第5章

语义分析与中间代码生成

高级程序语言中的语法定义了程序的构成。语法使用文法来描述，具有上下文无关性。程序只有语法正确还远远不够，还要语义正确才可以。语言中的语义是指约束规则，以确保程序无二义性，被翻译成目标代码之后，能在目标机器上正常运行。例如，对出现在数值运算表达式中的变量，必须事先定义，并且事先为其赋初值，这就是一条语义规则。程序必须遵循这条语义规则，否则在运行时会出现异常或错误。语义通常具有上下文相关性，例如变量要先定义后使用，函数调用要与函数定义相匹配。语义分析就是对输入的源程序进行检查，看它是否遵循了语义规则。如果没有遵循，就将其作为程序错误指出来，以便程序员对其进行改正。

编译的过程中，通常并不是将源代码直接翻译成目标机器代码，而是先将其翻译成中间代码，然后再将中间代码翻译成目标机器代码。中间代码是用中间语言写出的程序。可以将中间语言理解为逻辑计算机（也称概念计算机）的通用机器语言。这种处理带来的好处是：用任何一门高级程序语言编写的程序，都只需要经过两步编译，便能得到任何一种机型的可执行程序，使得高级程序语言与机器之间既彼此相互独立，又能对接组合。

高级程序语言的特点是通过引入新的数据类型和概念来实现更高级别的抽象，从而使得编程更加简单，程序更加简洁和健壮，程序更具可读性，更加通俗易懂。例如，面向对象语言中，通过引入类的概念、继承的概念、虚函数的概念，可以使得语言有更强的表达能力，更强的抽象能力，也可以使得程序具有更好的可重用性，更强的动态适应性。

中间语言的特点是面向机器计算，同时又抛开机器的物理特性，将机器抽象成由计算器和存储器两单元组成。在中间语言中，程序由代码和数据两部分组成，二者通常彼此分开存储。代码是一个指令序列。相对于高级程序语言，中间语言的一个显著差异是不再有名称概念，指令和数据都通过其逻辑地址来标识。目前使用广泛的中间语言有 LLVM 中的 IR（Intermediate Representation）、Java 中的字节码，以及.NET 中的 IL。中间代码生成要做的事情是，将用高级语言编写的程序翻译成用中间语言表达的程序。

从语法制导的翻译来看，语义分析和中间代码生成都是翻译目标，都是在语法分析的过程中附带完成的，在物理上并不构成独立的环节。因此本章内容就是针对翻译目标，针对典型的程序结构体来设计 SDT，定义数据结构，实现翻译函数。将语义分析和中间代码生成放到一起来处理，这是因为在语义分析中，中间代码的生成可以顺带完成。除此之外，在定义数据结构时，还考虑了程序调试的实现。

本章基于语法制导的翻译来讲解语义分析和中间代码生成的实现方法。5.1 节讲解语义分析和中间代码生成的具体含义和实施策略。5.2 节讲解类型和变量的语义分析框

架，5.3～5.5 节分别讲解类型和变量定义的 SDT 设计、变量使用的 SDT 设计，以及运算的语义分析和中间代码生成。5.6 节讲解类型系统。5.7 节和 5.8 节分别讲解分支语句的中间代码生成，以及函数调用的语义分析和中间代码生成。

5.1　语义分析和中间代码生成简介

为了让读者对语义分析和中间代码生成形成一个直观认识，现举例说明。代码 5.1 是一段 C++ 源程序。它首先定义了一个类 Student，然后定义了一个全局变量 p，接着实现了一个 main 函数。在 C++ 程序的函数实现中可随处定义变量，而且允许在不同层级中出现同名变量。代码 5.1 中就分别在第 5 行、第 11 行、第 15 行定义了变量 p。第 1 个 p 是全局变量，第 2 个 p 是函数中的局部变量，第 3 个 p 是函数内层中定义的局部变量。这种情形允许出现，因为它们分别在不同层级。

