实验指导

1. 如何完成本次实验

关于语义分析：本次实验需要你在上一次实验的基础上完成。更具体地说，需要在你上一次实验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。本次实验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，因为实验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。

总共有 17 种错误需要你检查出来，**1-12为必做，13-17(结构体部分)为选做**。注意：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。

关于IR表示：你可以在语义分析部分添加中间代码生成的内容，使编译器可以一边进行语义检查一边生成中间代码；也可以将中间代码生成的所有内容写到一个单独的文件里，等到语义检查全部完成并通过之后再生成中间代码。前者会让你的编译器效率高一些，后者会让你的编译器模块性更好一些。

确定了在哪里进行中间代码生成以后，下一步就要动手实现中间代码的数据结构（最好能写一系列可以直接生成一条中间代码的“构造函数”以简化后面的实现），然后再按照输出格式的要求自己编写函数将你的中间代码打印出来。接下来的任务是根据前文所给出的翻译模式完成一系列 translate 函数。下文中将给出 Exp 和 Stmt 应该如何翻译，你还需要自行考虑包括数组、数组与结构体定义、变量初始化、语法单元 CompSt、语法单元 StmtList 在内的翻译模式。

样例1为必做样例，样例2为选做样例。

2. 参考阅读：语义分析

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验任务。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 CMINUS 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验的一些建议。

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，$\forall A \rightarrow X_1X_2 \ldots X_n \in P$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。单就本次实验来看，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验所需的时间。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。

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 CMINUS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 CMINUS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 CMINUS语言中，我们默认要求你实现名等价。
- 从变量的类型推断表达式类型的规则。

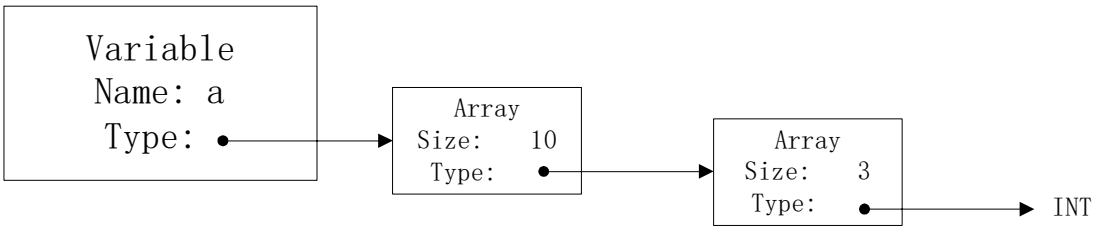
目前程序设计语言的类型系统分为两种：强类型系统（strongly typed system）和弱类型系统（weakly typed system）。前者在任何时候都不允许出现任何的类型错误，而后者可以允许某些类型错误出现在运行时刻。强类型系统的语言包括 Java、Python、LISP、Haskell 等，而弱类型系统的语言最典型的代表就是 C 和 C++语言[1]

