



课程项目

燕言言

QQ: 2214871526

邮箱: yanyanthunder@foxmail.com





课程项目



■ 课程项目

主要内容：实验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实验任务，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。

课程项目总共分为五个阶段：词法和语法分析、语义分析、**中间代码生成**、目标代码生成以及中间代码优化。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。其中，目标代码生成以及中间代码优化均基于第三次中间代码生成。

实验助教：

燕言言

QQ: 2214871526

邮箱: yanyanthunder@foxmail.com

何天行

QQ: 976792132

邮箱: 976792132@qq.com



提纲



■ 课程项目

- 实验内容
- 中间代码
- 实验设计

■ 虚拟机

- 虚拟机部署及使用



实验三



■ 实验目标

实验三内容是在词法分析、语法分析和语义分析程序的基础上，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。

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



```
1  FUNCTION main :
2  READ t1
3  var4 := t1
4  t2 := var4
5  t3 := #0
6  IF t2 > t3 GOTO label1
7  GOTO label2
8  LABEL label1 :
9  t4 := #1
10 WRITE t4
11 LABEL label2 :
12 t5 := #0
13 RETURN t5
```



实验三



■ C—假设

- 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
- 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
- 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
- 没有全局变量的使用，并且所有变量均不重名
- 函数不会返回结构体或数组类型的值
- 函数只会进行一次定义（没有函数声明）
- 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）



实验三



■ 输入、输出格式

➤ 输入格式

- 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

➤ 输出格式

- 中间代码生成器需要将运行结果输出到中间代码文件中，每行一条中间代码
- 如果源程序包含多个函数定义，则需通过FUNCTION语句将这些函数隔开
- 对每个特定的输入，并不存在唯一正确的输出。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，以“正确性”为主。第五次实验主要考察中间代码优化

- 第五次实验——中间代码优化涵盖了实验三中需要的优化技术



实验三



■ 选做内容

- 要求3.1：修改前面对C++源代码的假设2和3
 - 可以出现结构体类型的变量（但不会有结构体变量之间直接赋值）
 - 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）
- 要求3.2：修改前面对C++源代码的假设2和3
 - 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类值）



提纲



■ 课程项目

- 实验内容
- 中间代码
- 实验设计

■ 虚拟机

- 虚拟机部署及使用



实验三



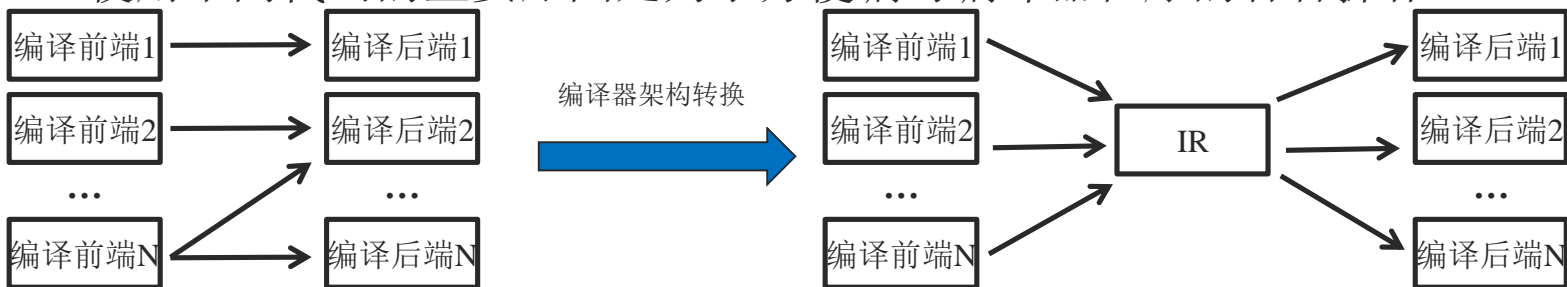
中间代码

编译器里最核心的数据结构之一就是中间代码（IR）。中间代码的定义会影响编译器实现的复杂度、运行效率以及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——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
- 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**）

中间代码作用：

- 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





实验三

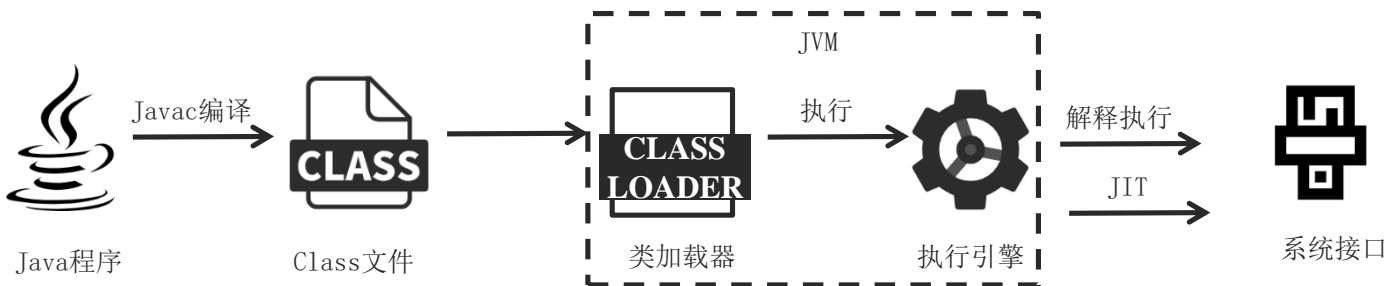


■ 中间代码

编译器里最核心的数据结构之一就是中间代码（IR）。任何语言编写的程序最终都需要经过编译器编译成机器码才能被计算机执行。

以Java字节码(Write Once, Run Anywhere)为例：

- Java程序经过javac编译以后生成对应的字节码
- Java 字节码是沟通 JVM 和 Java 程序的桥梁





实验三

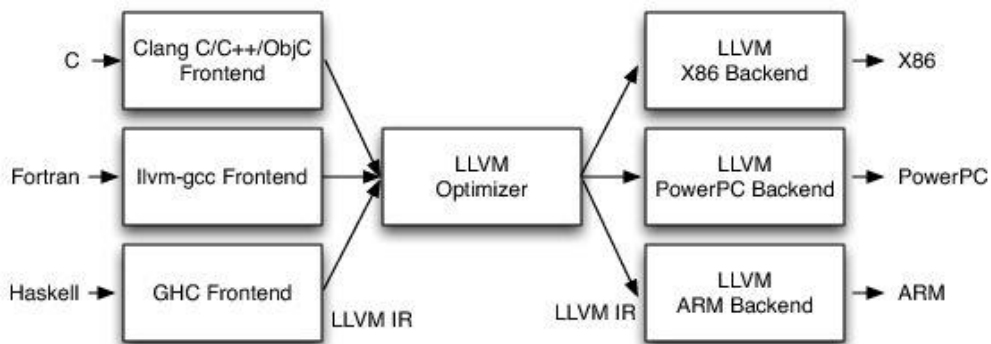


■ 中间代码

编译器里最核心的数据结构之一就是中间代码（IR）。传统的编译器将程序编译为汇编代码，然后调用各个平台汇编器编译为特定目标平台机器代码。引入中间表示后，就可以解耦编译器前端和后端。

以LLVM IR为例：

- 可以将C、Fortran等语言的源代码编译成LLVM中间代码（LLVM IR）
- LLVM后端对LLVM IR进行优化
- 后端编译优化后的LLVM IR到相应的平台的二进制程序





实验三



■ 中间代码

编译器里最核心的数据结构之一就是中间代码（IR）。对于一些动态类型语言如Python，JavaScript，大部分动态类型语言通常采用解释执行的方式执行源程序

以Python标准实现CPython生成执行字节码为例：

- CPython对Python词法语法语义分析，生成字节码
- 基于生成的字节码及其上下文，CPython栈式虚拟机创建栈帧
- CPython虚拟机加载栈帧，解释并执行运行时栈上的Python字节码





实验三



中间代码形式及操作规范

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 。
$x := \text{CALL } f$	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。



实验三



■ 中间代码形式

- LABEL: 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
- FUNCTION: 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
 - 赋值号左边的x一定是一个变量或者临时变量
 - 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 (需要在其前面添加“#”符号)
- 算术运算操作包括加、减、乘、除四种操作
- 赋值号右边的变量可以添加“&”符号对其进行取地址操作
- “*” 用于解引用



实验三



■ 中间代码形式

- 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
 - 有条件跳转语句先确定两个操作数**x**和**y**之间的关系，关系成立再跳转
- 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，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



