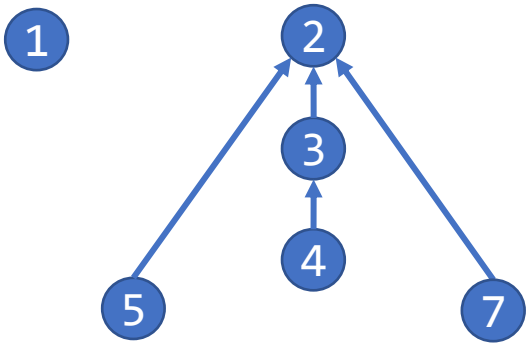
程序切片与规范化

数据依赖

- **数据依赖**（Data Dependency）：程序语句B引用了执行路径上前置语句A中的数据，则称B数据依赖于A
- 数据依赖是一种“定值引用”依赖关系

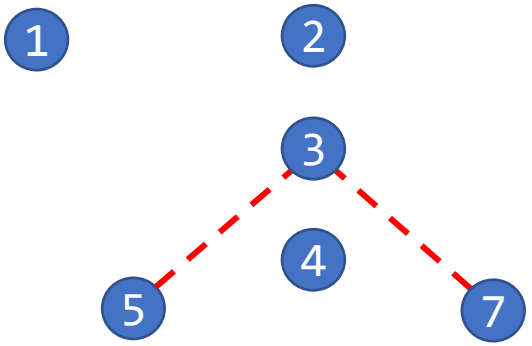
```
1.    str = read1();
2.    str = read2();
3.    valid = is_valid(str);
4.    if (valid) {
5.        foo(str);
6.    } else {
7.        bar(str);
8.    }
```



数据共享

- 数据共享（Data Sharing）：在同一条执行路径上，程序语句A和B均使用了同一个变量或来源于同一处的数据

```
1.    str = read1();
2.    str = read2();
3.    valid = is_valid(str);
4.    if (valid) {
5.        foo(str);
6.    } else {
7.        bar(str);
8.    }
```



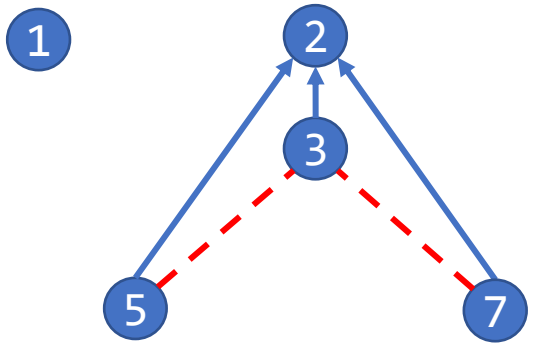
一个函数即为一个事务（Transaction）

- 后期的数据挖掘阶段，将要检查某种模式（如调用函数A的同时调用函数B）出现的频率是否足够高。如果对A和B的调用出现在函数C中，我们为其同时出现频次计1；如果又在函数D中同时被调用，频次加1
- 因此，检查某种模式是否出现，我们是以一个函数的实现作为基本识别单元。在数据挖掘中，将其定义为一个事务（Transaction）
- 对我们的目标而言，一个Transaction应该包含：
  - 我们所关注的程序元素（函数调用与函数返回）【参考第一次实验课的讲解】
  - 程序元素之间的数据依赖关系和数据共享关系

源代码角度的Transaction

- 针对如下的示例代码，在仅保留我们所关注的程序元素及元素关系之后，对应的Transaction如右图所示

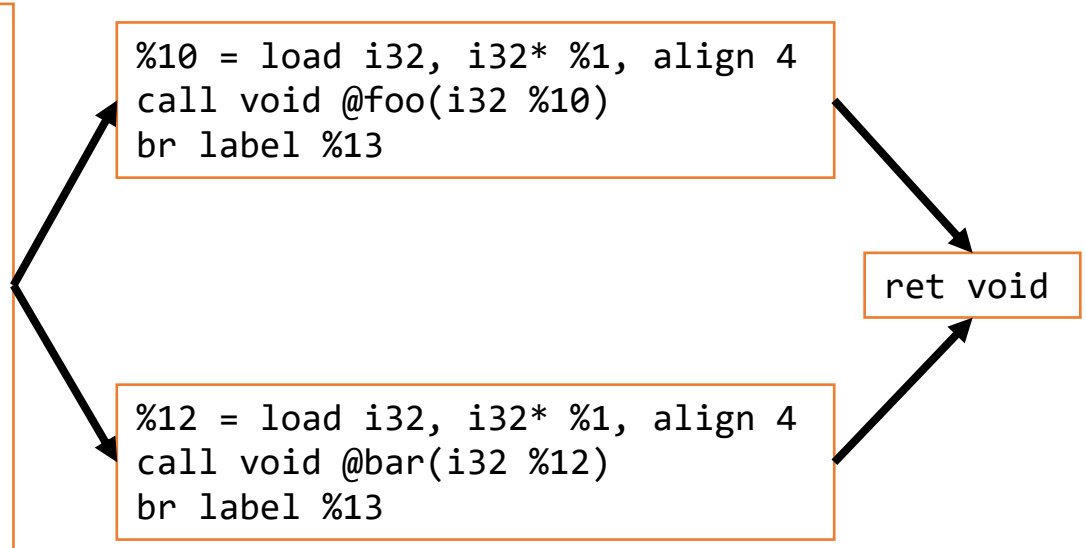
```
1.    str = read1();
2.    str = read2();
3.    valid = is_valid(str);
4.    if (valid) {
5.        foo(str);
6.    } else {
7.        bar(str);
8.    }
```



