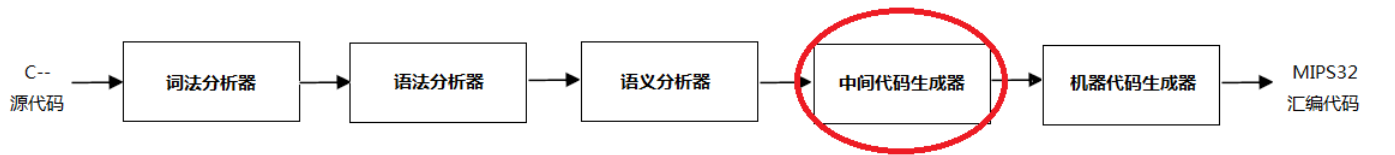


编译原理实习 3 中间代码生成

许畅 陈嘉 朱晓瑞

编译原理课程的实习内容是为一个小型的类 C 语言(C--)实现一个编译器。如果你顺利完成了本课程规定的实习任务，那么不仅你的编程能力将会得到大幅提高，而且你最终会得到一个比较完整的、能将 C--源代码转换成 MIPS 汇编代码的编译器，所得到的汇编代码可以在 SPIM Simulator 上运行。实习总共分为四个阶段：词法和语法分析、语义分析、中间代码生成和目标代码生成。每个阶段的输出是下一个阶段的输入，后一个阶段总是在前一个阶段的基础上完成，如下图所示：



实习 3 的任务是编写一个程序，该程序读入一个 C--源代码文件，在词法分析、语法分析和语义分析之后，生成该程序的中间代码。你的中间代码在编译器的内部表示可以选择用树形结构（抽象语法树）或者线形结构（三地址代码）等各种形式，但本次实验中为了方便对程序进行检查你需要将中间代码输出成一个线性形式，从而可以使用我们预先提供的小型虚拟机程序测试代码的运行结果。

本次实验中，你可以对输入文件做如下假设（注意：假设 2、3 可能因为你选择的分组任务而有所改变，请务必先阅读实验要求后再动手编程）：

- ◆ 假设 1：不会出现注释、八进制和十六进制常数、浮点类型的常数或者变量
- ◆ 假设 2：不会出现类型为结构体和高维数组（高于 1 维的数组）的变量
- ◆ 假设 3：任何参数都只能为简单变量，也就是说，数组和结构体不会作为参数传入某个函数中
- ◆ 假设 4：所有变量都具有全局作用域（这里“全局作用域”是指所有变量均不重名，并非变量在任何位置都能被访问），你还可以假设不存在全局变量
- ◆ 假设 5：函数不会返回数组类型和结构体类型的值
- ◆ 假设 6：函数只会进行一次定义（没有函数声明）
- ◆ 假设 7：输入文件不包含词法、语法、及语义错误

你的程序需要将 C--源代码翻译为中间代码，要求你输出的中间代码的形式及其提供的操作如下表所示：

Syntax	Comments
LABEL x :	Define a label named x
FUNCTION f :	Define a function named f
x := y	Assignment

$x := y + z$	Addition
$x := y - z$	Subtraction
$x := y * z$	Multiplication
$x := y / z$	Division
$x := \&y$	Result = ADDR[y]
$x := *y$	Result = MEM[address]
$*x := y$	MEM[address] = Result
GOTO x	Unconditional jump to label x
IF x [relop] y GOTO z	If (x [relop] y) then jump to label z [relop] $\in \{==, !=, <, >, <=, >=\}$
RETURN x	Exit current function and set x to be the return value
DEC x [size]	Memory block declaration [size] $\in \{4n n \in \mathbb{N}\}$
ARG x	Passing an argument x to a function
$x := \text{CALL } f$	Calling a function f and storing the returned value to a variable x
PARAM x	Parameter declaration
READ x	Read an integer from the console and store the value to a variable x
WRITE x	Write the value of an integer x to the console

表中的操作大致可以分为如下几类：

- ♦ 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。
- ♦ 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。
- ♦ 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 则既可以是变量/临时变量，也可以是立即数。如果是立即数的话，需要在前面添加"#"符号。例如，如果要将常数 5 赋给临时变量 t1，则可以写成 $t1 := \#5$ 。
- ♦ 算术运算操作包括加减乘除四则运算（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 则既可以是变量/临时变量也可以是立即数。如果是立即数的话, 需要在前面添加“#”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 $b := a + \#5$ 。

