



下载APP

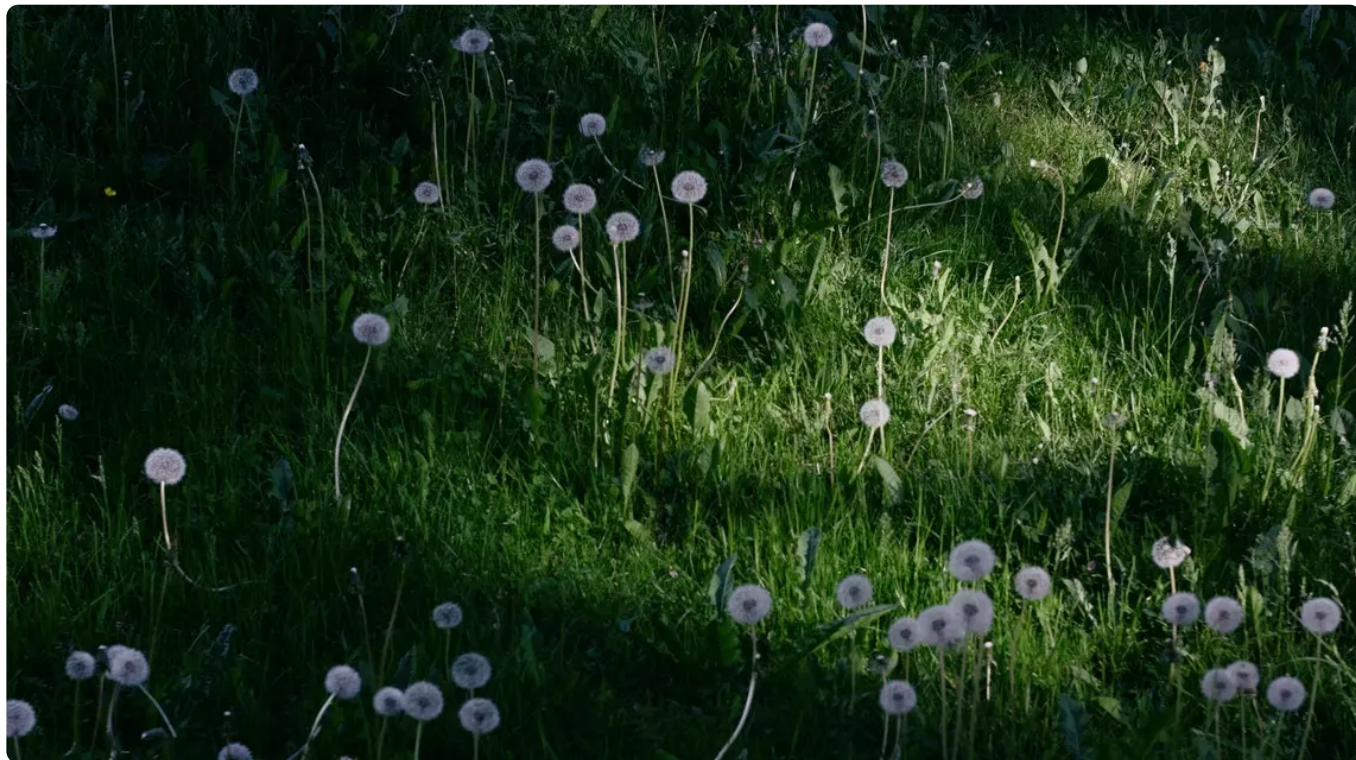


38 | 中端优化第1关：实现多种本地优化

2021-11-10 宫文学

《手把手带你写一门编程语言》

[课程介绍 >](#)



讲述：宫文学

时长 15:16 大小 13.99M



你好，我是宫文学。

上一节课，我们设计了 IR 的数据结构，并且分析了如何从 AST 生成 IR。并且，这些 IR 还可以生成 dot 文件，以直观的图形化的方式显示出来。

不过，我们上一节课只分析了 if 语句，这还远远不够。这节课，我会先带你分析 for 循环语句，加深对你控制流和数据流的理解。接着，我们就会开始享受这个 IR 带来的红利，用它来完成一些基本的本地优化工作，包括公共子表达式删除、拷贝传播和死代码删除，你初步体会基于 IR 做优化的感觉。



