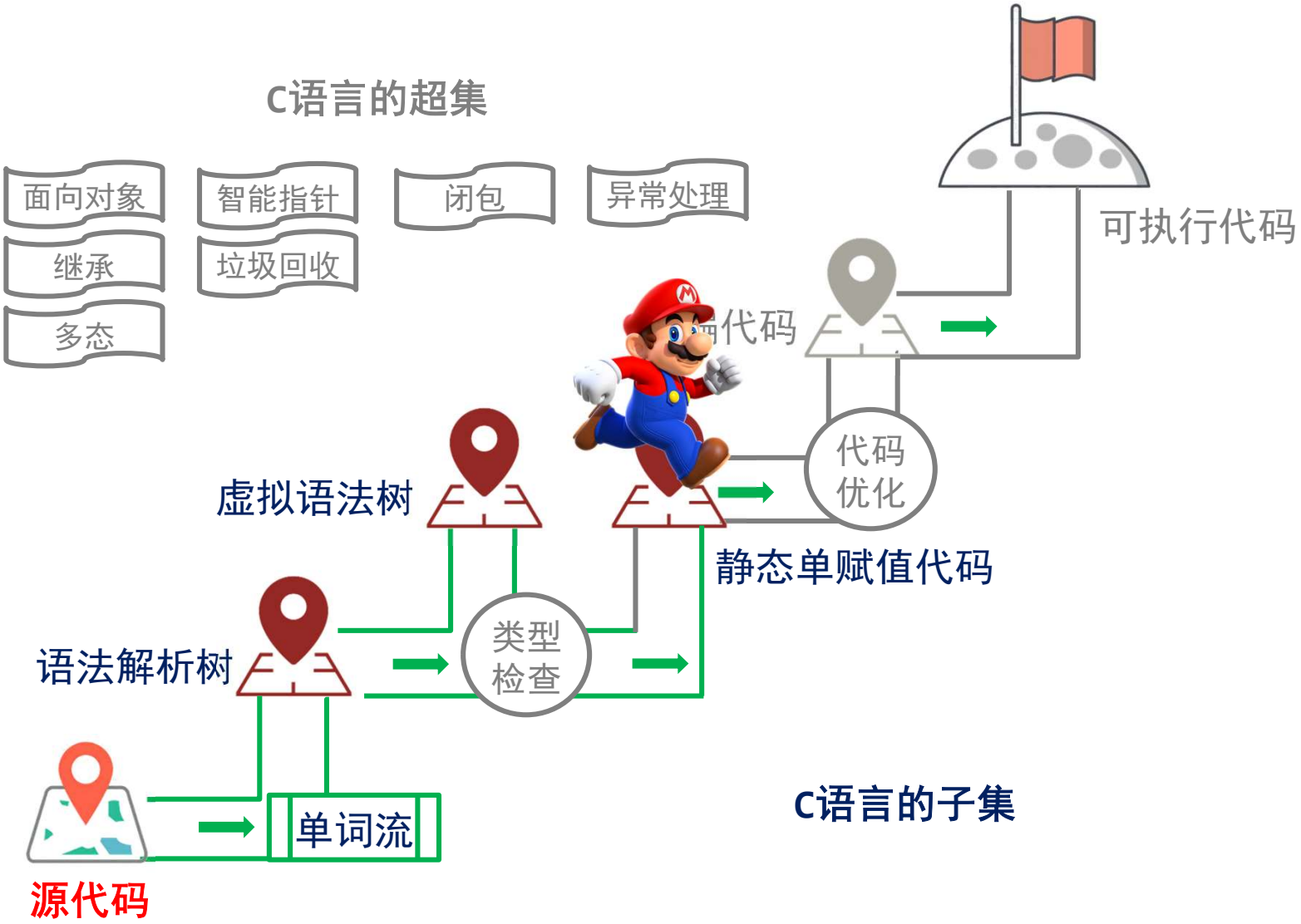
Lecture 4

语法制导和中间代码生成

徐 辉

xuh@fudan.edu.cn

学习地图


C语言的超集
面向对象
智能指针
闭包
异常处理
继承
垃圾回收
多态
可执行代码
代码
代码
优化
虚拟语法树
静态单赋值代码
语法解析树
类型
检查
单词流
源代码
C语言的子集

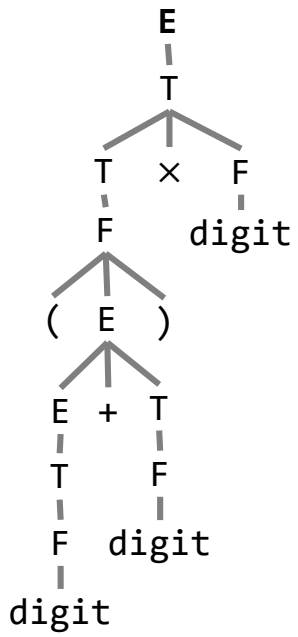
回顾：

- CFG解决了哪些问题？
  - 准确理解句子，生成语法解析树
- CFG语法分析尚未解决的问题
  - 如何生成目标代码？解析树怎么用？
  - 缺少上下文相关分析，可解析的程序未必正确

(a + b) × 2

Production

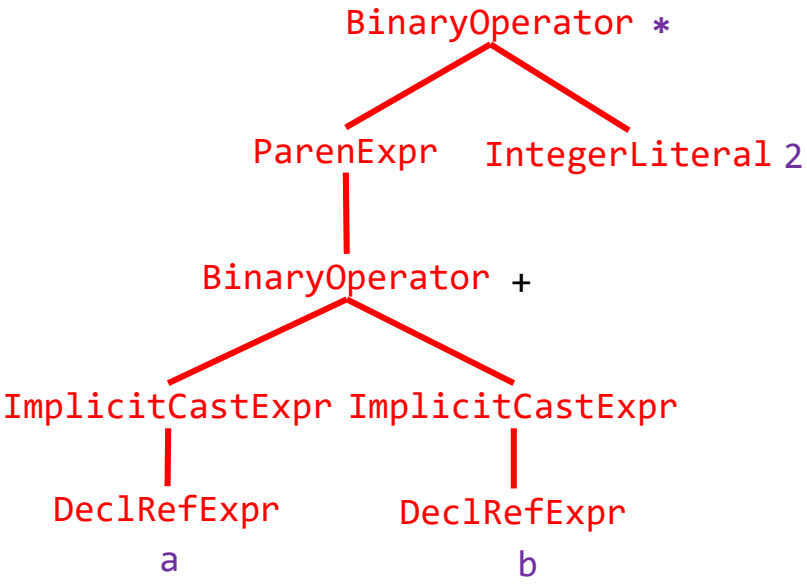
1) $E \rightarrow E + T$
2) $E \rightarrow T$
3) $T \rightarrow T \times F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow digit$



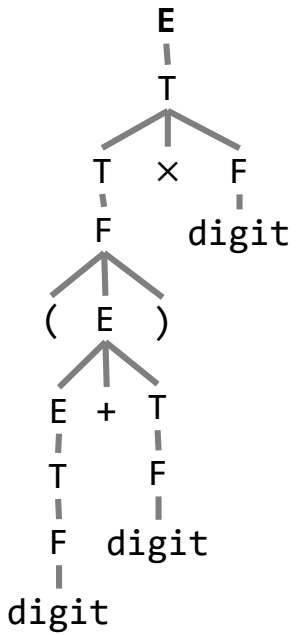
语法解析树

我们需要什么？

- 虚拟语法树（树型IR）
  - 语法解析树太复杂
- 线性IR
  - 复合计算机的计算方式



LLVM虚拟语法树



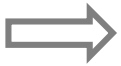
语法解析树

```
%6 = load i32, i32* %3, align 4
%7 = load i32, i32* %4, align 4
%8 = add nsw i32 %6, %7
%9 = mul nsw i32 %8, 2
```