- ◆ 赋值号右边的变量可以添加“&”符号对其进行取地址运算。例如, $b := \&a + \#8$ 代表将变量 a 的地址加上 8 然后赋给 b 。
- ◆ 当赋值语句右边的变量 y 添加了“*”符号时代表读取以 y 中的值作为地址的那个内存单元中的内容, 而当赋值语句左边的变量 x 添加了“*”符号时代表向以 x 中的值作为地址的那个内存单元中的写入内容。
- ◆ 跳转语句分为无条件跳转和有条件跳转。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、大于等于总共 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。
- ◆ 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。
- ◆ 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, 其中空间的大小以字节为单位。这个语句是专门为数组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者结构体的变量, 直接使用即可, 不要使用 **DEC** 语句对其进行声明。变量的命名规范与之前的实习相同。另外, 在中间代码中可不存在什么作用域的概念, 因此不同的变量一定要避免重名。
- ◆ 与函数调用有关的语句包括 **CALL**、**PARAM** 和 **ARG** 三个。其中 **PARAM** 语句在每一个函数开头使用, 对于函数中形参的数目和名称进行声明。例如, 一个函数 `func` 有三个形参 a, b 和 c , 则该函数的函数体内前三条语句一定是: **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。仍然以刚才的函数 `func` 为例子, 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, 则可以这样写: **ARG** z , **ARG** y , **ARG** x , $t1 := \text{CALL func}$ (注意 **ARG** 传入的参数的顺序和 **PARAM** 声明的参数的顺序正好相反)。

ARG 语句后面可以跟的除了变量以外也可以是以 `#` 开头的常数或是以 `&` 开头的某个变量的地址。

- ◆ 输入输出语句 **READ** 和 **WRITE** 用于跟控制台进行交互。**READ** 语句可以从控制台读入一个整型变量, 而 **WRITE** 语句可以将某一个整型变量写到控制台上。
- ◆ 关键字以及变量名都是大小写敏感的, 也就是说 `abc` 和 `AbC` 将会被作为两个不同的变量对待。

上述所有的关键字（例如 CALL、IF、DEC 等）都必须大写，否则虚拟机会将其看作一个变量名。

在本次实习中，你可能需要在前一次实习的代码中做如下更改：

- ♦ 在符号表中预先添加 `read()` 和 `write()` 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（代表读入的整数值）；`write()` 函数包含一个 `int` 类型的参数（代表要输出的整数值），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 程序拥有可以和控制台进行交互的函数接口，在具体翻译的过程中 `read()` 函数可以直接对应 `READ` 操作，`write()` 函数可以直接对应 `WRITE` 操作。

以上为每位同学的必做内容。除此之外，你的程序需要完成且**仅能完成**下列两个分组之一中的要求：

(1) 分组 3.1

修改对输入文件假设中的假设 2、3，使输入文件中

- ♦ 可以出现结构体类型的变量（采用名等价机制）。
- ♦ 结构体类型的变量可以作为函数的参数（但函数不会返回结构体类型的值）。

你的程序要能够翻译此类输入文件。

(2) 分组 3.2

修改对输入文件假设中的假设 2、3，使输入文件中

- ♦ 一维数组类型的变量可以作为函数参数（但函数不会返回一维数组类型的值）。
- ♦ 可以出现高维数组类型的变量（但高维数组类型的变量不会作为函数的参数或返回值）。

你的程序要能够翻译此类输入文件。

你可以根据分配到的**任务编号**在任务说明文件（`task.pdf`）中查找你在整个编译课程实验中**必须完成**的分组。除了必须完成的分组外，完成任何额外的分组都要**倒扣分**！

此外，本次试验还会考察你的程序输出的中间代码的执行**效率**，你需要思考如何优化中间代码。在你的程序可以生成**正确的中间代码**（“正确的中间代码”指的是中间代码在我们提供的小型虚拟机上运行结果正确）的前提下，如果你的中间代码在我们专门设计的 benchmark 上能够比 50%或 80%的其他同学的中间代码效率都要高，你将获得相应的额外分数的奖励。

在进行本次实习之前，请**仔细阅读**上面对中间代码的说明以及下面几个样例输入输出，确保你已经明确如何每一条中间代码应当如何使用，这些内容是你顺利完成本次实习的基本前提。