那么，我们先接着上一节课，继续把 for 循环从 AST 转换成 IR。

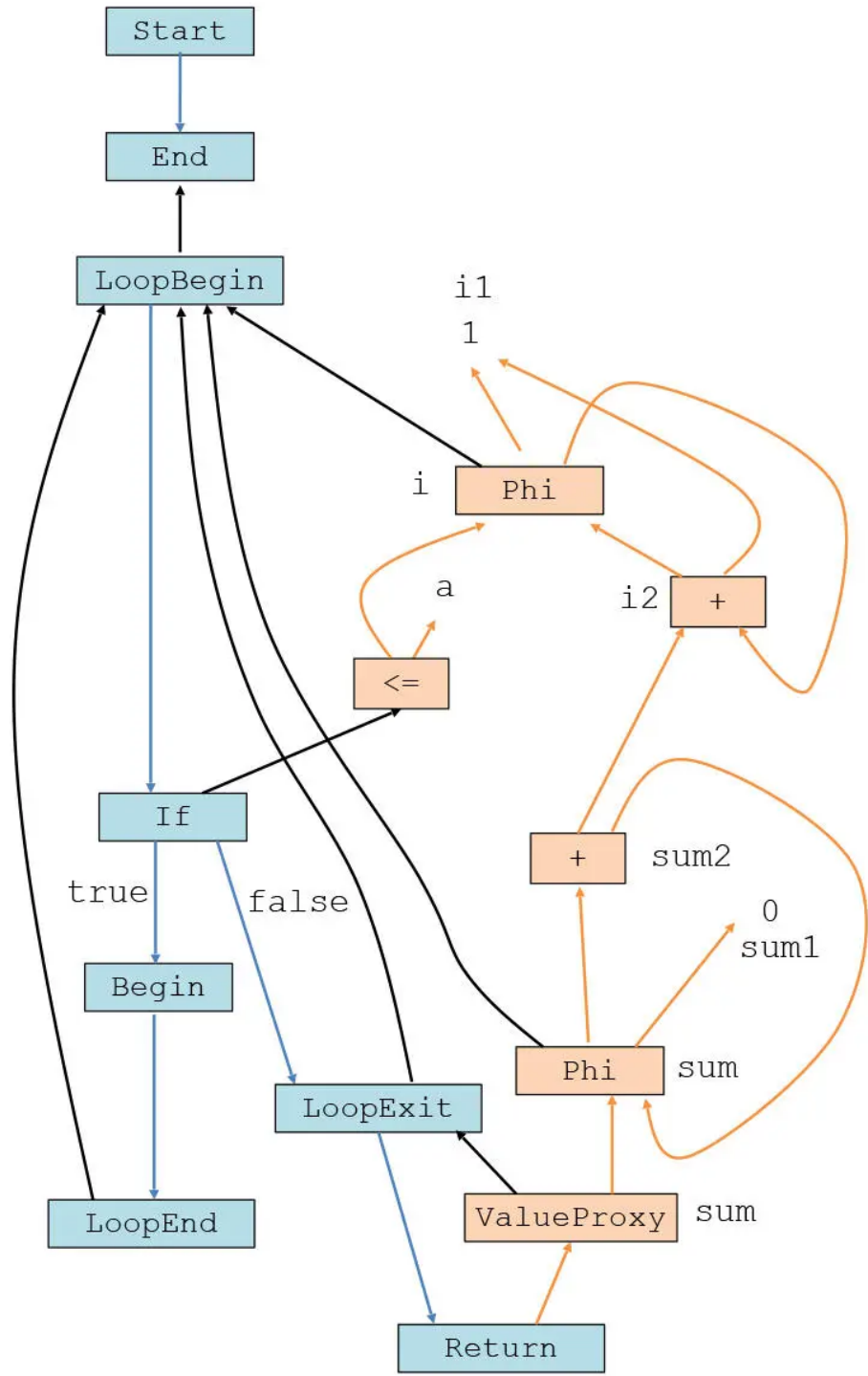
把 For 循环转换成 IR

同样地，我们还是借助一个例子来做分析。这个例子是一个实现累加功能的函数，bar 函数接受一个参数 a，然后返回从 1 到 a 的累加值。

[📄 复制代码](#)

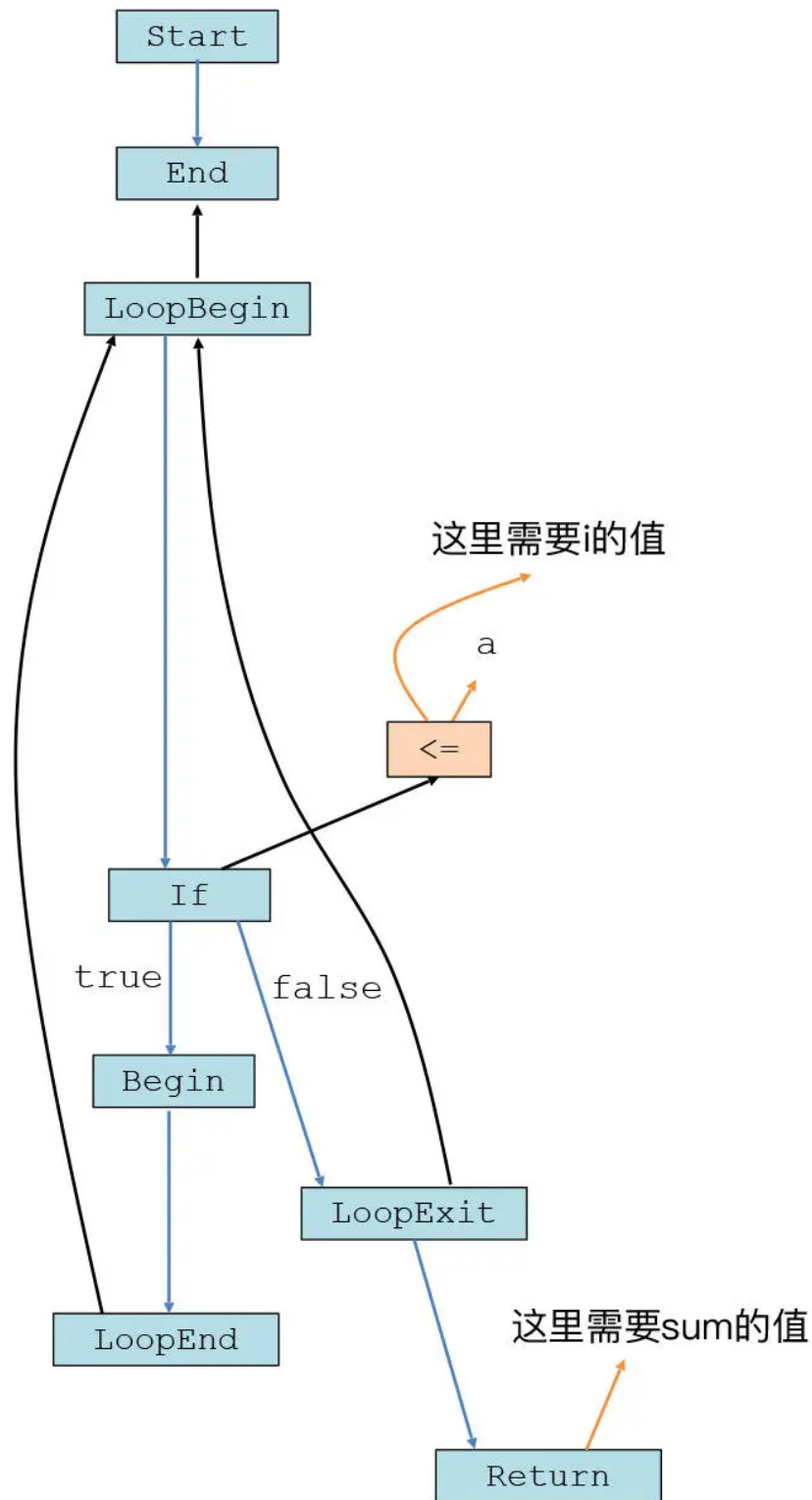
```
1 function bar(a:number):number{
2     let sum:number = 0;
3     for(let i = 1; i <= a; i++){
4         sum = sum + i;
5     }
6     return sum;
7 }
```