实验三



■ 中间代码形式

- 与函数调用有关的语句包括**CALL**、**PARAM**和**ARG**三种
 - **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
 - 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，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
 - 结构体或数组采用引用传递方式
- 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
 - **WRITE**语句可将一个整型变量的值写到控制台上
 - 在实验三中，在符号表中预先添加**read**和 **write**这两个预定义的函数
 - **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）



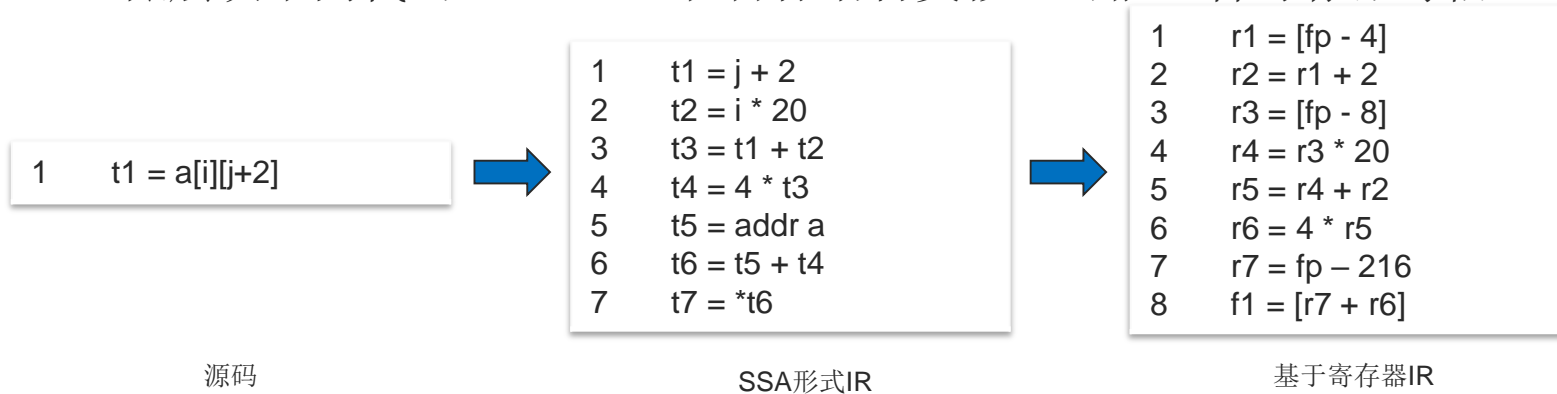
实验三



■ 中间代码分类一

中间代码设计和实现多种多样，不同编译器中间码通常不一样，同一编译器可以采用多种中间代码。从中间代码表现细节上，可以对其划分：

- 高层次中间代码(HIR)：和源码类似，保留了语言特性，可用于现骨干性分析和解释执行。比如数组、循环
- 中间层次中间代码(MIR)：介于源程序和目标语言之间，较难设计，但是可以进行相应的优化处理
- 低层次中间代码(LIR)：和目标语言类似，可能包含寄存器等信息





实验三



■ 中间代码分类二

中间代码设计和实现多种多样，不同编译器中间码通常不一样，同一编译器可以采用多种中间代码。从中间代码表示形式上，可以对其划分：

- 图形中间代码(**Graphical IR**)：此种IR将源程序信息嵌入到一张图中，以节点和边等元素组织代码信息。比如**AST**
- 线形中间代码 (**Linear IR**)：类似于某种抽象计算机的一个简单指令集。比如，汇编语言中语句和语句之间就是线性关系
- 混合型中间代码 (**Hybrid IR**)：混合了图形和线型两种中间代码，吸收前两种IR的优点
 - 中间代码由基本块组成，块内部采用线形表示，块之间采用图表示



实验三



■ 中间代码——线形

实验三中，可以逐条翻译源程序，并将中间结果保存至内存。等到全部翻译完成，就可以输出中间结果。线形IR可以用多种方式表示：

- 数组。数组中每一个元素表示一条中间代码。易于编程，但灵活性较低
- 指针数组。数组中每一个元素指向一条中间代码记录结构体。较为灵活
- 链表。可以使用双向链表表示中间码。虽然实现相对复杂，但是可扩展性较强

树形IR与AST类似

- 靠近树根的高层部分中间代码抽象层次较高，而靠近树根的低层次部分中间代码更加具体。
 - 可以参考语法树实现（不过我们实验一接住了bison实现了语法分析器，因此树形IR需要自己设计实现底层结构



实验三



表达式

translate Exp(Exp, sym_table, place) = case Exp of	
INT	value = get_value(INT) return [place := #value]
ID	variable = lookup(sym_table, ID) return [place := variable.name]
Exp1 ASSIGNOP Exp2 (Exp1 → ID)	variable = lookup(sym_table, Exp1.ID) t1 = new_temp() code1 = translate_Exp(Exp2, sym_table, t1) code2 = [variable.name := t1] + [place := variable.name] return code1 + code2
Exp1 PLUS Exp2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp1, sym_table, t1) code2 = translate_Exp(Exp2, sym_table, t2) code3 = [place := t1 + t2] return code1 + code2 + code3
MINUS Exp1	t1 = new_temp() code1 = translate_Exp(Exp1, sym_table, t1) code2 = [place := #0 - t1] return code1 + code2
Exp1 RELOP Exp2	label1 = new_label() label2 = new_label() code0 = [place := #0]
NOT Exp1	code1 = translate_Cond(Exp, label1, label2, sym_table)
Exp1 AND/OR Exp2	code2 = [LABEL label1] + [place := #1] return code0 + code1 + code2 + [LABEL label2]

- 为每个主要的语法单元“X”都设计相应的翻译函数“translate_X”。遍历语法树，调用对应的“translate_X”函数



实验三



语句

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]

- C—支持的语句类型：表达式语句、复合语句、返回语句、跳转语句、循环语句
- 条件跳转在翻译条件表达式时生成，跳转目标包括true和false两个



实验三



■ 条件表达式

- 条件表达式涉及关系运算符、逻辑运算符，需考虑每个运算符的优先级：

translate_Cond(Exp, label_true, label_false, sym_table) = case Exp of	
Exp ₁ RELOP Exp ₂	<pre>t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp1, sym_table, t1) code2 = translate_Exp(Exp2, sym_table, t2) op = get_relop(RELOP); code3 = [IF t1 op t2 GOTO label_true] return code1 + code2 + code3 + [GOTO label_false]</pre>
NOT Exp ₁	<pre>return translate_Cond(Exp1, label_false, label_true, sym_table)</pre>
Exp ₁ AND Exp ₂	<pre>label1 = new_label() code1 = translate_Cond(Exp1, label1, label_false, sym_table) code2 = translate_Cond(Exp2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2</pre>
Exp ₁ OR Exp ₂	<pre>label1 = new_label() code1 = translate_Cond(Exp1, label_true, label1, sym_table) code2 = translate_Cond(Exp2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2</pre>
(other cases)	<pre>t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [IF t1 != #0 GOTO label_true] return code1 + code2 + [GOTO label_false]</pre>



实验三



函数调用

translate Exp(Exp, sym_table, place) = case Exp of	
ID LP RP	<pre>function = lookup(sym_table, ID) if (function.name == "read") return [READ place] return [place := CALL function.name]</pre>
ID LP Args RP	<pre>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]</pre>
translate Args(Args, sym_table, arg_list) = case Args of	
Exp	<pre>t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list return code1</pre>
Exp COMMA Args ₁	<pre>t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list code2 = translate_Args(Args₁, sym_table, arg_list) return code1 + code2</pre>

- 我们定义了两个内置函数read和write，其声明可添加到符号表
- 调用translate_Args，生成实参IR。实参以列表传递（如Python可以用元组传递实参以及作为返回值）。没有参数可以传递空的列表



实验三



■ 中间代码生成——数组与结构体

- C—数组实现采用了最简单的C风格。访问数组的元素需要计算其对应的内存地址。以三维数组为例，为了访问`array[i][j][k]`，我们需要计算元素相对于数组首地址偏移量

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

- 结构体访问方式和数组类似，比如访问某个域。我们需要找到首地址，然后找到对应的域

$$\text{ADDR}(\text{st.field}_n) = \text{ADDR}(\text{st}) + \sum_{t=0}^{n-1} \text{SIZEOF}(\text{st.field}_t)$$