LLVM线性IR

展开While、If-Else等语法糖

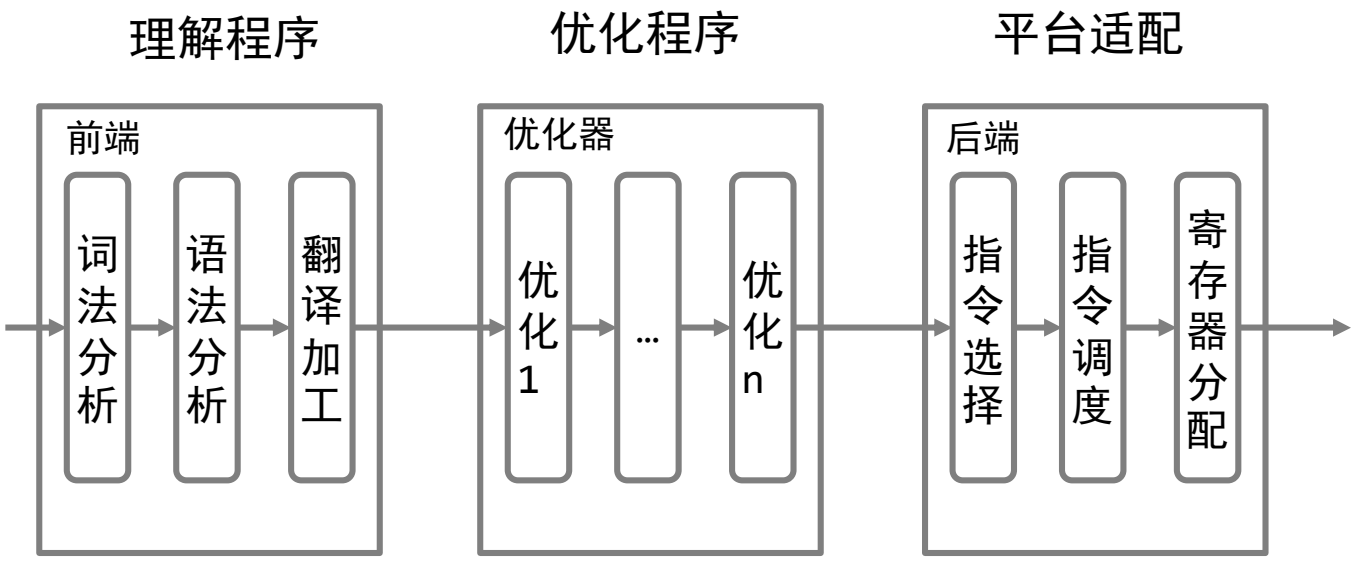
```
int main(){
  int s = 1, r = 1;
  while (r < 100){
    if (s < r)
      s = s+r;
    else r = s+r;
  }
  return r;
}
```



```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1, i32* %2, align 4
  store i32 1, i32* %3, align 4
  br label %4
4 (Basic Block):                                  ; preds = %19, %0
  %5 = load i32, i32* %3, align 4
  %6 = icmp slt i32 %5, 100
  br i1 %6, label %7, label %20
7 (Basic Block):                                  ; preds = %4
  %8 = load i32, i32* %2, align 4
  %9 = load i32, i32* %3, align 4
  %10 = icmp slt i32 %8, %9
  br i1 %10, label %11, label %15
11 (Basic Block):                                 ; preds = %7
  %12 = load i32, i32* %2, align 4
  %13 = load i32, i32* %3, align 4
  %14 = add nsw i32 %12, %13
  store i32 %14, i32* %2, align 4
  br label %19
15 (Basic Block):                                 ; preds = %7
  %16 = load i32, i32* %2, align 4
  %17 = load i32, i32* %3, align 4
  %18 = add nsw i32 %16, %17
  store i32 %18, i32* %3, align 4
  br label %19
19 (Basic Block):                                 ; preds = %15, %11
  br label %4
20 (Basic Block):                                 ; preds = %4
  %21 = load i32, i32* %3, align 4
  ret i32 %21
}
```

为什么不直接转换为汇编代码？

- 模块化考虑：
  - 前台负责理解程序：语言可以不同，中间代码相同
  - 后端负责翻译汇编：CPU指令集可以不同，中间代码相同
  - 中间代码格式相对稳定：方便优化算法设计和开发



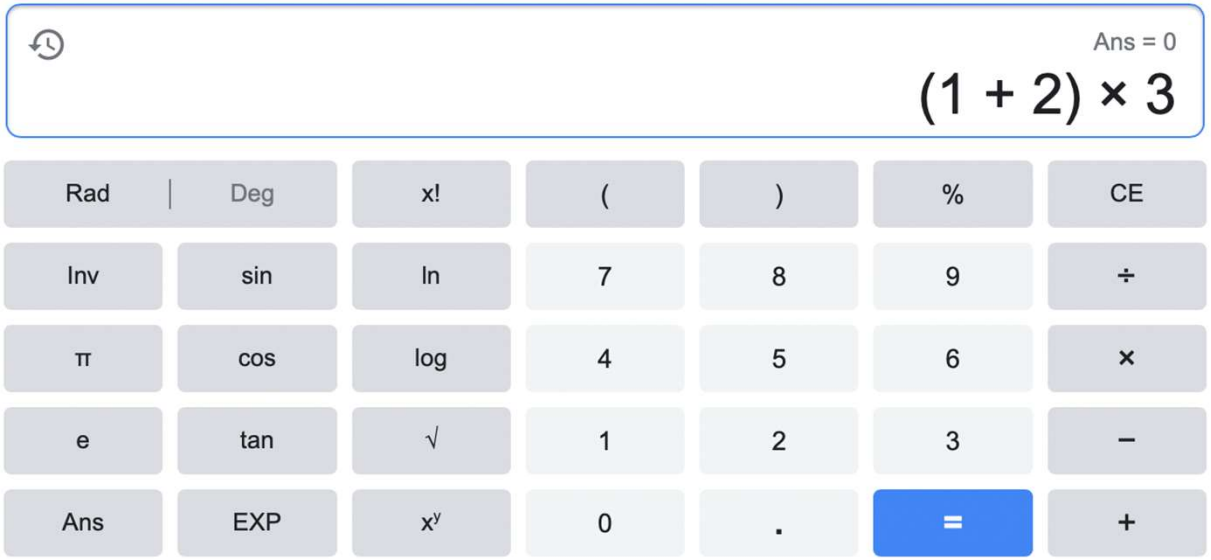
大纲

- 一、属性语法
- 二、中间代码生成
- 三、静态单赋值
- 四、LLVM IR案例分析

一、属性语法

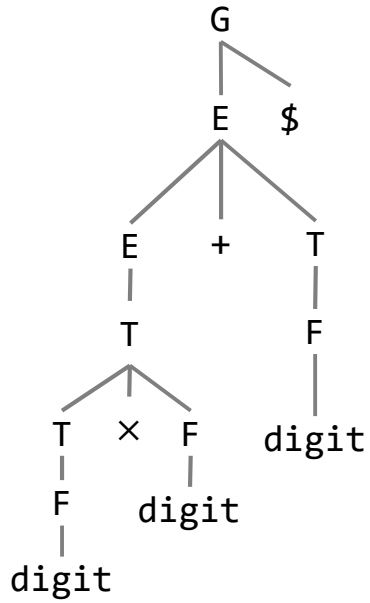
举例：计算器程序

- 计算器可看作一个简单的编译器
- 如何计算结果



如何完成上述计算？

Production	**Semantic Rules**
1) $L \rightarrow E\ \$$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 \times F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F = E.val$
7) $F \rightarrow digit$	$F = digit.lexval$



语法解析树：3×5+4$

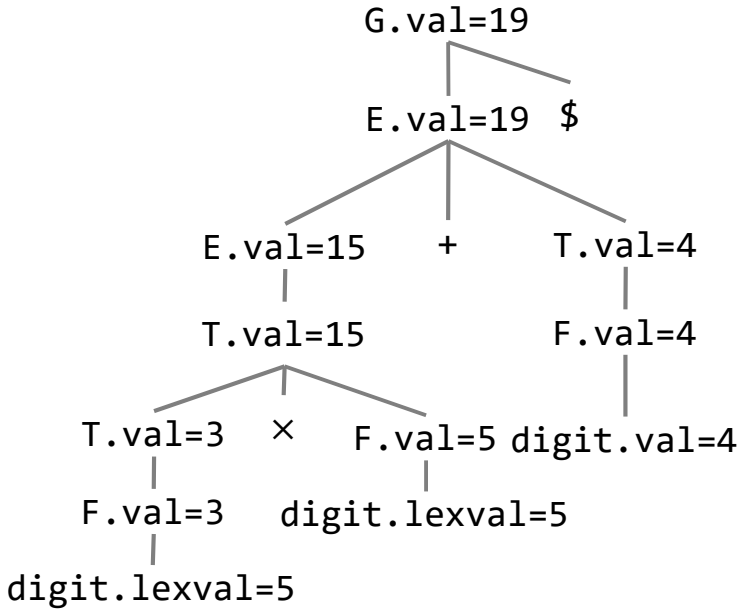
语法制导：Syntax-Directed Translation

- 语法制导定义（SDD，Syntax-Directed Definition）是由上下文无关文法、属性、和规则组成的。
  - 属性（attribute）：语法符号相关的信息
    - 数字、类型、引用、字符串（代码）等
    - 包括合成属性和继承属性
  - 规则（rule）：属性的计算方法

合成属性：Synthesized Attribute

- 解析树上非终结符节点A的属性是根据其子节点的语义规则定义的。

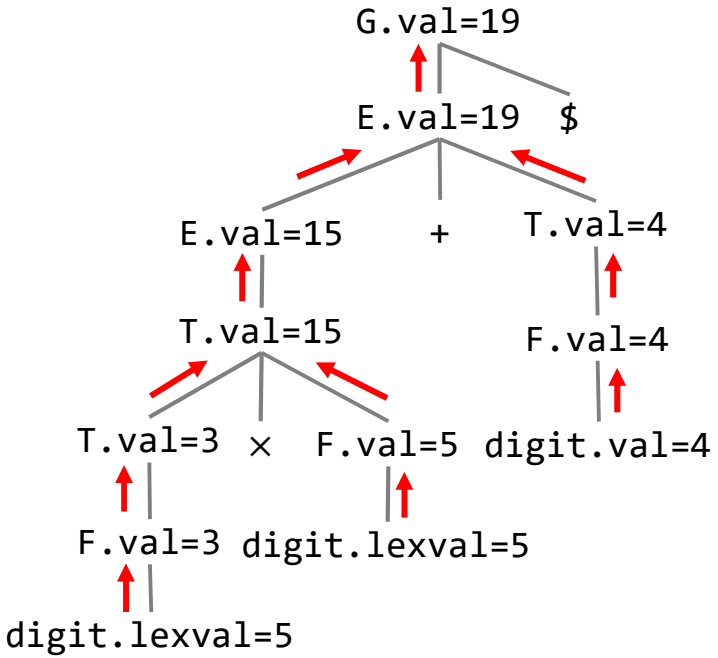
Production	**Semantic Rules**
1) $G \rightarrow E\ \$$	$G.val = E.val$
2) $E \rightarrow E_1 + \mathrm{T}$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 \times F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F = E.val$
7) $F \rightarrow digit$	$F.val = digit.lexval$