变量要先定义后使用，这是一条语义规则。对于同名变量，在使用时按照就近所指的语义规则来区分。也就是说，第 9 行中出现的 p 指第 5 行中定义的 p；第 12 行至第 14 行中出现的 p 指第 11 行中定义的 p；第 16 行中出现的 p 指第 15 行中定义的 p。对于类型也是如此。第 7 行可以看到类型 Student，于是先在 main 函数中检查是否有它的定义，若没有，则到上一层级去查找它的定义。由此可知，程序的层级结构是语义分析的基础。

代码 5.1　阐释语义分析和中间代码生成的 C++ 源程序示例

```
1    class Student {
2      char * name;
3      float tall;
4    }
5    int p = 2;
6    int main( )  {
7      Student * s1;
8      int a[20][40];
9      a[p][2] = 1;
10     s1 = new Student("Jim", 1.3);
11     char ** p;
12     p = & s1->name;
13     MyFun(a, p);
14     if(**p == 'J'&& s1->tall > 1.2)  {
15       float p = 0.3;
16       s1->tall += p;
17     }
18     int  i;
19     ...
20   }
```

5.1.1　程序的层级结构

从上述程序示例可知，程序具有层级结构，其基本单元为**块（block）**。块为语句序列，可以嵌套，因此整个程序为一棵**块树（block tree）**。既然是块树，自然就有**根块**、**子块**、**父块** 3 个

概念。以代码 5.1 为例,可以看出它包含 4 个语句块。首先是根块,也就是整个程序由 3 条语句构成。根块有 2 个子块。一个子块是 Student 类定义块,由第 2 行和第 3 行代码构成。另一个子块是 main 函数实现块,由第 7 行至第 19 行代码构成。main 函数实现块中也嵌有一个子块,该子块由第 15 行和第 16 行代码构成。根块是初始块。除了根块之外,其他块有明显的构成标识,即以左大括号开始,以与其配对的右大括号结束。

就语义分析而言,块有类别概念。块分为**根块**、**类定义块**、**函数实现块**,以及**被嵌块** 4 种类别。例如,代码 5.1 的第 2 行和第 3 行构成的块为类定义块,其中定义的变量为成员变量,定义的函数为成员函数;第 7 行至第 19 行所构成的块为函数实现块,其中定义的变量为局部变量;第 15 行和第 16 行构成的块为被嵌块。被嵌块中也可定义局部变量。根块中定义的变量为全局变量。

在每个块中,都可定义类型、变量、函数。为了语义分析,要标识出程序的块树,并记录每个块中定义的类型、变量、函数。可以给每个块设置唯一的序号来对其标识。根块为初始块,设其序号为 0。语义分析中有**当前块**概念。起始时,当前块为根块。随后每遇到新块标识(即左大括号)时,就创建一个新块,给其分配一个唯一的序号,以作标识。新块的父块为当前块。新块创建之后就成了当前块。随后当遇到块结束标识(即右大括号)时,当前块的父块又变回当前块。根据这一特性,需要构建一个**块栈(block stack)**来支持语义分析。

块栈中的元素为块。初始时,块栈中只有根块。语义分析过程中,栈顶元素为**当前块**。每创建一个新块,就将其压入块栈中,于是也就成了当前块。每遇到块结束标识时,就将栈顶元素从栈中弹出。于是被弹出块的父块又成了栈顶元素,自然也就成了当前块。从上述描述可知,块栈中相邻块之间在结构上呈父子关系。

以代码 5.1 为例,块以及块间父子关系的标识过程如下。初始时,块栈中只有根块,也为当前块。当语义分析到第 1 行末尾的左大括号时,创建一个块,设该块的 id 为 1,即块 1。块 1 的父块为块 0,即根块。将块 1 压入块栈后,块 1 成了当前块。当语义分析到第 4 行的右大括号时,块 1 结束。于是将块 1 从块栈中弹出,块 0 又成为了当前块。当语义分析到第 6 行末尾的左大括号时,又创建一个块,设该块的 id 为 2,即块 2。块 2 的父块为块 0。将块 2 压入块栈后,块 2 成了当前块。当语义分析到第 14 行末尾的左大括号时,又创建一个块,设该块的 id 为 3,即块 3。块 3 的父块为块 2。当语义分析到第 17 行末尾的右大括号时,块 3 结束。于是将块 3 从块栈中弹出,块 2 又成为了当前块。当语义分析到第 20 行末尾的右大括号时,块 2 结束。于是将块 2 从块栈中弹出,块 0 又成为了当前块。

