# 中间代码形式

非SSA 四元式

| op | src1 | src2 | dst | 备注 |
|---|---|---|---|---|
| DECL | bitsize | null | id | 声明空间为bitsize,标识符为id的变量 |
| ADD | src1 | src2 | dst | |
| SUB | src1 | src2 | dst | |
| MULT | src1 | src2 | dst | |
| DIV | src1 | src2 | dst | |
| MOD | src1 | src2 | dst | |
| NOT | src1 | null | dst | dst=(src1==0) |
| SLL | src1 | num | dst | |
| SRL | sr1c | num | dst | |
| STORE_ARRAY | ind | value | id | id[ind]=value |
| GET_ARRAY | id | ind | dst | dst=id[ind] |
| BNEZ | src1 | null | label | |
| BEQZ | src1 | null | label | |
| BLT | src1 | src2 | label | |
| BLE | src1 | src2 | label | |
| BGE | src1 | src2 | label | |
| BGT | src1 | src2 | label | |
| BEQ | src1 | src2 | label | |
| BNE | src1 | src2 | label | |
| JUMP | src/null | null | label | src不为null的时候需要将src加载到$v0 |
| PASS_ARG | arg | ind | func | 向函数func传递第ind个参数arg |
| GET_ARG | ind | is_array | dst | 接收第ind个参数dst，is_array是否为形参数组 |
| PASS_GET_ARG | passer | is_array | dst | 函数内联用，将passer传递给形参dst |
| FUNC_CALL | func | num | retDst/null | 调用函数func，参数个数num，返回到retDst |
| GET_RET_VALUE | null | null | retDst | 从$v0寄存器载入到retDst |
| RETURN | src/null | null | null | 返回值src |

| op | src1 | src2 | dst | 备注 |
|---|---|---|---|---|
| PRINTS | src | null | null | 打印字符串src |
| PRINTD | src | null | null | 打印数字src |
| GET_INT | null | null | dst | dst=getint() |
| SET | null | null | label | 设置标签 |
| END | null | null | null | 结束 |