■ 实验三任务分配

选做分类	队伍编号	选做内容
选做内容一	3, 14, 2, 20, 26, 8, 40, 42, 9, 19, 34, 4, 12, 5, 6, 7, 38, 16, 1, 31, 44, 39, 22, 47	修改假设2和3，C--支持结构体类型变量（但不会有结构体变量之间的赋值）及结构体变量可作为函数参数（但不会作为返回值）
选做内容二	41, 18, 45, 17, 21, 13, 30, 10, 23, 37, 29, 24, 25, 35, 27, 46, 28, 33, 11, 43, 36, 32, 15	修改假设2和3，C--支持一维数组类型变量作为函数参数（但不会作为返回值）及高维数组（但不会作为函数参数或返回值）



提纲



■ 课程项目

- 第三次实验介绍
- 中间代码
- 实验设计

■ 虚拟机

- 虚拟机部署
- 虚拟机使用



实验设计——中间代码生成



中间代码生成

- 左图所示的C--源码是符合文法定义的，我们需要将其翻译为中间代码，如右图所示

```
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 }
```

C--源码

```
1 FUNCTION main :
2 READ t1
3 v1 := t1
4 t2 := v1
5 t3 := #0
6 IF t2 > t3 GOTO label1
7 GOTO label2
8 LABEL label1 :
9 t4 := #1
10 WRITE t4
11 GOTO label3
12 LABEL label2 :
13 t5 := v1
14 t6 := #0
15 IF t5 < t6 GOTO label4
16 GOTO label5
17 LABEL label4 :
18 t8 := #1
19 t7 := #0 - t8
20 WRITE t7
21 GOTO label6
22 LABEL label5 :
23 t9 := #0
24 WRITE t9
25 LABEL label6 :
26 LABEL label3 :
27 t10 := #0
28 RETURN t10
```

可能生成的一种中间代码



实验设计——函数入口



■ 输入输出处理

- 我们的中间代码生成器需要接受一个C--源代码及一个输出中间代码文件名。词法、语法及语义分析后，开始生成中间代码

```
1  int main(int argc, char** argv) {
2      FILE* irFile; // 输出的中间代码文件
3      if (argc <= 1) return 1;
4      FILE* cmmFile = fopen(argv[1], "r");
5      if (!f) { perror(argv[1]); return 1; }
6      yyrestart(f);
7      yyparse(); //语法分析
8      if (!errorflag)
9          semantic(Root); // 语义分析
10     CodeList codelisthead = Intercode(Root); // 中间代码生成
11     if(argv[2] == NULL){ff = fopen("output", "w");}
12     else
13         ff =fopen(argv[2], "w"); // 构造输出中间代码文件
14     print_IR(codelisthead, ff); // 写入生成的中间代码
15     fclose(ff); // 关闭文件
16     return 0;
17 }
```

Main函数定义



实验设计——中间代码表示



■ 数据结构

➤ 首先，我们定义操作数的结构

```
1 struct _Operand{
2     enum{
3         Em_VARIABLE,    // 变量 (var)
4         Em_CONSTANT,    // 常量 (#1)
5         Em_ADDRESS,     // 地址 (&var)
6         Em_LABEL,       // 标签 (LABEL label1:)
7         Em_ARR,         // 数组 (arr[2])
8         Em_STRUCT,      // 结构体 (struct Point p)
9         Em_TEMP         // 临时变量 (t1)
10    } kind;
11    union{
12        int varno;        // 变量定义的序号
13        int labelno;     // 标签序号
14        int val;         // 操作数的值
15        int tempno;      // 临时变量序号 (唯一性)
16    } u;
17    Type type;           // 计算数组、结构体占用size
18    int para;            // 标识函数参数
19};
```

操作数结构体定义

```
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};
```

指导书给出的结构



实验设计——中间代码表示



■ 数据结构

➤ 接下来，我们定义中间代码的表示结构

```
1  struct _InterCode{
2      enum{
3          IC_ASSIGN,
4          IC_LABEL,
5          IC_PLUS,
6          ...
7          IC_CALL,
8          IC_PARAM,
9          IC_READ,
10         IC_WRITE,
11         IC_RETURN,
12         ...} kind;
13     union{
14         Operand op;
15         char *func;
16         struct{Operand right, left; } assign;
17         struct{Operand result, op1, op2; } binop; //三地址代码
18         ...
19         struct{Operand result; char *func;} call;
20     }u;
21 };
```

```
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 }
```

← 指导书给出的结构

中间代码数据结构



实验设计——中间代码表示



数据结构

最后，我们定义中间代码列表的表示结构

```
1 struct _CodeList{
2     InterCode code;      // 中间代码列表实现方式
3     CodeList prev, next;
4 };
5
6 struct _ArgList{         // 参数列表实现方式
7     Operand args;
8     ArgList next;
9 };
10
11 struct _Variable{       // 变量的实现方式
12     char* name;
13     Operand op;
14     Variable next;
15 };
16 int var_num, label_num, temp_num; // 新变量/新标签/新临时变量编号
17 CodeList code_head, code_tail;   // 双链表的首部和尾部
18 Variable var_head, var_tail;     // 变量表的首部和尾部
```

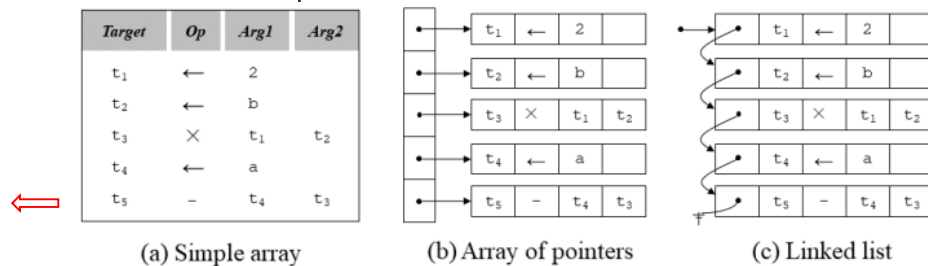


图5. 表示线性IR的三种基本数据结构。
指导书给出的结构

中间代码双链表实现方式



实验设计——中间代码表示



■ 中间代码翻译

- 中间代码翻译是遍历语法分析树，因此接口可以这么实现

```
1  CodeList Intercode(struct Node* Root){
2      首先判断AST是否为空;
3      接着判断AST结构是否正确（满足C-文法要求）;
4      if(Root==NULL){return NULL;}
5      if(strcmp(Root->nodeName, "Program")!= 0 ) return NULL;
6      初始化全局变量中间代码变量等列表;
7      code_head = code_tail = NULL;
8      var_tail = var_head = NULL;
9      设定变量、标签、临时变量标号;
10     var_num = 1; label_num = 1; temp_num = 1;
11     遍历AST的节点，依次调用相应的translate_x函数;
12     struct Node* node = Root -> firstChild;  // ExtDefList
13     while(node!= NULL){
14         node = node ->firstChild;
15         insert_code(translate_ExtDef(node));
16         递归遍历兄弟节点;
17         node = node->Sibc;
18     }
19     return code_head;
20 }
```

中间代码生成入口函数



实验设计——中间代码表示



中间代码翻译

➤ 遍历ExtDef节点，处理函数声明

```
1  CodeList translate_ExtDef(struct Node* ExtDef){
2      检查语法树是否为空;
3      检查是否为“;” ;
4      if(strcmp(ExtDef->firstChild->Sibc->nodeName, "SEMI")== 0 )
5          return NULL;
6      检查是否有全局标识符定义;
7      if(strcmp(ExtDef->firstChild->Sibc->nodeName, "ExtDecList")== 0 )
8          return NULL;
9      // 依次检查是否是函数定义;
10     // 实验三不包含全局变量，结构体定义也已经在实验二中处理并添加到符号表中了
11     if(checkp(ExtDef, 3, "Specifier", "FunDec", "CompSt")){
12         CodeList c1 = translate_FunDec(ExtDef->firstChild->Sibc);
13         CodeList c2 = translate_CompSt(ExtDef->firstChild->Sibc->Sibc);
14         // 合并函数声明和函数体
15         return concatenate(c1,c2);
16     } else{
17         fprintf(stderr, "error ExtDef!\n"); return NULL;
18     }
19 }
```

```
Program → ExtDefList
ExtDefList → ExtDef ExtDefList
           | ε
ExtDef → Specifier ExtDecList SEMI
       | Specifier SEMI
       | Specifier FunDec CompSt
```