这里，我先直接画出最后生成的 IR 图的样子：



你一看这个图，肯定会觉得有点眼花缭乱，摸不清头绪。不过没关系，这里面是有着清晰的逻辑的。

第一步，我们先来看控制流的部分。



在程序开头的时候，依然还是一个 Start 节点。

而下面的 LoopBegin 节点，则代表了整个 for 循环语句的开始。开始后，它会根据 for 循环的条件，确定是否进入循环体。这里，我们引入了一个 If 节点，来代表循环条件。If 节

点要依据一个 if 条件，所以这里有一条黑线指向一个条件表达式节点。

当循环条件为 true 的时候，程序就进入循环体。循环体以 LoopBegin 开头，以 LoopEnd 结尾。而当循环条件为 false 的时候，程序则要通过 LoopExit 来退出循环。最后再通过 Return 语句从函数中返回。

并且，LoopEnd 和 LoopExit 各自都有一条输入边，连接到 LoopBegin。这样，循环的开始和结束就能正确地配对，不至于搞混。

不过，你可能注意到了一个现象，Start 节点的后序节点并不马上是循环的开始 LoopBegin。为什么呢？因为其实有两条控制流能够到达 LoopBegin：一条是从程序开始的上方进去，另一条是在每次循环结束以后，又重新开始循环。所以 LoopBegin 相当于我们上一节见过的 Merge 节点，两条控制流在这里汇聚。而我们在控制流中，如果用一条蓝线往下连接其他节点，只适用于单一控制流和流程分叉的情况，不包括流程汇聚的情况。我们上节课也说过，每个 ControlNode 最多只有一个前序节点。

那控制流的部分就说清楚了。**第二步，我们就来看一下数据流。**

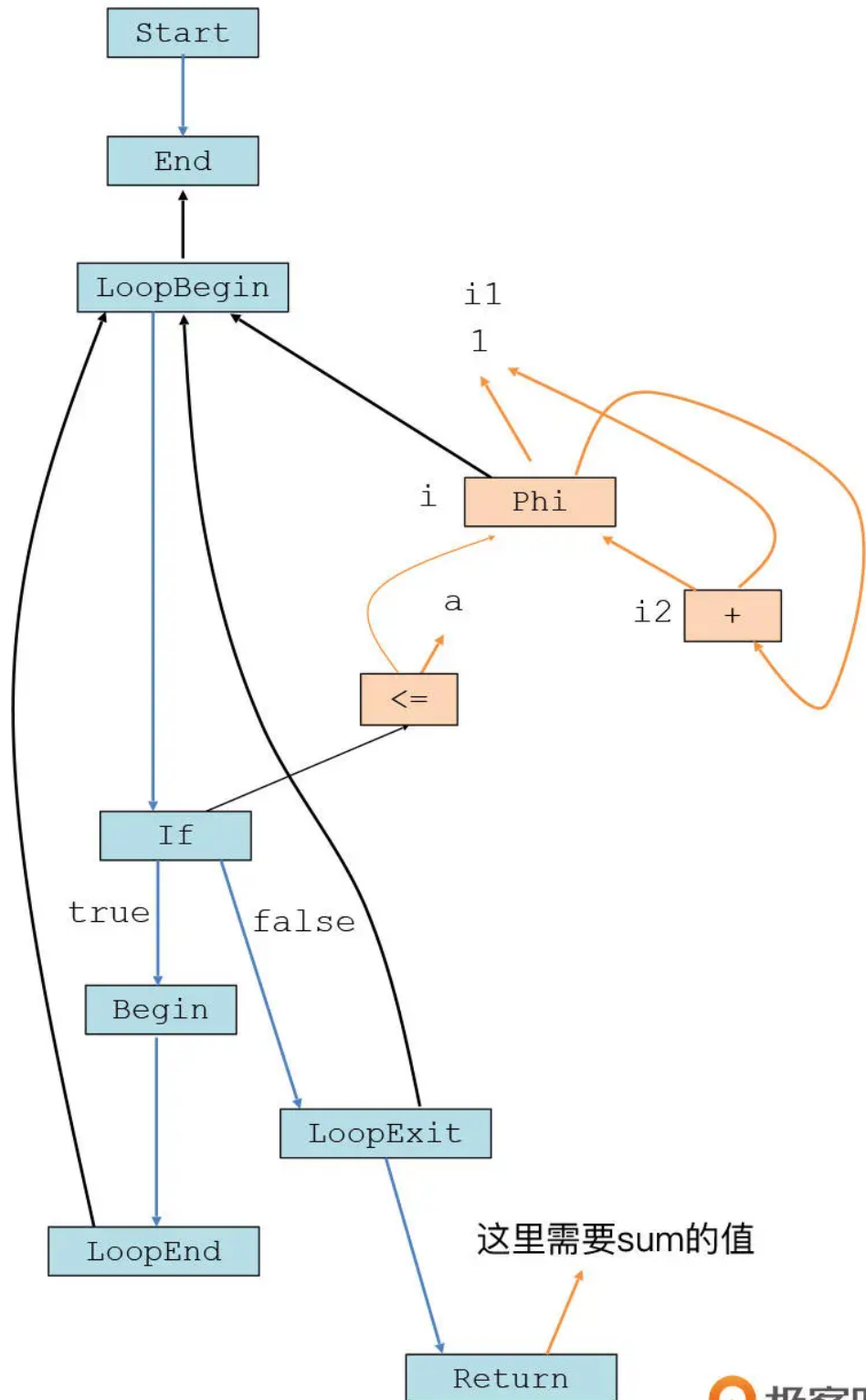
在数据流中，我们需要计算 i 和 sum 这两个变量。我们先看 i：

 复制代码

```
1 function bar(a:number):number{
2     let sum1:number = 0;
3     for(let i1 = 1; i1 <= a; i2 = i1 + 1){
4         sum2 = sum1 + i1;
5     }
6     return sum2;
7 }
```

这里，变量 i 被静态赋值了两次。一开始被赋值为 1，后来又通过 i++ 来递增。为了符合 SSA 格式，我们要把它拆分成 i1 和 i2 两个变量，然后再用 Phi 节点把它们聚合起来，用于循环条件的判断。