带注解的语法解析树：3×5+4$

S-attaibuted SDD

- 所有的属性都是合成属性的SDD；
- 适合自底向上（如LR）的解析算法，为什么？
  - 解析树构建采用“后序遍历”；
  - 遍历子节点后即满足了根节点属性计算依赖；
  - 解析和属性计算可以一趟完成。



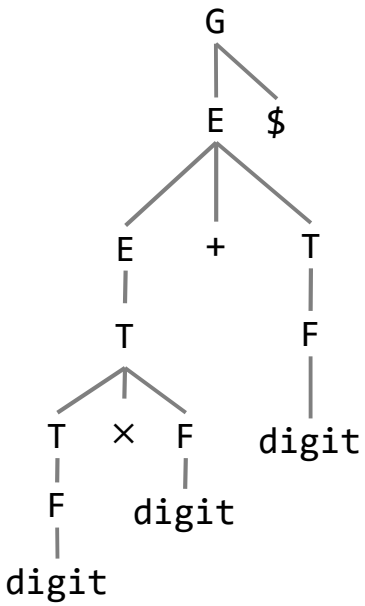
属性依赖关系图

属性依赖关系图：

- 点：语法解析树上符号的属性
- 边：依赖关系

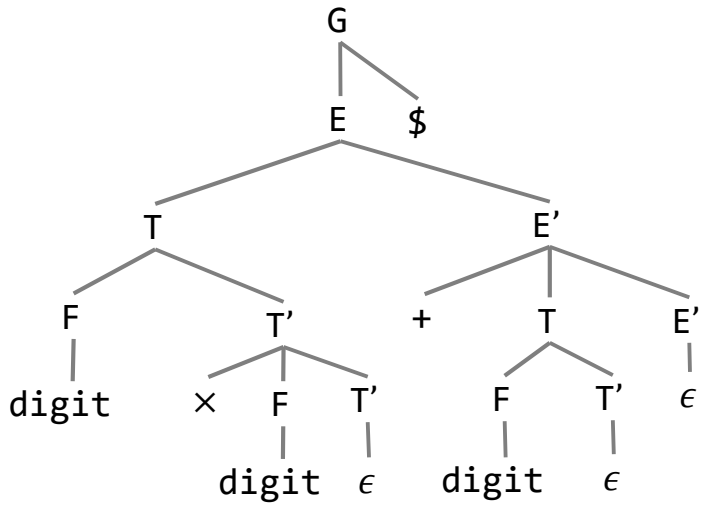
LL(1)语法如何处理

1) $G \rightarrow E\ \$$
2) $E \rightarrow E + \mathrm{T}$
3) $E \rightarrow T$
4) $T \rightarrow T \times F$
5) $T \rightarrow F$
6) $F \rightarrow (E)$
7) $F \rightarrow digit$



语法解析树

1) $G \rightarrow E\ \$$
2) $E \rightarrow TE'$
3) $E' \rightarrow +TE'$
4) $E' \rightarrow \epsilon$
5) $T \rightarrow FT'$
6) $T' \rightarrow \times FT'$
7) $T' \rightarrow \epsilon$
8) $F \rightarrow (E)$
9) $F \rightarrow digit$

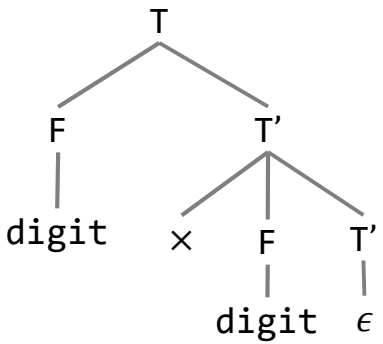


语法解析树

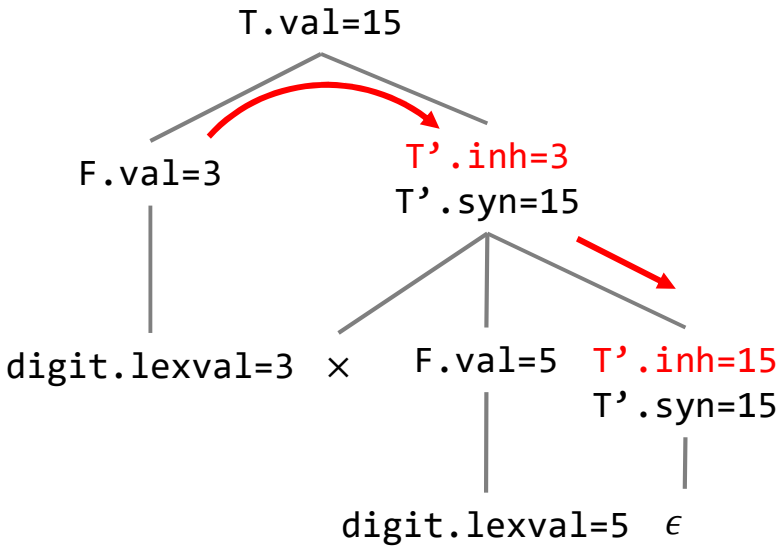
继承属性：Inherited Attaibute

- 解析树上节点β的属性是根据其父节点（$A \to \beta_1\beta_2\beta_3$）的生成式语义规则确定的。
  - 基于其父节点A
  - 或兄弟节点β_1、β_3

Production	Semantic Rules
1) $T \to FT'$	$T'.inh = F.val$
	$T.val = T'.syn$
2) $T' \to \times FT_1'$	$T_1'.inh = T'.inh \times F.val$
	$T'.syn = T_1'.syn$
3) $T' \to \epsilon$	$T'.syn = T'.inh$
4) $F \to digit$	$F.val = digit.lexval$



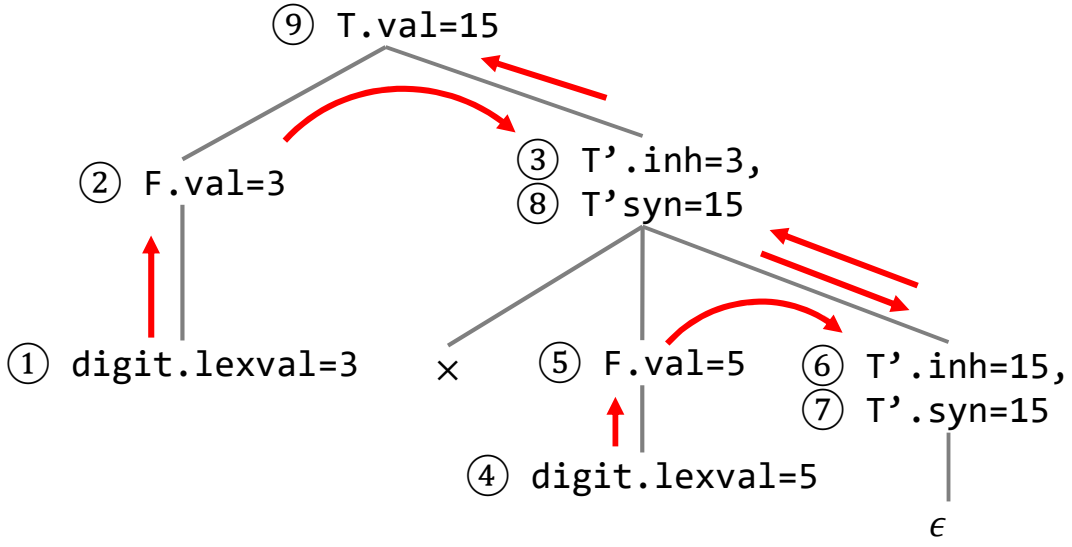
语法解析树：3×5



带注解的语法解析树：3×5

基于L-attributed SDD

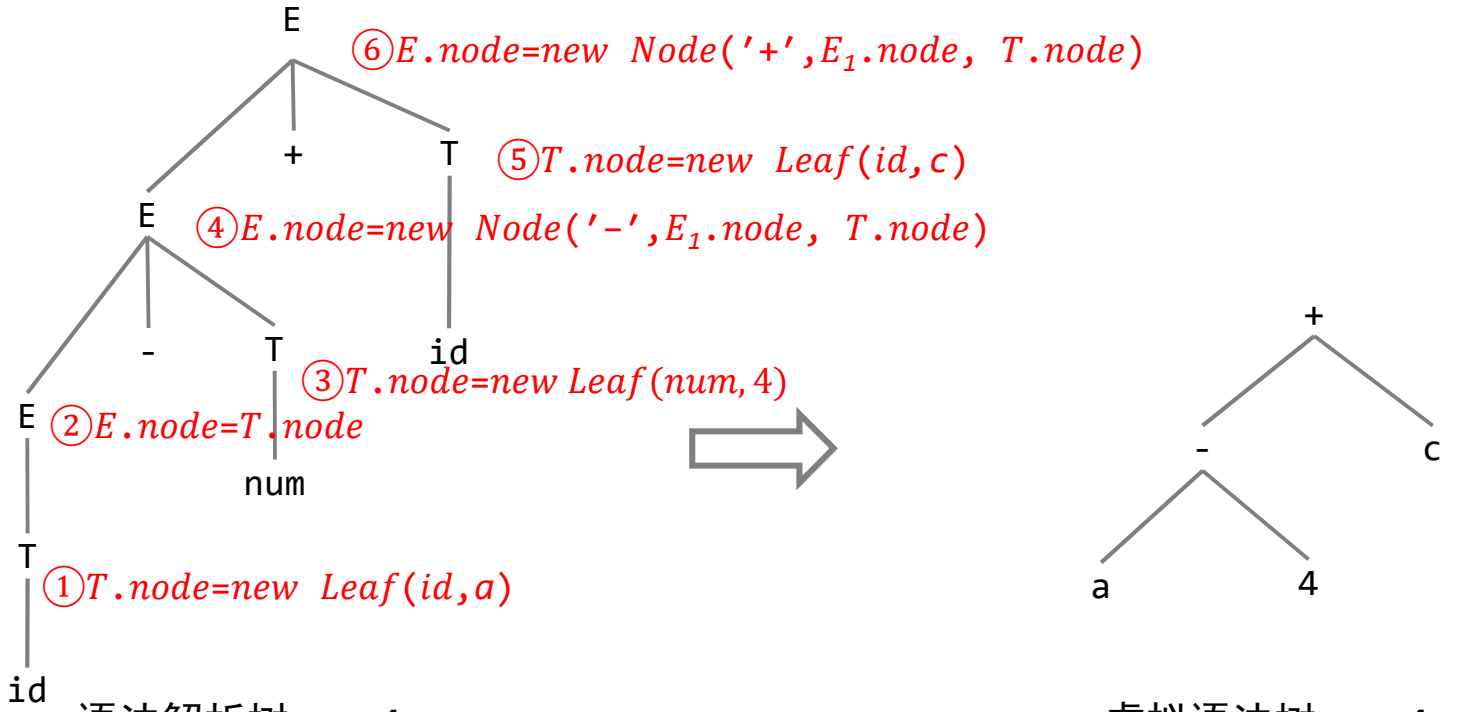
- 所有的属性都是合成属性，或对于$A \to \beta_1..\beta_i..\beta_n$中的任意$\beta_i$来说，其继承属性只依赖$A$或$\beta_1, \ldots, \beta_{i-1}$
- 适合自顶向下的解析算法，为什么？
  - 解析树构建采用“前序遍历”，最后访问右孩子节点；
  - L-Attributed SDD的继承属性计算依赖最后访问右孩子节点。