中间代码生成入口函数



实验设计——中间代码表示



中间代码翻译

➤ 遍历FunDec节点，处理函数声明

```
1  CodeList translate_FunDec(struct Node* FunDec){
2      if(checkp(FunDec, 3, "ID", "LP", "RP")){// 处理无参函数
3          // 初始化函数信息，获取函数名
4          InterCode ic = new_InterCode(IC_FUNC);
5          ic->u.func = FunDec ->firstChild -> Valstr;
6          CodeList cl = new_CodeList(ic);
7          return cl;
8      } else if(checkp(FunDec, 4, "ID", "LP", "VarList", "RP")){// 处理有参函数
9          // 初始化函数信息，获取函数名
10         InterCode ic = new_InterCode(IR_FUNC);
11         ic->u.func = FunDec ->firstChild -> Valstr;
12         CodeList cl = new_CodeList(ic);
13         // 获取参数列表
14         FieldList params = Type_get(FunDec->firstChild->Valstr)->u.function->param;
15         while (params!=NULL)
16             {
17                 // 构造函数参数列表判断参数类型（选做内容中需要支持结构体和数组作为参数）
18                 if(params->type->kind==BASIC){
19                     ...
20                 }else{ // 处理数组或者结构体
21                     }
22             }
23     } else{ fprintf(stderr, "error FunDec!\n"); return NULL;}
24 }
```

FunDec → ID LP VarList RP
| ID LP RP

函数声明处理



实验设计——中间代码表示



■ 中间代码翻译

- 遍历**CompSt**节点，处理函数体或者复合语句

```
1  CodeList translate_CompSt(struct Node* CompSt){
2      if(CompSt == NULL)
3          return NULL;
4      if(strcmp(CompSt->firstChild->Sibc->nodeName, "RC")==0)
5          return NULL;
6      // 处理DefList节点
7      CodeList c1 = NULL;
8      // 处理StmtList节点
9      CodeList c2 = NULL;
10     if(strcmp(CompSt->firstChild->Sibc->nodeName, "DefList")==0)
11         c1 = translate_DefList(CompSt->firstChild->Sibc);
12     else
13         c1 = translate_StmtList(CompSt->firstChild->Sibc);
14     if(strcmp(CompSt->firstChild->Sibc->Sibc->nodeName, "StmtList")==0)
15         c2 = translate_StmtList(CompSt->firstChild->Sibc->Sibc);
16     // 合并c1和c2
17     return concatenate(c1, c2);
18 }
```

CompSt → LC DefList StmtList RC



实验设计——中间代码表示



中间代码翻译

- **StmtSt**中比较复杂的就是**Stmt**，所以我们遍历**Stmt**节点，处理语句，生成中间代码（参考实验指导表3）

```
1  CodeList translate_Stmt(struct Node* Stmt){
2      if(strcmp(Stmt->firstChild->nodeName, "CompSt")==0){ // 处理复合语句
3          return translate_CompSt(Stmt->firstChild);
4      }else if(strcmp(Stmt->firstChild->nodeName, "Exp")==0){ // 处理表达式
5          return translate_Exp(Stmt->firstChild, NULL);
6      }else if(strcmp(Stmt->firstChild->nodeName, "RETURN")==0){ // 处理返回语句
7          Operand t1 = new_temp();
8          CodeList c1 = translate_Exp(Stmt->firstChild->Sibc, t1);
9          CodeList c2 = create_code_Op(t1, IC_RETURN);
10         return concatenate(c1, c2);
11     }else if(strcmp(Stmt->firstChild->nodeName, "IF")==0){ // IF ELSE
12         if(Stmt->firstChild->Sibc->Sibc->Sibc->Sibc->Sibc==NULL){ // true分支
13             ...
14         }else{ // true/false分支
15             ...
16         }
17     }else if(strcmp(Stmt->firstChild->nodeName, "WHILE")==0){ // while语句处理
18         ...
19     }
20     return NULL;
21 }
```

```
CompSt → LC DefList StmtList RC
StmtList → Stmt StmtList
| ε
Stmt → Exp SEMI
| CompSt
| RETURN Exp SEMI
| IF LP Exp RP Stmt
| IF LP Exp RP Stmt ELSE Stmt
| WHILE LP Exp RP Stmt
```

语句处理



实验设计——中间代码表示



中间代码翻译

- 表达式的产生式较多，工作量相对较大，我们会依次处理

```
1 CodeList translate_Exp(struct Node* Exp, Operand place){ // place表示值
2   if(strcmp(Exp->firstChild->nodeName,"INT")==0){ //INT
3     int val = Exp->firstChild->Valint;
4     InterCode ic = new_InterCode(IC_ASSIGN);
5     ic->u.assign.left = place; // 构建左值
6     ic->u.assign.right = new_constant(val); // 构建常量
7     return new_CodeList(ic);
8   }
9   if(strcmp(Exp->firstChild->nodeName,"ID")==0){
10    if(Exp->firstChild->Sibc==NULL){ // 处理标识符
11      Operand op = lookup_var(Exp->firstChild->Valstr); // 查找标识符
12      if(op->kind==Em_ARR || op->kind==Em_STRUCT ){ // 数组or结构体返回地址
13        InterCode ic = new_InterCode(IC_GET_ADDR);
14        ic->u.assign.left = place;
15        ic->u.assign.right = op;
16        return new_CodeList(ic);
17      }else{ // 一般变量直接赋值
18      }
19    }
20    ...
21  }
```

Expressions

```
Exp → Exp ASSIGNOP Exp
| Exp AND Exp
| Exp OR Exp
| Exp RELOP Exp
| Exp PLUS Exp
| Exp MINUS Exp
| Exp STAR Exp
| Exp DIV Exp
| LP Exp RP
| MINUS Exp
| NOT Exp
| ID LP Args RP
| ID LP RP
| Exp LB Exp RB
| Exp DOT ID
| ID
| INT
| FLOAT
Args → Exp COMMA Args
| Exp
```

整数及标识符处理



实验设计——中间代码表示



中间代码翻译

➤ 表达式的产生式较多，工作量相对较大，我们会依次处理

```
1 CodeList translate_Exp(struct Node* Exp, Operand place){ // place表示值
2     if(strcmp(Exp->firstChild->Sibc->nodeName,"LP")==0){ // 函数调用
3         if( strcmp(Exp->firstChild->Sibc->Sibc->nodeName,"Args")==0 ){
4             Function fun = Type_get(Exp->firstChild->Valstr)->u.function;
5             // 获取函数参数列表并处理实参
6             CodeList c1 = translate_Args(Exp->firstChild->Sibc->Sibc,&argList );
7             // 是否是write函数
8             if(strcmp(fun->name,"write")==0){
9                 InterCode ic = new_InterCode(IC_WRITE);
10                ic->u.op = argList->args;
11                return concatenate(c1,new_CodeList(ic));
12            }else{ // 用户自定义函数
13                CodeList c2 = NULL;
14                while (argList!=NULL)
15                {
16                    创建实参压栈IR;
17                }
18                创建函数调用IR;
19                合并操作;
20            }
21        }
22    }
23    ...
24 }
```

函数调用处理

Expressions

```
Exp → Exp ASSIGNOP Exp
| Exp AND Exp
| Exp OR Exp
| Exp RELOP Exp
| Exp PLUS Exp
| Exp MINUS Exp
| Exp STAR Exp
| Exp DIV Exp
| LP Exp RP
| MINUS Exp
| NOT Exp
| ID LP Args RP
| ID LP RP
| Exp LB Exp RB
| Exp DOT ID
| ID
| INT
| FLOAT
Args → Exp COMMA Args
| Exp
```



实验设计——中间代码表示



中间代码翻译

- 表达式的产生式较多，工作量相对较大，我们会依次处理

```
1  CodeList translate_Exp(struct Node* Exp, Operand place){ // place表示值
2      if(strcmp(Exp->firstChild->Sibc->nodeName,"DOT")==0) { // 结构体访问
3          // 构建临时变量
4          Operand baseAddr = new_temp();
5          baseAddr->kind = Em_ADDRESS;
6          CodeList c1 = translate_Exp(Exp->firstChild, baseAddr); // 获取结构体的地址
7          InterCode ic = new_InterCode(IR_PLUS);
8          Operand tmp = new_temp();
9          tmp->kind = Em_ADDRESS;
10         ic->u.binop.result = tmp;
11         ic->u.binop.op1 = baseAddr;
12         // 获取结构体域的偏移量
13         ic->u.binop.op2 = size_get_instruct(Exp->firstChild->Sibc->Sibc->Valstr);
14         CodeList c2 = new_CodeList(ic);
15         InterCode ic2 = new_InterCode(IC_ASSIGN);
16         ic2->u.assign.left = place;
17         ic2->u.assign.right = tmp;
18         CodeList c3 = new_CodeList(ic2); // 获取结构体域的地址
19         return CodePlus(3,c1,c2,c3);
20     } // Exp DOT ID 结构体
21     ...
22 }
```