每个块中定义的类型、变量、函数都要记录下来,用以支撑语义分析和中间代码生成。语法分析中每遇到类型定义,就将其添加到当前块的**类型表**中。每遇到变量的定义,就将其添加到当前块的**变量表**中。同理,每遇到函数的定义,就将其添加到当前块的**函数表**中。在一个块中,类型定义不允许出现重名,变量定义也是如此。函数则稍有不同,其标识信息除了名称之外,还有参数个数、参数类型、参数顺序。另外,在面向对象语言中,函数还可重载。

类型、变量、函数是语义分析中的核心内容,通常都有先定义后使用的语义规则。每遇到它们的使用,就要去查找其定义。以变量为例,首先是在当前块的变量表中查找。如果没有,就到当前块的父块中查找,若无则进一步往上追索,直至找到或者向上追索至根块。如果追溯至根块都未找到,则说明出现了变量未定义的语义错误。例如,语义分析至代码 5.1 第 9 行的变量 p 时,首先在当前块(即 main 函数实现块)的变量表中检查是否有其定义。发

现没有其定义，于是就到当前块的父块中去查找。当前块的父块为根块。在根块的变量表中有变量 p 的定义，于是此处的变量 p 是指全局变量 p。

依据上述分析，块应有如代码 5.2 所示的数据结构。其中字段 category 记录块的类别。块的类别有 4 种：根块、类型定义块、函数实现块、被嵌块，其取值分别用 ROOT、CLASS_DEF、FUNCTION_IMP、EMBEDDED 表示。category 取值的用意将在 5.1.3 节详解。一个类的详情表达在类型定义块中，包括成员变量和成员函数。一个类可继承其他类，有关继承的翻译和处理将在 5.8 节专题探讨。

字段 pTypeList、pVariableList、pFunctionList 分别记录块中定义的类型、变量、函数。字段 pFormalParamList 专门针对函数实现块而设置，记录形参。字段 srcRowIdFrom 和 srcRowIdTo 是为了程序调试目的而设置，记录块的开始行号和结束行号。程序调试时，在源程序中设置断点之后，基于行号就可计算出断点位于哪一个函数实现块中，便于内存数据块与源程序块的映射。字段 width 记录该块中所有变量的总宽度。这个字段在中间代码生成时用不到，只有在目标代码生成时才需要。总宽度表示所需内存空间大小，其使用将在后面讲解。

代码 5.2　块的数据结构定义

```
1  class Block  {
2     int blockId;                                  //块序号,起着标识块的作用
3     Block * pParentBlock;
4     BlockCategory category;
5     List<Type * > * pTypeList;                    //存储块中定义的类型;
6     List<Variable * > * pVariableList;            //对于类定义块:存储成员变量
                                                    //对于函数实现块:存储局部变量
7     List<Function * > * pFunctionList;            //存储块中定义的成员函数
8     List<Variable * > * pFormalParamList;         //仅只针对函数实现块:存储形参
9     int srcRowIdFrom;                             //从哪一行源程序起始
10     int srcRowIdTo;                              //到哪一行源程序结束
11     int width;
12  }
13  List<Block * > * pBlockList;
14  Stack <Block * > * pBlockStack;
```

语义分析中，每创建一个块，便将其添加到块表 pBlockList 中，于是块表中记录了程序的所有块。在中间语言中，对于类型、变量、函数，以及形参，不再有名称概念，其标识都使用逻辑地址。逻辑地址包括 3 部分：①类别符；②块 id；③行 id。例如，逻辑地址 m:1:2 指块 1 的变量表中第 2 行，该行记录了类定义中的一个成员变量。对于代码 5.1，就是 Student 类中的第 2 个成员变量 tall。