# 中间代码优化

## 全局类

### 分发

接收中间代码生成的四元式list，将四元式list划分为全局数据、主函数和函数。

### 发起函数内联

函数内联不是函数内优化，要放到全局类。

## 函数类

### 数据流分析

划分基本块，到达定义分析和活跃变量分析

### 调用优化器

循环调用常量传播优化、公共子表达式合并、复制传播优化、窥孔优化直到代码规模连续多次不减。

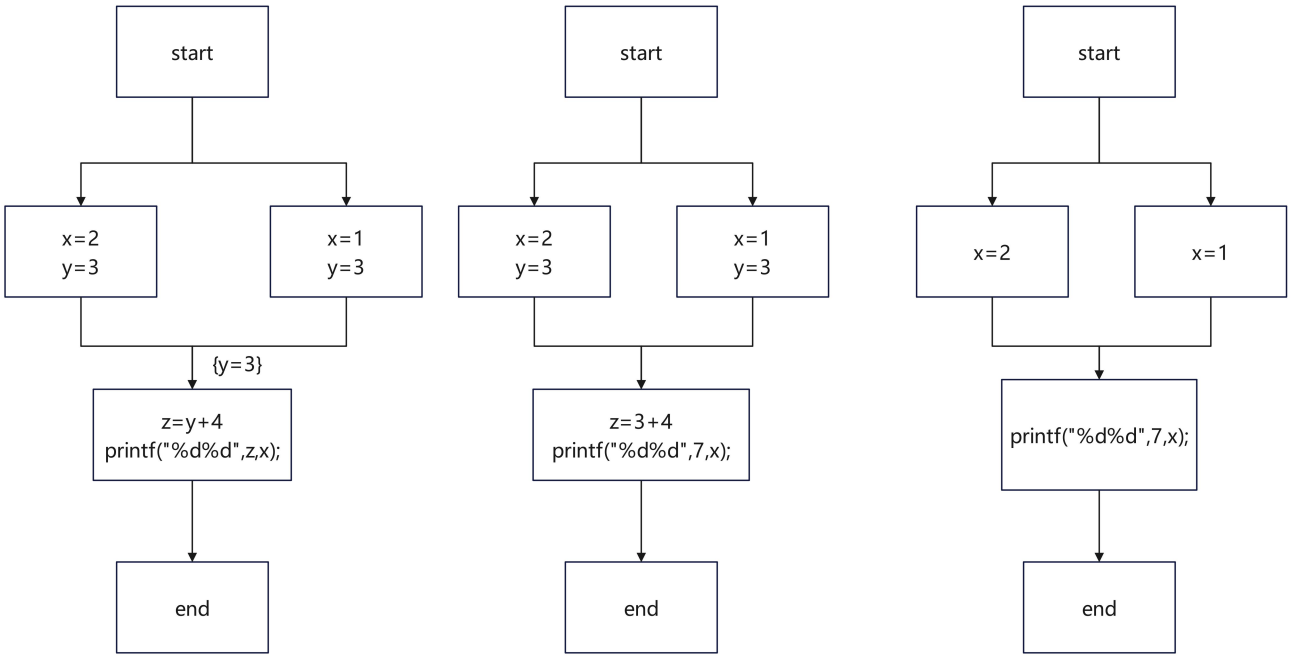
### 难点

活跃变量分析和到大定义分析有bug的时候很难debug，建议进行一定程度地输出，譬如Block的toString()方法输出相关信息这样可以在调试窗口查看，比如划分基本块的情况输出到一个文件里方便查看。

## 常量传播

常量传播优化器。
每个基本块维护一个常量表map，为取值为常量的变量到其常量值的映射。
我们已经有到达定义分析，所以就已经知道在块开始的时候可以到达的定义，对于非全局变量x，如果所有到达的定义都是同一个常量，可以认为在块开始的时候x是一个常量，用这些值初始化基本块的常量表。
遍历函数，不断更新常量表，并把可以替换的变量替换为常量。
譬如语句y=x+3，此时常量表有map.get(x)=9,则将语句替换为y=9+3。
把一些四元式的src替换为常量后，有的赋值在之后就不再使用了，删除这些冗余赋值。
这一步也同时完成了删除冗余赋值的任务。

**可视化**




### 难点

如果要引入全局变量的常量传播，一定要注意在一个函数调用之后所有全局变量的值都可能发生变化。

## 全局公共子表达式合并/有效表达式优化

### 核心思想

有效表达式优化器/子表达式合并优化器，所谓有效表达式validExp，指的是加减乘除余。
包含有效表达式的四元式，就是有效四元式。
譬如有效四元式:dst=b+c 有效表达式b+c。
如果我们当前有有效表达式b+c，然后遇到语句i:n=b+c，就可以进行合并。
那么什么时候我们有有效表达式b+c呢，要么在这个块内在语句i之前曾经有计算过b+c而且计算之后没有再对b c的赋值
要么这个块所有前序块都给出了有效表达式b+c，而且这个块在语句i之前没有再对b c赋值。

### 初始化

一个有效表达式，譬如b+c，一直到b和c再次被赋值之前都是有效的。
当我们读到表达式a=b+c，此时就生成了有效表达式b+c
当我们读到任意对b或c的赋值，此时就kill了有效表达式b+c
所以我们可以先遍历整个函数得到这个函数所有可能的有效表达式，同时也就得到了每个块能够生成的有效表达式gen。
我们称基本块i能够生成有效表达式b+c，当且仅当i存在指令dst = b ADD c且这条指令之后没有对b c的再次赋值。 搜集了全局所有可能的有效表达式之后，我们就可以获得基本块杀死的基本表达式kill。 我们称基本块j杀死了有效表达式b+c，当且仅当基本块j存在一个对b或c的赋值，且赋值之后没有指令dst = b ADD c。 我们对某基本块的所有前序基本块的输出有效表达式求交集，就得到这个基本块的输入有效表达式。
基本块的输出有效表达式:out=gen+(in-kill)。

类比于前向传播，反复迭代直到有效表达式流不在变化，于是现在我们就已知了所有基本块的输入有效表达式。
如果我们想要知道运行到某块的某条语句时有哪些有效表达式，只需要从这个块的起始位置开始正向遍历块，遇到赋值删除有关有效表达式，遇到有效四元式加入新的有效表达式。