Expressions

```
Exp → Exp ASSIGNOP Exp
    | Exp AND Exp
    | Exp OR Exp
    | Exp RELOP Exp
    | Exp PLUS Exp
    | Exp MINUS Exp
    | Exp STAR Exp
    | Exp DIV Exp
    | LP Exp RP
    | MINUS Exp
    | NOT Exp
    | ID LP Args RP
    | ID LP RP
    | Exp LB Exp RB
    | Exp DOT ID
    | ID
    | INT
    | FLOAT
Args → Exp COMMA Args
    | Exp
```

结构体域的地址计算



实验设计——中间代码表示



中间代码翻译

➤ 表达式的产生式较多，工作量相对较大，我们会依次处理

```
1 CodeList translate_Exp(struct Node* Exp, Operand place){ // place表示值
2   if(strcmp(Exp->firstChild->Sibc->nodeName,"LB")==0){ // 数组访问
3     if(strcmp(Exp->firstChild->firstChild->nodeName, "ID")==0){ // 非结构体数组
4       // 获取数组地址
5       Operand v1 = lookup_var(Exp->firstChild->firstChild->Valstr);
6       Operand baseAddr = new_temp();
7       baseAddr->kind = Em_ADDRESS;
8       InterCode ic;
9       ic->u.assign.left = baseAddr;
10      ic->u.assign.right = v1;
11      CodeList c0 = new_CodeList(ic);
12      Operand t1 = new_temp(); // t1 地址偏移量
13      Operand t2 = new_temp(); // t2 结果地址
14      t2->kind = Em_ADDRESS;
15      CodeList c1 = translate_Exp(Exp->firstChild->Sibc->Sibc,t1);
16      // 地址偏移量*4
17      ic = new_InterCode(IR_MUL);
18      ic->u.binop.result = t1;
19      ic->u.binop.op1 = t1;
20      ic->u.binop.op2 = new_constant(4);
21      ...
22    }
23  }
```

数组访问

Expressions

```
Exp → Exp ASSIGNOP Exp
| Exp AND Exp
| Exp OR Exp
| Exp RELOP Exp
| Exp PLUS Exp
| Exp MINUS Exp
| Exp STAR Exp
| Exp DIV Exp
| LP Exp RP
| MINUS Exp
| NOT Exp
| ID LP Args RP
| ID LP RP
| Exp LB Exp RB
| Exp DOT ID
| ID
| INT
| FLOAT
Args → Exp COMMA Args
| Exp
```




实验设计——中间代码表示



中间代码翻译

➤ 接下来，我们处理函数的参数

```
1  CodeList translate_Args(struct Node* Args, ArgList* arg_list){
2      Operand t1 = new_temp();
3      CodeList c1 = translate_Exp(Args->firstChild,t1);
4      // 返回时，根据函数参数依次访问判断，是取地址还是取值，只有数组和结构体要取地址
5      // write函数的参数为NULL，要避免空指针访问
6      // fdom指向函数形参列表
7      if(fdom!=NULL &&(fdom->type->kind == ARRAY ||
8          fdom->type->kind == STRUCTURE )){t1->kind = Em_ADDRESS;}; //当前param类型
9      ArgList newArgList = malloc(sizeof(struct _ArgList));
10     newArgList->args = t1;
11     newArgList->next = *arg_list;
12     *arg_list = newArgList;
13     if(Args->firstChild->Sibc==NULL){
14         return c1; // 只有一个参数直接返回
15     }else{
16         if(fdom!=NULL) fdom = fdom->tail; // write函数的参数为NULL，要避免空指针访问
17         // 处理下一对函数形参及实参
18         CodeList c2 = translate_Args(Args->firstChild->Sibc->Sibc,arg_list);
19         return merge(c1,c2);
20     }
21 }
```

Expressions

```
Exp → Exp ASSIGNOP Exp
    | Exp AND Exp
    | Exp OR Exp
    | Exp RELOP Exp
    | Exp PLUS Exp
    | Exp MINUS Exp
    | Exp STAR Exp
    | Exp DIV Exp
    | LP Exp RP
    | MINUS Exp
    | NOT Exp
    | ID LP Args RP
    | ID LP RP
    | Exp LB Exp RB
    | Exp DOT ID
    | ID
    | INT
    | FLOAT
Args → Exp COMMA Args
    | Exp
```

数组访问



实验设计——中间代码表示



中间代码翻译

➤ 接下来，我们处理函数的参数

```
1  CodeList translate_Cond(struct Node *Exp, Operand label_true, Operand label_false){
2      if(strcmp(Exp->firstChild->nodeName,"NOT")==0){ // 处理NOT逻辑运算符
3          return translate_Cond(Exp->firstChild->Sibc,label_false,label_true);
4      }else if(Exp->firstChild->Sibc==NULL){ // 处理表达式
5          Operand t1 = new_temp();
6          CodeList c1 = translate_Exp(Exp,t1);
7          InterCode ic = new_InterCode(IC_IFGOTO);
8          处理true分支;
9          strcpy(ic->u.if_goto.relop, "!=");
10         CodeList c2 = new_CodeList(ic);
11         CodeList gf = create_code_Op(label_false,IC_GOTO);
12         return CodePlus(3,c1,c2,gf);
13     } else{
14         if(strcmp(Exp->firstChild->Sibc->nodeName,"RELOP")==0){ // 处理关系运算符
15             ...
16         }else if(strcmp(Exp->firstChild->Sibc->nodeName,"AND")==0){ // 处理逻辑与
17             Operand label1 = new_label(); // 短路运算
18             CodeList c1 = translate_Cond(Exp->firstChild,label1,label_false);
19             CodeList c2 = translate_Cond(Exp->firstChild->Sibc->Sibc,label_true,label_false);
20             CodeList clabel1 = create_code_Op(label1,IC_LABEL);
21             return CodePlus(3,c1,clabel1,c2);
22         }else if(strcmp(Exp->firstChild->Sibc->nodeName,"OR")==0){ // 逻辑或
23             ...
24     }
```

Expressions

```
Exp → Exp ASSIGNOP Exp
| Exp AND Exp
| Exp OR Exp
| Exp RELOP Exp
| Exp PLUS Exp
| Exp MINUS Exp
| Exp STAR Exp
| Exp DIV Exp
| LP Exp RP
| MINUS Exp
| NOT Exp
| ID LP Args RP
| ID LP RP
| Exp LB Exp RB
| Exp DOT ID
| ID
| INT
| FLOAT
Args → Exp COMMA Args
| Exp
```



实验设计——中间代码生成



中间代码生成

- 左图所示的C--源码是符合文法定义的，我们需要将其翻译为中间代码，如右图所示

```
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  }
```

必做样例1

```
1  FUNCTION main :
2  READ t1
3  v1 := t1
4  t2 := v1
5  t3 := #0
6  IF t2 > t3 GOTO label1
7  GOTO label2
8  LABEL label1 :
9  t4 := #1
10 WRITE t4
11 GOTO label3
12 LABEL label2 :
13 t5 := v1
14 t6 := #0
15 IF t5 < t6 GOTO label4
16 GOTO label5
17 LABEL label4 :
18 t8 := #1
19 t7 := #0 - t8
20 WRITE t7
21 GOTO label6
22 LABEL label5 :
23 t9 := #0
24 WRITE t9
25 LABEL label6 :
26 LABEL label13 :
27 t10 := #0
28 RETURN t10
```

main函数代码



实验设计——中间代码生成



中间代码生成

- 左图所示的C--源码是符合文法定义的，我们需要将其翻译为中间代码，如右图所示

```
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 }
```