5.1.2　类型的语义分析

类型是语义分析的三大核心内容之一。常量和变量都有类型概念。函数作为变量的一个类别，同样有类型概念。类型有**基本类型**和**自定义类型**两种。在此基础上，还有**数组类型**和**指针类型**。基本类型有 int、float、char、bool 等。自定义类型指程序中定义的类型，例如代码 5.1 第 1 行定义的 Student 类就是自定义类型。语义分析中，当遇到源程序中出现类型

定义时,就将其记录到当前块的类型表中。例如,代码 5.1 开始处就定义了一个类 Student,于是就往根块的类型表中添加一行,记录 Student 类的定义。

类型表包含的字段如代码 5.3 所示。其中字段 typeId 是类型的标识字段,即类型 id,也称行 id。字段 name 专为基本类型和自定义类型而设置。只有这两种类型是通过名称来标识的。字段 category 记录类型的类别。类型的类别有 4 种:**基本类型(basic)、自定义类型(class)、数组类型(array)、指针类型(pointer)**。字段 pClassDefBlock 专为自定义类型而设置,存储其类型定义块的指针,以便通过它访问到类的成员变量和成员函数。字段 baseTypeAddr 专为数组类型和指针类型而设置。对于数组类型,baseTypeAddr 字段存储其元素的类型。对于指针类型,baseTypeAddr 字段存储所指对象的类型。字段 size 字段专为数组类型而设置,记录一维数组的元素个数。

代码 5.3　类型表的数据结构定义

```
1   class Type  {
2     int typeId;                    //类型 id
3     String name;                   //类型名称
4     TypeCategory category;         //取值有 BASIC、CLASS、ARRAY、POINTER
5     Block * pClassDefBlock;        //专门用于自定义类型
6     String baseTypeAddr;           //专门用于数组类型和指针类型
7     int size;                      //一维数组类型的元素个数
8     int srcRowId;                  //类型定义从源程序的哪一行开始
9     int width;                     //类型的宽度
10  }
```

数组类型和指针类型是两种特殊的类型,其定义可以迭代。例如,代码 5.1 第 8 行定义了一个二维数组 int a[20][40],可将其结构化成一个大小为 20 的一维数组,其元素的类型为另一个一维数组,即 int[40]。int[40]这个类型是一个大小为 40 的一维数组,其元素的类型为 int。对第 11 行定义的二阶指针 char **p,可将其结构化成一个一阶指针,其值所指对象的类型为另一个一阶指针,即 char*。char* 这个类型是一个一阶指针,其值所指对象的类型为 char。于是一个二维数组的定义包括两个一维数组类型的定义,一个二阶指针类型的定义也包括两个一阶指针类型的定义。

思考题 5-1　对于一个二维数组,在类型表中要添加两行,即定义两个一维数组类型对象。对于一个 n 维数组,是不是就会有 n 个一维数组类型对象的定义?对于一个二阶指针,在类型表中就要添加两行,即定义两个一阶指针类型对象。对于一个 n 阶指针,是不是就会有 n 个一阶指针类型对象的定义?

基本类型在任何地方都能使用,因此可将其定义记录在根块的类型表中,在编译初始化时就填好。如此处理之后,所有类型的定义便都记录到了类型表中。

就代码 5.1 而言,将其所涉及的类型定义汇集起来,如表 5.1 所示。头 3 行为基本类型 int、float、char 的定义,放在根块(块 id 为 0)的类型表中,设它们的 typeId 分别为 1、2、3,在初始时就添加在类型表中。第 4 行为 Student 类的定义,是在根块中定义的类型,因此其 blockId 为 0,其 typeId 字段值为 4。该行的 pClassDefBlock 字段值为块 1,表示其详细内容都记录在块 1 中。第 5 行是因源程序第 2 行的成员变量 name 的类型 char* 而添加。