LLVM IR角度如何构建依赖、共享关系

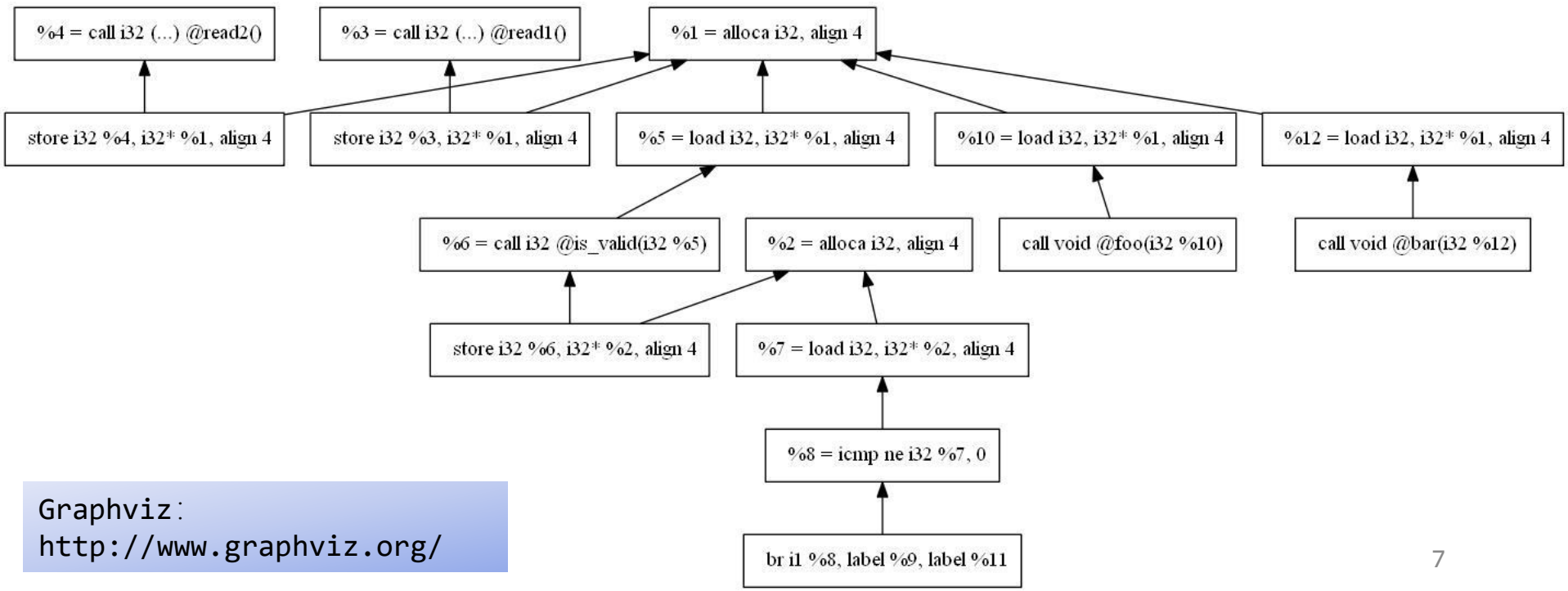
- LLVM IR已经提供了相关的功能，可以在IR上追踪某条指令（语句）是否使用了前置语句所定义的变量
- 下图展示了示例代码的控制流图（CFG）

```
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = call i32 (...) @read1()
store i32 %3, i32* %1, align 4
%4 = call i32 (...) @read2()
store i32 %4, i32* %1, align 4
%5 = load i32, i32* %1, align 4
%6 = call i32 @is_valid(i32 %5)
store i32 %6, i32* %2, align 4
%7 = load i32, i32* %2, align 4
%8 = icmp ne i32 %7, 0
br i1 %8, label %9, label %11
```



LLVM IR所构造的数据依赖关系图

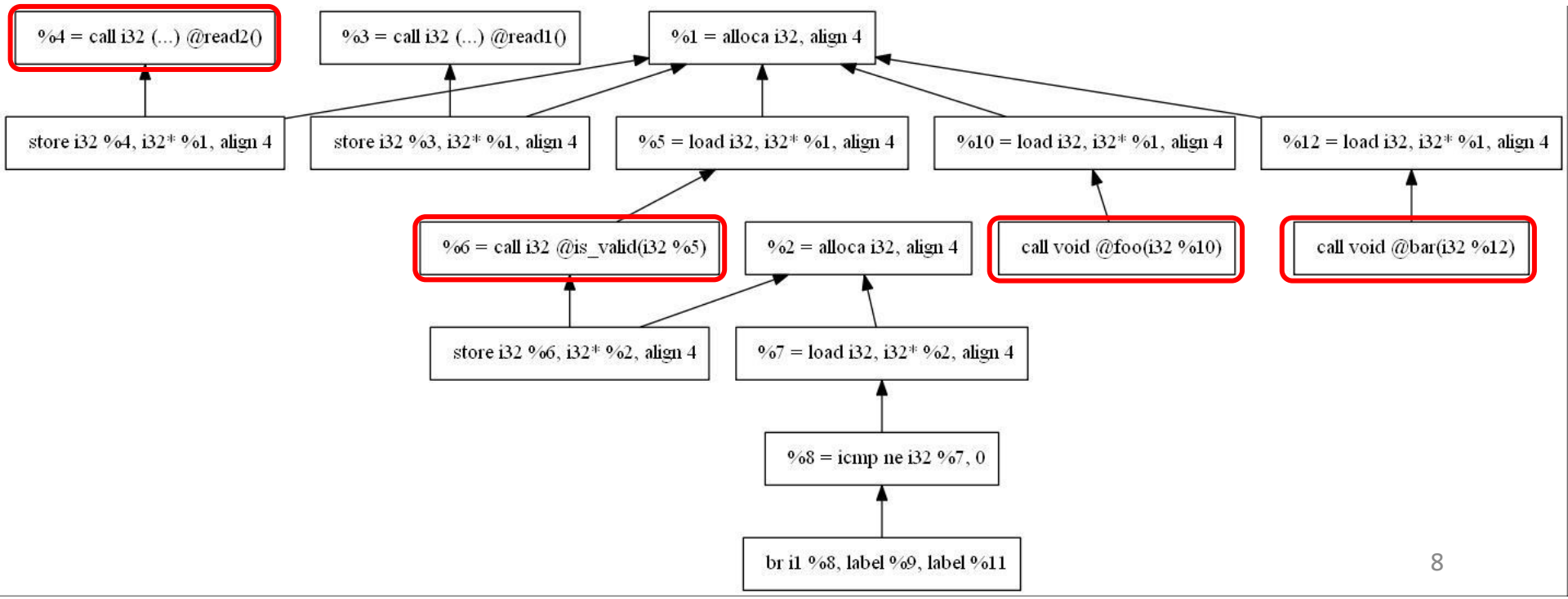
- 通过遍历函数中所有指令，调用相应的API，可以构造出如下的数据依赖图
  - 需要自己创建相关类，利用数组、set、map等数据结构建立图节点、节点联系
  - 下图是编写代码构造出图结构之后，将信息输出为graphviz所支持的dot格式，然后转成图片格式