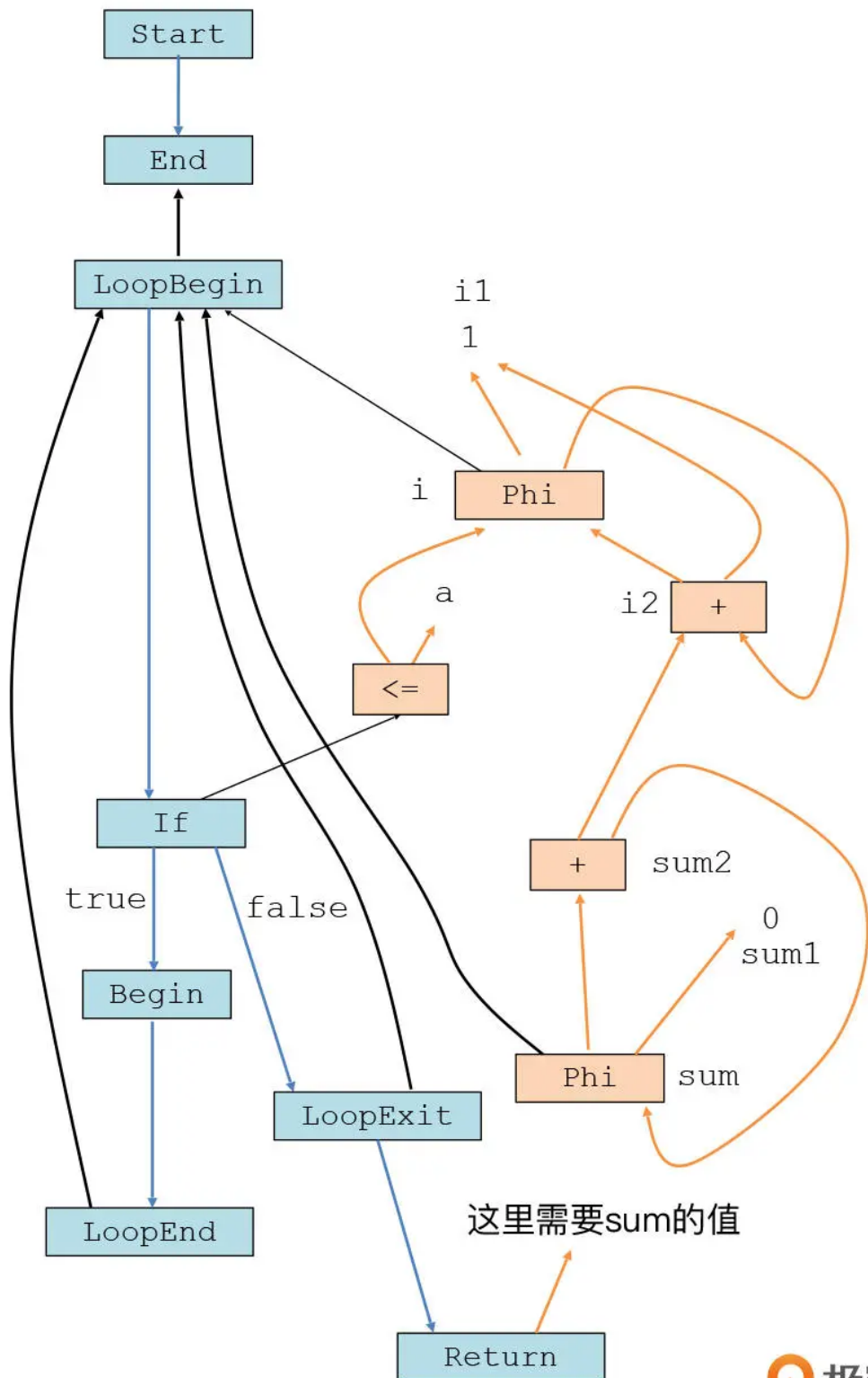
我们把与 i 有关的数据流加入到图中，就是下面这样：



我再解释一下这张图。 $i1=1$ 这个表达式，在刚进入循环时被触发，一次循环结束后，会触发 $i2 = i + 1$ 。所以，在 $i \leq a$ 这个条件中的 i ，在刚进入循环的时候，会选择 $i1$ ；而在循环体中循环过一次以后，会选择 $i2$ 。因此，我们图中这个 ϕ 节点有一条输入边指向 **LoopBegin**，用于判断控制流到底是从上面那条边进入的，还是从 **LoopEnd** 返回的。