表 5.1 中第 6 行因源程序第 7 行的局部变量 s1 的类型 Student* 而添加。在 main 函数

实现块(块 id 为 2) 中没有 Student 类的定义,于是就到其父块(块 id 为 0)的类型表查找,因此其 baseTypeAddr 字段值为 T:0:4,即 Student 类的逻辑地址,其中 T 表示类型表,0 为块 id,4 为行 id。第 7 行为 int[40]的一维数组形式定义。第 8 行为 int[20][40]的一维数组形式定义,其元素的类型为第 7 行定义的一维数组。第 9 行为 char* 类型的定义。尽管第 5 行也定义了 char* 类型,但它在块 1 中。块 1 不在块 2 至根块的块链中,因此对块 2 不可见。这就是第 9 行出现的缘由。第 10 行为 char** 的一阶指针形式定义,其值所指对象的类型,为第 9 行定义的内容。

表 5.1　类型定义的汇集表

行号	blockId	typeId	name	category	pClassDefBlock	baseTypeAddr	size	srcRowId	width
1	0	1	int	BASIC					4
2	0	2	float	BASIC					8
3	0	3	char	BASIC					1
4	0	4	Student	CLASS	1			1	12
5	1	1		POINTER		T:0:3		2	4
6	2	1		POINTER		T:0:4		7	4
7	2	2		ARRAY		T:0:1	40	8	160
8	2	3		ARRAY		T:2:2	20	8	3200
9	2	4		POINTER		T:0:3		11	4
10	2	5		POINTER		T:2:4		11	4

类型有宽度概念,单位为字节,用字段 width 记录。对于中间语言,有类型宽度概念,但没有具体值。其原因是中间语言是逻辑计算机的机器语言。只有在目标机器确定后,类型的宽度才确定,且为常量。因此字段 width 的值要等到将中间代码翻译成当目标代码时才来填写。例如,在 32 位机器中,int 类型的 width 为 4 字节,float 类型的 width 为 8 字节,char 类型的 width 为 1 字节,指针类型的 width 为 4 字节。自定义类型的 width 为其所有成员变量的 width 之和。因此表 5.1 中第 4 行记录的 Student 类,在 32 位机器上其 width 为 12。数组类型的 width 为其元素个数(即 size 字段值)与元素类型的 width 的乘积。于是在 32 位机器上,表 5.1 中第 8 行记录的 int[20][40]类型,其 width 为 3200。

在中间语言中,由于类型的 width 没有具体的值,因此使用宏定义来表示类型的宽度。例如,对于表 5.1 中第 8 行记录的数组类型,其宽度就用 WIDTH(T:2:3)表示,其中 T:2:3 为类型的逻辑地址,即块 2 的类型表中第 3 行。变量也有宽度概念,其 width 值就是其类型的 width 值。

在源程序中,数组类型和指针类型既可单独定义,也可将其和变量一同定义。代码 5.1 第 8 行中的 int a[20][40],其含义是:既定义了数组类型 int[20][40],又定义了该类型的变量 a。第 11 行的 char** p 也是如此,先定义一个二阶指针类型 char**,然后定义一个名称为 p 的变量,其类型为 char**。因此在类型表中,数组类型和指针类型可以没有名称。

在一个块的类型表中,自定义类型、数组类型、指针类型的标识字段各不相同。自定义

类型的标识字段为 name。数组类型和指针类型的定义可以没有名称。数组类型是通过其元素的类型以及元素的个数来标识的,因此在一个块中,数组类型是通过字段 baseTypeAddr 和 size 来标识的。一个指针类型可通过其所指对象的类型来标识。因此在一个块中,指针类型是通过字段 baseTypeAddr 来标识的。

类型表的字段 srcRowId 是为了程序调试目的而设置的,其含义是类型定义在源程序中所在的行号。程序调试时,通常既要了解一个变量的值,还要了解它的类型。程序中的任何一个变量都记录在其定义所在块的变量表中。变量表中有类型标识字段 typeAddr,基于该字段值就可以到类型表中查出其类型详情。再由类型表中的 srcRowId 字段值,就可以到源程序中定位其定义。

