符号只需要将该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以找到待删除的结点，因此时间复杂度也是$O(n)$。

链表的最大问题是它的查找和删除效率太低，一旦符号表中的符号数量较大，查表操作将

变得十分耗时。不过，使用链表的好处也是显而易见：它的结构简单，编程容易，可以被快速实现。如果你事先能够确定表中的符号数目较少（例如，在结构体定义中或在面向对象语言的一些短方法中），链表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、红黑树或伸展树[1]等）上查找一个符号的时间复杂度是O(log*n*)。插入一个符号相当于进行一次失败的查找而找到待插入的位置，时间复杂度也是O(log*n*)。删除一个符号可能需要做更多的维护操作，但其时间复杂度仍然维持在O(log*n*)的级别。

平衡二叉树相对于其它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(log*n*)的搜索效率已经完全可以接受）以及较好的空间效率（它所占用的空间随树中结点的增多而增长，不像散列表那样每张表都需要大量的空间）。平衡二叉树的缺点是编程难度高，成功写完并调试出一个能用的红黑树所需要的时间不亚于你完成实验二所需的时间。不过如果你真的想要使用类似于红黑树的数据结构，也可以从其它地方（例如Linux内核代码中）寻找别人写好的红黑树源代码。

散列表：

散列表是一种可以达到搜索效率极致的数据结构。一个好的散列表实现可以让插入、查找和删除的平均时间复杂度都达到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也很简单：申请一个大数组，计算一个散列函数的值，然后根据该值将对应的符号放到数组相应下标的位置即可。对于符号表来说，一个最简单的散列函数（即hash函数）可以把符号名中的所有字符相加，然后对符号表的大小取模。你可以寻找更好的hash函数，这里我们提供一个不错的选择，由P.J. Weinberger[2]所提出：

```
 1  unsignedinthash_pjw(char* name)
 2  {
 3    unsignedint val = 0, i;
 4    for (; *name; ++name)
 5    {
 6      val = (val << 2) + *name;
 7      if (i = val & ~0x3fff) val = (val ^ (i >> 12)) & 0x3fff;
 8    }
 9    return val;
10  }
```

[1] 《数据结构与算法分析——C语言描述》，Mark Allen Weiss著，冯舜玺译，机械工业出版社，第80、351和89页，2004年。

[2] http://en.wikipedia.org/wiki/Peter_J._Weinberger。

需要注意的是，代码第7行的常数（0x3fff）确定了符号表的大小（即16384），用户可根据实际需要调整此常数以获得大小合适的符号表。如果散列表出现冲突，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close addressing[1]方法，推荐使用），或再次计算散列函数的值而为当前符号寻找另一个槽的方式（称为open addressing或者rehashing[2]方法）来解决。如果你还知道一些更酷的技术，如multiplicative hash function以及universal hash function[3]，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和编程难度上的优异表现，它已经成为符号表的实现中最常被采用的数据结构。

Multiset Discrimination：

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比输入程序中出现的所有符号的数量还要多，较为浪费。如果我们能只为输入程序中出现的每个符号单独分配一个编号和空间，那岂不是既省空间又不会有冲突吗？Multiset discrimination[4]就是基于这种想法。在词法分析部分，我们先统计输入程序中出现的所有符号（包括变量名、函数名等），然后把这些符号按照名字进行排序，最后申请一张与符号总数量一样大的符号表，查表功能可通过基于符号名的二分查找实现。

3.2.3 支持多层作用域的符号表

如果你的编译器不需要支持变量的作用域（即不需要实现要求2.2），那可以跳过本节内容，不会对实验二的完成产生负面的影响。否则，请考虑下面这段代码：

```
1  …
2  int f()
3  {
4    int a, b, c;
5    …
6    a = a + b;
7    if (b > 0)
8    {
9      int a = c * 2;
10     b = b - a;
11   }
12   …
13 }
```

[1] 《计算机程序设计艺术 第3卷 排序与查找》，Donald E. Knuth著，苏运霖译，国防工业出版社，第496页，2002年。

[2] 《计算机程序设计艺术 第3卷 排序与查找》，Donald E. Knuth著，苏运霖译，国防工业出版社，第501页，2002年。

[3] 《算法导论》，Thomas H. Corman等著，潘金贵、顾铁成、李成法和叶懋译，机械工业出版社，第138和139页，2007年。

[4] 《Engineering a Compiler》，第2版，Keith D. Cooper和Linda Torczon著，Morgan Kaufmann出版社，第256和751页，2011年。