对于 $i2 = i + 1$ 中的 i ，也是一样。它在一开始等于 $i1$ ，循环过一次以后，就等于 $i2$ 了。

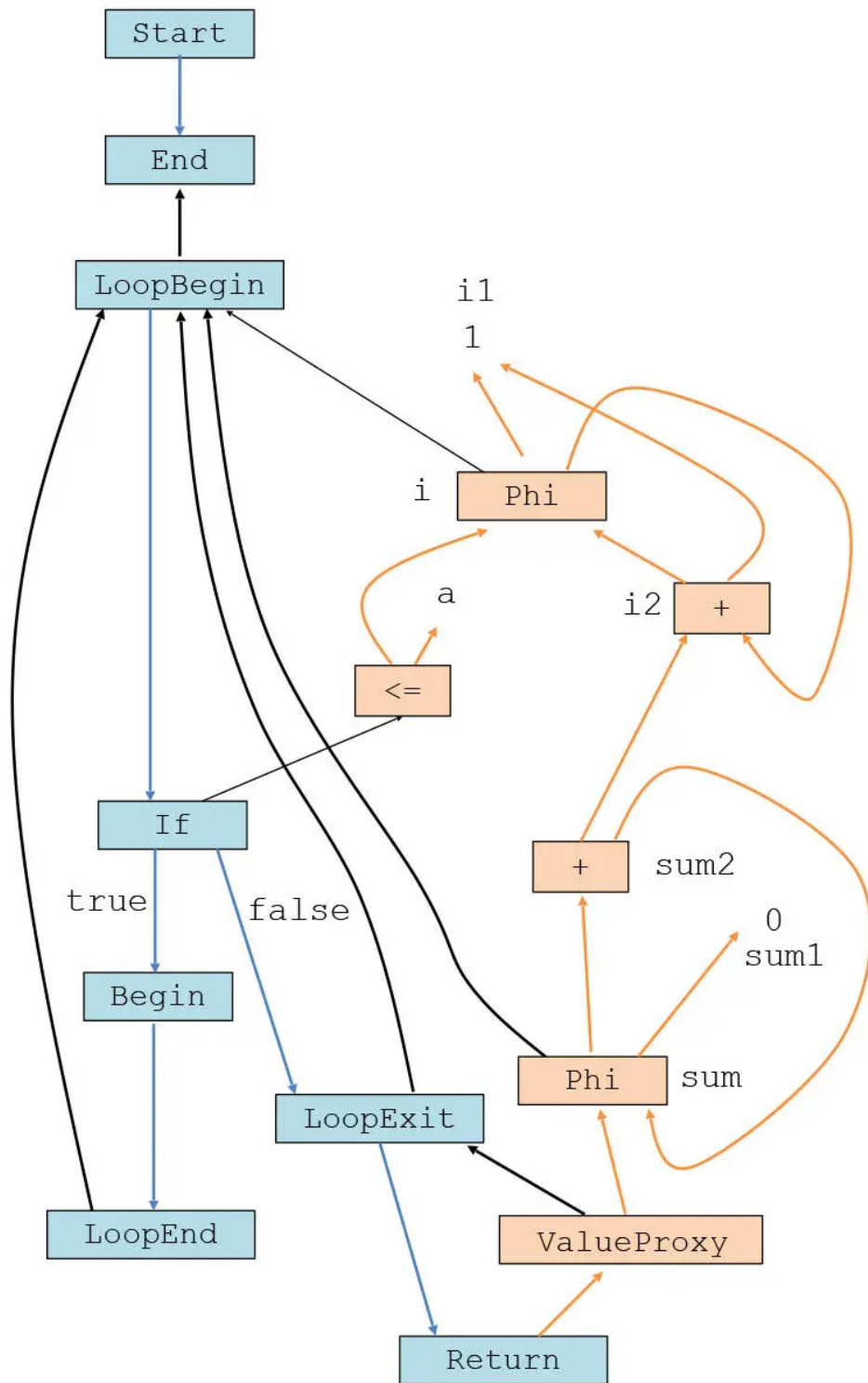
我们可以用同样的方式加入与 sum 变量有关的数据流：



这张图中，sum1 在循环体外被赋值为 0，后来在循环体内，则是执行 $\text{sum2} = \text{sum} + i$ 。这里的 sum，也是刚进入循环体的时候取 sum1，循环过一次以后就取 sum2，所以这里也需要一个 Phi 节点。

到这里，借助 Phi 节点，sum 的值也已经算出来了。那么在最后的 return 语句中，是不是就可以直接把这个值返回了呢？

不可以。为什么呢？因为 return 语句是在 for 循环语句之后的，而我们刚才计算的 sum 值，是循环体内的 sum 值。我们在程序里，必须要保证是在退出循环以后再获取的这个值，不能违背这个控制流带来的约束。所以，我们添加了一个 ValueProxy 节点，以 LoopExit 作为输入，确保这个值的计算是在循环之外。但它实际的值，就是刚才由 Phi 节点计算出的 sum 的值。




到此为止，整个 for 循环的 IR 就生成完毕了。一开始，你感觉会有点复杂，但如果你逐渐习惯了控制流和数据流的思维方式，分析起来就会越来越快了。

不过，回报和付出总是相匹配的。我们花了这么大代价来生成这个 IR，会让某些优化工作变得异常简单，接下来我们就来体会一下吧！

公共子表达式删除

首先，我们看看怎么利用这个 IR 来删除公共子表达式。

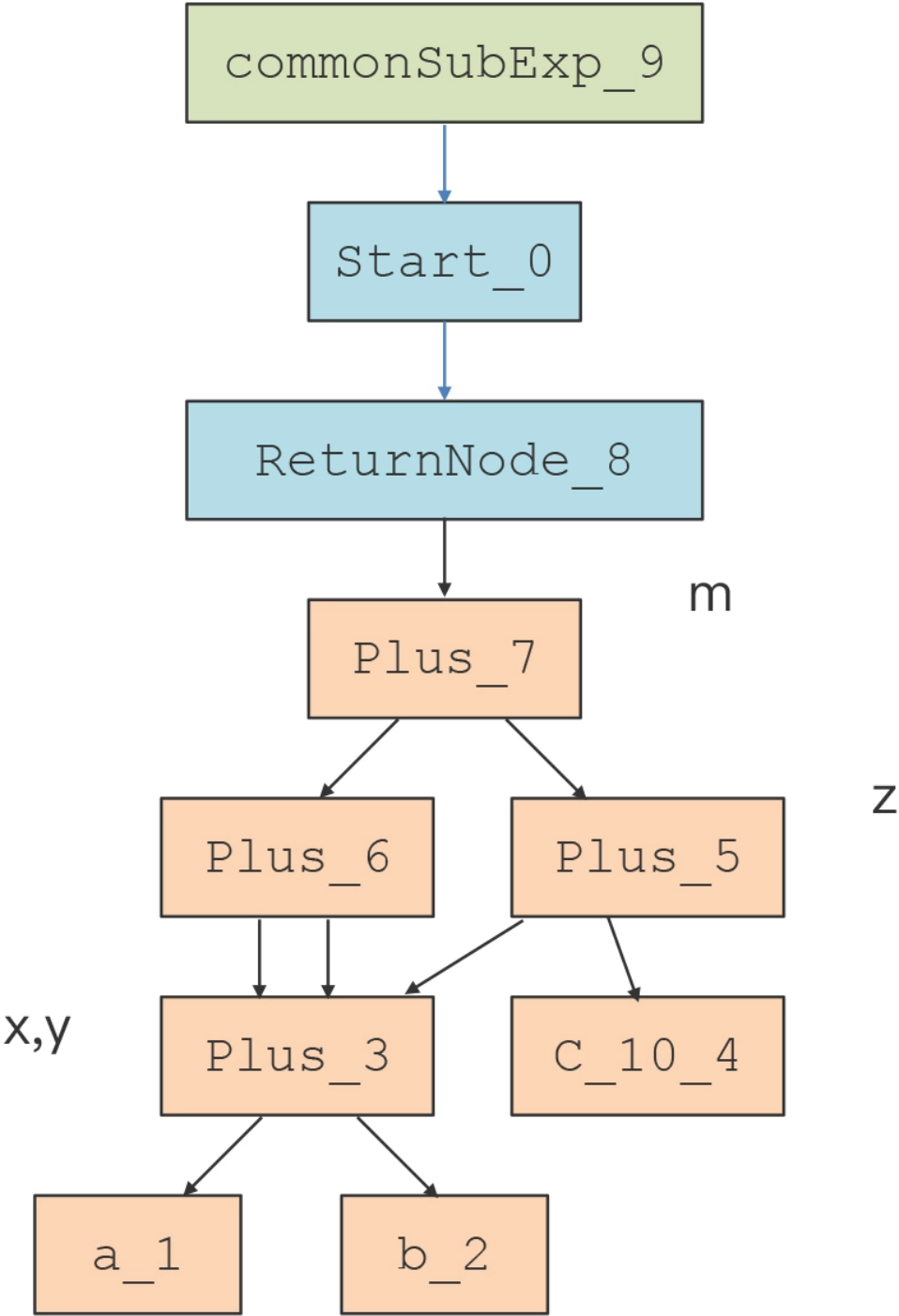
我们来看下面这个示例程序。这个程序中有两个变量 x 和 y ，它们的定义都是 $a + b$ ，所以它们有公共的子表达式。并且，变量 z 的定义中也有 $a+b$ 这个公共的子表达式。