5.1.3 变量的语义分析

语义分析中,当遇到一个变量定义时,程序为其指定的类型一定要在类型表中存在。如果不存在,就表明源程序有类型未定义的语义错误;如果存在,就要将该变量的定义记录到当前块的变量表中。变量表的数据结构如代码 5.4 所示。其中字段 typeAddr 记录类型的逻辑地址,标识变量的类型。变量表用于变量的语义分析,对于变量,语义规则有如下 4 条:①变量要先定义后使用;②同一个块中不允许出现两个变量重名;③同名变量在使用时就近所指;④读取变量的值之前要先给变量赋值。块标识和块间父子关系标识为变量的语义分析提供了支持。

代码 5.4 变量表的数据结构定义

```
1  class Variable {
2    int variableId;              //变量序号: 标识变量
3    String name;                 //变量名称
4    String typeAddr;             //类型的逻辑地址
5    bool hasValue;               //已经赋值
6    bool used;                   //已经读取
7    int srcRowId;                //定义从源程序的哪一行开始
8    int offset;                  //相对地址
9  }
```

语义分析中,每遇到变量定义,便执行如下语义操作。首先检查其类型是否已定义,然后检查当前块的变量表中是否已经有该变量名的定义。如果有,就说明当前变量为重名变量,出现了语义错误;如果没有,就说明当前变量的定义合规,于是就往当前块的变量表中添加 1 行,记录该变量的定义。

语义分析中,每遇到变量使用,则执行变量是否已定义的语义检查。其方法是:到当前块的变量表中,基于变量名称检查是否有该变量的定义。如果有,就找到了它的定义,并能得到它的逻辑地址,语义分析就此完成;如果没有,那么就到当前块的父块中检查是否有该变量的定义。以此法则梯次向上追溯,直至找到,或者追溯至根块。如果最终在根块的变量表中都没有找到其定义,就表明源程序出现了变量未定义的语义错误。

找到变量的定义之后,如果是给变量赋值,那么就将其 hasValue 字段值设为 true;如果是读取变量的值,那么就要检查 hasValue 字段值是否为 true。如果不为 true,则表明源程序违背了变量要先赋值后读取的语义规则。读取之后,将其 used 字段设为 true,表示变量

在程序中被使用。当整个程序的语义分析完毕之后，如果一个变量的 used 字段值还为 false，则表明它在程序中未被使用，是多余可去掉的变量。

语义分析中，每当遇到变量使用时，就要生成中间代码。在中间语言中，变量不再有名称概念，所有变量都用其**逻辑地址**来标识。中间语言也称通用机器语言，其中的类型和变量有宽度概念，但是没有具体值，只有在目标机器（即物理机器）确定时才会有具体值。例如，int 类型的 width 在 16 位物理机上为 2 字节，在 32 位物理机上为 4 字节，而在 64 位物理机上则为 8 字节。正因为如此，中间语言中的变量只能用**逻辑地址**标识。一旦目标机器确定，基本类型的 width 便为常量，于是所有类型和变量的 width 也为常量，此时可将中间代码中变量的逻辑地址替换成**存储地址**。此时的存储地址为**相对地址**，也称**偏移量（offset）**，即变量表中的 offset 字段值。

往当前块的变量表中添加一个变量定义时，行 id（即 variableId 字段值）便标识了该变量。从全局来看，块 id 和行 id 便构成了变量的逻辑地址。例如，逻辑地址 1:2:1，指的是块 2 的变量表中第 1 行记录的变量。一旦目标机器确定，逻辑地址即被替换成相对地址，即 offset 字段值。

将代码 5.1 中 4 个块的变量表中的变量汇集到一起，如表 5.2 所示。其中有 8 个变量的定义，前两行为 Student 类型定义块中成员变量的定义；第 3 行为全局变量 p 的定义；第 4 行至第 7 行为块 2 中定义的 4 个局部变量；第 8 行为块 3 中定义的局部变量 p。字段 typeAddr 表示变量的类型，即类型的逻辑地址。

表 5.2　变量定义汇集表示例

blockId	VariableId	name	typeAddr	srcRowId	offset
1	1	name	T:1:1	2	0
1	2	tall	T:0:2	3	4
0	1	p	T:0:1	5	0
2	1	s_1	T:2:1	7	0
2	2	a	T:2:3	8	4
2	3	p	T:2:5	11	3204
2	4	i	T:0:1	18	3208
3	4	p	T:0:2	15	3208