属性语法的应用

- 对CFG语法规则的含义进行定义或解释：
  - 对应的计算指令是什么？
  - 应如何转换为相应的中间代码？
    - AST
  - 是否暗含上下文敏感信息？
    - 类型约束

基于S-attaibuted SDD构建AST

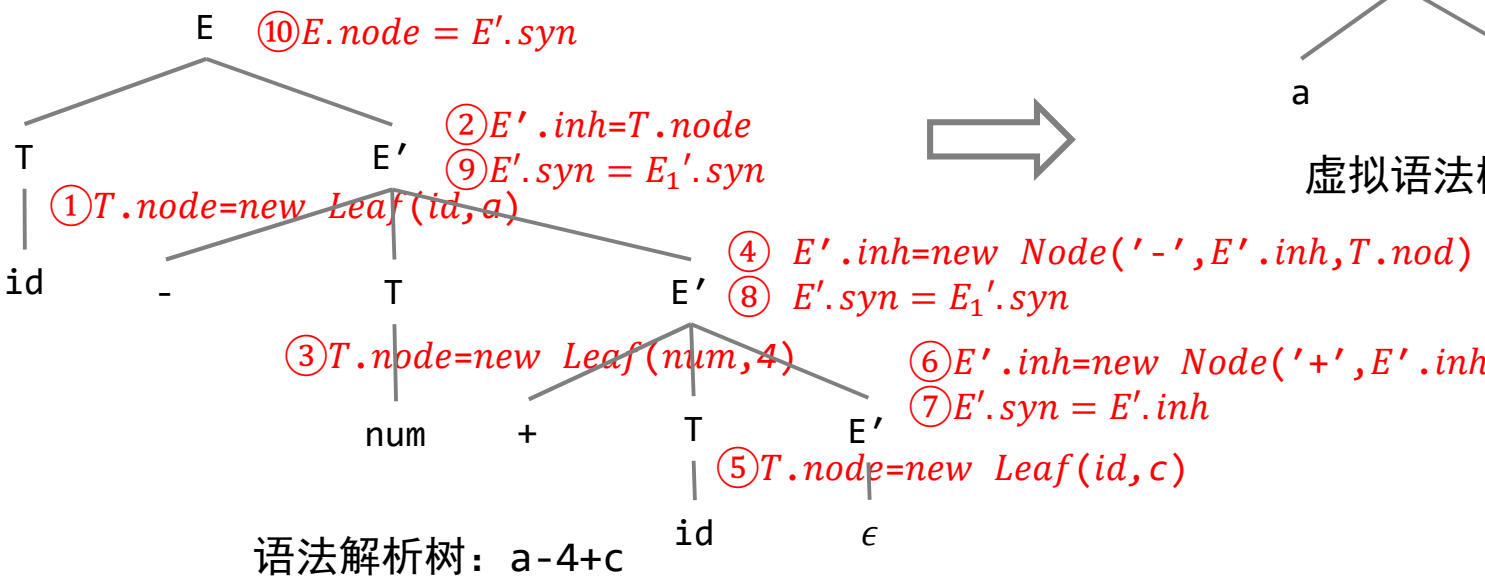


语法解析树：a-4+c

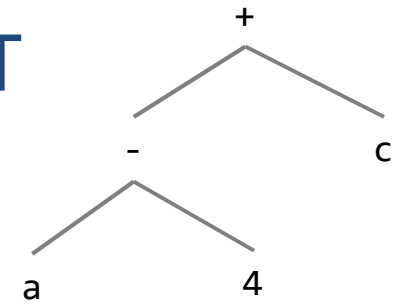
虚拟语法树：a-4+c

Production	**Semantic Rules**
1) $E \rightarrow E_1 + \mathrm{T}$	$E.node = new\ Node('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = new\ Node('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$T.node = E.node$
5) $T \rightarrow id$	$T.node = new\ Leaf(id, id.entry)$
6) $T \rightarrow num$	$T.node = new\ Leaf(num, num.val)$

基于L-attaibuted SDD构建AST



语法解析树：a-4+c

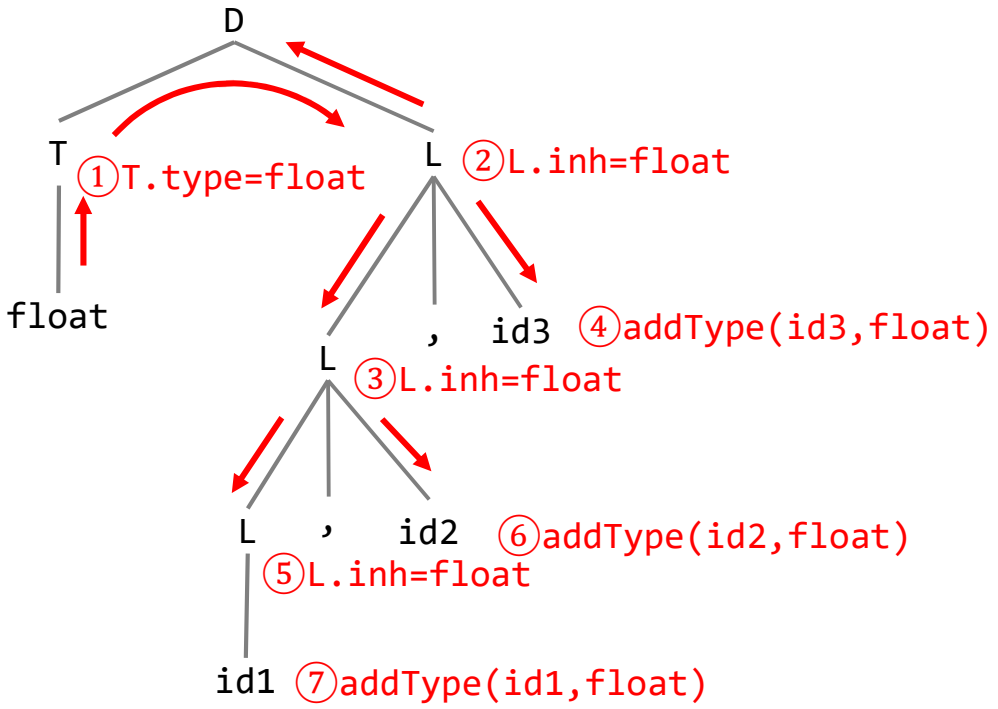


虚拟语法树：a-4+c

Production	**Semantic Rules**
1) $E \to T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2) $E' \to +T\ E_1'$	$E'.inh = new\ Node('+', E'.inh, T.nod)$ $E'.syn = E_1'.syn$
3) $E' \to -T\ E_1'$	$E_1.inh = new\ Node('-', E'.inh, T.node)$ $E'.syn = E_1'.syn$
4) $E' \to \epsilon$	$E'.syn = E'.inh$
5) $T \to (E)$	$T.node = E.node$
6) $T \to id$	$T.node = new\ Leaf(id, id.entry)$
7) $T \to num$	$T.node = new\ Leaf(num, num.val)$

生成类型约束

Production	Semantic Rules
1) $D \rightarrow T\ L$	$L.inh = T.type$
2) $T \rightarrow int$	$T.type = integer$
3) $T \rightarrow float$	$T.type = float$
4) $L \rightarrow L_1, id$	$L_1.inh = L.inh$ $addType(id.entry, L.inh)$
5) $L \rightarrow id$	$addType(id.entry, L.inh)$