## 遍历块

正向遍历函数的四元式，如果是有效四元式(加减乘除余的四元式)(下称有效表达式a)，通过quat2block获取其所在的基本块i。
遍历基本块i在有效四元式a之前的四元式，初始时可用的有效表达式就是块的输入有效表达式，之后遇到赋值，需要删除被赋值变量相关的有效表达式，遇到有效四元式，需要增添有效表达式。
核心在于得到两个信息：

- 到达有效四元式a的时候到底有哪些有效表达式(因为我们只知道每个块开始的时候有哪些可用的有效表达式)
- 以及这个有效表达式是来自块i在四元式a之前的某个表达式还是块i的前序块

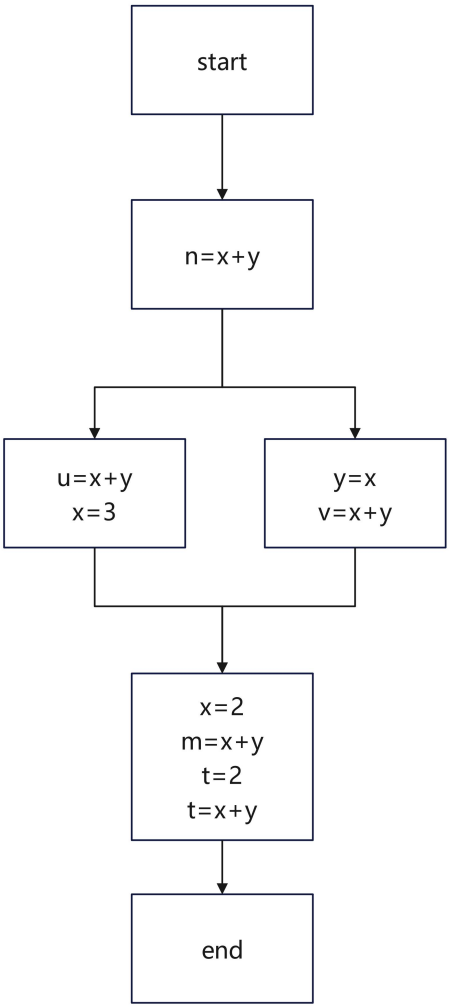
## travel

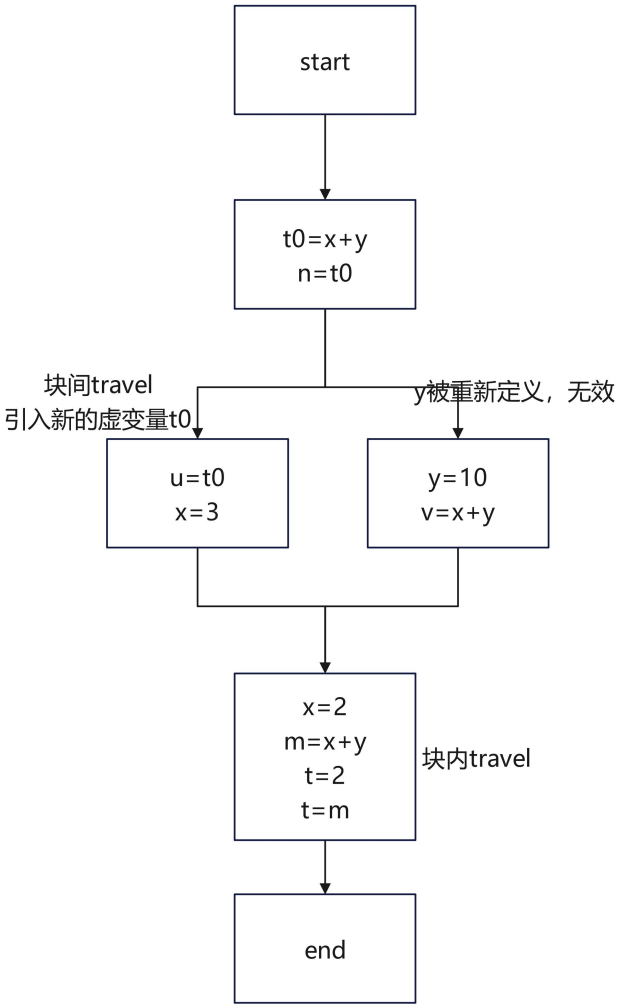
当我们发现有效四元式a处，有可以合并的有效表达式，譬如a:dst=b+c，a处有可用有效表达式b+c
我们需要区分两种情况：