对于函数实现块，其中定义的变量除了局部变量之外，还有形参。因此，在函数实现块中，除了有变量表之外，还要有**形参表**。形参表依次记录了每个形参的定义。不将形参放到变量表的原因是：形参是函数调用时从调用者传递给被调用者的数据，和变量表中的变量可能不在一个存储空间中。

变量表中 offset 字段的值只有在目标机器确定之后才能填写，也就是要后延到目标代码生成时才填写。填写方法如下。对于根块、类定义块、函数实现块，其变量表中的第 1 个变量，其 offset 字段值都为 0。对于第 2 个变量，其 offset 字段值的计算方法是，第 1 个变量的 offset 字段值加上第 1 个变量的 width 就是第 2 个变量的 offset 字段值。以此类推，第 i

个变量的 offset 字段值加上第 i 个变量的 width,就是第 $i+1$ 个变量的 offset 字段值。

现举例说明变量的 offset 字段值计算。设目标机器为 32 位机,于是 int 类型的 width 为 4,float 类型的 width 为 8,char 类型的 width 为 1,指针类型的 width 为 4。代码 5.1 中,main 函数实现块(块 2)中共定义了 4 个变量。第 1 个变量 s_1(逻辑地址为 1:2:1)的 offset 字段值为 0,因其类型为指针,width 为 4,于是第 2 个变量 a(逻辑地址为 1:2:2)的 offset 字段值为 4。变量 a 的 width 为 3200,于是第 3 个变量 p(逻辑地址为 1:2:3)的 offset 字段值为 3204。变量 p 的 width 为 4,于是第 4 个变量 i(逻辑地址为 1:2:4)的 offset 字段值为 3208,如表 5.2 所示。

对于函数实现块中的被嵌块,在创建后,其起始行的 variableId 字段值不能像其他类别的块那样设置为 1,而要初始化成其父块中已有变量的个数后再加上 1。例如,设代码 5.1 根块的 blockId 为 0,main 函数实现块的 blockId 为 2。当语义分析到第 14 行末尾的左大括号时,便要创建一个新块,其 blockId 为 3。此时块 2 的变量表中已有变量的个数为 3,于是块 3 中第一个变量的行 id(即 variableId 字段值)要初始化成 4。这就是第 15 行定义的变量 p 的逻辑地址为 1:3:4 的原因。

被嵌块中第 1 个变量的 offset 字段值计算依赖其行 id 以及被嵌块的父块。以代码 5.1 为例,被嵌块(块 3)的父块为 main 函数实现块(块 2),其第 1 个变量 p 的行 id 为 4。其 offset 字段值的计算方法如下:到父块(块 2)的变量表中,找到行 id 为 3 的变量,其 offset 字段值加上其 width,就为被嵌块中第 1 个变量 p 的 offset 字段值。该值为 3208。

注意:被嵌块(块 3)的局部变量 p 与块 2 中的局部变量 i(见代码 5.1 第 18 行)有相同的 offset 字段值,都为 3208,即它们共享存储空间。这完全可行,因为它们位于不同的块中,而且变量 i 的定义出现在变量 p 所在块的后面。也就是说,程序执行到第 18 行时,块 3 已经结束了,其中定义的变量在逻辑上也已释放,不复存在了。

计算完变量的 offset 字段值之后,还要计算块宽。块宽用 width 字段来存储。对于根块和类型定义块,其 width 的计算非常简单,为其变量表中最后一个变量的 offset 字段值与其 width 值之和。对于函数实现块,如果其中没有被嵌块,那么其 width 也为其变量表中最后一个变量的 offset 字段值与其 width 值之和。如果其中含有被嵌块,那么 width 就要另外考虑。还是接着上例说明。main 函数实现块(块 2)的变量表中最后一个变量的 offset 字段值与其 width 值之和为 3212。被嵌块(即块 3)的变量表中最后一个变量的 offset 字段值与其 width 值之和为 3216。main 函数实现块(块 2)因为有子块(块 3),其 width 值要在自己的 width 值与其子块的 width 值中取最大值,即在(3212,3216)中取最大值。这里的最大值为 3216。因此,main 函数实现块(即块 2)的 width 值为 3216。