需要注意的是，由于在后面的实习中还会扩展甚至修改本次实习你已经写好的代码，因此保持一个良好的代码风格、系统地设计代码结构和各模块之间的接口等对于整个实习来讲可以说是相当重要的。

输入格式

程序的输入是一个文本文件，其中包含有 C-- 的源代码。

本次实习要求你的程序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。

输出格式

本次实习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，则通过 `FUNCTION` 语句将这些函数隔开。`FUNCTION` 语句和 `LABEL` 语句格式类似，具体例子见样例输出。

对于每一个特定的输入文件而言，并不存在唯一正确的输出。我们将使用虚拟机小程序对你输出的代码的正确性进行测试，任何能被虚拟机执行并且执行结果正确的输出都将被接受。此外，虚拟机程序会统计你的程序所执行过的各种操作的次数，以此来估计你生成的代码的效率。

测试环境

你的程序将在如下环境中被编译并运行：

- ◆ GNU Linux Release: Ubuntu 12.04, kernel version 3.2.0-29
- ◆ GCC version 4.6.3
- ◆ GNU Flex version 2.5.35
- ◆ GNU Bison version 2.5

一般而言，只要避免使用过于冷门的特性，使用其他版本的 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。

提交要求

本次实习要求你提交如下内容：

- ◆ Flex、Bison 以及 C 语言的可以被正确编译运行的源代码

◆ 一份 PDF 格式的实验报告，内容主要包括：

- 你的程序实现了哪些功能？简要说明你是如何实现这些功能的。如果因为你的说明不够充分而导致助教没有对你所实现的功能进行测试，那么后果自负。
- 你所提交上来的程序应当如何编译？不管你使用了脚本也好，准备了 Makefile 也好甚至是单独地逐条命令手工输入进行编译也好，请**详细说明**具体需要键入哪些命令——无法顺利编译将会使你丢失相应分数，并且如果不去找助教进行修正，后面的正确分也会因你的程序无法运行而全部丢失，请谨记这一点。
- 你的实验报告长度**不得超过 3 页**！因此，你需要好好考虑一下该往实验报告里写些什么。我们的建议是，实验报告中需要你重点描述的应当是你所提交的工作中的亮点，应当是那些你认为**最个性化**、最具有**独创性**的内容，而那些比较简单的、任何人都可以做出来的内容可以不提或者只简单的提一下，尤其要避免去大段大段地向报告里贴代码。

为了避免大家通过减小字号来变相加长页数限制，我们规定实验报告中所出现的最小字号不得小于五号字（英文 11 号字）。

必做内容样例

样例输入 1

```
int main()
{
    int n;
    n = read();
    if (n > 0) write(1);
    else if (n < 0) write (-1);
    else write(0);
    return 0;
}
```

样例输出 1

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


```
FUNCTION main :  
  READ t1  
  v1 := t1  
  t2 := #0  
  IF v1 > t2 GOTO label1  
  GOTO label2  
  LABEL label1 :  
  t3 := #1  
  WRITE t3  
  GOTO label3  
  LABEL label2 :  
  t4 := #0  
  IF v1 < t4 GOTO label4  
  GOTO label5  
  LABEL label4 :  
  t5 := #1  
  t6 := #0 - t5  
  WRITE t6  
  GOTO label6  
  LABEL label5 :  
  t7 := #0  
  WRITE t7  
  LABEL label6 :  
  LABEL label3 :  
  t8 := #0  
  RETURN t8
```

需要注意的是，虽然这段样例输出里使用的变量遵循着字母 t 后跟一个数字（t1、t2等）或者字母 v 后跟一个数字（v1、v2等）的命名方式，行标名也遵循了 label 后跟一个数字的方式，但这并不是强制要求的。也就是说，你在自己的程序中完全可以使用其它的名字而不会影响到虚拟机的运行。

可以发现，这一段代码中存在着很多可以优化的地方。首先，0这个常数我们赋给了 t2、t4、t7、t8四个临时变量，实际上赋值一次就可以了。其次，对于 t6的赋值来说我们可以直接写成 t6:=-1而不必多进行一次减法运算。另外，程序中的标号也似乎有些冗余。如果你的编译器足够“聪明”，可能会将上述代码简单地优化成这样：

```

FUNCTION main :
READ  t1
v1 := t1
t2 := #0
IF v1 > t2  GOTO label1
IF v1 < t2  GOTO label2
WRITE t2
GOTO label3
LABEL label1 :
t3 := #1
WRITE t3
GOTO label3
LABEL label2 :
t6 := #-1
WRITE t6
LABEL label3 :
RETURN  t2