 复制代码

```
1 //删除公共子表达式
2 function commonSubExp(a:number, b:number):number{
3     let x = a + b;
4     let y = a + b;
5     let z = a + b + 10;
6     let m = x + y + z
7     return m;
8 }
```

如果用我们的 IR 来删除这个示例程序中的公共子表达式，我们甚至都不需要等到优化阶段，而是在生成 IR 的时候，顺带手就可以做了。

你可以运行一下 `node play example_opt1.ts --dumpIR` 命令，生成下面的图。我手工在节点旁边标注了一下变量名称。你能看到，图中只有一张子图代表“ $a+b$ ”这个公共子表达式，而且它被多个变量的定义引用了。



那具体这个公共子表达式是怎么被共享的呢？首先，为了保存我们的 IR 图，我们设计了一个 graph 类，里面保存了所有节点的列表。

复制代码

1 //IR图

```
2 export class Graph{
3     nodes:IRNode[] =[];
4 }
```

然后，在遍历 AST 生成 IR 的时候，我们会先生成针对某个 AST 节点的 `DataNode`，然后再加入到 `Graph` 中。这个节点实际上就代表了一个子图。□我们以加法运算节点为例，这个子图包含了一个 `BinaryOpNode`，还有 `left` 和 `right` 这两个 input。

但是，这个子图可能在 `Graph` 中已经存在了。比如，在上面的示例程序中，当处理变量 `x` 的定义的时候，程序就为“`a+b`”这个表达式生成了一个 `BinaryOpNode`，它的左右两个 input 分别是参数 `a` 和 `b`，然后我们把这个节点加入到了 `Graph` 中。而当处理变量 `y` 的定义的时候，程序也会生成一个 `BinaryOpNode`，它的左右两个 input 也是参数 `a` 和 `b`。

这个时候，我们就没有必要把第二个 `BinaryOpNode`，或者说子图，加入到 `Graph` 中了，我们直接用之前那个子图就行了。所以，我们要添加一个功能，用来比较两个 `DataNode` 节点是不是相同的。如果我们准备加入的节点在 `Graph` 中已经存在，那就返回原来的节点。这部分具体实现，你可以参考 [@ir.ts](#) 中的 `Graph` 类中的 [@addDataNode\(\)](#) 方法。另外，为了比较两个节点是否相同，我还为每个 `DataNode` 都实现了一个 `equals()` 方法。

接下来，你可以继续看看变量 `z` 的定义。在变量 `z` 定义中也存在“`a+b`”这个子表达式，它直接引用了原来的 `DataNode` 节点，然后再跟常量 `10` 相加。

最后，在变量 `m` 的定义中，我们先使用了一个临时变量来计算“`x+y`”。你在图中能看到，这个临时变量的两个 input 都指向了代表“`a+b`”的 `DataNode`。这就说明变量 `x` 和 `y` 引用的都是同一个 `DataNode`。

这部分的具体实现是这样的，在 [@IRGenerator](#) 程序的 [@visitVariable\(\)](#) 方法中，根据变量的符号，我们可以从 `Graph` 中把对应的 `DataNode` 都查出来。这是因为，`IRGenerator` 在处理好 AST 以后，会生成一个 `IRModule`，而 `IRModule` 中就保存每个变量跟 `DataNode` 的对应关系。