Graphviz:
http://www.graphviz.org/

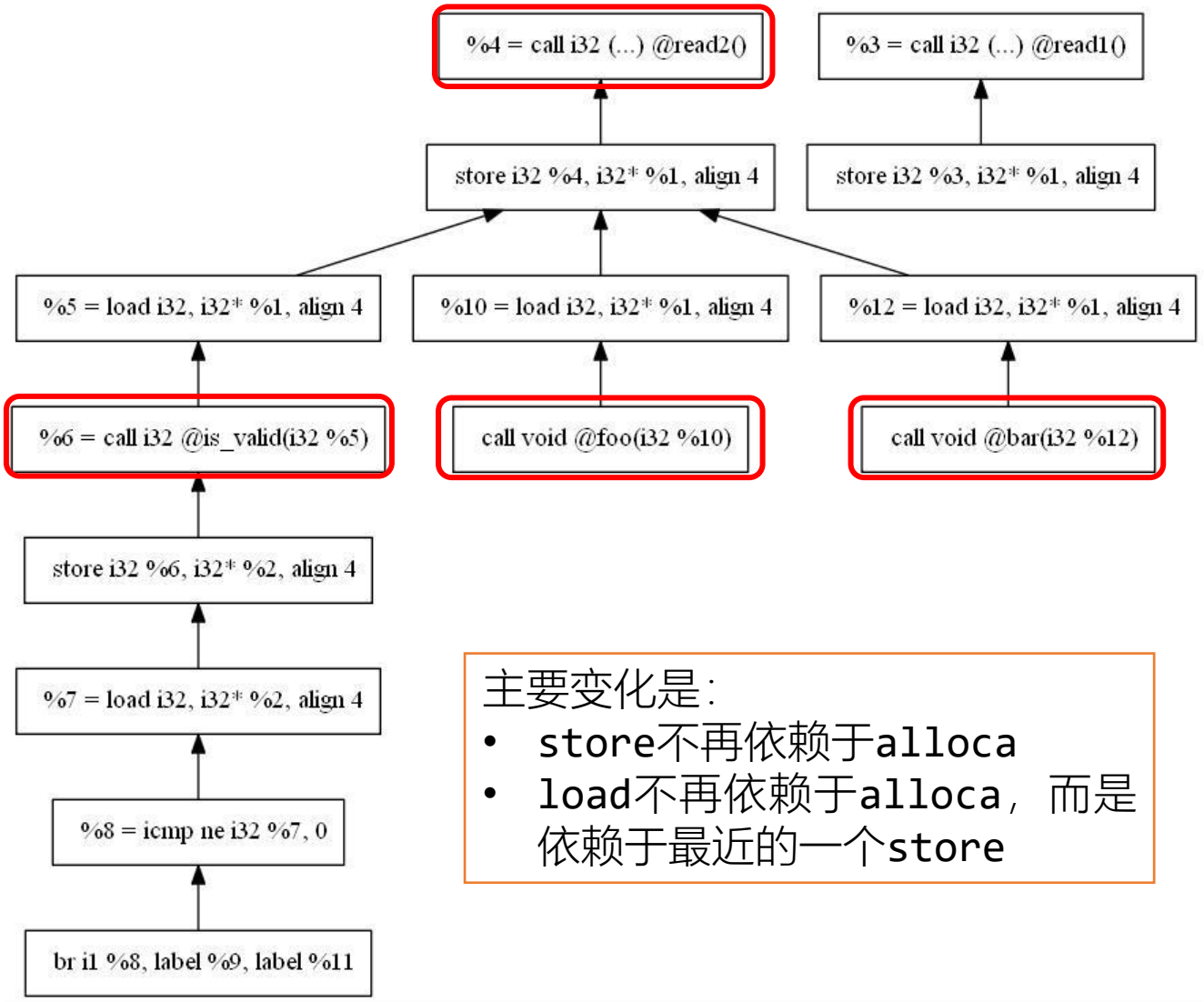
上述数据依赖图存在的问题

- 检查该数据依赖图，可以发现：
  - `foo()`、`bar()`、`is_valid()`语句并不依赖于`read2()`
  - 但是可以较为容易地判断`is_valid()`与`foo()`、`bar()`存在数据共享关系，无法判断`foo()`和`bar()`不应该存在数据共享关系