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。

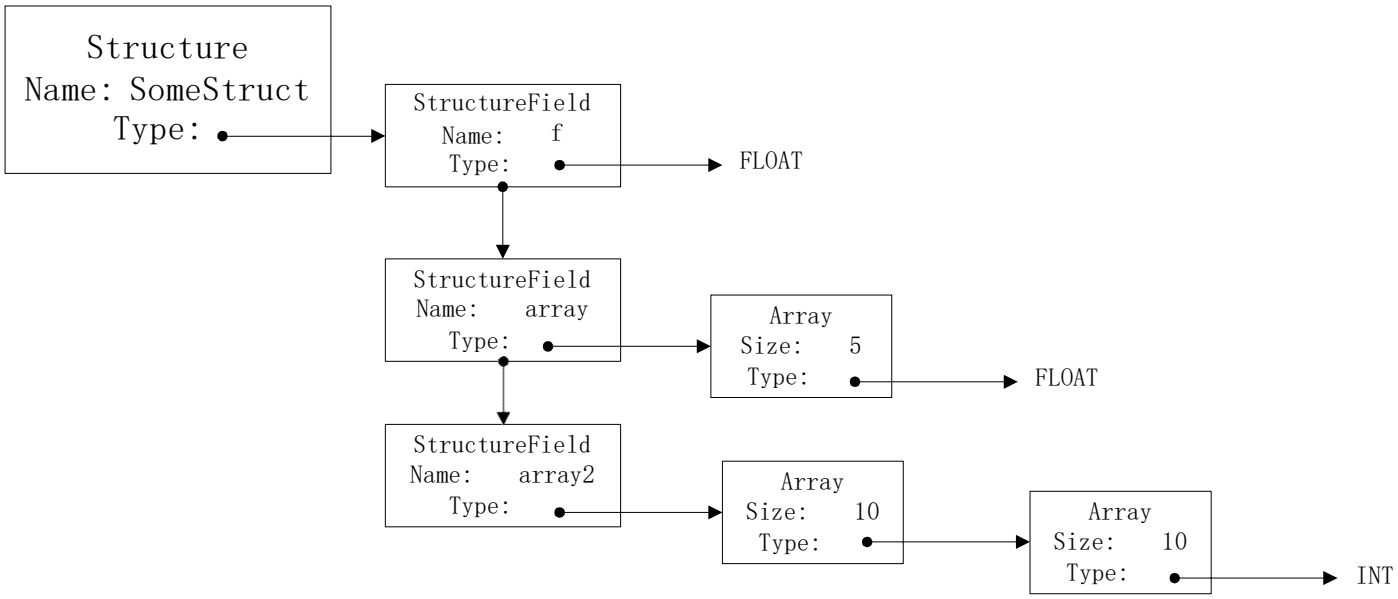
关于什么样的类型系统是好的，人们进行了长期的、激烈的而又没有结果的争论（这一争论被称作“Type War”）。动态类型检查语言更适合快速开发、构建程序原型（因为这类语言往往不需要指定变量的类型3），而使用静态类型检查语言写出来的程序常拥有更少的 bug（因为这类语言往往不允许多态）。强类型系统语言更加健壮，而弱类型系统语言更加高效。总之，不同的类型系统特点不一，目前还没有哪一种选择在所有情况下都比其它选择要来得更好。介绍完了这么多基本概念，接下来我们就来考察实现上的问题。如果整个语言中只有基本类型，那么类型的表示将会极其简单：我们只需用不同的常数代表不同的类型即可。但是，在引入了数组（尤其是多维数组）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 CMINUS语言中的类型（代码风格取自虎书）：

```
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
};
```

[1]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。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）

3. 参考阅读：中间代码生成

编译器里核心的数据结构之一就是中间代码（Intermediate representation，简称 IR）。中间代码应当包含哪些信息，这些信息又应当有怎样的内部表示将会极大地影响到编译器的代码复杂程度、编译器的运行效率以及编译出来的目标代码的运行效率。

广义地说，编译器中根据输入程序所构造出来的绝大多数结构都被称为中间代码（或者更精确地译作“中间表示”）。例如，我们之前所构造的词法流、语法树、带属性的语法树等等，都可以看作是一种中间代码。使用中间代码的主要原因当然是为了方便我们在编程时的各种操作，毕竟如果我们在需要有关输入程序的任何信息时都只能重新去读入并处理输入程序的源代码的话， 不仅会使得编译器的编写变得麻烦，也会大大减慢其运行效率。况且经过前面两次实验我们都应该清楚，输入程序里包含的绝大部分编译器需要的信息都没有被显式地表达出来。

狭义地说，中间代码是编译器从源语言到目标语言之间采用的一种过渡性质的代码形式（这时它常被称作 intermediate code）。看到这里你可能会产生疑问：为什么编译器不能直接把输入程序直接翻译成目标代码，而是要额外地多一道手续，引入中间代码呢？这难道不是自己给自己找麻烦吗？实际上，引入中间代码有两个主要的好处： 一方面，中间代码将编译器自然地分成了前端和后端两个部分。当我们需要改变编译器的源语言或者目标语言时， 如果采用中间代码，那么我们只需要替换原有编译器的前端或者后端，而不需要重写整个编译器。另一方面，即使源语言和目标语言是固定的，采用中间代码也有利于编译器的模块化。人们将编译器设计中的那些复杂但相关性又不是很大的任务分别放在前端和后端的各个模块之中，这样既简化了模块内部的处理，又使得我们能单独对每个模块进行调试与修改而不影响到其它模块。下文中，如果不是特别说明，那么所有出现的“中间代码”都指的是本段所介绍的狭义的中间代码。