```

样例输入 2

```

int fact(int n)
{
    if (n == 1)
        return n;
    else
        return (n*fact(n-1));
}
int main()
{
    int m, result;
    m = read();
    if (m > 1)
        result = fact(m);
    else
        result = 1;
    write(result);
    return 0;
}

```


样例输出 2

这是一个读入 m 并输出 m 阶乘的小程序。其对应的中间代码可以是：

```
FUNCTION fact :  
  PARAM v1  
  IF v1 == #1 GOTO label1  
  GOTO label2  
  LABEL label1 :  
  RETURN v1  
  LABEL label2 :  
  t1 := v1 - #1  
  ARG t1  
  t2 := CALL fact  
  t3 := v1 * t2  
  RETURN t3  
  
FUNCTION main :  
  READ t4  
  v2 := t4  
  IF v2 > #1 GOTO label3  
  GOTO label4  
  LABEL label3 :  
  ARG v2  
  t5 := CALL fact  
  v3 := t5  
  GOTO label5  
  LABEL label4 :  
  v3 := #1  
  LABEL label5 :  
  WRITE v3  
  RETURN #0
```

这个例子主要展示了如何处理包含多个函数以及函数调用的输入文件。在我们的样例输出中，`main` 函数没有使用 `fact` 函数中使用的任何变量，`fact` 中也没有使用 `main` 函数中的变量，但这不是必须的。前面说过，我们的虚拟机程序里没有作用域的概念，因此理论上在中间代码的任何位置都可以使用任何一个变量或者直接使用新的变量。不过我们并不推荐跨函数使用变量，其原因会在下次实习中解释。

分组内容样例

样例输入 1

```
struct Operands
{
    int o1;
    int o2;
};

int add(struct Operands temp)
{
    return (temp.o1 + temp.o2);
}

int main()
{
    int n;
    struct Operands op;
    op.o1 = 1;
    op.o2 = 2;
    n = add(op);
    write(n);
    return 0;
}
```

样例输出 1

样例程序中出现了结构体类型，以及结构体类型的变量作为函数参数的用法。对于**需要**完成分组 3.1 的同学，样例程序对应的中间代码可以是：

```
FUNCTION add :
PARAM v1
t2 := *v1
t7 := v1 + #4
t3 := *t7
t1 := t2 + t3
RETURN t1
FUNCTION main :
DEC v3 8
t9 := &v3
*t9 := #1
t12 := &v3 + #4
*t12 := #2
ARG &v3
```

```
t14 := CALL add
v2 := t14
WRITE v2
RETURN #0
```

对于**不需要**完成分组 3.1 的同学，你的程序应不能翻译样例程序，因此你需要在标准输出中给出如下提示信息：

```
Can not translate the code: Contain structure
and function parameters of structure type!
```

样例输入 2

```
int add(int temp[2])
{
    return (temp[0] + temp[1]);
}

int main()
{
    int op [2];
    int r[1][2];
    int i = 0, j = 0;
    while (i<2)
    {
        while (j<2)
        {
            op[j] = i + j;
            j = j + 1;
        }
        r[0][i] = add(op);
        write(r[0][i]);
        i = i + 1;
        j = 0;
    }
    return 0;
}
```

样例输出 2

样例程序中出现了高维数组类型，以及一维数组类型的变量作为函数参数的用法。对于**需要**完成分组 3.2 的同学，样例程序对应的中间代码可以是：

```

FUNCTION add :
PARAM v1
t2 := *v1
t11 := v1 + #4
t3 := *t11
t1 := t2 + t3
RETURN t1
FUNCTION main :
DEC v2 8
DEC v3 8
v4 := #0
v5 := #0
LABEL label1 :
IF v4 < #2 GOTO label2
GOTO label3
LABEL label2 :
LABEL label4 :
IF v5 < #2 GOTO label5
GOTO label6
LABEL label5 :
t18 := v5 * #4
t19 := &v2 + t18
t20 := v4 + v5
*t19 := t20
v5 := v5 + #1
GOTO label4
LABEL label6 :
t31 := v4 * #4
t32 := &v3 + t31
ARG &v2
t33 := CALL add
*t32 := t33
t41 := v4 * #4
t42 := &v3 + t41
t35 := *t42
WRITE t35
v4 := v4 + #1
v5 := #0
GOTO label1
LABEL label3 :
RETURN #0