自己重新建立数据依赖

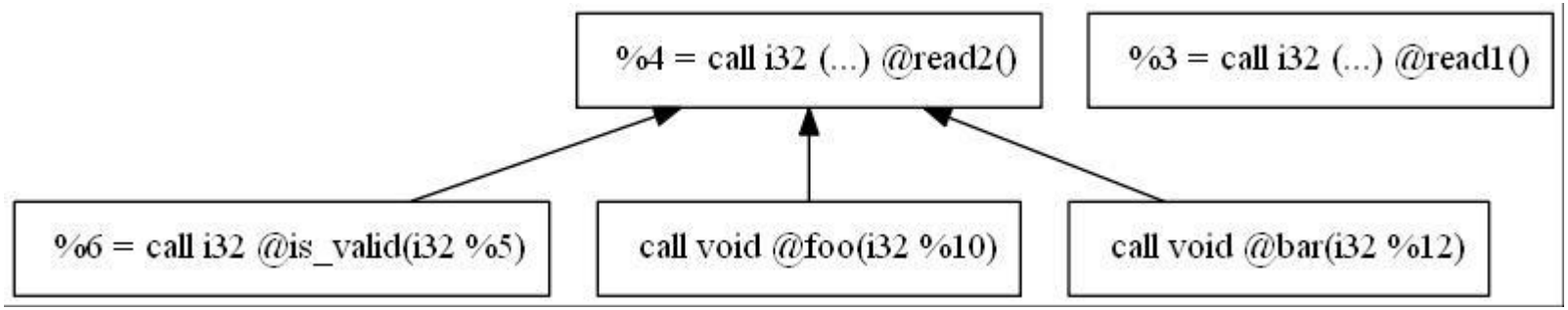
- 下图展示了根据一定方式重构数据依赖图的结果
- 可以看出
  - `is_valid()`、`foo()`、`bar()`均正确地数据依赖于`read2()`



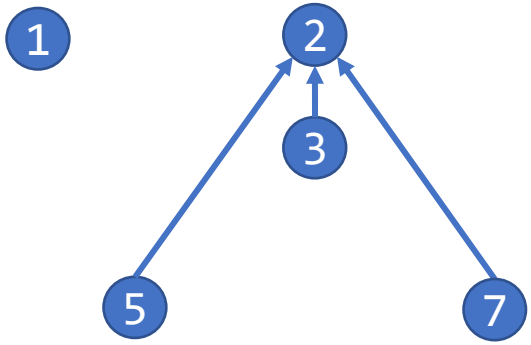
主要变化是：

- `store`不再依赖于`alloca`
- `load`不再依赖于`alloca`，而是依赖于最近的一个`store`

- 下图展示了简化后的数据依赖图（仅保留我们所关注的程序元素——函数调用、有意义的函数返回）




```
1.    str = read1();
2.    str = read2();
3.    valid = is_valid(str);
4.    if (valid) {
5.        foo(str);
6.    } else {
7.        bar(str);
8.    }
```



利用数据流分析构建数据依赖图

- 对原有依赖关系的改变集中于alloca、store、load这几种指令上，下图展示了此种变化
  - 我们认为store指令是对其第二个操作数（下例中的%1）的重新赋值，而不是依赖于%1的定义（alloca指令）

原始IR

```
%1 = alloca i32, align 4

%3 = call i32 (...) @read1()
store i32 %3, i32* %1, align 4

%4 = call i32 (...) @read2()
store i32 %4, i32* %1, align 4

%5 = load i32, i32* %1, align 4
%6 = call i32 @is_valid(i32 %5)
```

新的IR概念（概念上做出改变，但是IR本身不变）

```
%1 = alloca i32, align 4

%3 = call i32 (...) @read1()
store i32 %3, i32* %1, align 4
  ==> *%1 = %3
%4 = call i32 (...) @read2()
store i32 %4, i32* %1, align 4
  ==> *%1 = %4
%5 = load i32, i32* %1, align 4
%6 = call i32 @is_valid(i32 %5)
```

基本块内部的数据流分析

- 在做出上述概念性改变之后，我们可以通过顺序遍历一个基本块内部的所有指令，构造出该基本块内部的数据依赖关系
- 通过对数据流问题求解的方式来进行
- 对每一条语句s，定义其输入、输出状态分别为：IN[s]和OUT[s]
  - 对于前向数据流分析（按指令顺序逐条分析），存在一个传递函数f，使得OUT[s] = f(IN[s])，即：根据输入状态处理当前语句s后得到相应的输出状态
  - 例：对于第一个store指令（Stmt3），我们有
  - IN[stmt3] = {%1 -> Stmt1, %3 -> Stmt2}
  - 在处理完该指令后，我们有 OUT[stmt3] = {%1 -> %3, %3 -> Stmt2}

```
1   %1 = alloca i32, align 4

2   %3 = call i32 (...) @read1()
3   store i32 %3, i32* %1, align 4
       ==> *%1 = %3
```

基本块之间的数据流分析

- 在一个基本块内部，每条语句的输入状态即为前一条语句的输出状态

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], \text{ for all } i = 1, 2, \ldots, n-1$$

- 将语句层面上的数据流分析扩展到基本块上，可以定义一个基本块的输入、输出状态分别为IN[B]、OUT[B]，且存在一个传递函数f_B，使得OUT[B] = f_B(IN[B])
  - 假设基本块有n条语句（s_1, s_2, …, s_n），可知，IN[B]即为当前基本块第一条语句的输入状态IN[s_1]，而OUT[B]为当前基本块的最后一条语句的输出状态OUT[s_n]

