编译原理实验指导书

Compilers Principles Experiment Instruction Book

陈贺敏　王林

燕山大学软件工程系

实验 4 语义分析和中间代码生成

4.1 实验目的

（1）熟悉语义分析和中间代码生成过程。

（2）加深对语义翻译的理解。

4.2 实验任务

此次实验任务有两个：首先是**语义分析**任务，即在词法分析和语法分析程序的基础上编写一个程序，对输入的源代码进行语义分析和类型检查，并打印分析结果；然后，在语义正确的基础上，实现一种**中间代码的生成**。

任务 1：审查每一个语法结构的静态语义，即验证语法正确的结构是否有意义。此部分不再借助已有工具，需手写代码来完成。

任务 2：在词法分析、语法分析和语义分析程序的基础上，将输入源代码翻译成中间代码。（A、B 任务二选一）

A) 将 C--源代码翻译为中间代码，理论上中间代码在编译器的内部表示可以选用**树形结构（抽象语法树）**或者**线形结构（三地址代码）**等形式，为了方便检查你的程序，我们要求将中间代码输出成线性结构，从而可以使用我们提供的虚拟机小程序（附录 B）来测试中间代码的运行结果。

B) 编写一个中间代码生成程序，能将算术表达式等翻译成逆波兰、三元组或四元组形式（选择其中一种形式即可）

4.3 实验内容

4.3.1 实验要求

任务 1：语义分析

在本任务中，以 C--语言为例（选择其他语言同样可以事先进行相关假设），可对其做如下假设，你可以认为这些就是 C--语言的特性：

1) **假设 1**：整型（int）变量不能与浮点型（float）变量相互赋值或者相互运算。

2) **假设 2**：仅有 int 型变量才能进行逻辑运算或者作为 if 和 while 语句的条件；仅有 int 型和 float 型变量才能参与算术运算。

3) **假设 3**：任何函数只进行一次定义，无法进行函数声明。

4) **假设 4**：所有变量（包括函数的形参）的作用域都是全局的，即程序中所有变量均不能重名。

5) **假设 5**：函数无法进行嵌套定义。

以上假设 1 至 5 也可视为要求，违反即会导致各种语义错误，不过我们只对后面讨论的

11 种错误类型进行考察。此外，**你可以安全地假设输入文件中不包含注释、八进制数、十六进制数、以及指数形式的浮点数、不包含结构体，也不包含任何词法或语法错误。**

你的程序需要对输入文件进行语义分析（输入文件中可能包含函数、一维和高维数组）并检查如下类型的错误：

1) **错误类型 1**：变量在使用时未经定义。

2) **错误类型 2**：函数在调用时未经定义。

3) **错误类型 3**：变量出现重复定义，或变量与前面定义过的结构体名字重复。

4) **错误类型 4**：函数出现重复定义（即同样的函数名出现了不止一次定义）。

5) **错误类型 5**：赋值号两边的表达式类型不匹配。

6) **错误类型 6**：操作数类型不匹配或操作数类型与操作符不匹配（例如整型变量与数组变量相加减）。

7) **错误类型 7**：return 语句的返回类型与函数定义的返回类型不匹配。

8) **错误类型 8**：函数调用时实参与形参的数目或类型不匹配。

9) **错误类型 9**：对非数组型变量使用“[…]”（数组访问）操作符。

10) **错误类型 10**：对普通变量使用“(…)”或“()”（函数调用）操作符。

11) **错误类型 11**：数组访问操作符“[…]”中出现非整数（例如 a[1.5]）。

其中，要注意的是关于数组类型的等价机制，同 C 语言一样，只要数组的基类型和维数相同我们即认为类型是匹配的，例如 int a[10][2]和 int b[5][3]即属于同一类型。

任务 2：中间代码生成

在 **A 任务**中，我们对输入的 C--语言源代码文件做如下假设：

1) **假设 1**：不会出现注释、八进制或十六进制整型常数、浮点型常数或者变量。

2) **假设 2**：不会出现类型为结构体或高维数组（高于 1 维的数组）的变量。

3) **假设 3**：任何函数参数都只能为简单变量，也就是说，结构体和数组都不会作为参数传入函数中。

4) **假设 4**：没有全局变量的使用，并且所有变量均不重名。

5) **假设 5**：函数不会返回结构体或数组类型的值。

6) **假设 6**：函数只会进行一次定义（没有函数声明）。

7) **假设 7**：输入文件中不包含任何词法、语法或语义错误（函数也必有 return 语句）。

你的程序需要将符合以上假设的 C--源代码翻译为中间代码，中间代码的形式及操作规范如表 1 所示，表中的操作大致可以分为如下几类：

1)标号语句 LABEL 用于指定跳转目标，注意 LABEL 与 x 之间、x 与冒号之间都被空格或

制表符隔开。

2)函数语句 FUNCTION 用于指定函数定义，注意 FUNCTION 与 f 之间、f 与冒号之间都被空格或制表符隔开。

3)赋值语句可以对变量进行赋值操作（注意赋值号前后都应由空格或制表符隔开）。赋值号左边的 x 一定是一个变量或者临时变量，而赋值号右边的 y 既可以是变量或临时变量，也可以是立即数。如果是立即数，则需要在其前面添加“#”符号。例如，如果要将常数 5 赋给临时变量 t1，可以写成 t1 := #5。

4)算术运算操作包括加、减、乘、除四种操作（注意运算符前后都应由空格或制表符隔开）。赋值号左边的 x 一定是一个变量或者临时变量，而赋值号右边的 y 和 z 既可以是变量或临时变量，也可以是立即数。如果是立即数，则需要在其前面添加“#”符号。例如，如果要将变量 a 与常数 5 相加并将运算结果赋给 b，则可以写成 b := a + #5。