好了，现在你已经了解了如何基于我们的 IR 来删除公共子表达式了。接下来，我们再看看它在处理其他优化任务时是否也同样方便。我们看一下拷贝传播。

拷贝传播

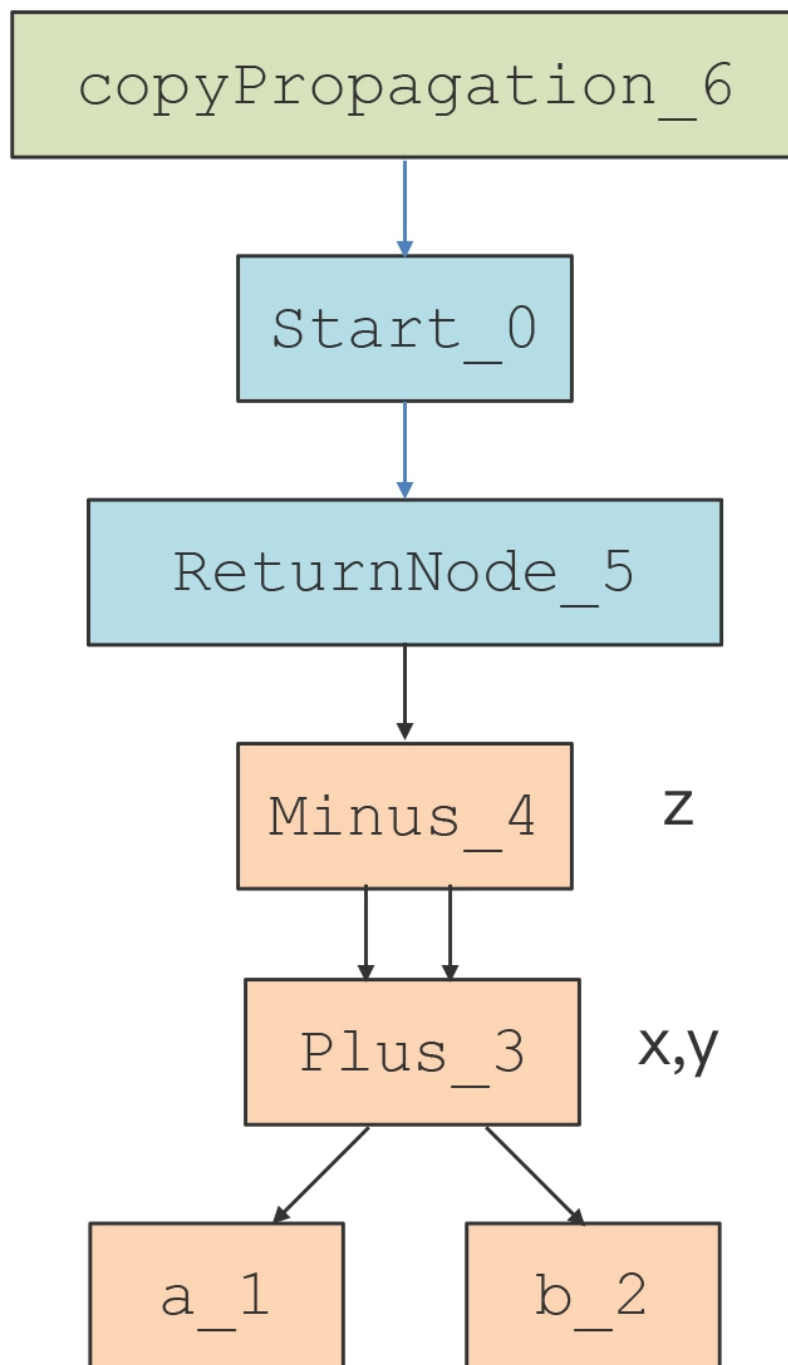
实际上，基于我们的 IR 来处理拷贝传播，也是手到擒来，几乎不需要做什么额外的工作。

我们看一段示例代码。在这段代码中，变量 x 的定义是 $a+b$ 。然后，我们又用了 x 来定义 y ，那你推理一下就知道，现在 y 也应该等于 $a+b$ 。

[复制代码](#)

```
1 // 拷贝传播
2 function copyPropagation(a:number, b:number):number{
3     let x = a + b;
4     let y = x;
5     let z = y - x;
6     return z;
7 }
```

你仍然可以用我们现在的编译器加上 `-dumpIR` 选项来生成 `.dot` 图，我把它放在下面了。



你会看到，变量 x 和 y 都引用了相同的 `DataNode`。这里具体的实现你可以看一下 [IRGenerator](#) 中的 `visitVariableDecl()` 方法。在声明变量 y 的时候，我们会获取变量初始化表达式对应的 `DataNode`，再把它跟该变量绑定。而变量 y 的初始化表达式就是 x ， x

对应的 `DataNode` 就是图中的 `Plus` 节点，所以这个节点也跟变量 `y` 关联到了一起。拷贝传播就是这么在处理变量声明的过程中自然而然地发生了。


最后，你在图中再看一下变量 `z` 的定义。你会看到，减法运算的左右两个 `input` 都是指向了同一个 `DataNode`。所以，接下来我们就可以自然而然地做一个优化了，直接计算出 `z=0` 就可以了。在做优化的时候，我们经常会遇到这种情况，就是一个优化的处理结果，为其他优化创造了机会。就像当前的例子，拷贝传播的结果就是给减法运算的优化创造了机会。

不过，在实际的优化算法中，我们通常会让 IR 经历多个 `Pass` 的处理，每个 `Pass` 处理一种优化场景。并且，经常同一种优化算法会被使用多次，原因就是在做完优化 A 以后，可能又制造出了优化 B 的机会。