```

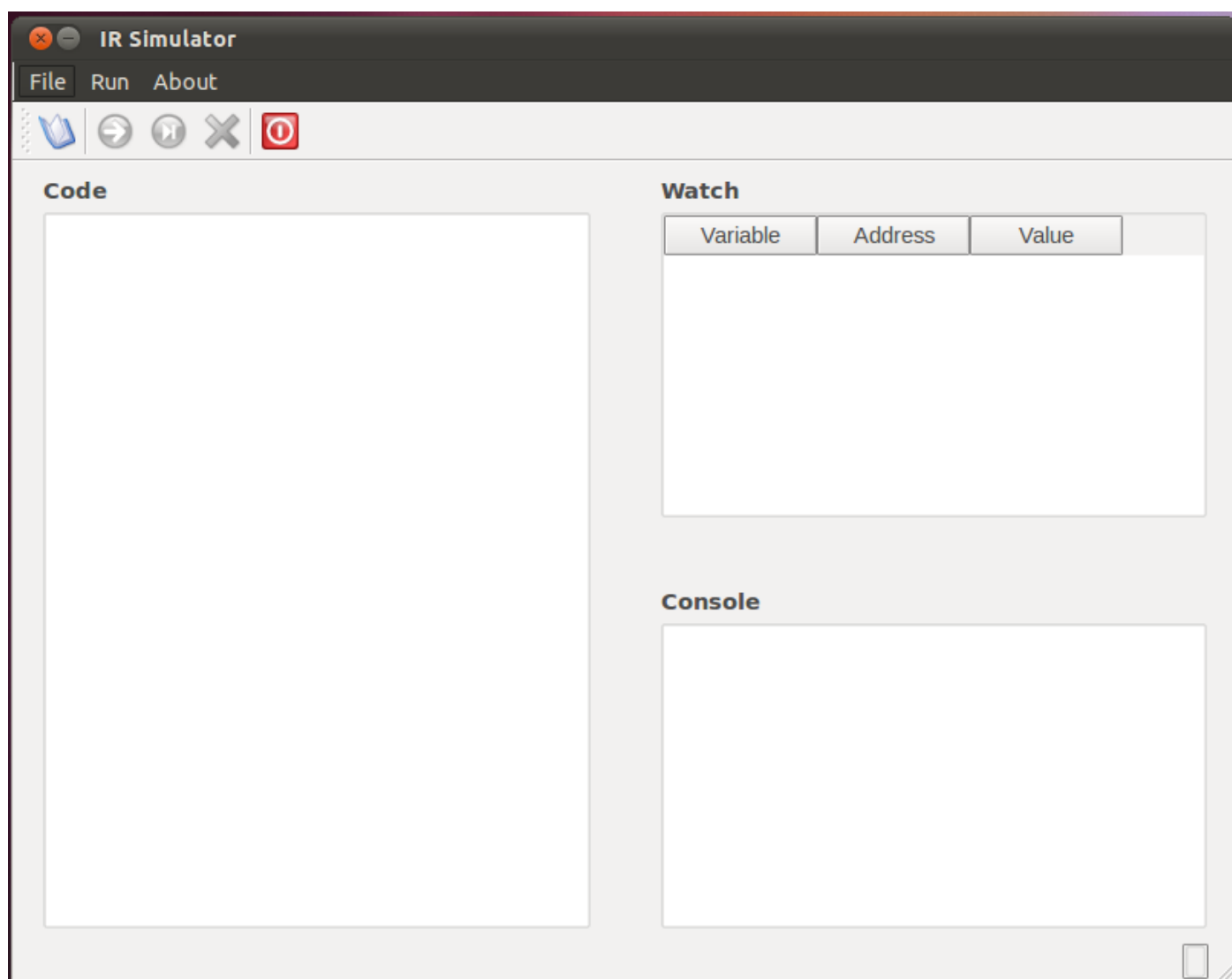
对于**不需要**完成分组 3.2 的同学，你的程序应不能翻译样例程序，因此你需要在标准输出中给出如下提示信息：


Can not translate the code: Contain multidimensional array and function parameters of array type!