- 这个有效表达式来自块内 travelWithinBlock
- 这个有效表达式来自块的前序块 travelOverBlocks

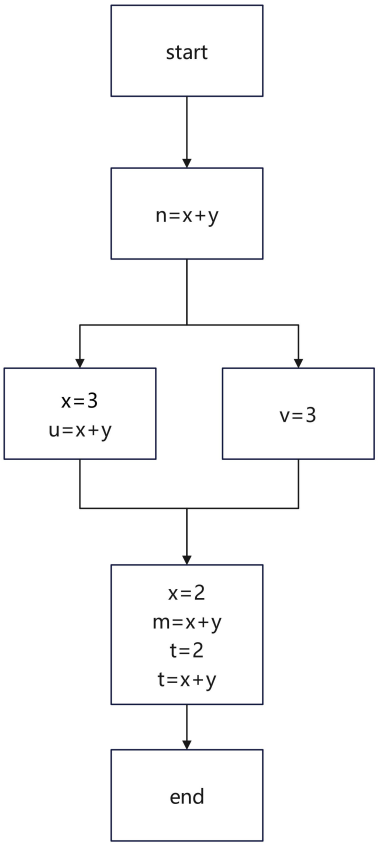
**travel的可视化**

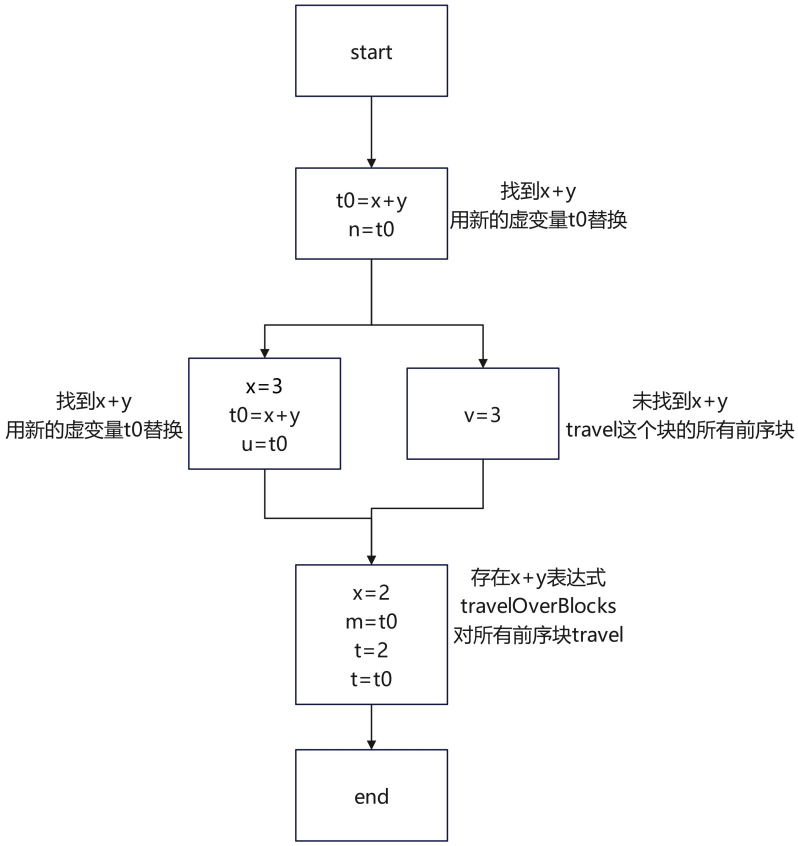
travel的过程比较复杂，但是可视化比较简洁，我们给出travelWithinBlock和travelOverBlocks的可视化：
块内travel和简单的块间travel