必做样例2

```
1  FUNCTION fact :
2  PARAM v1
3  t1 := v1
4  t2 := #1
5  IF t1 == t2 GOTO label1
6  GOTO label2
7  LABEL label1 :
8  t3 := v1
9  RETURN t3
10 GOTO label3
11 LABEL label2 :
12 t5 := v1
13 t8 := v1
14 t9 := #1
15 t7 := t8 - t9
16 ARG t7
17 t6 := CALL fact
18 t4 := t5 * t6
19 RETURN t4
```

fact函数

```
20 FUNCTION main :
21 READ t10
22 v2 := t10
23 t11 := v2
24 t12 := #1
25 IF t11 > t12 GOTO label4
26 GOTO label5
27 LABEL label4 :
28 t14 := v2
29 ARG t14
30 t13 := CALL fact
31 v3 := t13
32 GOTO label6
33 LABEL label5 :
34 t15 := #1
35 v3 := t15
36 LABEL label6 :
37 t16 := v3
38 WRITE t16
39 t17 := #0
40 RETURN t17:
```

main函数



实验设计——中间代码生成



中间代码生成

- 左图所示的C--源码是符合文法定义的，我们需要将其翻译为中间代码，如右图所示

```
1 struct Operands
2 {
3     int o1;
4     int o2;
5 };
6 int add(struct Operands temp)
7 {
8     return (temp.o1 + temp.o2);
9 }
10 int main()
11 {
12     int n;
13     struct Operands op;
14     op.o1 = 1;
15     op.o2 = 2;
16     n = add(op);
17     write(n);
18     return 0;
19 }
```

选做样例1

```
1 FUNCTION add :
2 PARAM v1
3 t4 := v1
4 t5 := t4 + #0
5 t2 := *t5
6 t6 := v1
7 t7 := t6 + #4
8 t3 := *t7
9 t1 := t2 + t3
10 RETURN t1
```

fact函数

```
11 FUNCTION main :
12 DEC v2 8
13 t9 := &v2
14 t10 := t9 + #0
15 t8 := t10
16 t11 := #1
17 *t8 := t11
18 t13 := &v2
19 t14 := t13 + #4
20 t12 := t14
21 t15 := #2
22 *t12 := t15
23 t17 := &v2
24 ARG t17
25 t16 := CALL add
26 v3 := t16
27 t18 := v3
28 WRITE t18
29 t19 := #0
30 RETURN t19
```

main函数



实验设计——中间代码生成



中间代码生成

- 左图所示的C--源码是符合文法定义的，我们需要将其翻译为中间代码，如右图所示

```
1  int add(int temp[2])
2  {
3      return (temp[0] + temp[1]);
4  }
5  int main()
6  {
7      int op[2];
8      int r[1][2];
9      int i = 0, j = 0;
10     while (i < 2)
11     {
12         while (j < 2)
13         {
14             op[j] = i + j;
15             j = j + 1;
16         }
17         r[0][i] = add(op);
18         write(r[0][i]);
19         i = i + 1;
20         j = 0;
21     }
22     return 0;
23 }
```

选做样例2

```
1  FUNCTION add :
2  PARAM v1
3  t4 := v1
4  t5 := #0
5  t5 := t5 * #4
6  t6 := t4 + t5
7  t2 := *t6
8  t7 := v1
9  t8 := #1
10 t8 := t8 * #4
11 t9 := t7 + t8
12 t3 := *t9
13 t1 := t2 + t3
14 RETURN t1
```

fact函数



实验设计——中间代码生成



中间代码生成

```
15 FUNCTION main :  
16 DEC v2 8  
17 DEC v3 8  
18 v4 := #0  
19 v5 := #0  
20 LABEL label1 :  
21 t10 := v4  
22 t11 := #2  
23 IF t10 < t11 GOTO label2  
24 GOTO label3  
25 LABEL label2 :  
26 LABEL label4 :  
27 t12 := v5  
28 t13 := #2  
29 IF t12 < t13 GOTO label5  
30 GOTO label6  
31 LABEL label5 :  
32 t15 := &v2  
33 t16 := v5  
34 t16 := t16 * #4  
35 t17 := t15 + t16  
36 t14 := t17
```

main函数1

```
37 t19 := v4  
38 t20 := v5  
39 t18 := t19 + t20  
40 *t14 := t18  
41 t22 := v5  
42 t23 := #1  
43 t21 := t22 + t23  
44 v5 := t21  
45 GOTO label4  
46 LABEL label6 :  
47 t25 := &v3  
48 t28 := #0  
49 t26 := #0  
50 t27 := t26 * #8  
51 t28 := t28 + t27  
52 t26 := v4  
53 t27 := t26 * #4  
54 t28 := t28 + t27  
55 t29 := t28 + t25  
56 t24 := t29  
57 t31 := &v2  
58 ARG t31
```

main函数2

```
59 t30 := CALL add  
60 *t24 := t30  
61 t33 := &v3  
62 t36 := #0  
63 t34 := #0  
64 t35 := t34 * #8  
65 t36 := t36 + t35  
66 t34 := v4  
67 t35 := t34 * #4  
68 t36 := t36 + t35  
69 t37 := t36 + t33  
70 t32 := *t37  
71 WRITE t32  
72 t39 := v4  
73 t40 := #1  
74 t38 := t39 + t40  
75 v4 := t38  
76 t41 := #0  
77 v5 := t41  
78 GOTO label1  
79 LABEL label3 :  
80 t42 := #0  
81 RETURN t42
```

main函数3



提纲



■ 课程项目

- 第三次实验介绍
- 中间代码
- 实验设计

■ 虚拟机

- 虚拟机部署及使用

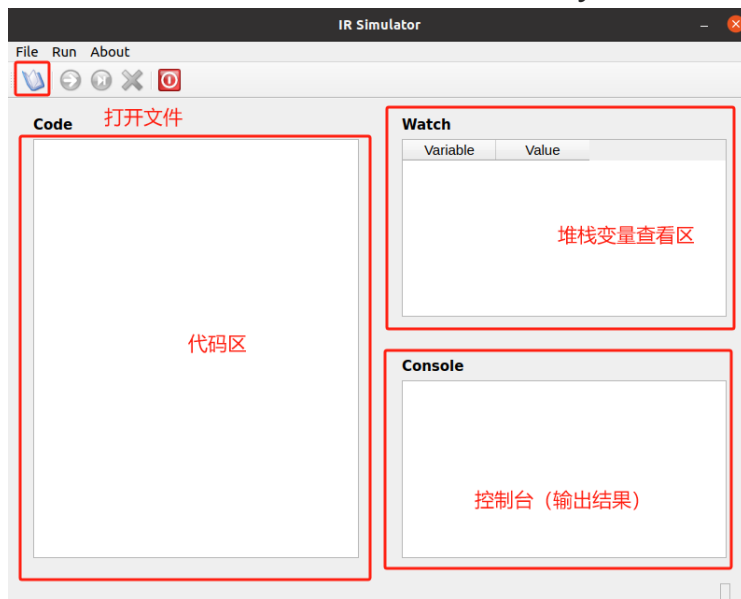


实验三



■ 虚拟机扩展

- 虚拟机(IR Simulator)使用Python3实现（版本 ≥ 3.8 ），依赖于PyQt5。
 - 中间代码解释器，类似于Java VM，Python VM



<https://www.pythonguis.com/installation/install-pyqt-linux/>



实验三



虚拟机依赖环境安装

- 虚拟机(IR Simulator)使用Python3实现（版本 ≥ 3.8 ），依赖于PyQt5 [1]
 - 安装PyQt5: `pip install PyQt5 PyQt5-tools`。如果报错AttributeError: module 'sipbuild.api' has no attribute 'prepare_metadata_for_build_wheel'，可以升级pip版本[2]

```
lulu@ubuntu:~/lab3/irsim$ python3 -m pip install PyQt5 PyQt5-tools
Collecting PyQt5
  Using cached PyQt5-5.15.10.tar.gz (3.2 MB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing wheel metadata ... error
ERROR: Command errored out with exit status 1:
  command: /usr/bin/python3 /tmp/tmpkrquhzhg prepare_metadata_for_build_wheel
/tmp/tmpvuizzh5e
  cwd: /tmp/pip-install-bj_lm5wp/PyQt5
  Complete output (31 lines):
  Traceback (most recent call last):
    File "/tmp/tmpkrquhzhg", line 126, in prepare_metadata_for_build_wheel
      hook = backend.prepare_metadata_for_build_wheel
  AttributeError: module 'sipbuild.api' has no attribute 'prepare_metadata_for_build_wheel'
```

```
lulu@ubuntu:~/lab3/irsim$ python3 -m pip -V
pip 20.0.2 from /usr/lib/python3/dist-packages/pip (python 3.8)
lulu@ubuntu:~/lab3/irsim$ python3 -m pip install --upgrade pip
Collecting pip
  Downloading pip-23.3.1-py3-none-any.whl (2.1 MB)
  | 2.1 MB 583 kB/s
Installing collected packages: pip
Successfully installed pip-23.3.1
lulu@ubuntu:~/lab3/irsim$ python3 -m pip install PyQt5 PyQt5-tools
Defaulting to user installation because normal site-packages is not writeable
Collecting PyQt5
  Downloading PyQt5-5.15.10-cp37-abi3-manylinux_2_17_x86_64.whl.metadata (2.1 kB)
Collecting PyQt5-tools
  Downloading pyqt5_tools-5.15.9.3.3-py3-none-any.whl (29 kB)
Collecting PyQt5-sip<13,>=12.13 (from PyQt5)
  Downloading PyQt5_sip-12.13.0-cp38-cp38-manylinux_2_5_x86_64_manylinux1_x86_64.whl.metadata (504 bytes)
Collecting PyQt5-Qt5>=5.15.2 (from PyQt5)
  Downloading PyQt5_Qt5-5.15.2-py3-none-manylinux2014_x86_64.whl (59.9 MB)
  59.9/59.9 MB 6.1 MB/s eta 0:00:00
```