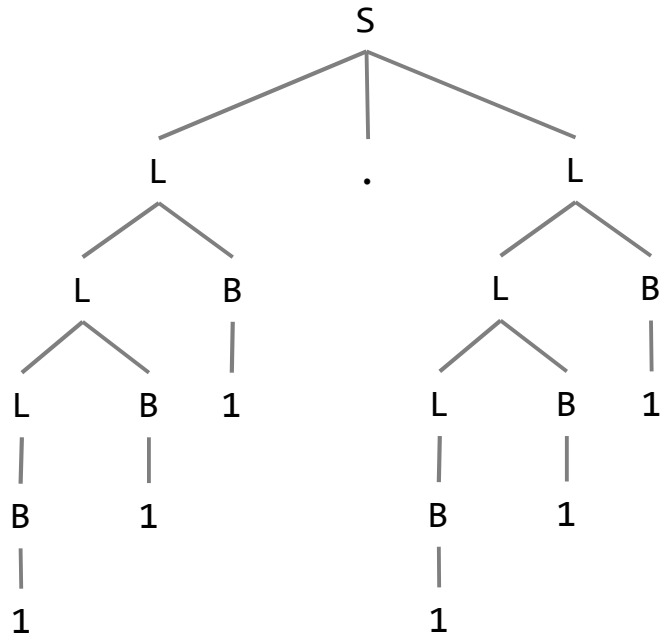


float id1, id2, id3

练习

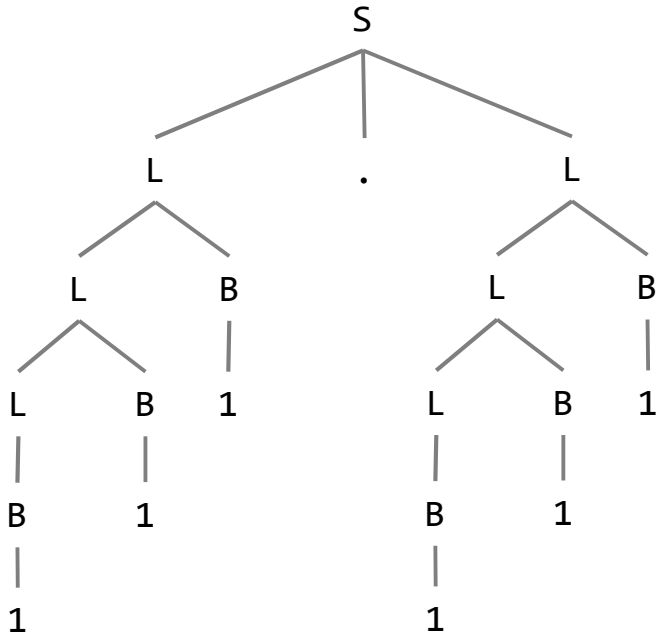
- 下列语法可以解析二进制数。
  1) 设计S-attaibuted SDD将其转化为十进制数；
  2) 设计L-attaibuted SDD将其转化为十进制数。
     1) 将整数/小数部分的信息传递给子树。
     - 例如：如101.101的对应的十进制数是5.625。

[1] $S \rightarrow L.L_1$
[2] $\quad\quad | L$
[3] $L \rightarrow L_1 B$
[4] $\quad\quad | B$
[5] $B \rightarrow 0$
[6] $\quad\quad | 1$



参考答案：S-attaibuted SDD

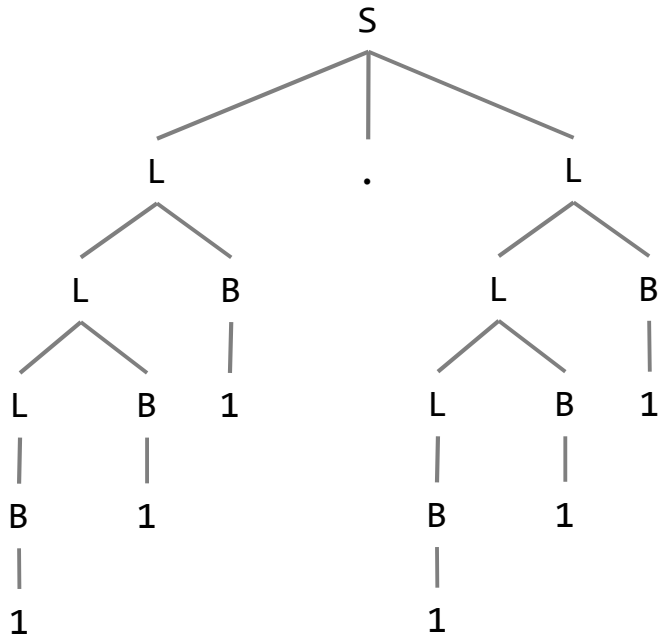
[1]	$S \rightarrow L.L_1$	$\{S.val = L.val + L_1.val/L_1.frac\}$
[2]	$\mid L$	$\{L.val = L.val\}$
[3]	$L \rightarrow L_1 B$	$\{L.val = L_1.val \times 2 + B.val; L.frac = L_1.frac \times 2\}$
[4]	$\mid B$	$\{L.val = B.val; L.frac = 2\}$
[5]	$B \rightarrow 0$	$\{B.val = 0\}$
[6]	$\mid 1$	$\{B.val = 1\}$



- 主要问题：会冗余计算整数部分的$L.frac$。

参考答案：L-attaibuted SDD

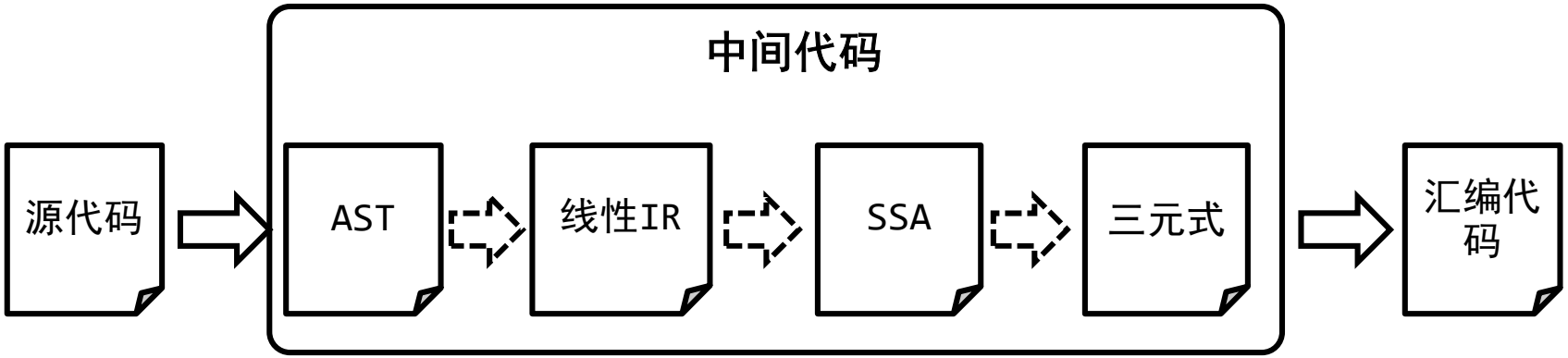
[1]	$S \rightarrow L.L_1$	$\{S.val = L.val + L_1.val; L.isFrac = false; L_1.isFrac = true\}$
[2]	$\mid L$	$\{S.val = L.val; L.isFrac = false\}$
[3]	$L \rightarrow L_1 B$	$\{ L_1.isFrac = L.isFrac; L.pos = L_1.pos + 1;$ $L.val = L_1.isFrac?\ L_1.val + B.val/2^{L_1.pos} : L_1.val * 2 + B.val$
[4]	$\mid B$	$\{L.pos = 1; L.val = L.isFrac?\ B.val/2 : B.val\}$
[5]	$B \rightarrow 0$	$\{B.val = 0\}$
[6]	$\mid 1$	$\{B.val = 1\}$



二、中间代码生成

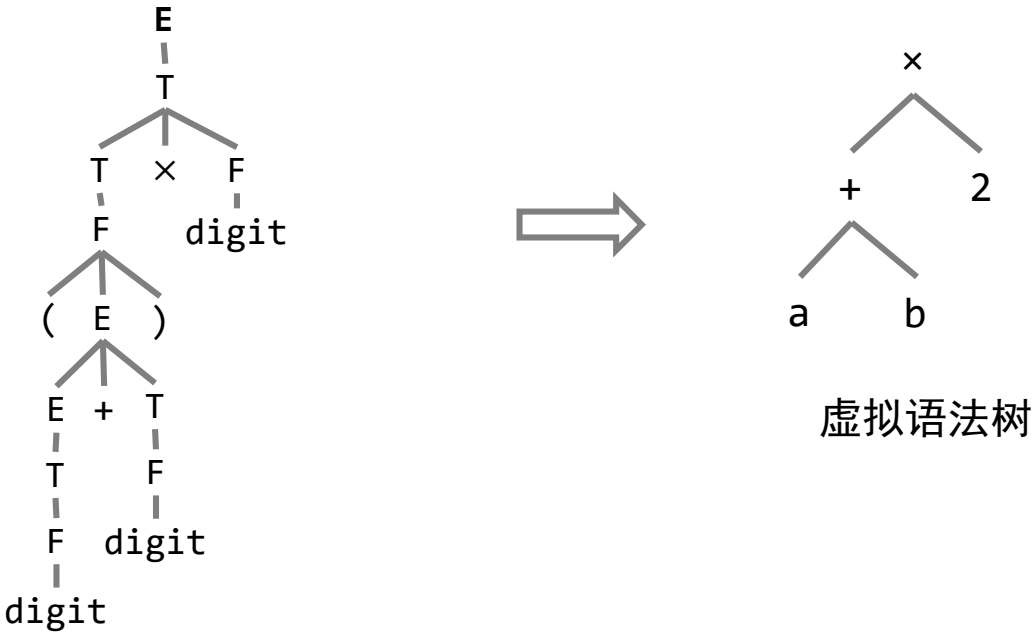
中间代码生成过程

- 主要目标是生成接近CPU指令的SSA；
- 基于SSA代码更容易进行代码优化。



创建虚拟语法树AST

- Concrete Syntax：程序员实际写的代码
  - 解析源代码得到的语法解析树比较大，它是对源代码的完整表示。
- Abstract Syntax：编译器实际需要的内容
  - 虚拟语法树，消除推导过程中的一些步骤或节点得到抽象语法树。
    - 运算符和关键字不再是叶子结点
    - 单一展开形式塌陷，如E->T->F->digit
    - 去掉括号等冗余信息