复杂的块间travel




### 难点

块内travel与块间travel的判断。
以及块间travel的时候要能够申请临时变量。(也就意味着在优化的时候要有申请临时变量的途径)

## CopyOptimizer

复制传播优化

### 复制对

当有直接赋值，譬如x=y时，就生成了复制对<x,y>(无序对，<x,y>等同于<y,x>)
如果出现任意对x或y的赋值，就杀死了复制对<x,y>

### initialCopy

我们可以看到，复制对与有效表达式很像，我们完全效仿有效表达式流的分析方式。
首先收集整个函数的可能复制对，同时得到各个基本块的生成复制对gen。
我们称基本块生成了复制对<x,y>，当且仅当块内存在直接赋值x=y或者y=x且赋值之后没有再对x或y赋值
我们称基本块杀死了复制对<x,y>，当且仅当块内存在对x或y的赋值而且赋值之后没有重新生成<x,y>。
我们对某基本块的所有前序基本块的复制对求交集，就得到了这个基本块的输入复制对。
基本块的输出复制对：out=gen+(in-kill)
类比于前向传播，反复迭代直到复制对流不在变化，于是现在我们就已知了所有基本块的输入复制对。
如果我们想要知道运行到某块的某条语句时有哪些复制对，只需要从这个块的起始位置开始正向遍历块，遇到赋值删除有关复制对，遇到直接赋值加入新的复制对。

**并查集**

不难发现，当我们拥有复制对<x,y>和<y,z>的时候，其实我们就同时拥有了复制对<x,z>，想要实现这种传递性，并查集是很好的选择。
考察这个并查集需要支持的操作，有：

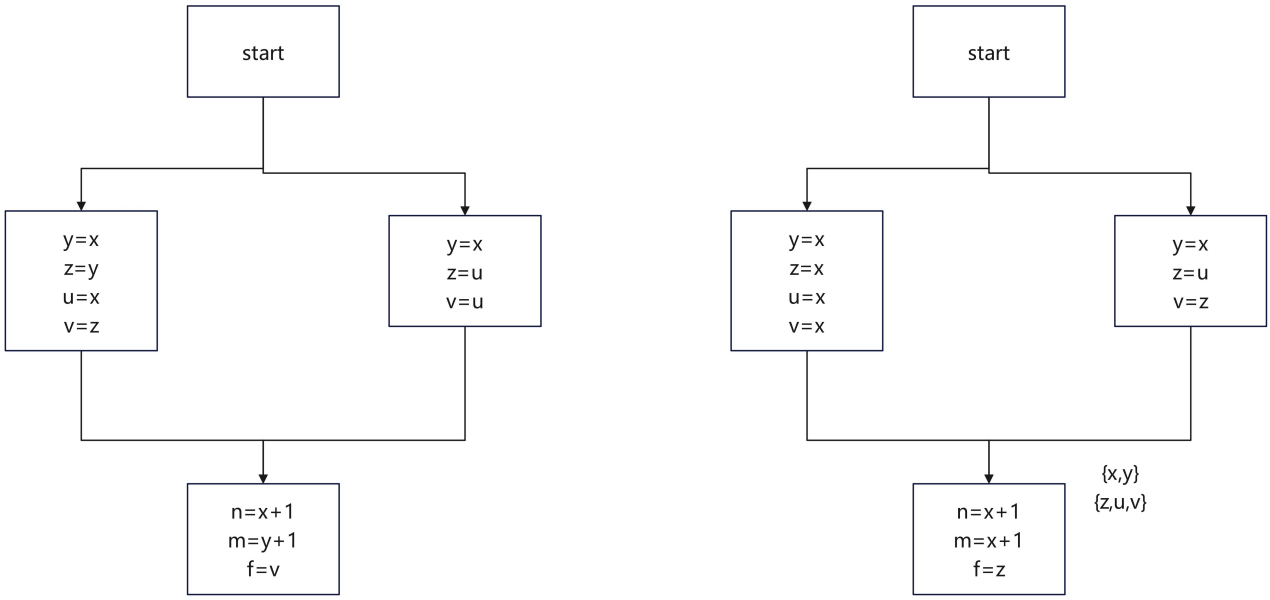
- 加入复制对，例如读到语句y=x要加入<x,y>
- 并查集的交集，例如某块的两个前驱块的并查集分别为{{x,y,z},{u,v,w}}和{{x,y,u},{z,v,w}}，则交集为{{x,y},{z},{u},{v,w}}
- 删除某变量的复制关系，例如读到对x的赋值需要把x踢出其所在的并查集，例如{{x,y,z},{u,v,w}},我们读到x=34，并查集会变为{{x},{y,z},{u,v,w}}