中间代码的分类

中间代码的设计可以说更多的是一门艺术而不是技术。不同的编译器所使用的中间代码可以是千差万别，而就算是同一个编译器内部也可以使用多种不同的中间代码——有的中间代码与源语言更加接近，有的中间代码与目标语言更加接近。编译器需要在不同的中间代码之间进行转换，有时候为了处理的方便甚至会在将中间代码 1 转换为中间代码 2 之后，对中间代码 2 进行优化然后又转换回中间代码 1。这些不同的中间代码虽然对应了同一个输入程序，但它们却体现了输入程序不同层次上的细节信息。举一个实际的例子：gcc 内部首先会将输入程序转换成一棵抽象语法树，然后将这棵树转换成另一种被称为 GIMPLE 的树形结构，在 GIMPLE 之上建立静态单赋值（SSA）式的中间代码以后，又会将它转换为一种非常底层的 RTL（Register Transfer Language）代码，最后才把 RTL 变为汇编代码。

我们可以从不同的角度对现存的这些花样繁多的中间代码进行分类。从代码所体现出的细节上，我们可以将中间代码分为如下三类：

- 高层次中间代码（High-level IR, HIR）
  这种中间代码更多地体现了比较高层次的细节信息，因此往往和高级语言比较类似，保留了不少包括数组、循环在内的源语言的特征。高层次中间代码常在编译器的前端部分使用，并在之后被转换为更加低层次的中间代码。高层次中间代码常被用于进行相关性分析（dependence analysis)和解释执行。我们熟悉的 Java bytecode、Python .pyc bytecode 以及目前使用得非常广泛的 LLVM IR 都属于高层次 IR
- 中层次中间代码（Medium-level IR, MIR）
  这个层次的中间代码的形式介于源语言和目标语言之间。它既体现了许多高级语言的一般特性，又可以被方便地转换成低级语言的代码。正是由于 MIR 的这个特性，我们认为它是三种 IR 中最难设计的一种 IR。在这个层次上，变量和临时变量可能已经有了区分，控制流也可能已经被简化为无条件跳转、有条件跳转、函数调用和函数返回四种。另外， 对中层 次中间代码可以进行绝大部分优化处理， 例如公共子表达式消除（common-subexpression elimination)、代码移动（code motion)、代数运算简化（algebraic simplification）等。
- 低层次中间代码（Low-level IR, LIR）
  低层中间代码基本上与目标语言已经非常接近，它在变量的基础上可能会加入寄存器的细节信息。事实上，LIR 中的大部分代码和目标语言中的指令往往存在着一一对应的关系。即使没有对应，二者之间的转换也应该属于一趟指令选择就能完成的任务。RTL 就属于一种非常典型的低层次 IR。下图给出了一个完成相同功能的三种 IR 的例子（从左到右依次为 HIR、MIR 和 LIR)：

```
t1 = a[i][j+2]      t1 = j + 2          r1 = [fp - 4]
                    t2 = i * 20         r2 = r1 + 2
                    t3 = t1 + t2        r3 = [fp - 8]
                    t4 = 4 * t3         r4 = r3 * 20
                    t5 = addr a         r5 = r4 + r2
                    t6 = t5 + t4        r6 = 4 * r5
                    t7 = *t6            r7 = fp - 216
                                        f1 = [r7 + r6]
```

从表示方式来看，我们又可以将中间代码分成如下三类：

- 图形中间代码（Graphical IR）

  这种类型的中间代码将输入程序的信息嵌入到一张图中，以节点、边等元素来组织代码信息。由于要表示和处理一般的图代价稍大，人们经常会使用特殊的图，例如树或者有向无环图（DAG）来代替一般的图。

  一个典型的树形中间代码的例子就是抽象语法树（Abstract Syntax Tree，AST）。抽象语法树中省去了语法树里那些不必要的节点，将输入程序的语法信息以一种更加简洁的形式呈现出来，对于我们的处理来说是十分方便的。其它树形代码的例子有 GCC 里所使用的 GIMPLE。这种中间代码将各种操作都组织在一棵树里，在最后一次实验的指令选择部分我们会看到这种表示方式会简化其中的某些处理。

- 线形中间代码（Linear IR）

  线形结构的代码我们见得非常多，例如我们经常使用的 C 语言、Java 语言和汇编语言中语句和语句之间也就是一个线性关系。你可以将这种中间代码看成某种抽象计算机的一个简单的指令集。这种结构最大的优点是表示简单、处理高效，而缺点是代码和代码之间的先后关系有时会模糊整段程序的逻辑，让某些优化操作变得复杂。