计算根块、函数实现块、类型定义块的宽度值,是为了分配运行时所需的物理内存空间。局部变量的物理内存空间分配具有实时动态性,其含义是:只有执行进入一个函数时,才为其局部变量一次性分配物理内存空间;当执行到函数返回语句时,便释放其局部变量所占的物理内存空间。因此,在为一个函数实现块生成中间代码时,初始指令就是为其局部变量申请存储空间,末尾指令就是释放局部变量所占的存储空间。这两项操作都需要知道存储空间大小,这就是要计算函数实现块 width 的原因。对于根块、类型定义块,其情形也是如此。

思考题 5-2 类型的 width、变量的 offset 只有在目标机器确定之后才来填写,表明该项工作属于目标代码生成环节。这是不是说明、块表、类型表、变量表、函数表,形参表也是

中间代码的组成部分？

思考题 5-3　对于全局变量、类的实例对象、函数的局部变量，在运行时，会将其视作一个整体，来为其一次性分配内存空间。基于这种处理，思考为什么根块、类型定义块、函数实现块的第 1 个变量的 offset 字段值为 0？为什么被嵌块的第 1 个变量的 offset 字段值要按上述方案处理？为什么不要计算被嵌块的宽度？

5.1.4　函数的语义分析

函数是变量的另一种形式，和数据变量一样，既需要定义，也要使用，同样也要先定义后使用，同样也有类型概念和作用域概念。函数的使用表现为函数调用。函数的语义分析和数据变量的语义分析几乎相同，只是稍微复杂一点。复杂性首先体现在函数的标识上。函数的标识不仅有名称，还有形参的个数、类型、顺序。复杂性的另一表现为：函数除了定义和使用之外，还有实现。对于被调用的函数，必须要有实现。对于根块和类型定义块，除了有**变量表**之外，还要有**函数表**，记录函数的定义信息，以支持函数的语义分析。

思考题 5-4　有关函数的语义规则有哪些？请列举出 4 条。函数表的数据结构中应包含哪些字段，才足以支持函数的语义分析？

对于类(class)的定义，除了成员变量定义之外，还有成员函数的定义。如果将根块也视作一个类，那么全局变量就是其成员变量，全局函数就是其成员函数。如果给这个类取一个名字，叫根类，那么 C 程序中就只有一个根类，而 C++ 程序中则允许在根类中再定义类，也允许在其他类中再定义类。从类的视角来看 C 程序，可知其中只给出了根类的成员函数的实现，没有另外单独给出其定义，其定义隐含在其实现中。因此，语义分析时，要提取其定义，写入根块的函数表中。另外，根类中的成员变量和成员函数能在程序中任何地方使用，这等于说它们都有面向对象中的 public 属性。因此，无论是 C 语言之类的面向过程语言，还是 C++ 语言之类的面向对象语言，可使用统一的语义分析和中间代码生成实现框架。

注意：根类相对于自定义类，有着其自身独特的特性：只有一个实例对象，而且始终存在，无生命周期概念。

成员函数的定义和实现可以合二为一，如代码 5.5 中所示的成员函数 area。成员函数的实现也可和其定义分离，移至类定义块之外，如代码 5.6 所示。这种分开能使类定义更加紧凑简洁，增强程序的可读性，便于代码维护。代码 5.5 中的函数实现块，其父块为类定义块，于是在其中自然能够使用成员变量。与其相对地，代码 5.6 中的函数实现块，其父块不为类定义块。该情形下，要把函数实现块的父块设为类定义块，其原因是代码 5.5 和代码 5.6 的语义完全相同。当成员函数的实现写在类定义块之外时，便要标识它属于哪一个类。在 C++ 中，标识方法便是在函数名之前加上类名，并用“::”将它们区分开，例如代码 5.6 中第 5 行所示。

代码 5.5　成员函数的实现形式之一

```
1  class Rectangle  {
2    int width, height;
3    int area()  {
4      return width * height;
5    }
6  }
```