14 …

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中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中b的值[1]。那么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“{”和“}”包含的语句块（在C--中就相当于发现了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。当语句块中出现任何表达式使用到某个变量时，编译器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。如果查遍所有的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**。该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或红黑树数据结构的符号表实现。假如你的符号表采用的是散列表数据结构，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**，它不会申请多个符号表，而是自始至终在单个符号表上进行动态维护。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“{”和“}”包含的语句块，而在这个语句块中又有新的变量定义时，它会将该变量插入f的符号表里。当语句块中出现任何表达式使用某个变量时，编译器就查找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，则要求符号表返回最近的那个定义。每当编译器离开某个语句块时，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的要求。图1是一个满足要求的基于十字链表和open hashing散列表的Imperative Style的符号表设计。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，它从另一维度也引入链表将符号表中属于同一层

[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。

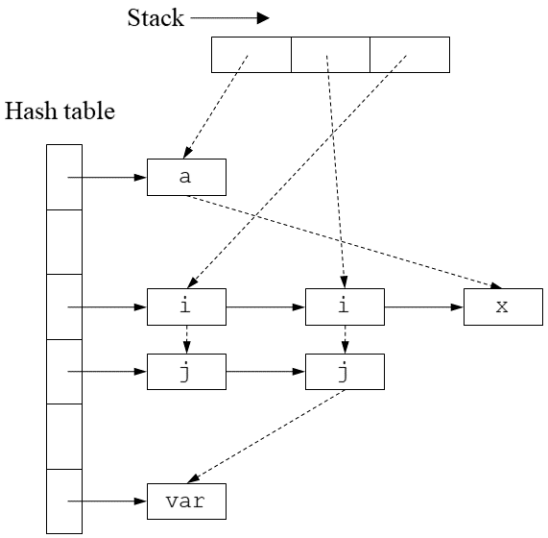

图1. 基于十字链表和open hashing散列表的符号表。

作用域的所有变量都串起来。在图中，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。其中i、j这两个变量有同名情况，被分配到散列表的同一个槽内。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，如此一来便可以保证：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该层中新定义的变量；每次离开一个语句块，则需要顺着代表该层语句块的链表将所有本层定义变量全部删除。

如何处理作用域是语义分析的一大重点也是难点。考虑到实现难度，实验二并没有对作用域作过多要求，但现实世界中的动态作用域将更难实现，某些与作用域相关的问题甚至涉及代码生成与运行时刻环境!

3.2.4 类型表示

“类型”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其不支持的操作时，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：

1) 一组基本类型。在C--语言中，基本类型包括int和float两种。

2) 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来构造新

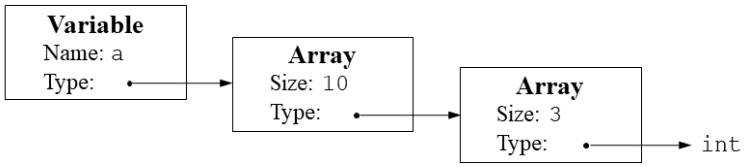

图2. 多维数组的链表表示示例。

的类型。

3) 判断两个类型是否等价的机制。在C−−语言中，默认要求实现名等价，如果你的程序需要完成要求2.3，则需实现结构等价。

4) 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[1]。

编译器尝试去发现输入程序中的类型错误的过程被称为是**类型检查**。根据进行检查的时刻的不同，类型检查可被划分为两类，即**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，某些运行时刻的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更好，人们进行了长期、激烈而又没有结果的争论。动态类型检查语言更适合快速开发和构建程序原型（因为这类语言往往不需要指定变量的类型[2]），而使用静态类型检查语言写出来的程序通常拥有更少的错误（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，目前还没有哪种选择在所有情况下都比其它选择来得更好。

[1] 有关类型系统强弱的定义在不同的文献中不尽相同，例如另一种说法是，强类型系统要求每个变量在定义时都必须赋予一个类型，并且语言本身很少做隐式类型转换。按照这种标准，C和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。