5)赋值号右边的变量可以添加“&”符号对其进行取地址操作。例如，b := &a + #8 代表将变量 a 的地址加上 8 然后赋给 b。

6)当赋值语句右边的变量 y 添加了“*”符号时代表读取以 y 的值作为地址的那个内存单元的内容，而当赋值语句左边的变量 x 添加了“*”符号时则代表向以 x 的值作为地址的那个内存单元写入内容。

7)跳转语句分为无条件跳转和有条件跳转两种。无条件跳转语句 GOTO x 会直接将控制转移到标号为 x 的那一行，而有条件跳转语句（注意语句中变量、关系操作符前后都应该被

表 1. 中间代码的形式及操作规范

语法	描述
LABEL x :	定义标号 x。
FUNCTION f :	定义函数 f。
x := y	赋值操作。
x := y + z	加法操作。
x := y - z	减法操作。
x := y * z	乘法操作。
x := y / z	除法操作。
x := &y	取 y 的地址赋给 x。
x := *y	取以 y 值为地址的内存单元的内容赋给 x。
*x := y	取 y 值赋给以 x 值为地址的内存单元。
GOTO x	无条件跳转至标号 x。
IF x [relop] y GOTO z	如果 x 与 y 满足[relop]关系则跳转至标号 z。
RETURN x	退出当前函数并返回 x 值。
DEC x [size]	内存空间申请，大小为 4 的倍数。
ARG x	传实参 x。
x := CALL f	调用函数，并将其返回值赋给 x。

PARAM x	函数参数声明。
READ x	从控制台读取 x 的值。
WRITE x	向控制台打印 x 的值。

空格或制表符分开）则会先确定两个操作数 x 和 y 之间的关系（相等、不等、小于、大于、小于等于、大于等于共 6 种），如果该关系成立则进行跳转，否则不跳转而直接将控制转移到下一条语句。

8)返回语句 RETURN 用于从函数体内部返回值并退出当前函数，RETURN 后面可以跟一个变量，也可以跟一个常数。

9)变量声明语句 DEC 用于为一个函数体内的局部变量声明其所需要的空间，该空间的大小以字节为单位。这个语句是专门为数组变量和结构体变量这类需要开辟一段连续的内存空间的变量所准备的。例如，如果我们需要声明一个长度为 10 的 int 类型数组 a，则可以写成 DEC a 40。对于那些类型不是数组或结构体的变量，直接使用即可，不需要使用 DEC 语句对其进行声明。变量的命名规范与之前的实验相同。另外，在中间代码中不存在作用域的概念，因此不同的变量一定要避免重名。

10)与函数调用有关的语句包括 CALL、PARAM 和 ARG 三种。其中 PARAM 语句在每个函数开头使用，对于函数中形参的数目和名称进行声明。例如，若一个函数 func 有三个形参 a、b、c，则该函数的函数体内前三条语句为：PARAM a、PARAM b 和 PARAM c。CALL 和 ARG 语句负责进行函数调用。在调用一个函数之前，我们先使用 ARG 语句传入所有实参，随后使用 CALL 语句调用该函数并存储返回值。仍以函数 func 为例，如果我们需要依次传入三个实参 x、y、z，并将返回值保存到临时变量 t1 中，则可分别表述为：ARG z、ARG y、ARG x 和 t1 := CALL func。注意 ARG 传入参数的顺序和 PARAM 声明参数的顺序正好相反。ARG 语句的参数可以是变量、以#开头的常数或以&开头的某个变量的地址。注意：当函数参数是结构体或数组时，ARG 语句的参数为结构体或数组的地址（即以传引用的方式实现函数参数传递）。

11)输入输出语句 READ 和 WRITE 用于和控制台进行交互。READ 语句可以从控制台读入一个整型变量，而 WRITE 语句可将一个整型变量的值写到控制台上。

除以上说明外，注意关键字及变量名都是大小写敏感的，也就是说“abc”和“AbC”会被作为两个不同的变量对待，上述所有关键字（例如 CALL、IF、DEC 等）都必须大写，否则虚拟机小程序会将其看作一个变量名。

在此任务中，你可能需要在语义分析的程序中做如下更改：在符号表中预先添加 read 和 write 这两个预定义的函数。其中 read 函数没有任何参数，返回值为 int 型（即读入的整数值），write 函数包含一个 int 类型的参数（即要输出的整数值），返回值也为 int 型

（固定返回 0）。添加这两个函数的目的是让 C--源程序拥有可以与控制台进行交互的接口。在中间代码翻译的过程中，read 函数可直接对应 READ 操作，write 函数可直接对应 WRITE 操作。

在 **B 任务**中，你的程序应具有通用性，即能接受各种不同的算术表达式等语法成分。对于语法正确的算术表达式，能生成所选形式的相应序列，并输出结果；对不正确的表达式，能检测出错误。

4.3.2 输入格式

任务 1：你的程序的输入是一个包含源代码的文本文件，程序需要能够接收一个输入文件名作为参数。

任务 2：对于 **A 任务**，你的程序的输入是一个包含 C--源代码的文本文件，你的程序需要能够接收一个输入文件名和一个输出文件名作为参数。例如，假设你的程序名为 cc、输入文件名为 test1、输出文件名为 out1.ir，程序和输入文件都位于当前目录下，那么在 Linux 命令行下运行./cc test1 out1.ir 即可将输出结果写入当前目录下名为 out1.ir 的文件中。

对于 **B 任务**，你的程序输入是包含各种算术表达式的文本文件。

4.3.3 输出格式

任务 1：要求通过标准输出打印程序的运行结果。对于那些没有语义错误的输入文件，你的程序不需要输出任何内容。对于那些存在语义错误的输入文件，你的程序应当输出相应的错误信息，这些信息包括错误类型、出错的行号以及说明文字，其格式为：

Error type [错误类型] at Line [行号]: [说明文字].