基本块之间的数据流分析

- 如果基本块A的后继结点时基本块C，那么C的输入状态必然跟A的输出状态有关，但并不一定是基本块内部语句之间那种直接传递的关系，因为一个基本块的前驱或后继结点不一定是唯一的
- 我们通过求解如下的数据流方程来获取每个基本块的输入、输出状态
  - 每个基本块的输入状态是其所有前驱结点的输出状态的并集

$$\mathrm{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \mathrm{OUT}[P]$$

$$\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$$

IN和OUT有何作用

- 如果某个基本块C中使用了一个变量%x，但是%x并不是在当前基本块中（使用它之前）定义的，即：没有%x = …或store …, i32* %x之类的语句，如果所有基本块的IN和OUT都已经计算好，那么IN中就应该包含了所有可以到达基本块C的对于%x的定义
  - 对于示例代码，最终可以发现%10 = load i32, i32* %1, …语句数据依赖于前一个基本块中的store …, i32* %1, …语句，也就为read2和foo之间建立了数据依赖关系

```
%1 = alloca i32, align 4

%3 = call i32 (...) @read1()
store i32 %3, i32* %1, align 4
  ==> *%1 = %3
%4 = call i32 (...) @read2()
store i32 %4, i32* %1, align 4
  ==> *%1 = %4
%5 = load i32, i32* %1, align 4
%6 = call i32 @is_valid(i32 %5)
```

```
%10 = load i32, i32* %1, align 4
call void @foo(i32 %10)
br label %13
```

Definition和Use

- 定义（Definition）表示对某个变量的数值进行更新；使用（Use）表示使用了某个此前定义过的变量
- 我们需要计算每个基本块定义了哪些变量、使用了哪些变量
  - 后者最终用于完善数据依赖图
- 需要注意，在基本块层面，如果某个变量被重新定义（重赋值或者如前所述的store指令），此后对该变量的使用不再计入Use中；如果一个基本块内对同一个变量进行了多次定义，则只有最后一次定义被计入Definition中（因为后继结点中使用该变量时，不会用到前边几次的、被覆盖了的定义）
- 如何构建一个基本块的Definition与Use：逐条语句分析，根据语句的语义查看是否定义了变量、使用了变量

gen-kill

- 为了方便概念描述，下文将使用数据流分析中的gen-kill模式
- gen：一条语句产生了某个变量的定义，被认为是该语句的gen；同理，一个基本块产生了一系列变量的定义，这些定义点被视作当前基本块的gen集合
- kill：一条语句销毁（杀死）了某变量在此前的所有定义，被认为是该语句的kill；一个基本块销毁（杀死）了某变量在此基本块的所有前驱结点中的定义，这些被销毁的定义点被视作该基本块的kill集合

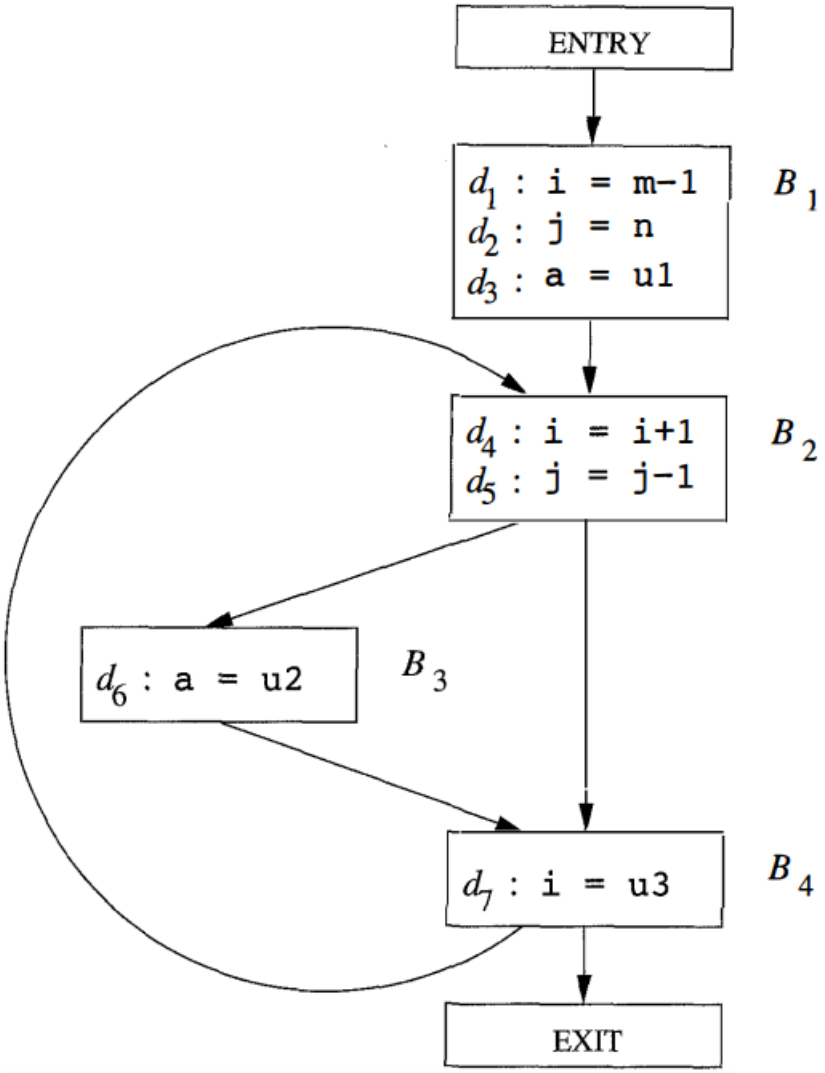
迭代式计算IN和OUT

- 我们推荐使用如下的迭代式算法计算一个函数的IN和OUT
  - 在实际实现中，需要考虑IN、OUT、gen、kill应定义为何种数据结构，整个数据依赖图应如何实现
  - 同时注意gen、kill与前述的Definition、Use的关系
  - 通过该方法获得一个函数的较为完整的数据依赖关系之后，需要去除其中的无关元素，仅保留Transaction所需的部分

1. **for** (每个基本块*B*)　　OUT[*B*] = {}
2. *changed* = true
3. **while** (*changed*)
4. 　　*changed* = false
5. 　　**for** (每个基本块*B*)
6. 　　$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$
7. 　　$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$
8. 　　**end for**
9. 　　若存在基本块*B*，OUT[*B*]相比for循环之前有所改变，则*changed* = true
10. **end while**