[2] 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）。

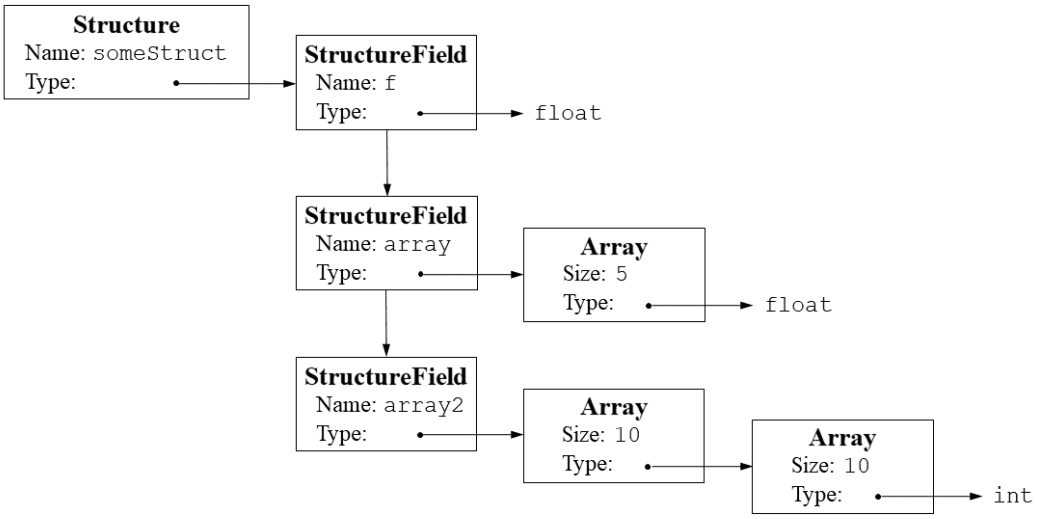

图3. 结构体的链表表示示例。

介绍完基本概念后，我们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，类型的表示就不那么简单了。想像一下如果某个数组的每一个元素都是结构体类型，而这个结构体中又有某个域是多维数组，那么该如何去表示呢？

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结点，每个链表结点存两个内容：数组元素的类型，以及数组的大小。例如，int a[10][3]可以表示为图2所示的形式。

结构体同样也可以使用链表保存。例如，结构体struct SomeStruct { float f; float array[5]; int array2[10][10]; }可以表示为图3所示的形式。

在代码实现上，你可以使用如下定义的Type结构来表示C--语言中的类型：

```
1  typedef struct Type_* Type;
2  typedef struct FieldList_* FieldList;
3
4  struct Type_
5  {
6    enum { BASIC, ARRAY, STRUCTURE } kind;
7    union
8    {
9      // 基本类型
10     int basic;
11     // 数组类型信息包括元素类型与数组大小构成
12     struct { Type elem; int size; } array;
13     // 结构体类型信息是一个链表
14     FieldList structure;
15   } u;
16 };
17
18 struct FieldList_
19 {
20   char* name;  // 域的名字
21   Type type;  // 域的类型
```

```
22    FieldList tail;  // 下一个域
23  };
```

同作用域一样，类型系统也是语义分析的一个重要的组成部分。C--语言属于强类型系统，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，你会发现实现编译器的复杂程度会陡然上升。一个严谨的类型检查机制需要通过将类型规则转化为形式系统，并在这个形式系统上进行逻辑推理。为了控制实验的难度我们可以无需这样费事，但应该清楚实用的编译器内部类型检查要复杂的多。

3.2.5 语义分析提示

实验二需要在实验一的基础上完成，特别是需要在实验一所构建的语法树上完成。实验二仍然需要对语法树进行遍历以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在Bison代码中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，而把和语义分析相关的代码都放到一个单独的文件中去。如果采用前一种做法，所有语法结点的属性值请尽量使用综合属性；如果采用后一种做法，就没有这些限制。

每当遇到语法单元ExtDef或者Def，就说明该结点的子结点们包含了变量或者函数的定义信息，这时候应当将这些信息通过对子结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，说明该结点及其子结点们会对变量或者函数进行使用，这个时候应当查符号表以确认这些变量或者函数是否存在以及它们的类型是什么。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。实验二要求检查的错误类型较多，因此你的代码需要处理的内容也较复杂，请仔细完成。还有一点值得注意，在发现一个语义错误之后不要立即退出程序，因为实验要求中有说明需要你的程序有能力查出输入程序中的多个错误。

实验要求的必做内容共有17种语义错误需要检查，大部分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而没有左值。例如，赋值表达式x = 3是合法的，但3 = x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，你可以只从语法层面来检查左值错误：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种语义错误都与结构体有关，结构体我们前面提到过，可以使用链表进行表示。

要求2.1与函数声明有关，函数声明需要你在语法中添加产生式，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。要求2.2涉及作用域，作用域的实现方法前文已经讨论过。要求2.3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。