说明文字的内容没有具体要求，但是错误类型和出错的行号一定要正确，因为这是判断输出的错误提示信息是否正确的唯一标准。请严格遵守实验要求中给定的错误分类，否则将影响你的实验评分。

输入文件中可能包含一个或者多个错误（但每行最多只有一个错误），你的程序需要将它们全部检查出来。当然，有些时候输入文件中的一个错误会产生连锁反应，导致别的地方出现多个错误（例如，一个未定义的变量在使用时由于无法确定其类型，会使所有包含该变量的表达式产生类型错误），我们只会去考察你的程序是否报告了较本质那个的错误（如果难以确定哪个错误更本质一些，建议你报告所有发现的错误）。但是，如果源程序里有错而你的程序没有报错或报告的错误类型不对，又或者源程序里没有错但你的程序却报错，都会影响你的实验评分。

任务 2：对于 **A 任务**，要求你的程序将运行结果输出到文件。输出文件要求每行一条中间代码，每条中间代码的含义如前文所述。如果输入文件包含多个函数定义，则需要通过

FUNCTION 语句将这些函数隔开。FUNCTION 语句和 LABEL 语句的格式类似，具体例子见后面的样例。

对每个特定的输入，并不存在唯一正确的输出。我们将使用虚拟机小程序对你的中间代码的正确性进行测试。任何能被虚拟机小程序顺利执行并得到正确结果的输出都将被接受。此外，虚拟机小程序还会统计你的中间代码所执行过的各种操作的次数，以此来估计你的程序生成的中间代码的效率。

对于 **B 任务**，要求你的程序能输出生成的逆波兰、三元组或四元组序列。

4.3.4 测试环境

如果你选择的系统是 Ubuntu，你的程序将在如下环境中被编译并运行：

1) GNU Linux Release: Ubuntu 12.04, kernel version 3.2.0-29；

2) GCC version 4.6.3；

3) GNU Flex version 2.5.35；

4) GNU Bison version 2.5.

一般而言，只要避免使用过于冷门的特性，使用其它版本的 Linux 或者 GCC 等，也基本上不会出现兼容性方面的问题。注意，实验的检查过程中不会去安装或尝试引用各类方便编程的函数库（如 glib 等），因此请不要在你的程序中使用它们。

4.3.5 提交要求

1) 可被正确编译运行的源程序。

2）一份实验报告，内容包括：实验目的、实验任务、实验内容，算法描述，程序结构，主要变量说明，程序清单，调试情况及各种情况运行结果截图，心得体会。

4.3.6 样例

任务 1：语义分析

请仔细阅读样例，以加深对实验要求以及输出格式要求的理解。

样例 1：

输入：

```
1   int main()
2   {
3       int i = 0;
4       j = i + 1;
5   }
```

输出：

样例输入中变量“j”未定义，因此你的程序可以输出如下的错误提示信息：

Error type 1 at Line 4: Undefined variable "j".

样例 2：

输入：

```
1    int main()
2    {
3        int i = 0;
4        inc(i);
5    }
```

输出：

样例输入中函数“inc”未定义，因此你的程序可以输出如下的错误提示信息：

Error type 2 at Line 4: Undefined function "inc".

样例 3：

输入：

```
1    int main()
2    {
3        int i, j;
4        int i;
5    }
```

输出：

样例输入中变量“i”被重复定义，因此你的程序可以输出如下的错误提示信息：

Error type 3 at Line 4: Redefined variable "i".

样例 4：

输入：

```
1    int func(int i)
2    {
3        return i;
4    }
5
6    int func()
7    {
8        return 0;
9    }
10
11   int main()
12   {
13   }
```

输出：

样例输入中函数“func”被重复定义，因此你的程序可以输出如下的错误提示信息：

Error type 4 at Line 6: Redefined function "func".

样例 5：

输入：

```
1   int main()
2   {
3       int i;
4       i = 3.7;
5   }
```

输出：

样例输入中错将一个浮点常数赋值给一个整型变量，因此你的程序可以输出如下的错误提示信息：

Error type 5 at Line 4: Type mismatched for assignment.

样例 6：

输入：

```
1   int main()
2   {
3       float j;
4       10 + j;
5   }
```

输出：

样例输入中表达式“10 + j”的两个操作数的类型不匹配，因此你的程序可以输出如下的错误提示信息：

Error type 6 at Line 4: Type mismatched for operands.

样例 7：

输入：

```
1   int main()
2   {
3       float j = 1.7;
4       return j;
5   }
```

输出：

样例输入中“main”函数返回值的类型不正确，因此你的程序可以输出如下的错误提示信息：

Error type 7 at Line 4: Type mismatched for return.

样例 8：

输入：

```
1  int func(int i)
2  {
3      return i;
4  }
5
6  int main()
7  {
8      func(1, 2);
9  }
```

输出：

样例输入中调用函数“func”时实参数目不正确，因此你的程序可以输出如下的错误提示信息：

Error type 8 at Line 8: Function "func(int)" is not applicable for arguments "(int, int)".

样例 9：

输入：

```
1  int main()
2  {
3      int i;
4      i[0];
5  }
```

输出：

样例输入中变量“i”非数组型变量，因此你的程序可以输出如下的错误提示信息：

Error type 9 at Line 4: "i" is not an array.

样例 10：

输入：

```
1  int main()
2  {
3      int i;
4      i(10);
5  }
```

输出：

样例输入中变量“i”不是函数，因此你的程序可以输出如下的错误提示信息：

Error type 10 at Line 4: "i" is not a function.

样例 11：

输入：

```
1  int main()
2  {
3      int i[10];
```

```
4       i[1.5] = 10;
5   }
```

输出：

样例输入中数组访问符中出现了非整型常数“1.5”，因此你的程序可以输出如下的错误提示信息：

Error type 11 at Line 4: "1.5" is not an integer.