[1]. <https://www.pythonguis.com/installation/install-pyqt-linux/>

[2]. https://learnshareit.com/attributeerror-module-sipbuild-api-has-no-attribute-prepare_metadata_for_build_wheel/

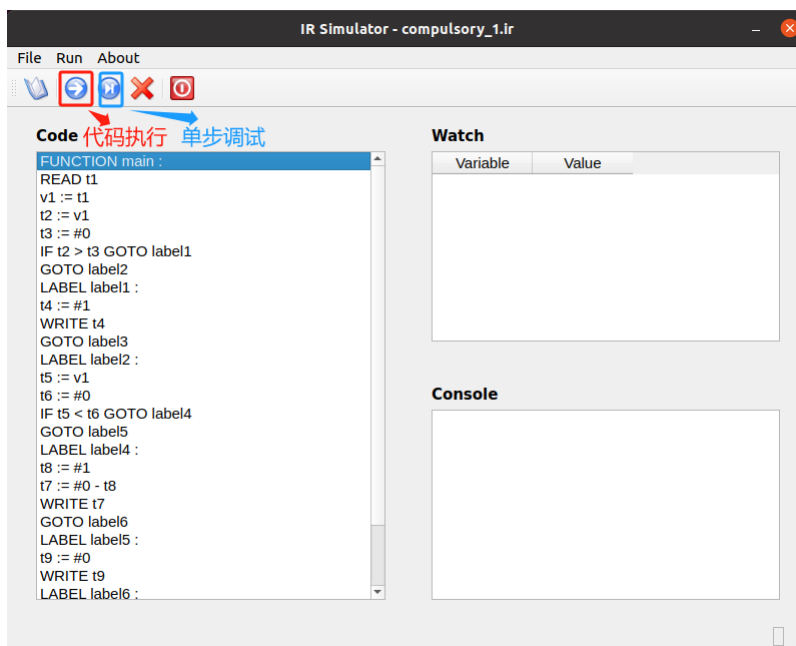


实验三

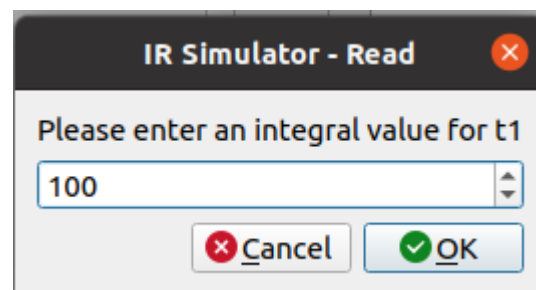


虚拟机扩展

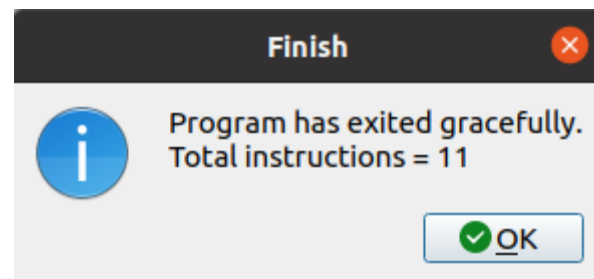
- 虚拟机执行命令: `python3 irsim.py`或`python3 irsim.pyc`
 - 加载IR, 每一个IR都有一个main入口函数。执行完会有相应的指令数提示



程序运行



程序输入



程序运行结束提示信息

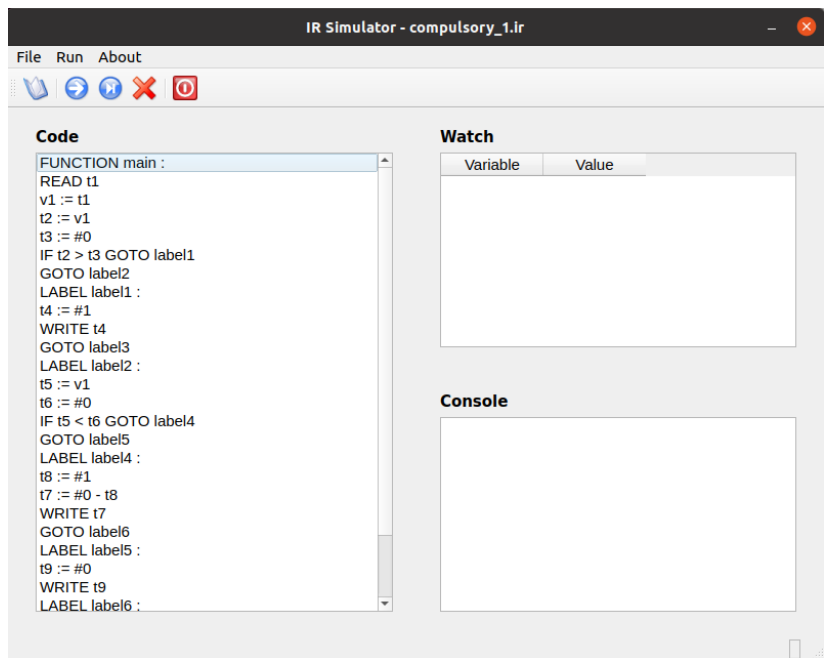


实验三

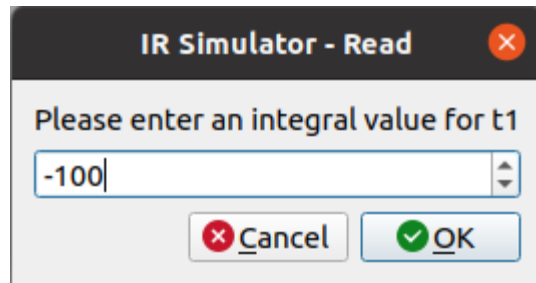


虚拟机使用

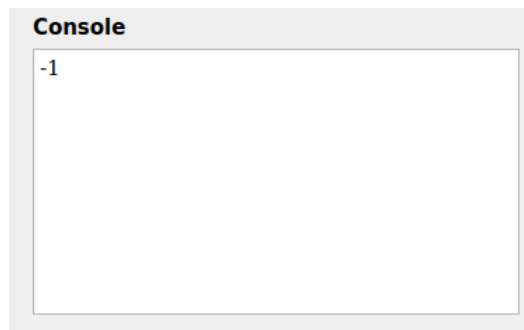
- 虚拟机执行命令: `python3 irsim.py`或`python3 irsim.pyc`
 - 加载IR, 每一个IR都有一个main入口函数。执行完会有相应的指令数提示