示例

- 程序的控制流图如右图所示，计算各基本块的IN、OUT信息
- **注意**：LLVM IR中没有右图中的不包含任何语句的ENTRY和EXIT基本块

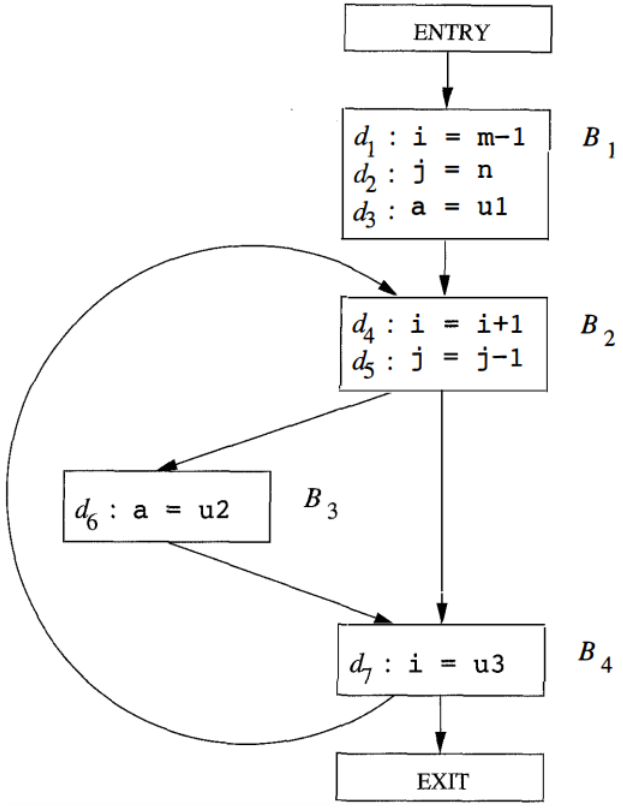


- 初始化

BB	OUT
ENTRY	{}
1	{}
2	{}
3	{}
4	{}
EXIT	{}

- 第一轮

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$$
$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

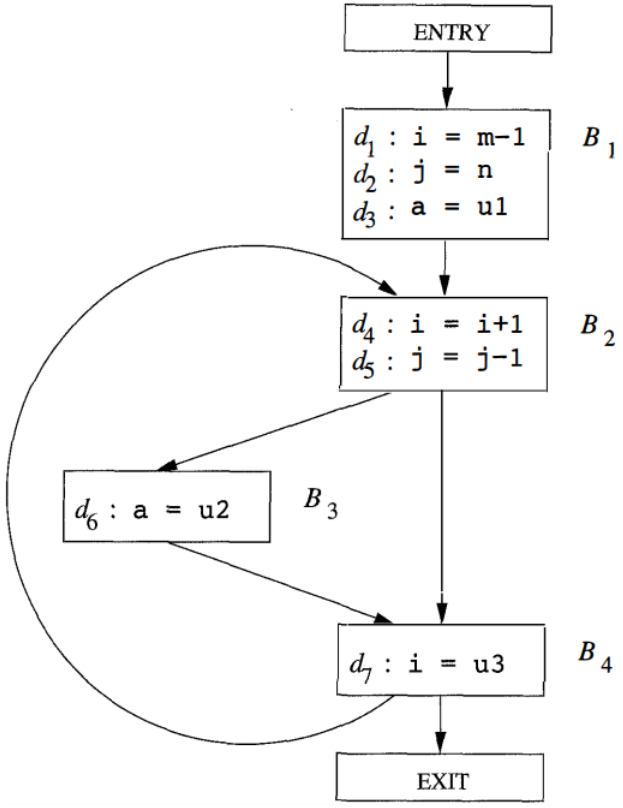


BB	IN	gen	kill	OUT
1		d1, d2, d3		d1, d2, d3
2	d1, d2, d3	d4, d5	d1, d2	d3, d4, d5
3	d3, d4, d5	d6	d3	d4, d5, d6
4	d3, d4, d5, d6	d7	d4	d3, d5, d6, d7
EXIT	d3, d5, d6, d7			d3, d5, d6, d7

所有基本块的OUT相比于初始化结束都有所改变，所以进入下一轮迭代。

- 第二轮

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$$
$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

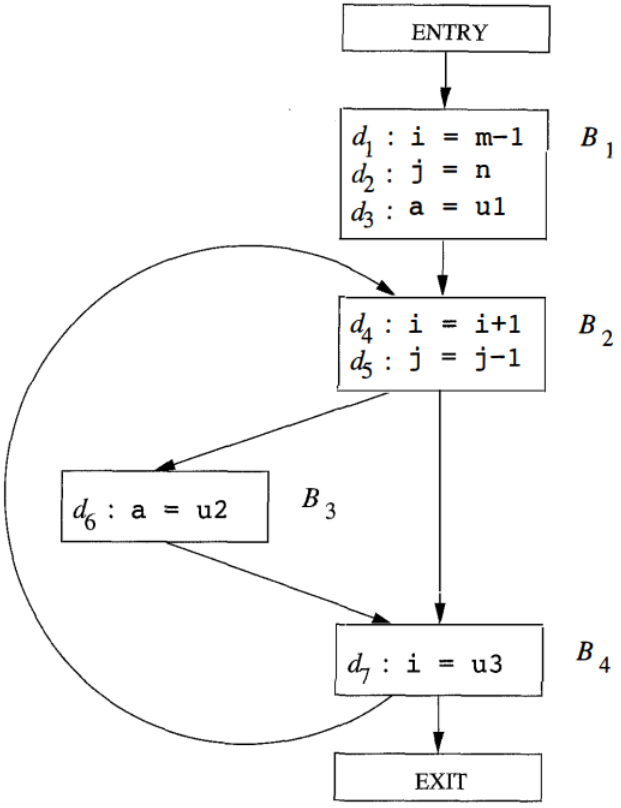


BB	IN	gen	kill	OUT
1		d1, d2, d3		d1, d2, d3
2	d1, d2, d3, d5, d6, d7	d4, d5	d1, d2, d7	d3, d4, d5, d6
3	d3, d4, d5, d6	d6	d3	d4, d5, d6
4	d3, d4, d5, d6, d7	d7	d4	d3, d5, d6, d7
EXIT	d3, d5, d6, d7			d3, d5, d6, d7