虚拟语法树

语法解析树：(a + b) × 2

AST的本质

- 记录程序信息的数据结构；
- 更接近我们之前定义的有问题的CFG语法；
- AST使用树形结构记录不同运算之间的先后顺序；
- 需要事先约定不同类型节点的子树结构和遍历顺序。

[1] $Expr \rightarrow Expr\ Bop\ Expr$
[2] $\quad | num$
[3] $\quad | (Expr)$
[4] $Bop \rightarrow +$
[5] $\quad | -$
[6] $\quad | \times$
[7] $\quad | \div$

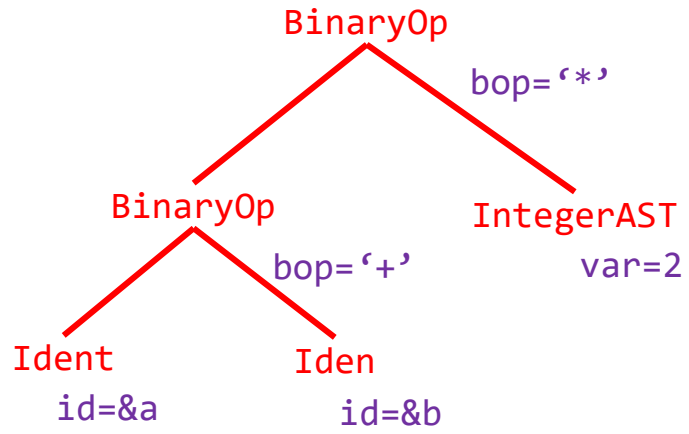
$< regex > ::= < union > | < concat >$
$\quad | < closure > | < term >$
$< union > ::= < regex >$ "|" $< regex >$
$< concat > ::= < regex >< regex >$
$< closure > ::= < regex > *$
$< term > ::= < group > | < alphanum >$
$< group > ::= (< regex >)$

AST的节点类型

- 每一个实例化的AST结点类型都有固定的子树结构。

```
class ExprAST{}
class BinOpAST : ExprAST{
    char bop;
    ExprAST* lhs, rhs;
}
class IntegerAST : ExprAST{
    int var;
}
class IdentAST : ExprAST{
    char id;
}
```

四则运算的AST节点

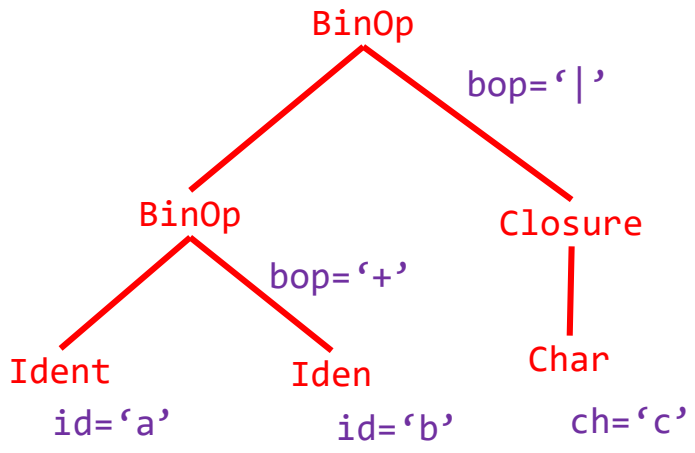


(a+b)*2虚拟语法树

更多AST的例子

```
class RegexAST{}
class BinOpAST : RegexAST{
    char bop;
    ExprAST* lhs, rhs;
}
class ClosureAST : RegexAST{
    RegexAST* reg;
}
class CharAST : RegexAST{
    char ch;
}
```

Regex的AST节点



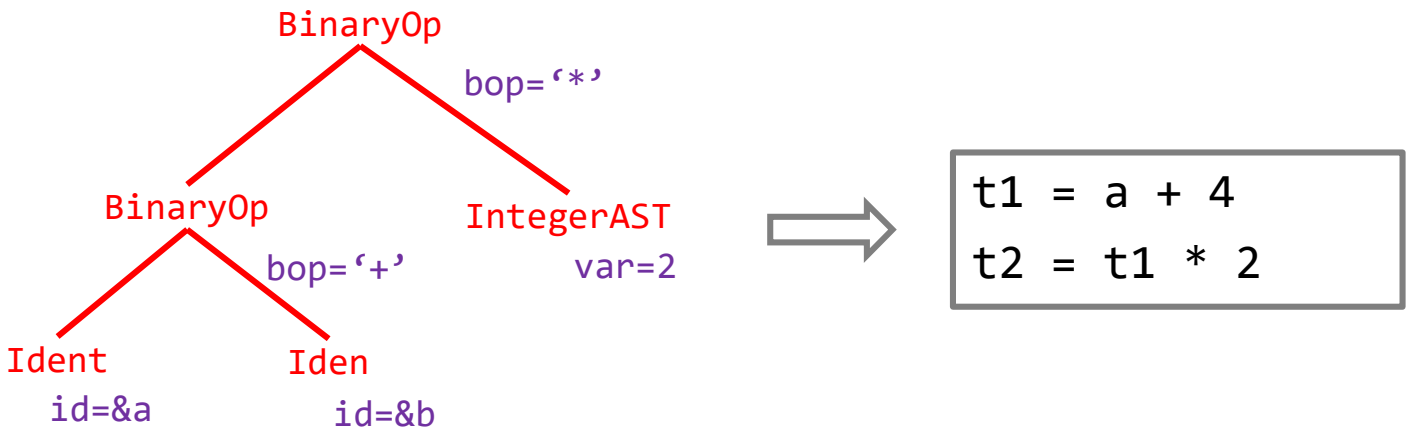
ab|c*虚拟语法树

同理，If语句的AST...

```
class IfStmtAST : StmtAST{
    CondStmtAST* cond;
    CompoundStmtAST* thenblock;
    CompoundStmtAST* elseblock;
}
```

翻译成线性IR

- 递归下降将AST翻译为线性IR；
- 三地址代码是线性IR，由指令和地址组成；
- 地址可以是：
  - 变量名
  - 常量
  - 编译器生成的临时变量或存储单元



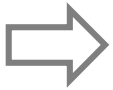
(a+b)*2虚拟语法树

翻译成线性IR

- 基本的三地址IR
  - 二元运算符（binary operator）赋值：x = y op z
  - 一元运算符（unary operator）赋值：x = op y
  - 拷贝赋值：x = y
  - 数组操作：x = y[i]; x[i] = y
  - 指针和地址操作：x = &y; x = *y; *x = y
- 需要特殊处理：
  - 控制流语句：If/If-Else/While/For/Switch-Case
  - 函数调用：y = f(x1,…,xn)

控制流语句：If-Else

```
if(x==0)
    x = 1;
else
    x = -1;
y = x * a;
```



```
b = x==1;
ifFalse b goto falseBB
trueBB:
    x = 1;
    goto nextBB;
falseBB:
    x = -1;
    goto nextBB;
nextBB:
    y = x * a;
```

如何生成线性IR？

- 递归下降遍历AST树
- 或直接基于属性语法

CFG语法规则

IfStmt → if ($cond$)
trueBB
else
falseBB

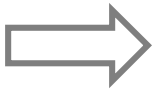
属性语法

$$trueLabel = newBBLabel()$$
$$falseLabel = newBBLabel()$$
$$trueBB.next = falseBB.next = IfStmt.next$$
$$IfStmt.code = cond.code \,||\, trueLabel \,||\, trueBB.code \,||\, codegen('goto'\ trueBB.next) \,||\, falseLabel \,||\, falseBody.code \,||\, codegen('goto'\ falseBB.next)$$

```
    cond.code
trueLabel:
    ifBB.code
    goto IfStmt.next
falseLabel:
    elseBB.code
    goto IfStmt.next
IfStmt.next:
```

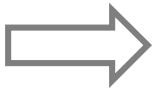
控制流语句：While/For

```
while(i++<100)
    x = x + 2;
y = x * a;
```



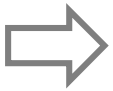
```
condBB:
    b = x<100;
    i = i+1;
ifFalse b goto nextBB
trueBB:
    x = 1;
    goto condBB;
nextBB:
    y = x * a;
```

```
for(;i!=100;i++)
    x = x + 2;
y = x * a;
```



控制流语句：Switch-Case

```
swith(i){
    case 0:
        x = 1;
        break;
    case 1:
        x = 100;
        break;
    default:
        x = -1;
        break;
}
y = x * a;
```

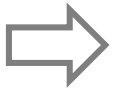


```
switch i, bbDefault[
    0, bbCase0;
    1, bbCase1;
]
bbCase0:
    x = 1;
    goto bbNext
bbCase1:
    x = 0;
    goto bbNext
bbDefault:
    x = -1;
    goto bbNext;
bbNext:
    y = x*a;
```

过程调用翻译成线性IR

- 结合CPU函数调用（calling convention）的特点；
  - 先将参数分别存入寄存器/栈
  - 然后跳转到被调函数
- 一般会在SSA之后才进行翻译。

```
y = f(x1,x2,…,xn)
```