任务 2：中间代码生成

A 任务：

样例 1：

输入：

```
1   int main()
2   {
3    int n;
4    n = read();
5    if (n > 0) write(1);
6    else if (n < 0) write (-1);
7    else write(0);
8    return 0;
9   }
```

输出：

这段程序读入一个整数 n，然后计算并输出符号函数 sgn(n)。它所对应的中间代码可以是这样的：

```
 1   FUNCTION main :
 2   READ t1
 3   v1 := t1
 4   t2 := #0
 5   IF v1 > t2 GOTO label1
 6   GOTO label2
 7   LABEL label1 :
 8   t3 := #1
 9   WRITE t3
10   GOTO label3
11   LABEL label2 :
12   t4 := #0
13   IF v1 < t4 GOTO label4
14   GOTO label5
15   LABEL label4 :
16   t5 := #1
```

```
17  t6 := #0 - t5
18  WRITE t6
19  GOTO label6
20  LABEL label5 :
21  t7 := #0
22  WRITE t7
23  LABEL label6 :
24  LABEL label3 :
25  t8 := #0
26  RETURN t8
```

需要注意的是，虽然样例输出中使用的变量遵循着字母后跟一个数字（如 t1、v1 等）的方式，标号也遵循着 label 后跟一个数字的方式，但这并不是强制要求的。也就是说，你的程序输出完全可以使用其它符合变量名定义的方式而不会影响虚拟机小程序的运行。

可以发现，这段中间代码中存在很多可以优化的地方。首先，0 这个常数我们将其赋给了 t2、t4、t7、t8 这四个临时变量，实际上赋值一次就可以了。其次，对于 t6 的赋值我们可以直接写成 t6 := # -1 而不必多进行一次减法运算。另外，程序中的标号也有些冗余。如果你的程序足够“聪明”，可能会将上述中间代码优化成这样：

```
1   FUNCTION main :
2   READ t1
3   v1 := t1
4   t2 := #0
5   IF v1 > t2 GOTO label1
6   IF v1 < t2 GOTO label2
7   WRITE t2
8   GOTO label3
9   LABEL label1 :
10  t3 := #1
11  WRITE t3
12  GOTO label3
13  LABEL label2 :
14  t6 := #-1
15  WRITE t6
16  LABEL label3 :
17  RETURN t2
```

样例 2：

输入：

```
1   int fact(int n)
2   {
```

```
3     if (n == 1)
4     return n;
5     else
6     return (n * fact(n - 1));
7   }
8   int main()
9   {
10    int m, result;
11    m = read();
12    if (m > 1)
13    result = fact(m);
14    else
15    result = 1;
16    write(result);
17    return 0;
18  }
```

输出：

这是一个读入 m 并输出 m 的阶乘的小程序，其对应的中间代码可以是：

```
1   FUNCTION fact :
2   PARAM v1
3   IF v1 == #1 GOTO label1
4   GOTO label2
5   LABEL label1 :
6   RETURN v1
7   LABEL label2 :
8   t1 := v1 - #1
9   ARG t1
10  t2 := CALL fact
11  t3 := v1 * t2
12  RETURN t3
13
14  FUNCTION main :
15  READ t4
16  v2 := t4
17  IF v2 > #1 GOTO label3
18  GOTO label4
19  LABEL label3 :
20  ARG v2
```

```
21  t5 := CALL fact
22  v3 := t5
23  GOTO label5
24  LABEL label4 :
25  v3 := #1
26  LABEL label5 :
27  WRITE v3
28  RETURN #0
```

这个样例主要展示如何处理包含多个函数以及函数调用的输入文件。

4.4 实验指导

4.4.1 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语义分析任务的一些建议。

4.4.1.1 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 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→ 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知道 X1…Xn 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，$\forall A \to X_1 ... X_n \in P$ 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。