最后，我们再看看死代码删除的情况，看看我们的 IR 又会带来什么惊喜。

死代码删除

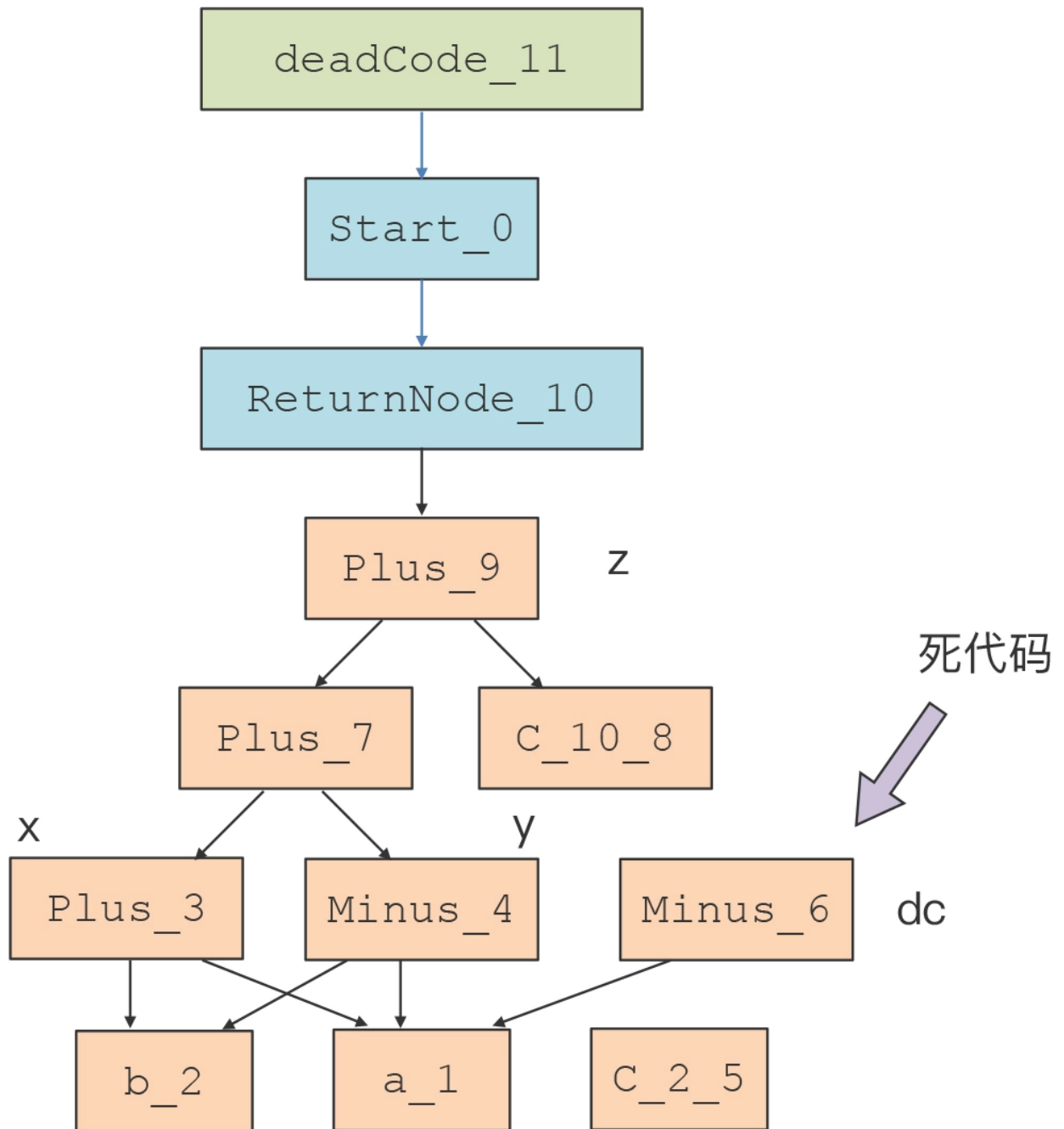
我们还是看一个存在死代码的例子程序。这个例子中有 `x`、`y`、`z` 和 `dc` 共 4 个变量。你用肉眼看一下就能发现，定义 `dc` 变量的这行代码是多余的。因为在定义出 `dc` 以后，再也没有代码用到它了。

 复制代码

```
1 //删除死代码
2 function deadCode(a:number, b:number):number{
3     let x = a + b;
4     let y = a - b;
5     let dc = a - 2;
6     let z = x + y + 10;
7     return z;
8 }
```

我之前给你介绍过变量活跃性分析的数据流方法。我们可以自底向上地遍历这个代码块，并不断更新一个“活跃变量”的集合。等分析到声明 `dc` 这一行的时候，我们会发现当前活跃变量集合里是没有 `dc` 的，这样就知道这行代码是死代码了。

如果使用我们现在的 IR，那应该如何检测死代码呢？我们还是先看编译器生成的 IR 图，看看死代码在图中有什么特点。



我在图中标出了作为死代码的 dc 变量。你从图中可以直观地看到，这个节点有一个显著的特点，就是没有其他节点引用它，因此它不是任何其他节点的 input。

你应该记得，我们在 `DataNode` 中设置了一个 `uses` 属性，指向所有使用该节点的其他节点，是一个反向的链接。那这个时候，其实 `dc` 变量对应的 `DataNode` 的 `uses` 列表是空的。所以，只要是 `uses` 为空的节点，我们就可以把它从图中去掉。而我们把 `Minus_6` 去掉以后，常量 2 也没有任何节点使用了，所以我们可以把它去掉。

你看，现在我们要去除死代码的话，简单到**只是查询 `DataNode` 的 `uses` 属性是否为空集合**就行了。是不是太方便了？具体实现你可以看看 `ir.ts` 中的 `DeadCodeElimination` 类。

不过，需要注意的是，上面只是产生死代码的其中一个场景，还有另一个场景是出现在 `return`、`break` 等语句之后的代码，也都是死代码。这种类型死代码，也是在生成 IR 的时候就可以去掉的。也就是，在遇到 `return` 语句以后，我们不再为同一个块中的其他语句生成 IR 就行了。

课程小结

今天的内容就是这些。今天这节课，我首先接着上一节分析了如何为 `for` 循环语句生成 IR，让你熟悉另一种常用的 IR 结构，接着分析了如何基于该 IR 实现几种常见的本地优化算法。我希望你记住以下的重点：

首先，在 `For` 循环中，`LoopBegin` 和 `Merge` 节点一样，都是实现了多个控制流的汇聚。`LoopEnd` 代表一次循环的结束，而 `LoopExit` 代表退出循环，它们都要跟 `LoopBegin` 配对。对于循环变量，我们需要用 `Phi` 节点来获取其不同控制流分支上的取值。

第二，在生成 IR 的过程中，我们顺手就可以实现对公共子表达式的删除，这要实现 `DataNode` 的比较。并且要求在 `DataNode` 加入 `Graph` 的过程中，不能存在相同的 `DataNode`，或者子图。

第三，在生成 IR 的过程中，我们通过处理变量声明，也可以自然而然地实现拷贝的传播。

第四，如果一个 IR 的 `uses` 属性是一个空集合，那我们就可以判断出它是一个没有用的变量，可以把它删除掉，这就实现了死代码删除的功能。

最后，一种优化工作的结果会为其他的优化创造机会。所以，编译器在优化一个 IR 的时候，会前后多次调用同一个优化算法。

思考题

今天我们讨论的这些优化的例子，都是本地优化的情况，也就是在同一个基本块中代码做优化，没有考虑控制流跳转的情况。那你不能分析一下，当存在 if 语句和循环语句的情况下，能不能也像这节课这样实现公共子表达式的删除、常量传播和死代码删除？

欢迎你把这节课分享给更多感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

[🔗 这节课示例代码的目录在这里！](#)

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 37 | 从AST到IR：体会数据流和控制流思维

下一篇 39 | 中端优化第2关：全局优化要怎么搞？

精选留言 (1)

 写留言



奋斗的蜗牛

2021-11-10

如果考虑控制流，我认为比较DataNode时，要加入控制节点的比较