- 混合型中间代码（Hybrid IR）

  顾名思义，混合型中间代码主要混合了图形和线形两种中间代码，期望结合这两种代码的优点、避免二者的缺点。例如，我们可以将中间代码组织成一个个基本块，块内部采用线形表示，块与块之间采用图表示，这样既可以简化块内部的数据流分析，又可以简化块与块之间的控制流分析。

本次实验中，你被要求按照格式输出一系列已经设计好了的内容。虽说实验要求中规定的这个代码格式类似于 线形的中层次中间代码，但不要据此就认为我们在限制你对于中间代码的选择——实验要求里仅仅是一个输出格式， 而你的编译器只要能按要求输出规定的内容即可。至于程序内部采用何种形式的中间代码，而这些中间代码中又体 现了多少细节信息，都完全取决于你自己的喜好。

刚拿到实验任务时，你可能会想要一边对语法树进行处理一边使用fprintf()函数把要输出的代码内容打印出来。这种做法其实并不好，因为当你将代码内容打印出来的那一刻你就已经失去了对这些代码进行进一步调整、优化的机会。更加合理的做法应该是将所生成的中间代码先保存到内存中，等到全部翻译完毕、优化也都做完再使用一个专门的打印函数把内存中的中间代码打印出来。

翻译模式——基本表达式

本次实验的任务说起来很简单：你只需要根据语法树产生出一段段的中间代码，然后将中间代码按照输出格式打印出来即可。中间代码如何表示以及如何打印我们都已经讨论过了，现在需要解决的问题是：如何将语法树变成中间代码呢？最简单也是最常用的方式仍是遍历语法树中的每一个节点，当发现语法树中有特定的结构出现时，就产生出相应的中间代码。和语义分析一样，中间代码的生成需要借助语法制导翻译。具体到代码上，我们可以为每一个主要的语法单元 X 都设计相应的翻译函数 translate_X，对语法树的遍历过程也就是这些函数之间的互相调用的过程。每一种特定的语法结构都对应了固定模式的翻译“模板”，下面我们将针对一些典型的语法树的结构的翻译“模板”进行说明，这些内容你也可以在教材上找到，理论课上应该也会讲。我们先从语言最基本的结构——表达式开始：下表列出了与表达式相关的一些结构的翻译模式。假设我们有函数 translate_Exp()，它接受三个参数：语法树的节点 Exp、符号表 sym_table 以及一个变量名 place，并返回一段语法树当前节点及其子孙节点对应的中间代码（或是一个指向存储中间代码内存区域的指针）。根据语法单元 Exp 所采用的产生式的不同，我们将生成不同的中间代码：

- 如果 Exp 产生了一个整数 INT，那么我们只需要为传入的 place 变量赋成前面加上一个‘#’ 的相应的数值即可。
- 如果 Exp 产生了一个标识符 ID，那么我们只需要为传入的 place 变量赋成 ID 对应的变量名（或者该变量对应到中间代码中的名字）即可。
- 如果 Exp 产生了赋值表达式 Exp_1 ASSIGNOP Exp_2，由于之前提到过作为左值的 Exp_1 只能是三种情况之一（单

个变量访问、数组元素访问或者是结构体特定域的访问），而对于数组和结构体的翻译模式我们将放在后文讨论，故这里仅列出当 Exp_1 -> ID 时应该如何进行翻译。我们需要通过查表找到 ID 对应的变量，再对 Exp_2 进行翻译（运算结果储存在临时变量 t1 中），最后将 t1 中的值赋于 ID 所对应的变量并将结果再存回 place，然后把刚翻译好的这两段代码合并随后返回即可。如果 Exp 产生了算数运算表达式 Exp_1 PLUS Exp_2，则先对 Exp_1 进行翻译（运算结果储存在临时变量 t1 中），再对 Exp_2 进行翻译（运算结果储存在临时变量 t2 中），最后生成一句中间代码 place := t1 + t2，并将刚翻译好的这三段代码合并随后返回即可。使用类似的翻译模式我们也可以对减法、乘法和除法表达式进行翻译。