4.4.1.2 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，等等）。你可以针对每个语句块都新建一张表，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**语义分析任务**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**语义分析任务**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、红黑树或伸展树等）上查找一个符号的时间复杂度是 O(logn)。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(logn)。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(logn)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(logn)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语义分析所需的时间。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 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  }
```

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 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 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，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 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。

4.4.1.3 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。

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。

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），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。

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。想像一下多维数组该如何去表示呢？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 1 所示的形式。

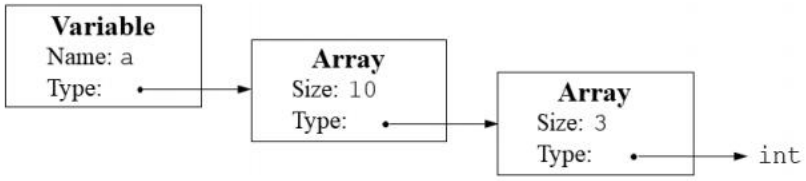

图 1 多维数组的链表表示示例

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。

4.4.1.4 语义分析提示

语义分析需要在词法和语法分析的基础上完成，特别是需要在语法分析时所构建的语法树上完成。语义分析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。语义分析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 11 种语义错误需要检查，它们只涉及到查表与类型操作，其中类型不考虑比较复杂的结构体，即默认输入的源程序里不包含结构体，以此来降低实验的难度。

4.4.2 中间代码生成

编译器里最核心的数据结构之一就是中间代码（Intermediate Representation 或 IR）。中间代码应包含哪些信息，这些信息又应有怎样的内部表示？这些问题会极大地影响编译器代码的复杂程度、编译器的运行效率、以及编译生成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数据结构都被称为中间代码（或可更精确地译为“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，都可视为中间代码。使用中间代码的主要原因是为了方便编写编译器程序的各种操作。如果我们在需要有关输入程序的任何信息时都只能去重新读入并处理输入程序源代码的话，编译器的编写将会变得非常麻烦，同时也会大大降低其运行效率。

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）。你可能会有疑问：为什么编译器不能把输入程序直接翻译成目标代码，而是要额外引入中间代码呢？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为前端和后端两个部分。当我们需要改变编译器的源语言或目标语言时，如果采用了中间代码，我们只需要替换原编译器的前端或后端，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那些复杂但相关性不大的任务分别放在前端和后端的各个模块中，这既简化了模块内部的处理，又使我们能单独对每个模块进行调试与修改而不影响其它模块。下文中，如果不特别说明，“中间代码”都是指狭义的中间代码。

4.4.2.1 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。不同编译器所使用的中间代码可能是千差万别的，即使是同一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接近，有的中间代码与目标语言更接近。编译器需要在不同的中间代码之间进行转换，有时为了处理的方便，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，然后将该树转换为另一种被称为 GIMPLE 的树形结构。在 GIMPLE 之上它建立静态单赋值式的中间代码之后，又会将其转换为一种非常底层的 RTL（Register Transfer Language）代码，最后才把 RTL 转换为汇编代码。

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。从中间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：

1) **高层次中间代码（High-level IR 或 HIR）**：这种中间代码体现了较高层次的细节信息，因此往往和高级语言类似，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。我们所熟悉的 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。

2) **中层次中间代码（Medium-level IR 或 MIR）**：这个层次的中间代码在形式上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为低级语言的代码。正是由于 MIR 的这个特性，它是三种 IR 中最难设计的一种。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操作。另外，对中层次中间代码可以进行绝大部分的优化处理，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。

3) 低层次中间代码（Low-level IR 或 LIR）：低层次中间代码与目标语言非常接近，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，二者之间的转换也属于一趟指令选择就能完成的任务。前面提到的 RTL 就属于一种非常典型的低层次 IR。

图 2 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]
```

图 2 三种不同层次的中间代码示例

从表示方式来看，我们又可以将中间代码分成如下三类：

1) **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，以结点和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会很大，人们经常会使用特殊的图，例如树或有向无环图（DAG）。一个典型的树形中间代码的例子就是抽象语法树（Abstract Syntax Tree 或 AST）。抽象语法树中省去了语法树里不必要的结点，将输入程序的语法信息以一种更加简洁的形式呈现出来。其它树形中间代码的例子有 GCC 中所使用的 GIMPLE。这类中间代码将各种操作都组织在一棵树中。

2) **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就是线性关系。你可以将这种中间代码看成是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，而缺点就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很复杂。

3) 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并避免二者的缺点。例如，我们可以将中间代码组织成许多基本块，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。

在中间代码生成 A 任务中，你需要按照格式输出中间代码。虽然实验要求中规定的中间代码格式类似于线形的中层次中间代码，但这只是输出格式，而你的程序内部可以采用任何形式的中间代码，而这些中间代码中又体现了多少细节信息，则完全取决于你自己的设计。

4.4.2.2 中间代码的表示（线形）

在 A 任务中，你可能会一边对语法树进行处理一边把要输出的代码内容打印出来。这种做法其实并不好，因为当代码内容被打印出来的那一刻起，我们就已经失去了对这些代码进行调整和优化的机会。更加合理的做法是将所生成的中间代码先保存到内存中，等全部翻译完毕，优化也都做完后再使用一个专门的打印函数把在内存中的中间代码打印出来。既然生成好的中间代码会被放到内存中，那么如何保存这些代码以及为其设计怎样的数据结构就是值得考虑的问题了。我们下面对一种典型的线形 IR 的实现细节进行介绍，关于树形 IR 的

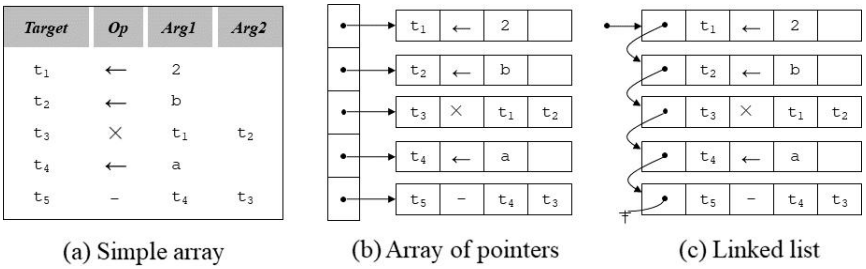

图 3 表示线性 IR 的三种基本数据结构

实现细节我们将放到下一节介绍。

相对而言，线形 IR 是实现起来最简单，而且打印起来最方便的中间代码形式。由于代码本身是线形的，我们可以使用几种最基本的线形数据结构来表示它们，如图 3 所示。

其中图 3(a)为一个大的静态数组，数组中的每个元素（图中的一行）就是一条中间代码。使用静态数组的好处是写起来编程方便，缺点是灵活性不足。中间代码的最大行数受限，而且代码的插入、删除以及调换位置的代价较大。图 3(b)同样为一个大数组，但数组中的每个元素并不是一条中间代码，而是一个指向中间代码指针。虽然采用这种实现时代码行数也会受限，不过它和图 3(a)的实现相比则大大减少了调换代码位置的开销。图 3(c)是一个纯链表的实现， 图中画出来的链表是单向的，但我们更建议使用双向循环链表。链表以增加实现的复杂性为代价换得了极大的灵活性，可以进行高效的插入、删除以及调换位置操作，并且几乎不存在代码最大行数的限制。