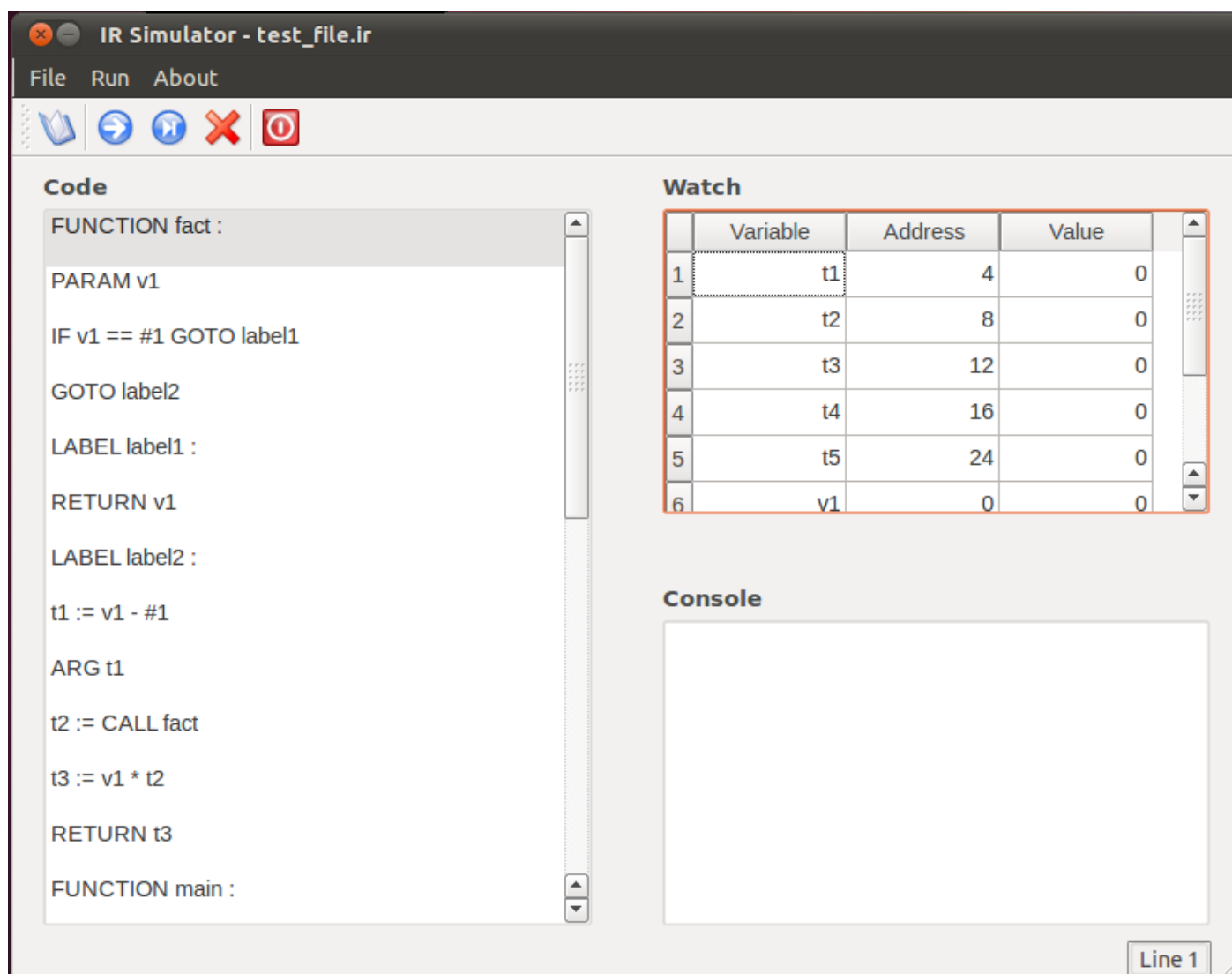
附 虚拟机小程序使用说明




我们所提供的这个简单的虚拟机程序名字叫做 IR Simulator，它本质上就是一个中间代码的解释器。这个程序使用 Python 写成，图形界面部分则借助了跨平台的诺基亚 Qt 库。由于实验环境为 Linux，因此该程序只在 Linux 下进行发布。注意，运行虚拟机程序前要确保本机上装有 Qt 运行环境。你可以在 Linux 终端下使用“`sudo apt-get install python-qt4`”命令获取 Qt 运行环境。课程网站上可以下载虚拟机的源码，将压缩包中的三个文件解压到同一目录中并在当前目录下执行“`python irsim.py`”命令即可启动虚拟机。