- 如果 Exp 产生了取负表达式 MINUS Exp_1，则先对 Exp_1 进行翻译（运算结果储存在临时变量 t1 中），再生成一句中间代码 place := #0 - t1 从而对 t1 取负，最后将翻译好的这两段代码合并并返回。使用类似的翻译模式我们也可以对括号表达式进行翻译。
- 如果 Exp 产生了条件表达式（包括与或非运算以及比较运算的表达式），我们会调用 translate_Cond()函数进行（短路）翻译。如果条件表达式为真，那么为 place 赋值 1；否则，为其赋值 0。由于条件表达式的翻译可能和跳转语句有关，下表中并没有明确 translate_Cond()应该如何实现，这一点我们将放在后文介绍。

translate_Exp(Exp, sym_table, place) = case Exp of	
INT	value = get_value(**INT**) return [*place := #value*][4]
ID	variable = lookup(sym_table, **ID**) return [*place := variable.name*]
Exp_1 ASSIGNOP Exp_2[5] (Exp_1 → ID)	t1 = new_temp() variable = lookup(sym_table, **Exp_1.ID**) code1 = translate_Exp(**Exp_2**, sym_table, t1) code2 = [variable.name := *t1*] +[6] [*place* := variable.name] return code1 + code2
Exp_1 PLUS Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(**Exp_1**, sym_table, t1) code2 = translate_Exp(**Exp_2**, sym_table, t2) code3 = [*place := t1 + t2*] return code1 + code2 + code3
MINUS Exp_1	t1 = new_temp() code1 = translate_Exp(**Exp_1**, sym_table, t1) code2 = [*place := #0 - t1*] return code1 + code2
Exp_1 RELOP Exp_2 NOT Exp_1 Exp_1 AND Exp_2 Exp_1 OR Exp_2	label1 = new_label() label2 = new_label() code0 = [*place := #0*] code1 = translate_Cond(**Exp**, label1, label2, sym_table) code2 = [*LABEL label1*] + [*place := #1*] return code0 + code1 + code2 + [*LABEL label2*]

用方括号括起来的内容代表新建一条具体的中间代码，下同。
这里 Exp 的下标只是用来区分产生式 Exp→Exp ASSIGNOP Exp 中多次重复出现的 Exp，下同。
这里的加号相当于连接运算，意为将两段代码连接成一段，下同

翻译模式——语句

CMINUS的语句包括表达式语句、复合语句、返回语句、跳转语句和循环语句，它们的翻译模式如下表所示：

translate_Stmt(Stmt, sym_table) = case Stmt of	
Exp SEMI	return translate_Exp(**Exp**, sym_table, NULL)
CompSt	return translate_CompSt(**CompSt**, sym_table)
RETURN Exp SEMI	t1 = new_temp() code1 = translate_Exp(**Exp**, sym_table, t1) code2 = [*RETURN t1*] return code1 + code2
IF LP Exp RP Stmt$_1$	label1 = new_label() label2 = new_label() code1 = translate_Cond(**Exp**, label1, label2, sym_table) code2 = translate_Stmt(**Stmt$_1$**, sym_table) return code1 + [*LABEL label1*] + code2 + [*LABEL label2*]
IF LP Exp RP Stmt$_1$ ELSE Stmt$_2$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(**Exp**, label1, label2, sym_table) code2 = translate_Stmt(**Stmt$_1$**, sym_table) code3 = translate_Stmt(**Stmt$_2$**, sym_table) return code1 + [*LABEL label1*] + code2 + [*GOTO label3*] + [*LABEL label2*] + code3 + [*LABEL label3*]
WHILE LP Exp RP Stmt$_1$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(**Exp**, label2, label3, sym_table) code2 = translate_Stmt(**Stmt$_1$**, sym_table) return [*LABEL label1*] + code1 + [*LABEL label2*] + code2 + [*GOTO label1*] + [*LABEL label3*]

细心的你可能已经注意到了，无论是 if 语句还是 while 语句，上表中列出的翻译模式都不包含条件跳转。如果没有条件跳转，如何对 if 和 while 语句的条件表达式进行判断呢？其实我们可以在翻译条件表达式的同时生成这些条件跳转语句。translate_Cond()函数负责对条件表达式进行翻译，其翻译模式如下图。