这些操作用并查集都不难实现。

### optimize

对于每个块，以块的复制对并查集为初始情况正向遍历块的四元式，遇到直接赋值(例如y=x)，添加到并查集，遇到赋值(譬如x=34)，将被定义变量踢出其并查集。
块内对变量的使用，我们的替换方式是替换为其所在并查集内编号最小的变量，如此有利于构成更多的公共子表达式进行合并。

## 可视化




## 难点

并查集可能是一个难点，需要一些数据结构的知识来完成并查集的添加、删除和求交集操作。 删除冗余表达式的时候记得不要删除对数组和全局变量的赋值。

# inlineOptimizer

函数内联优化器
我们进行函数内联的时候，函数是用funcOptimizer包裹起来的，因此发起内联的对象和被内联的对象都是funcOptimizer。

## 发起内联

函数可以发起对另一个非递归函数inline的内联，inline应该不是一个递归函数。
发起内联的函数需要做的事情包括：搜集要传递的参数，删除PASS_ARG和FUNC_CALL四元式，调用被内联函数方法merged获取四元式list插入恰当位置，承接返回值。

## 被内联

被内联的函数会被发起内联的函数调用merged方法，返回要插入对方的四元式list。
被内联的函数要做的事情包括：把GET_ARG换为PASS_GET_ARG，在函数最后插入一个标签用于return跳转，return语句将返回值放到$v0后，替换为跳转到return跳转标签。

## 示例

内联前：

```
void recursion(int x)
{
    if (x < 0)
        return;
    recursion(x--);
}

int func(int x,int y)
{
    printf("%d",x);
    recursion(y);
    return y;
}

int main()
{
    int a=1,b=2,c=3,d=4;
    a=func(a,b);
    c=func(c,d);
    return 0;
}
```

内联后：

```
void recursion(int x)
{
    if (x < 0)
        return;
    recursion(x--);
```

```
}

int main()
{
    int a=1,b=2,c=3,d=4;
    x_0=a;
    y_0=b;
    printf("%d",x_0);
    recursion(y_0);
    a=y_0;
    x_1=c;
    y_1=b;
    printf("%d",x_1);
    recursion(y_1);
    c=y_1;
    return 0;
}
```

## peepHoleOptimizer

本项目的优化在于清除优化时产生的冗余
不难发现，在上述优化过程中很容易产生如下代码：

```
t0 = x + 3
y = t0
```

且t0在之后不再被使用，此时可以进行简单合并为

```
y = x + 3
```

### 难点

无

# 优化的组合

我们似乎没有实现DAG和局部子表达式合并，是这样吗?
我们以书上的一个局部子表达式合并的例子来看看这些优化的组合会有什么效果,假定该块在活跃变量分析的out为{a,b,,d}：

```
t1=-c
t2=b*t1
t3=-c
c=b*t3
```

```
t4=t2+c
a=t4
```

公共子表达式合并:

```
t1=-c
t2=b*t1
t3=t1
c=b*t3
t4=t2+c
a=t4
```

复制传播:

```
t1=-c
t2=b*t1
t3=t1
c=b*t1
t4=t2+c
a=t4
```

再次公共表达式合并:

```
t1=-c
t2=b*t1
t3=t1
c=t2
t4=t2+c
a=t4
```

再次复制传播:

```
t1=-c
t2=b*t1
t3=t1
c=t2
t4=c+c
a=t4
```

最后常量传播+窥孔优化消除冗余赋值:

```
t1=-c
c=b*t1
```

```
a=c+c
```

可以看到，多次迭代之后我们实际上实现了局部公共子表达式合并