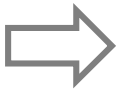


```
param x1;
param x2;
…
param xn;
y = call F,n
```

静态单赋值IR：SSA

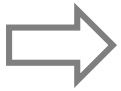
- 每个变量仅赋值一次，再次赋值需要重命名，如x1、x2；
- 同一变量的不同控制流采用不同变量名；
- 汇合节点使用Phi函数。

```
a = b * -c;
a = a + 1;
b = a + 1;
```



```
a = b * -c;
a1 = a + 1;
b1 = a1 + 1;
```

```
if(b)
    x = 1;
else
    x = -1;
y = x * a;
```



```
ifFalse b goto falseBB
trueBB:
    x1 = 1;
    goto nextBB;
falseBB:
    x2 = -1;
    goto nextBB;
nextBB:
    y = Phi(x1, x2) * a;
```

四元式和三元式

- 将线性IR转换为四元式；
- 四元式（尤其是SSA）的结果项多为临时变量，可在三元式中消除
  - 采用指令位置替代

三地址线性IR　　四元式　　三元式

op	arg1	arg2
mov	a	r1
add	b	r1
*	2	r1

两地址形式

```
mov   -4(%rbp), %eax
add   -8(%rbp), %eax
shl   $1, %eax
```

实际汇编代码

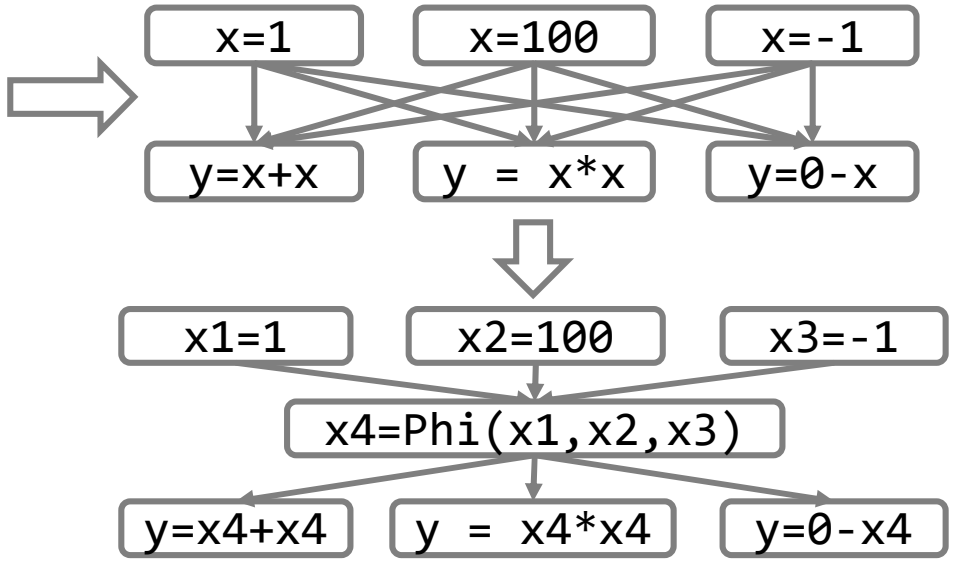
三、静态单赋值

Static Single Assignment

SSA

- 1988年由Barry K. Rosen等人提出
- 传统数据流分析需要很多pass
- 通过SSA简化变量的def-use关系
  - 分析数据流关系无需再考虑CFG;
  - 原始程序的def-use关系数量是$O(n^2)$;
  - SSA的def-use数量减少为$O(n)$。

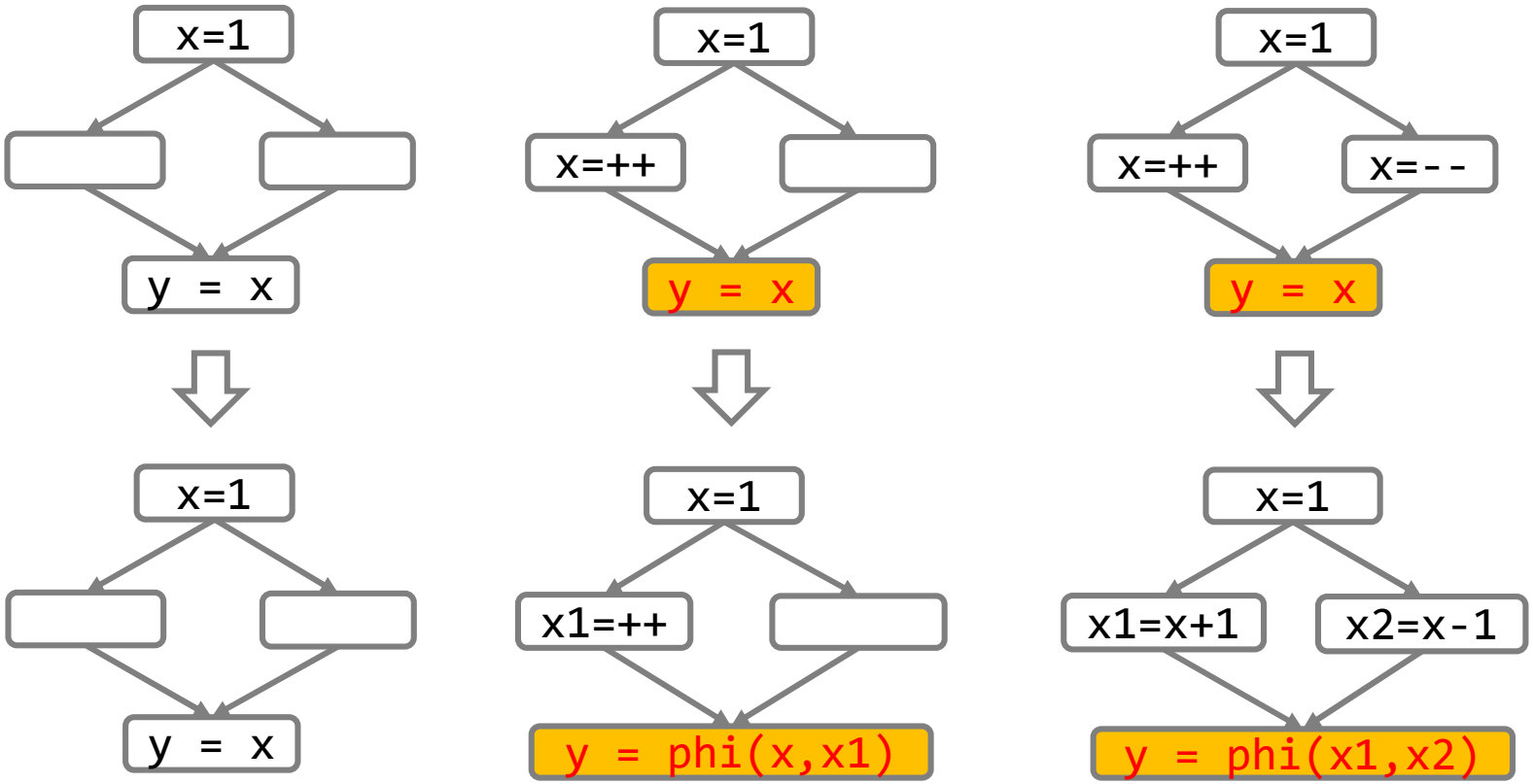
```
switch…
case 0: x = 1; break;
case 1: x = 100; break;
default: x = -1; break;
…
switch…
case 1: y = x+x; break;
case 2: y = x*x; break;
default: y = 0 – x; break;
```