基本块2的OUT相比于第一轮迭代结束都有所改变(增加了d6)，所以进入下一轮迭代。

- 第三轮

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$$
$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

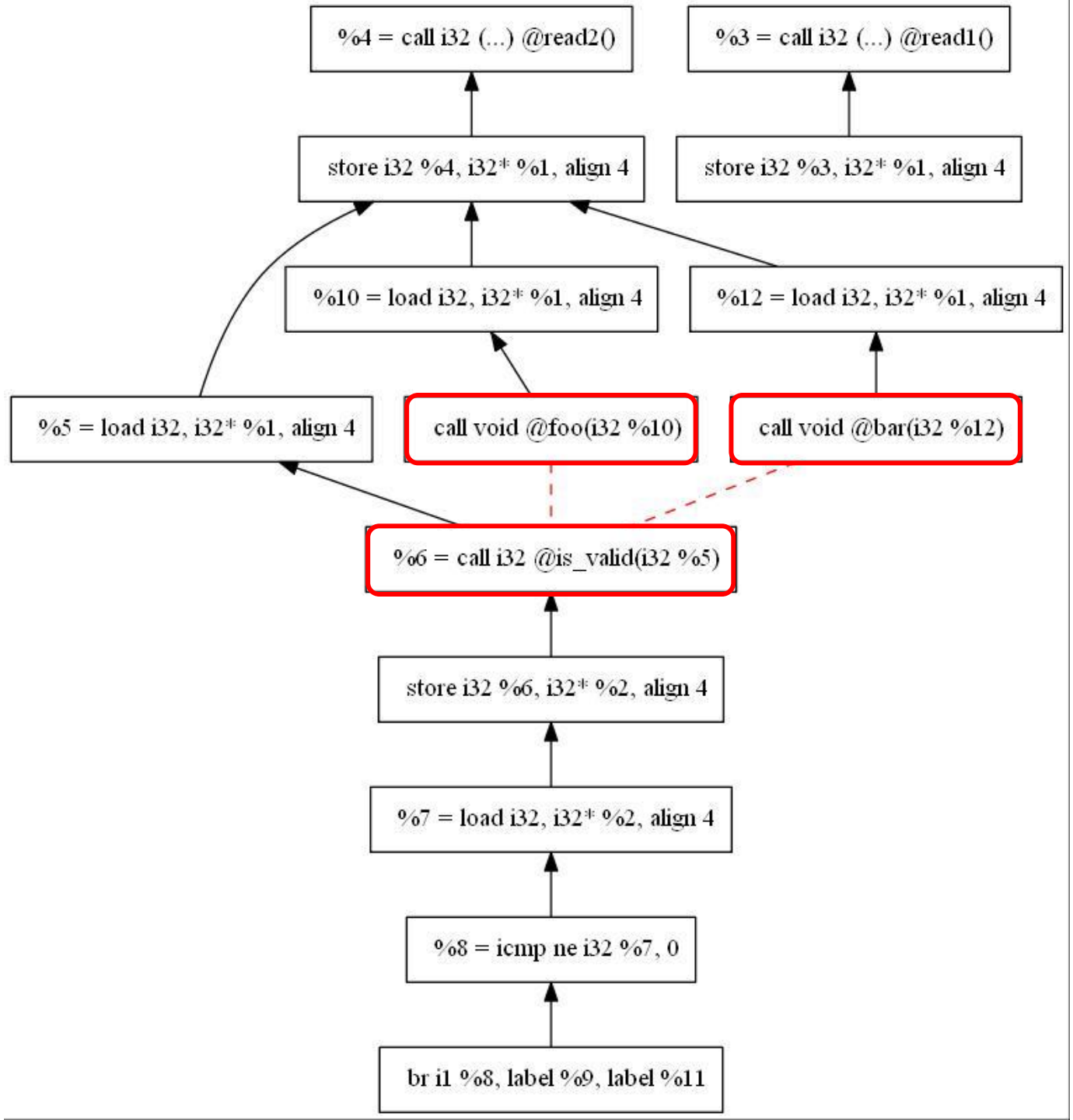


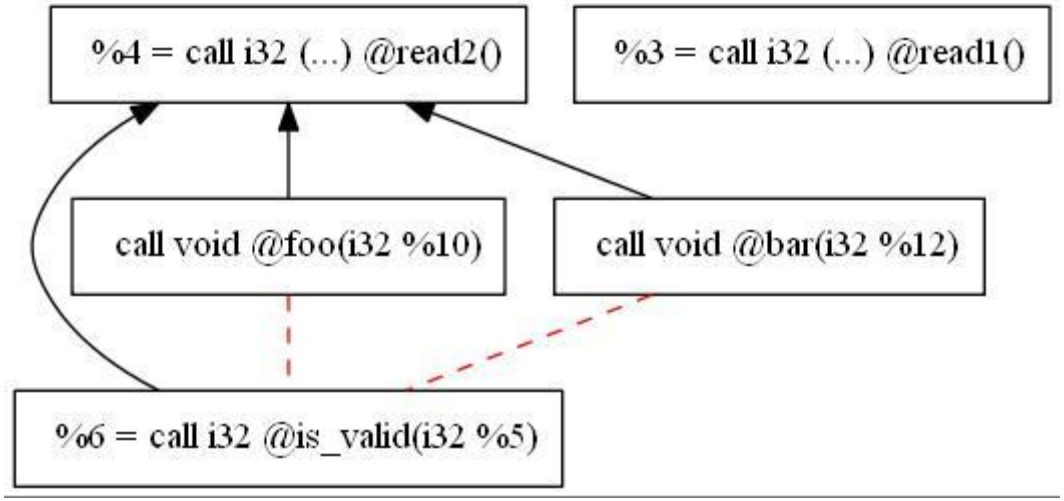
BB	IN	gen	kill	OUT
1		d1, d2, d3		d1, d2, d3
2	d1, d2, d3, d5, d6, d7	d4, d5	d1, d2, d7	d3, d4, d5, d6
3	d3, d4, d5, d6	d6	d3	d4, d5, d6
4	d3, d4, d5, d6, d7	d7	d4	d3, d5, d6, d7
EXIT	d3, d5, d6, d7			d3, d5, d6, d7