假设单条中间代码的数据结构定义为：

```
1   typedef struct Operand_* Operand;
2   struct Operand_ {
3     enum { VARIABLE, CONSTANT, ADDRESS, … } kind;
4     union {
5       int var_no;
6       int value;
7       …
8     } u;
9   };
10
11   struct InterCode
12   {
13     enum { ASSIGN, ADD, SUB, MUL, … } kind;
14     union {
```

```
15      struct { Operand right, left; } assign;
16      struct { Operand result, op1, op2; } binop;
17      …
18    } u;
19  }
```

那么，图 3(a)中的实现可以写成：

```
InterCode codes[MAX_LINE];
```

图 3(b)中的实现可以写成：

```
InterCode* codes[MAX_LINE];
```

图 3(c)中的（双向链表）实现可以写成：

```
struct InterCodes { InterCode code; struct InterCodes *prev, *next; };
```

要想打印出线形 IR 非常简单，只需从第一行代码开始逐行访问，根据每行代码 kind 域的不同值按照不同的格式打印即可。对于数组，逐行访问其实就意味着一个 for 循环；对于链表，逐行访问则意味着以一个 while 循环顺着链表的 next 域进行迭代。

4.4.2.3 中间代码的表示（树形）

树形 IR 看上去可能让人感到复杂，但仔细想想就会发现其实它与线形 IR 一样直观。树形结构天然具有层次的概念，在靠近树根的高层部分的中间代码其抽象层次较高，而靠近树叶的低层部分的中间代码则更加具体。例如，中间代码 t1 := v2 + #3 可用树形结构表示为如图 4 所示。

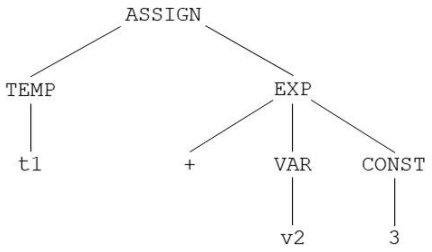

图 4 中间代码 t1 := v2 + #3 的树形结构表示

我们也可以从另一个角度来理解树形 IR。在上次实验中，我们曾为输入程序构造过语法树，这棵语法树所对应的程序设计语言是编译器的源语言；而在这里，树形结构同样可以看作是一棵语法树，它所对应的程序设计语言则是我们的中间代码。源语言的语法相当复杂，但此次实验要求我们输出的中间代码的语法规则却是简单的，因此树形结构的中间代码的设计与实现也会比语法树更简单。

之前我们已经做过有关语法树的实验，你对于树形结构该如何实现应当非常熟悉。正如前面所述，树形 IR 可以看作是一种基于中间代码的语法树（或抽象语法树），因此其数据

结构以及实现细节与语法树非常类似。有了写语法树的经验，写树形 IR 只需在原有基础上稍加修改即可，不会带来太多困难。

树形 IR 的打印要比线形 IR 复杂一些，该任务类似于给定一棵输入程序的语法树需要将该输入程序打印出来。你需要对树形 IR 进行（深度优先）遍历，根据当前结点的类型递归地对其各个子结点进行打印。从另外一个角度看，从之前实验中构造的语法树到实验三要求输出的中间代码之间总要经历一个由树形到线形的转换，使用线形 IR 其实就是将这步转换提前到构造 IR 时，而使用树形 IR 则是将这步转换推后到输出时才进行。

4.4.2.4 运行时环境初探

在一个程序员眼里，程序设计语言中可以有很多机制，包括类型（基本类型、数组和结构体），类和对象，异常处理，动态类型，作用域，函数调用，名空间等等，而且每个程序似乎都有使用不完的内存空间。但很显然，程序运行所基于的底层硬件不能支持这么多机制。一般来说，硬件只对 32bits 或 64bits 位整数、IEEE 浮点数、简单的算术运算、数值拷贝，以及简单的跳转提供直接的支持，并且其存储器的大小也是有限的、结构也是分层的。程序设计语言中 的其它机制则需要编译器、汇编/链接器、操作系统等共同努力，从而让程序员们产生一种幻觉，认为他们眼中所看到的所有机制都是被底层硬件直接支持的。

使用程序设计语言所书写出来的变量、类、函数等都是些抽象层次比较高的概念，为了能使用硬件直接支持的底层操作来表示这些高抽象层次的概念，仅靠编译时刻字面上的代码翻译是远远不够的，我们需要能够生成额外的目标代码，使程序在运行时刻可以维护起一系列的结构以支撑起程序设计语言中的各种高级特性。这些程序员一般不可见、但又确实存在于运行时刻的结构就被称为运行时环境（Runtime Environment）。运行时环境与源语言、目标语言和目标机器都紧密相关，其中包含很多细节，此次实验中我们以介绍原理为主，附带介绍一些简单结构（例如数组和结构体）的实现方式。

高级语言中的 char、short 和 int 等类型一般会直接对应到底层机器上的一个、两个或四个字节，而 float 和 double 类型则会对应到底层机器上的四个和八个字节，这些类型都可以由硬件直接提供支持。底层硬件中没有指针类型，但指针可以用四个字节（32bits 位机器）或者八个字节（64bits 位机器）整数表示，其内容即为指针所指向的内存地址。

以上是一些比较基本的类型，下面我们来考察一维数组的表示。最熟悉的表示数组的方式是 C 风格的（如图 5 所示），即数组中的元素一个挨着一个并占用一段连续的内存空间。

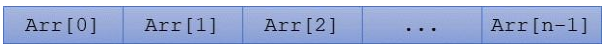

图 5　C 语言中一维数组的内存表示方式

图 6 Java 语言中一维数组的内存表示方式

当然这不是唯一的表示方法。Java 在编译 bytecode 时就会采取另外一种布局，将数组长度放在起始位置（Pascal 中的 string 类型数据也是这样保存的），如图 6 所示。