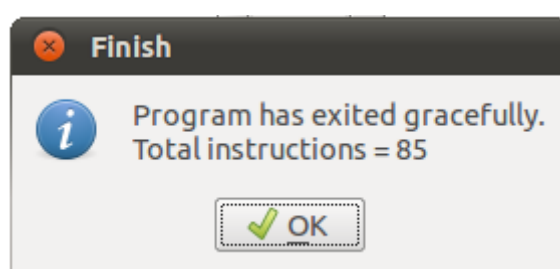
IR Simulator 的界面如下图所示：



整个界面分为三个部分：左侧的代码区、右上方的监视区和右下方的控制台。代码区负责显示已经载入的代码，监视区会显示代码中所包含的所有变量的当前值，控制台区供 `WRITE` 函数进行输出时用。我们可以点击上方工具栏中的  按钮或者点击菜单项 `File->Open` 打开一个保存有中间代码的文本文件（注意该文件的后缀名必须是 `.ir`）。如果中间代码中包含语法错误，则程序会弹出对话框提示出错位置并且拒绝载入代码；如果文件中没有错误，那么你将看到类似下面的内容：



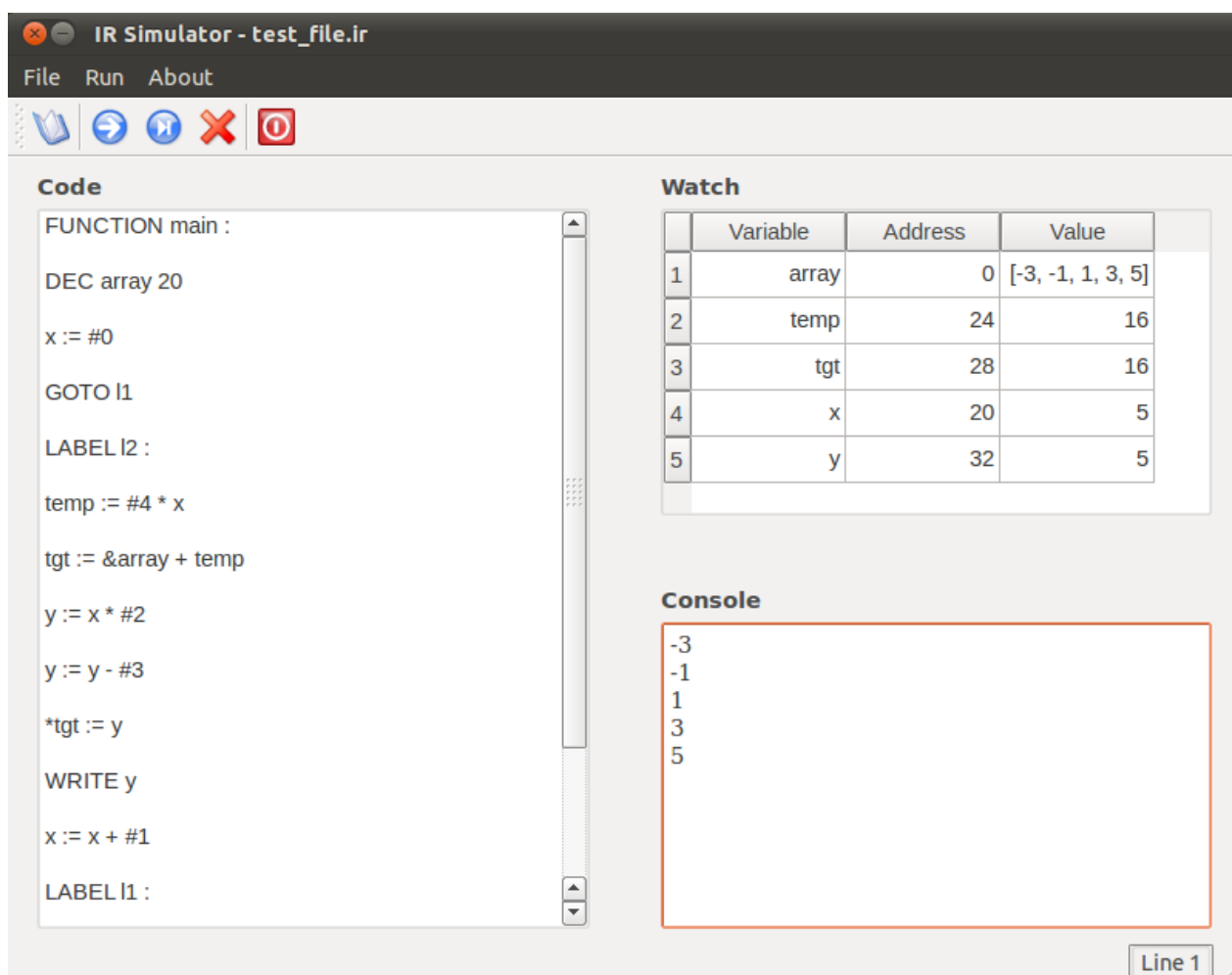
此时代码已经完全被载入到了左侧代码区，而右侧的监视窗口也已经初始化完毕，所有变量的初始值默认都为 0。此时你就可以单击工具栏上的  按钮或通过菜单选择 Run->Run（快捷键 F5）直接运行代码，或者单击  按钮/菜单项 Run->Step（快捷键 F8）来对代码进行单步执行。单击  按钮/菜单项 Run->Stop 来停止运行并初始化运行状态。如果运行出现错误，那么程序将弹出对话框提示出错的行号以及原因（一般的出错原因都是因为访问了一个不存在的内存地址，或者在某个函数中缺少 RETURN 语句等）；如果一切正常，你将会看到如下对话框：