对于条件表达式的翻译，理论课上可能会花比较长的时间进行介绍，尤其是与回填有关的内容更是重点。不过，无论你如何仔细地阅读下表，你仍然找不到和回填相关的任何内容。原因其实非常简单：下表中我们将跳转的两个目标 label_true 和 label_false 作为继承属性（函数参数）进行处理，这种情况下每当我们在条件表达式内部需要跳转到外面时，跳转目标都已经从父节点那里通过传参数得到了，直接填上即可。所谓回填，只用于将 label_true 和 label_false 作为综合属性处理的情况，注意这两种处理方式的区别。（你们可以采用回填的方式）

translate_Cond(Exp, label_true, label_false, sym_table) = case Exp of	
Exp_1 RELOP Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(**Exp_1**, sym_table, t1) code2 = translate_Exp(**Exp_2**, sym_table, t2) op = get_relop(**RELOP**); code3 = [*IF t1 op t2 GOTO label_true*] return code1 + code2 + code3 + [*GOTO label_false*]
NOT Exp_1	return translate_Cond(**Exp_1**, label_false, label_true, sym_table)
Exp_1 AND Exp_2	label1 = new_label() code1 = translate_Cond(**Exp_1**, label1, label_false, sym_table) code2 = translate_Cond(**Exp_2**, label_true, label_false, sym_table) return code1 + [*LABEL label1*] + code2
Exp_1 OR Exp_2	label1 = new_label() code1 = translate_Cond(**Exp_1**, label_true, label1, sym_table) code2 = translate_Cond(**Exp_2**, label_true, label_false, sym_table) return code1 + [*LABEL label1*] + code2
(other cases)	t1 = new_temp() code1 = translate_Exp(**Exp**, sym_table, t1) code2 = [*IF t1 != #0 GOTO label_true*] return code1 + code2 + [*GOTO label_false*]

翻译模式——函数调用

函数调用是由语法单元 Exp 推导而来的，因此为了翻译函数调用表达式我们需要继续完善 translate_Exp()：

translate_Exp(Exp, sym_table, place) = case Exp of	
ID LP RP	function = lookup(sym_table, **ID**) if (function.name == "read") return [*READ place*] return [*place := CALL function.name*]
ID LP Args RP	function = lookup(sym_table, **ID**) arg_list = NULL code1 = translate_Args(**Args**, sym_table, arg_list) if (function.name == "write") return code1 + [*WRITE arg_list[1]*] for i = 1 to length(arg_list) code2 = code2 + [ARG arg_list[i]] return code1 + code2 + [*place := CALL function.name*]

由于实验要求中规定了两个需要特殊对待的函数 read()和 write()，故当我们从符号表里找到 ID 对应的函数名时不能直接生成 CALL，而是应该判断函数名是否是 read 或者 write。对于那些非 read()和 write()的带参数的函数而言，我们还需要 translate_Args()函数将计算实参的代码翻译出来，并构造这些参数所对应的临时变量列表 arg_list。translate_Args()的实现如下：

translate_Args(Args, sym_table, arg_list) = case Args of	
Exp	t1 = new_temp() code1 = translate_Exp(**Exp**, sym_table, t1) arg_list = t1 + arg_list return code1
Exp COMMA $Args_1$	t1 = new_temp() code1 = translate_Exp(**Exp**, sym_table, t1) arg_list = t1 + arg_list code2 = translate_Args(**$Args_1$**, sym_table, arg_list) return code1 + code2

翻译模式——数组

CMINUS语言的数组实现采取的是最简单的 C 风格。数组以及结构体不同于一般变量的一点在于，访问某个数组元素或者访问结构体的某个域需要牵扯到内存地址的运算。以三维数组为例，假设有数组 int array[7][8][9]，为了访问数组元素 array[3][4][5]，我们首先需要找到三维数组 array 的首地址（直接对变量 array 取地址即可），然后找到二维数组 array[3]的首地址（array 的地址加上 3 乘以二维数组的大小 8×9 再乘以 int 类型的宽度 4），然后找到一维数组 array[3][4]的首地址（array[3]的地址加上4 乘以一维数组的大小9 再乘以int类型的宽度4），最后找到整数array[3][4][5]的地址（array[3][4]的地址加上 5 乘以 int 类型的宽度 4）。整个运算过程可以表示为：

$$\mathrm{ADDR(array[i][j][k])} = \mathrm{ADDR(array)} + \sum_{t=0}^{i-1} \mathrm{SIZEOF(array[t])} + \sum_{t=0}^{j-1} \mathrm{SIZEOF(array[i][t])} + \sum_{t=0}^{k-1} \mathrm{SIZEOF(array[i][j][t])}$$