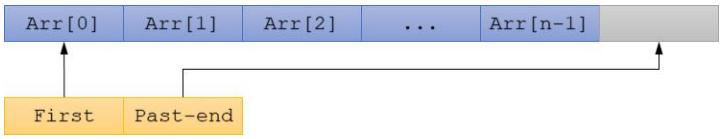

图 7 D 语言中一维数组的内存表示方式

图 8　C 语言中多维数组的内存表示方式

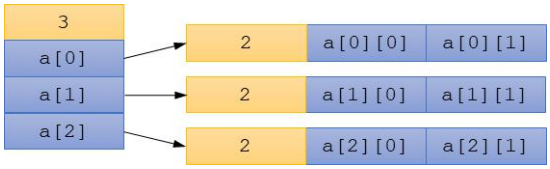

图 9　Java 语言中多维数组的内存表示方式

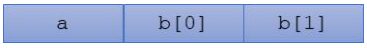

图 10　结构体的内存表示示例

另外还有一种表示方式为 D 语言所采用：数组变量本身仅由两个指针组成，一个指向数组的开头，另一个指向数组的末尾之后，数组的所有信息存在于另外一段内存之中，如图 7 所示。可以看出，无论是哪一种表示方式，数组元素在内存中总是连续存储的，这当然是为了使数组的访问能够更快（只需计算基地址+偏移量，然后取值即可）。多维数组可以看作一维数组的数组，C 风格的表示方法仍然是使用一段连续的内存空间，如图 8 所示。

而 Java 中每个一维数组是一个独立的对象，因此多维数组中的各维一般不会聚在一起，如图 9 所示。

结构体的表示与数组类似，最常见的办法是将各个域按定义的顺序连续地存放在一起。比如 struct { int a; float b[2]; }在内存中的表示如图 10 所示。

图 11　C—语言中结构体的内存表示的错误示例

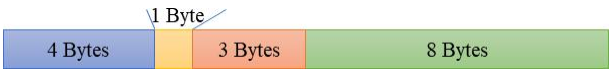

图 12 C—语言中结构体的内存表示的正确示例

我们此次实验中只包含 int 和 float 两种类型，而这两种类型的宽度都是四字节，这省去了我们许多的麻烦。如果 C—语言允许其它宽度的类型存在又会如何呢？例如，struct { int a; char b; double c; }这个结构体在内存中会排成如图 11 所示的表示方式吗？

答案是不会。如果没有特别指定，GCC 总会将结构体中的域对齐到字边界。因此在 char b 和 double c 之间会有 3 字节的空间被浪费掉，如图 12 所示。

x86 平台允许变量在存储时不对齐，但 MIPS 平台则要求对齐，这是我们需要注意的。最后我们简单讨论一下函数的表示。众所周知，在调用函数时我们需要进行一系列的配套工作，包括找到函数的入口地址、参数传递、为局部变量申请空间、保存寄存器现场、返回值传递和控制流跳转等等。在这一过程中，我们需要用到的各种信息都必须有地方能够保存下来，而保存这些信息的结构就称为活动记录（Activation Record）。因为活动记录经常被保存在栈上，故它往往也被称为栈帧（Stack Frame）。活动记录的建立是维护运行时刻环境的重点，编译器一般也会为它生成大段的额外代码，这意味着函数调用的开销一般会很大。所以对于功能简单的函数来说，内联展开往往是一种有效的优化方法。在本任务中我们并不需要关心活动记录是如何建立的，只需要压入相应的参数然后调用 CALL 语句即可。但要注意的是， 若数组或结构体作为参数传递，需谨记数组和结构体都要求采用引用调用（Call by Refer- ence）的参数传递方式，而非普通变量的值调用（Call by Value）。

表 2 基本表达式的翻译模式

translate_Exp(Exp, sym_table, place) = case Exp of	
INT	value = get_value(INT) return [place := #value][2]
ID	variable = lookup(sym_table, ID) return [place := variable.name]
Exp_1 ASSIGNOP Exp_2[3] ($Exp_1 \rightarrow$ ID)	variable = lookup(sym_table, Exp_1.ID) t1 = new_temp() code1 = translate_Exp(Exp_2, sym_table, t1) code2 = [variable.name := t1] +[4] [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	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]
NOT Exp_1	
Exp_1 AND Exp_2	
Exp_1 OR Exp_2	

4.4.2.5 翻译模式（基本表达式）

中间代码生成的任务比较简单，你只需根据语法树产生出中间代码，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？

最简单也是最常用的方式仍是遍历语法树中的每一个结点，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于实验二中我们已经提到的工具：语法制导翻译（SDT）。具体到代码上，我们可以为每个主要的语法单元“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互相调用的过程。每种特定的语法结构都对应了固定模式的翻译“模板”，下面我们针对一些典型的语法树结构的翻译“模板”进行说明。这些内容你也可以在课本上找到，课本上介绍的翻译模式与下面我们介绍的可能略有不同，但核心思想是一致的[1]。

我们先从语言最基本的结构表达式开始。

表 2 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：语法树的结点 Exp、符号表 sym_table 以及一个变量名 place，并返回一段语法树当前结点及其子孙结点对应的中间代码（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：

1)如果 Exp 产生了一个整数 INT，那么我们只需要为传入的 place 变量赋值成前面加上一个“#”的相应数值即可。

2) 如果 Exp 产生了一个标识符 ID，那么我们只需要为传入的 place 变量赋值成 ID 对应的变量名（或该变量对应的中间代码中的名字）即可。

3) 如果 Exp 产生了赋值表达式 Exp1 ASSIGNOP Exp2，由于之前提到过作为左值的 Exp1 只能是三种情况之一（单个变量访问、数组元素访问或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在后面讨论，故这里仅列出当 Exp1 → 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，然后对 Exp2 进行翻译（运算结果储存在临时变量 t1 中），再将 t1 中的值赋于 ID 所对应的变量并将结果再存回 place，最后把刚翻译好的这两段代码合并随后返回即可。