所有基本块的OUT相比于第二轮迭代结束都**没有**改变，所以分析终止。

数据共享

- 对于前述的示例代码，我们需要发现`is_valid`与`foo`有数据共享关系，`is_valid`与`bar`有数据共享关系，但是`foo`与`bar`没有数据共享关系（两者不在同一条执行路径上）
- 仅从数据依赖图是无法区分出`foo`与`bar`是否有数据共享关系的
- 使用类似的（稍微复杂点的）数据流分析方法可以构造出这种数据共享关系
  - 我们将提供一个参考实现，同学们可以在此基础上实现自己的代码或者将其修改集成到自己的代码中去
  - 下两页展示了包含数据共享关系的依赖图（完整的与去除无关元素之后的）


%4 = call i32 (...) @read2()
%3 = call i32 (...) @read1()
store i32 %4, i32* %1, align 4
store i32 %3, i32* %1, align 4
%10 = load i32, i32* %1, align 4
%12 = load i32, i32* %1, align 4
%5 = load i32, i32* %1, align 4
call void @foo(i32 %10)
call void @bar(i32 %12)
%6 = call i32 @is_valid(i32 %5)
store i32 %6, i32* %2, align 4
%7 = load i32, i32* %2, align 4
%8 = icmp ne i32 %7, 0
br i1 %8, label %9, label %11


%4 = call i32 (...) @read2()
%3 = call i32 (...) @read1()
call void @foo(i32 %10)
call void @bar(i32 %12)
%6 = call i32 @is_valid(i32 %5)

规范化

- 同一个函数，在不同上下文中被调用时，其参数、返回值所使用的变量可能不一致，但是我们应该将其识别为对同一个函数的多次调用
- 针对C语言代码、我们的待检测目标代码（单文件），对同名函数的调用必然是针对同一个函数
  - 但是在真实项目中以及C++代码中，不能仅凭函数名区分是否是对同一个函数的调用
- 一种规范化方式是将参数及返回值用相应的数据类型替代，如指令`%6 = call i32 @is_valid(i32 %5)`，根据其声明，可以规范化为`int is_valid(int)`

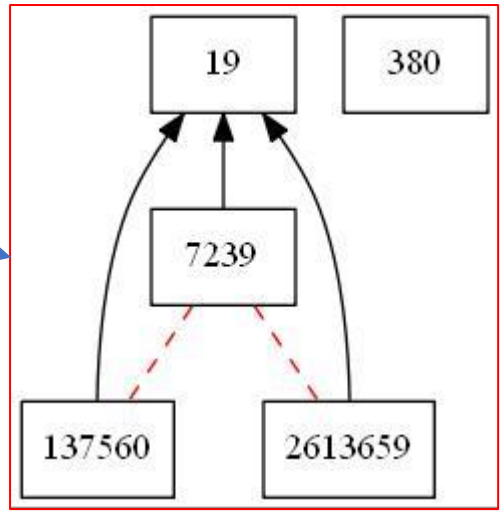
字符串→整数

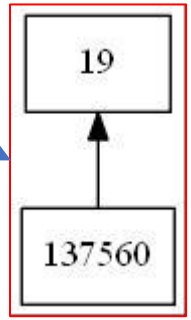
- 后续的数据挖掘部分，要想确认某种模式是否出现了多次，需要进行大量的比较操作，如果是进行字符串比较，我们认为效率较低。一般是将字符串转化为整数，后续所有的比较操作都是整数比较
- 因此需要寻找一种方式，能够将相同字符串转化为同一个整数，而不同字符串转化为不同整数
- 可以有各种不同的实现，如寻找一种哈希方法，将字符串转化为不冲突的哈希值
  - 请自行试验并完成

示例

- 对于如下示例代码，两个函数对应的Transaction如图所示

```
void Test2_func() {
     int str = read1();
     str = read2();
     int valid = is_valid(str);
     if (valid) {
          foo(str);
     } else {
          bar(str);
     }
     return;
}
void Test2_ff() {
     int rr = read2();
     foo(rr);
}
```





其他注意事项

- LLVM IR中有些函数调用是其内部定义的函数，在最终生成可执行代码时会进行相应转化
  - 打开调试开关“-g”时会生成相应的函数调用
  - 一些常用的库函数也会被转化为其内部函数调用的形式，如memcpy的调用，在LLVM IR中不是call memcpy(…)的形式（请自行试验）
  - 内部函数被称为Intrinsic
  - 请注意，在编程实现的过程中需要排除大多数内部函数，如调试模式引入的函数不应用于后续的数据挖掘；在没有进一步说明前，我们仅保留memcpy所对应的内部调用形式，即：将memcpy的内部调用形式保留在Transaction中
- 我们将会提供一个数据依赖图的参考定义（类），以及基于此定义实现的将图结构转化为Graphviz代码的方法
  - 在自行实验测试过程中发现参考实现代码有错的，请自行修改（我们不再提供修改后的参考实现）