这提示我们代码运行已经正常结束，并且本次运行总共执行了 85 条指令。由于本次实习会考察优化中间代码的性能，如果你想要拿到这些分数那么最好想方设法使 Total instructions 后面跟的这个数字尽可能小。

另外，关于这个虚拟机程序还有几点需要注意：

- ♦ 在模拟运行前请确保你输出的中间代码不会陷入死循环或者无穷递归中。虚拟机程序并不会对这些情况进行判断，因此一旦当它执行了包含死循环的代码，你就可以直接考虑如何将虚拟机强制退出了。
- ♦ 互相对应的 **ARG** 语句和 **PARAM** 语句数量一定要相等，否则有可能会出现问题。
- ♦ 由于程序实现上的原因，单步执行时虚拟机并不会在 **GOTO**、**IF**、**RETURN** 等与跳转相关的语句上停留，而是会将控制直接转移到跳转目标那里，遇到这种情况不要认为是虚拟机出现了问题。
- ♦ 该虚拟机的存储模型极其简单：在读入一段中间代码之后，它会按顺序统计代码中出现的变量的个数和大小，随后根据变量所占的总空间大小开辟一段从 0 开始编址的连续的线性空间（上限为 1MB，超出 1MB 的话程序会持续出现内存访问错误，此时建议重新载入源文件）。监视窗口中每个变量的“**Address**”域就是指该变量在这段连续空间中的位置，而函数调用中的参数和返回值则不会占用这块空间而是会使用单独的一个调用栈。使用如此简单的存储模型的意义在于，它可以使我们的精力集中到生成的中间代码本身的行为上，而不必去关心像多级存储、调用栈布置这些本次实习涉及不到的问题。
- ♦ 使用 **DEC** 语句定义的变量在监视窗中的显示和普通的变量是有区别的：



注意监视窗中变量 **array** 的值：与其它变量不同，**array** 中的内容被一对中括号[]给包裹起来了。有

时候,因为有时DEC出来的变量内容较多,Value一栏可能无法完全显示,这时你可以用鼠标调整Value栏的宽度使得所有内容都能被显示出来。

- ♦ 1.02版之后的虚拟程序修改了变量地址的计算方式。所有函数内部的局部变量现在都只在程序运行到该函数体之后才会去为其分配存储空间,在此之前无论是其地址还是值在Watch窗口中都将显示为N/A。存储空间采用由低地址到高地址的栈式分配方式,递归调用的局部变量将不会影响到上层函数的相应变量的值。如果想要修改上层函数中变量的值,需要向被调用的函数传递想要修改的变量的地址作为参数,也就是我们常说的call by reference。不过,Watch窗口中显示的变量总是当前这一层变量的值,只要不退回上一层,我们就无法看到上层函数局部变量的值是多少。