---

[1] 使用模板并不是生成中间代码的唯一方法，也存在着其他方法（例如构造控制流图）可以完成翻译任务，只不过使用模板是最简单和被介绍得最为广泛的方法。

[2] 用方括号括起来的内容表示新建一条具体的中间代码。

[3] 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。

[4] 这里的加号相当于连接运算，表示将两段代码连接成一段。

4) 如果 Exp 产生了算术运算表达式 Exp1 PLUS Exp2，则先对 Exp1 进行翻译（运算结果储存在临时变量 t1 中），再对 Exp2 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。

5) 如果 Exp 产生了取负表达式 MINUS Exp1，则先对 Exp1 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 从而实现对 t1 取负，最后将翻译好的这两段代码合并后返回。使用类似的翻译模式我们也可以对括号表达式进行翻译。

表 3 语句的翻译模式

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]

6) 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），我们则会调用 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与跳转语句有关，表中并没有明确说明 translate_Cond 该如何实现，这一点我们在后面介绍。

表 4 条件表达式的翻译模式

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]

4.4.2.6 翻译模式（语句）

C--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，它们的翻译模式如表 3 所示。

你可能注意到，无论是 if 语句还是 while 语句，表 4 中列出的翻译模式都不包含条件跳转。其实我们是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，其翻译模式如表 4 所示。

对于条件表达式的翻译，课本上已经花了较大的篇幅进行介绍，尤其是与回填有关的内容更是重点。不过，表 4 中没有与回填相关的任何内容。原因很简单：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，在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结点那里通过参数得到了，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。

表 5 函数调用的翻译模式

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]] + [place := #0] for i = 1 to length(arg_list) code2 = code2 + [ARG arg_list[i]] return code1 + code2 + [place := CALL function.name]

表 6 函数参数的翻译模式

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

4.4.2.7 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，如表 5 所示。

由于实验要求中规定了两个需要特殊对待的函数 read 和 write，故当我们从符号表中找到 ID 对应的函数名时不能直接生成函数调用代码，而是应该先判断函数名是否为 read 或 write。对于那些非 read 和 write 的带参数的函数而言，我们还需要调用 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 的实现如表 6 所示。

4.4.2.8 翻译模式（数组与结构体）

C−−语言的数组实现采取的是最简单的 C 风格。数组和结构体不同于一般变量的一点在于，访问某个数组元素或结构体的某个域需要牵扯到其内存地址的运算。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：

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

上式很容易推广到任意维数组的情况。

结构体的访问方式与数组非常类似。例如，假设要访问结构体 struct { int x[10]; int y, z; } st 中的域 z，我们首先找到变量 st 的首地址，然后找到 st 中域 z 的首地址（st 的地址加上数组 x 的大小（4×10）再加上整数 y 的宽度 4）。我们可以把一个有 n 个域的结构体看成为一个有 n 个元素的“一维数组”，它与一般一维数组的不同点在于，一般一维数组的每个元素的大小都是相 同的，而结构体的每个域大小可能不一样。其地址运算的过程可以表示为：

$$\mathrm{ADDR}(\mathrm{st}.field_n) = \mathrm{ADDR}(\mathrm{st}) + \sum^{n-1} \mathrm{SIZEOF}(\mathrm{st}.field_t)。$$

将数组的地址运算和结构体的地址运算结合起来也并不是太难的事。假如我们有一个结构体，该结构体的某个元素是数组，为了访问这个数组中的某个元素，我们需要先根据该数组在结构体中的位置定位到这个数组的首地址，然后再根据数组的下标定位该元素。反之，如果我们有一个数组，该数组的每个元素都是结构体。为了访问某个数组元素的某个域，我们需要先根据数组的下标定位要访问的结构体，再根据域的位置寻找要访问的内容。这个过程中唯一需要关注的是，我们应记录并区分在访问过程中使用到的临时变量哪些代表地址，哪些代表内存中的数值。如果弄错，会导致代码的运行结果出错或者非法的内存访问。当然，上述访问方式需要经历多次地址计算，如果我们能通过其它手段将这多次地址计算合并成一次，那么得到的中间代码的效率就会得到一定的提高。

4.4.2.8 中间代码生成的提示

中间代码生成需要在语义分析的基础上完成。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；也可以将关于中间代码生成的所有内容写到一个单独的文件中，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之后，下一步就要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。完成之后建议先自行写一个测试程序，在这个测试程序中使用构造函数人工构造一段代码并将其打印出来，然后使用我们提供的虚拟机小程

序简单地测试一下，以确保自己的数据结构和打印函数都能正常工作。准备工作完成之后，再继续做下面的内容。

接下来的任务是根据前面介绍的翻译模式完成一系列的 translate 函数。我们已经给出了 Exp 和 Stmt 的翻译模式，你还需要考虑包括数组、结构体、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。需要你自己考虑的内容实际上并不太多，最关键的一点在于一定要读懂前面介绍的几个 translate 函数的意思。如果读懂了，那么无论是将它们实现到你的编译器中还是书写新的 translate 函数，或者是对这些 translate 函数进行改进，都将是比较容易的。如果没读懂，例如没想明白某些 translate 函数中出现的 place 参数究竟有什么用，那么建议你还是先别急着动手写代码。如果顺利完成了所有 translate 函数，并将其连同中间代码的打印函数添加到了你的编译器中，那么中间代码生成的要求就差不多完成了。建议在这里多写几份测试用例检查你的编译器，如果发现了错误需要及时纠正。

最后，虚拟机小程序将以总共执行过的中间代码条数为标准来衡量你的编译器所输出的中间代码的运行效率。因此如果要进行代码优化，重点应该放在精简代码逻辑以及消除代码冗余上。