必做样例1



程序输入



程序输出结果

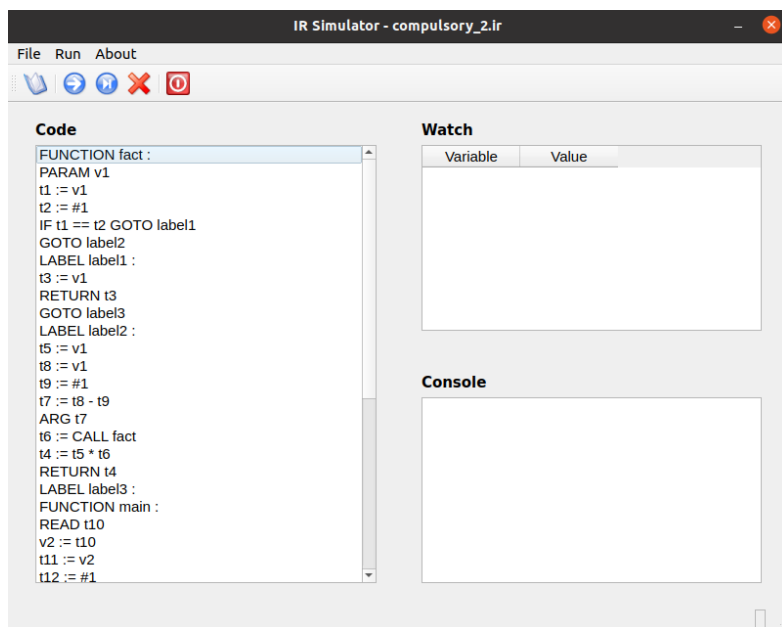


实验三

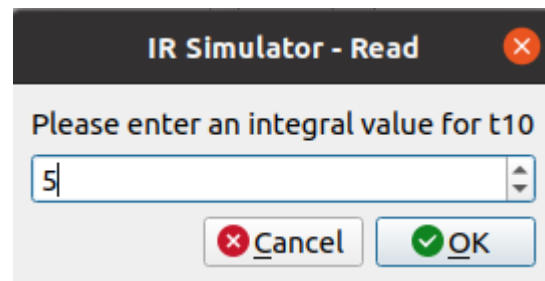


虚拟机使用

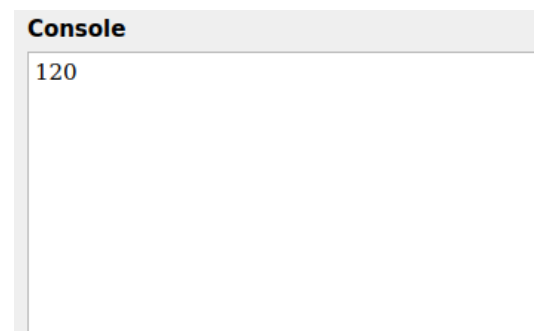
- 虚拟机执行命令: `python3 irsim.py`或`python3 irsim.pyc`
 - 加载IR, 每一个IR都有一个main入口函数。执行完会有相应的指令数提示



必做样例2



程序输入



程序输出结果

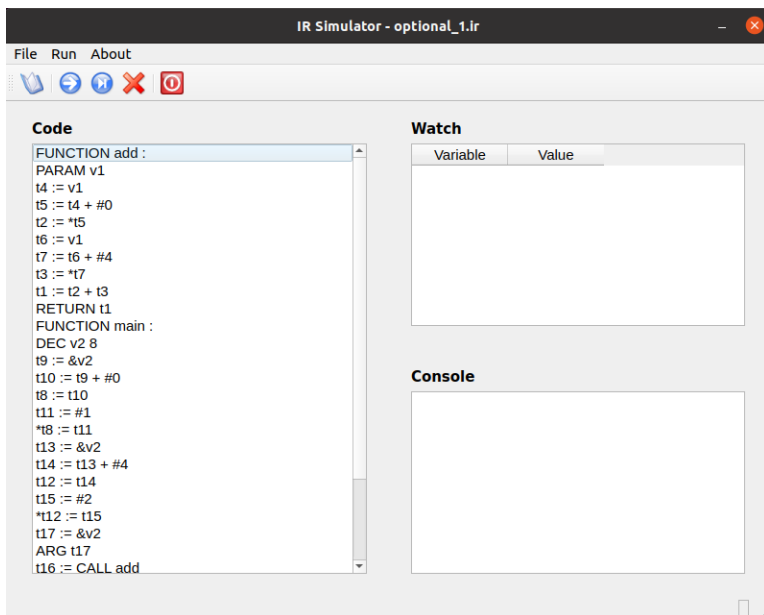


实验三



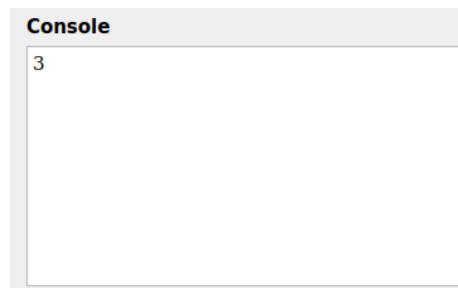
虚拟机使用

- 虚拟机执行命令: `python3 irsim.py`或`python3 irsim.pyc`
 - 加载IR, 每一个IR都有一个main入口函数。执行完会有相应的指令数提示



选做样例1

无程序输入



程序输出结果

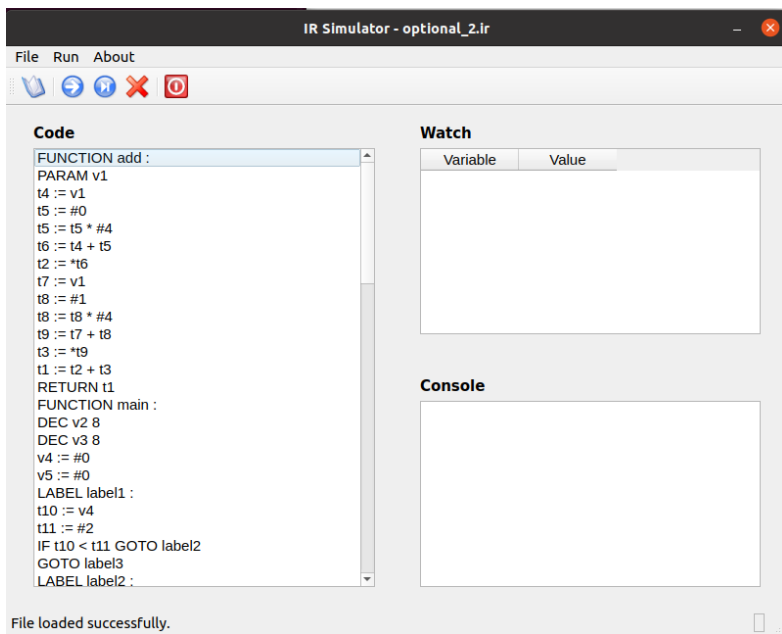


实验三



虚拟机使用

- 虚拟机执行命令: `python3 irsim.py`或`python3 irsim.pyc`
 - 加载IR, 每一个IR都有一个main入口函数。执行完会有相应的指令数提示



选做样例2

无程序输入



程序输出结果

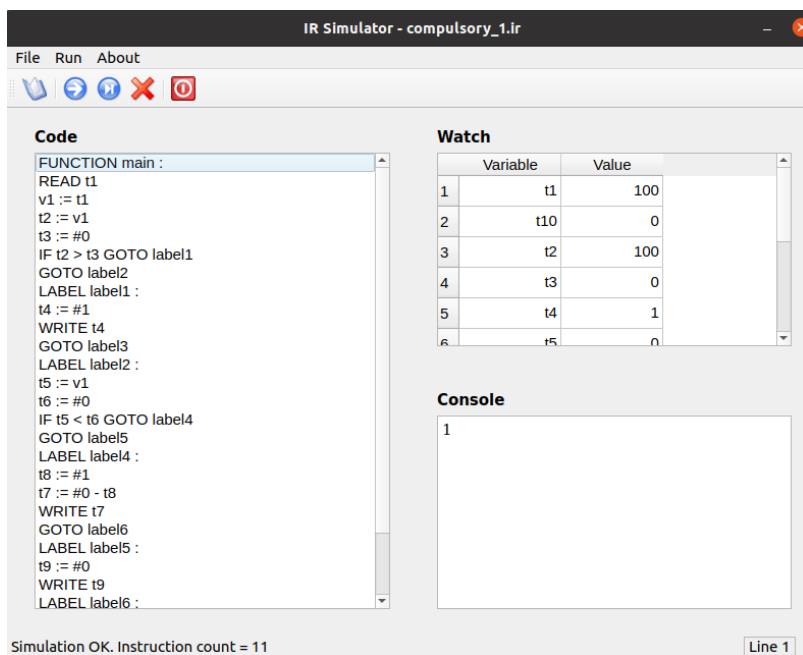



实验三



虚拟机扩展

- 新的虚拟机版本是在原有虚拟机基础上升级改造的，如果大家使用过程中发现了问题，可以联系我或者自行修复（可以有相应的补偿），然后提醒我一下，我把新的虚拟机共享给大家





誠朴雄偉
勵學敦行

谢谢大家