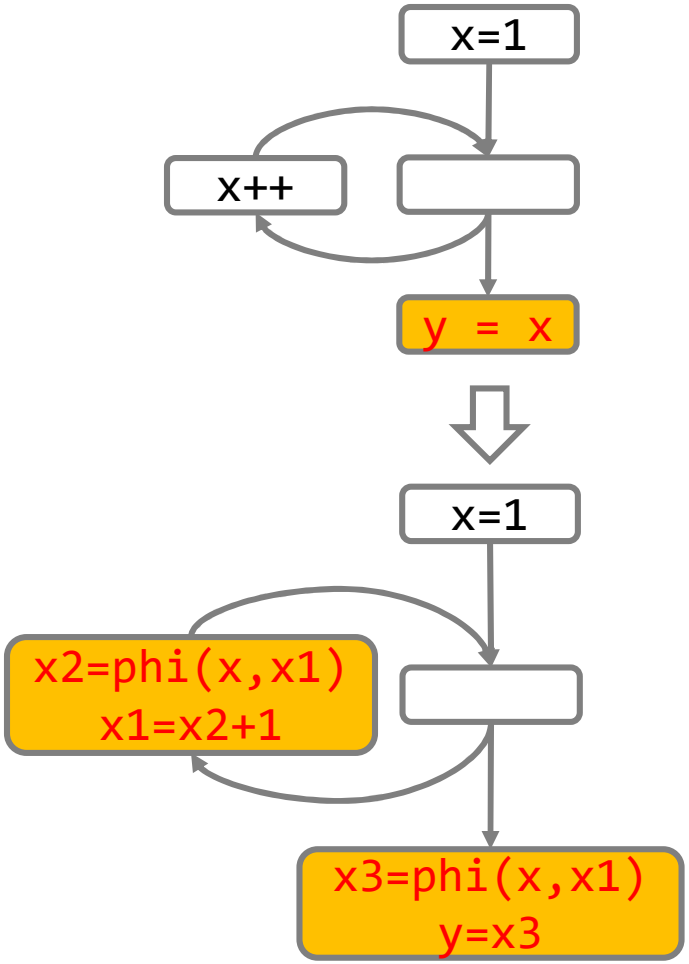
SSA的特点和构建思路

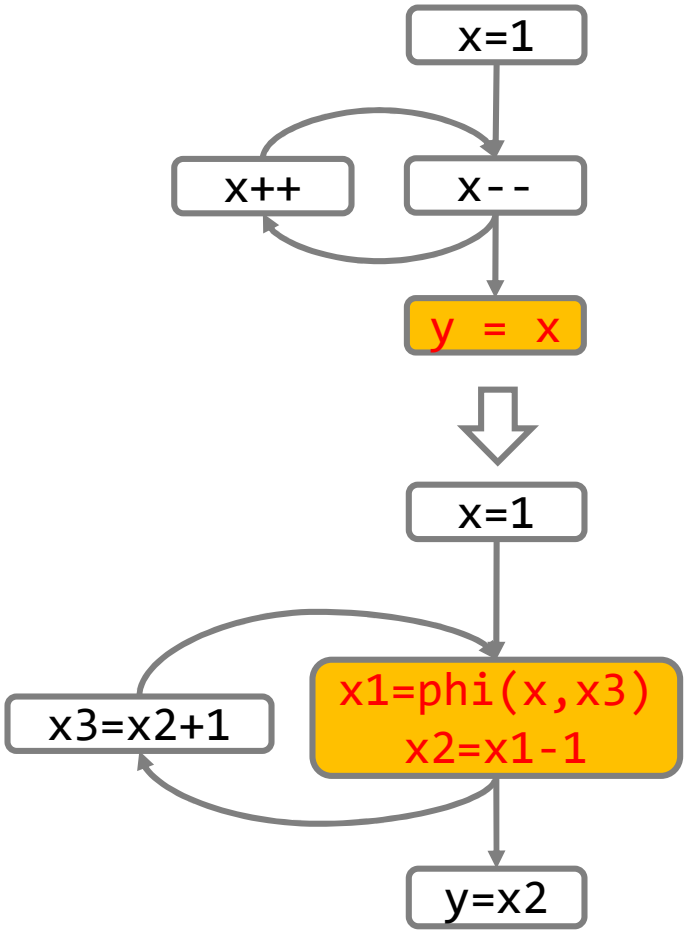
- SSA的要求：
  - 每个变量仅被赋值1次；
  - 每个变量在使用前已经被定义；
  - 使用phi函数解决控制流带来的[def_1,def_2]-use问题。
- 关键问题：
  - 哪些节点需要使用phi函数？
  - 对哪些变量使用phi函数？
  - 每个变量的ssa标识符是什么？

哪些节点需要使用phi函数？



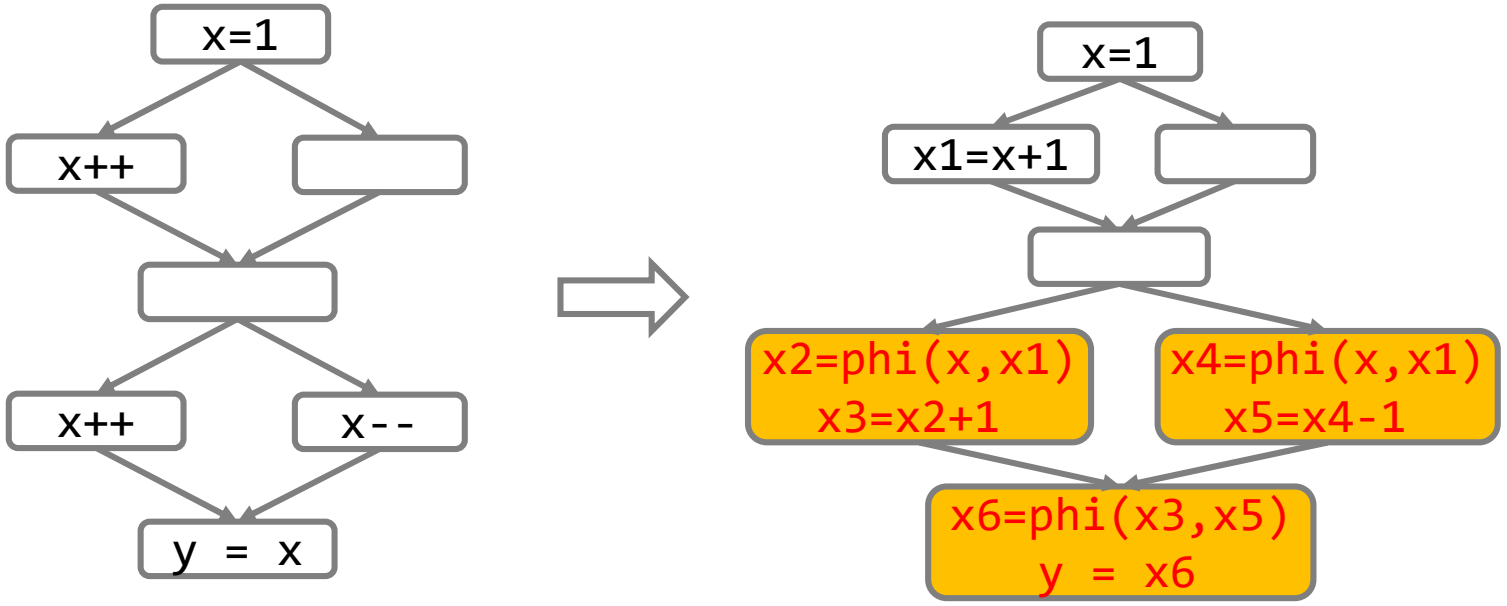
哪些节点需要使用phi函数？





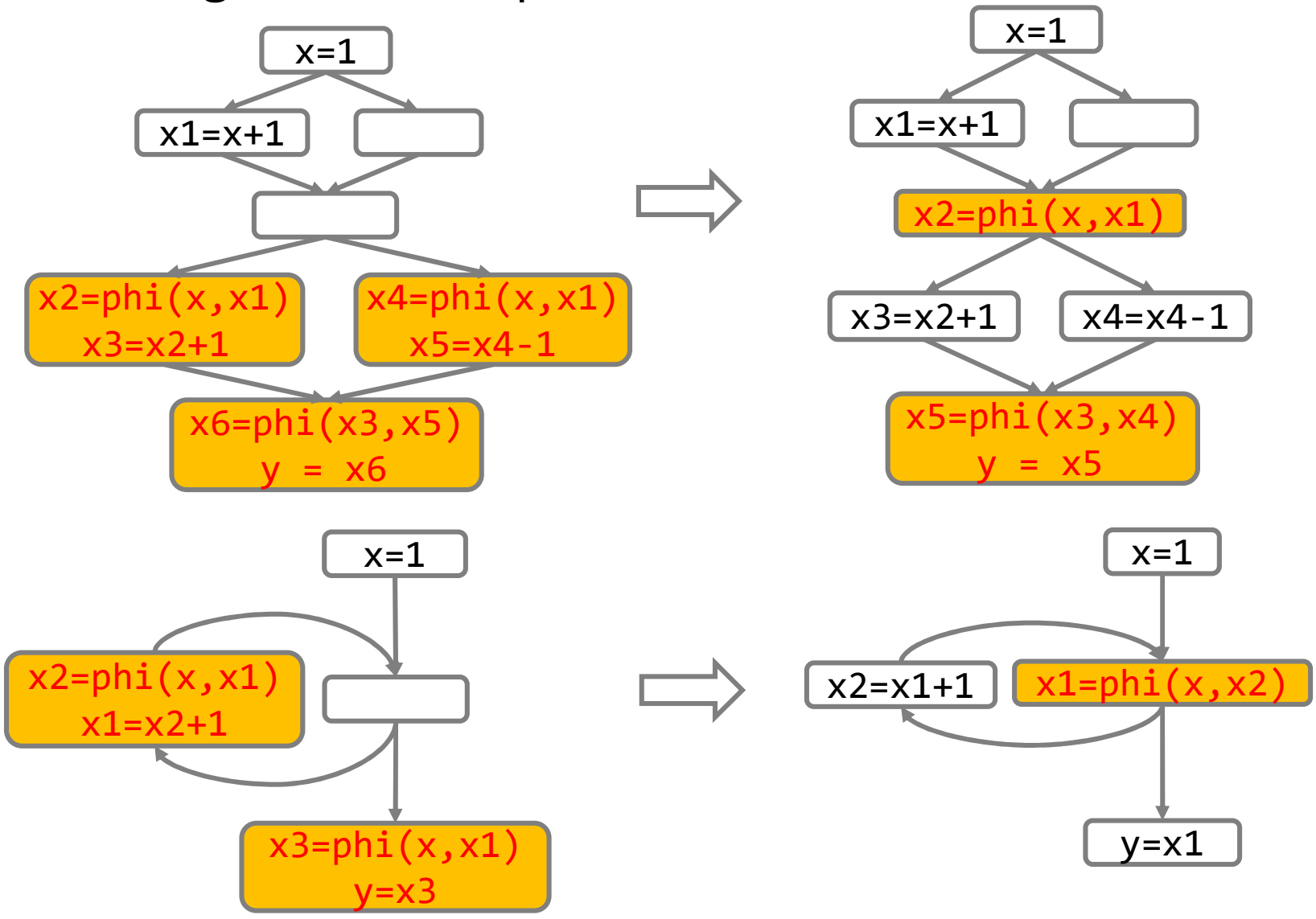
需要放置phi函数的条件

- 必要条件：
  - 该代码块use(x)
  - 且有多个def(x)可到达该代码块。
    - 中间未经过其它def(x)
- 问题：并未简化def-use关系

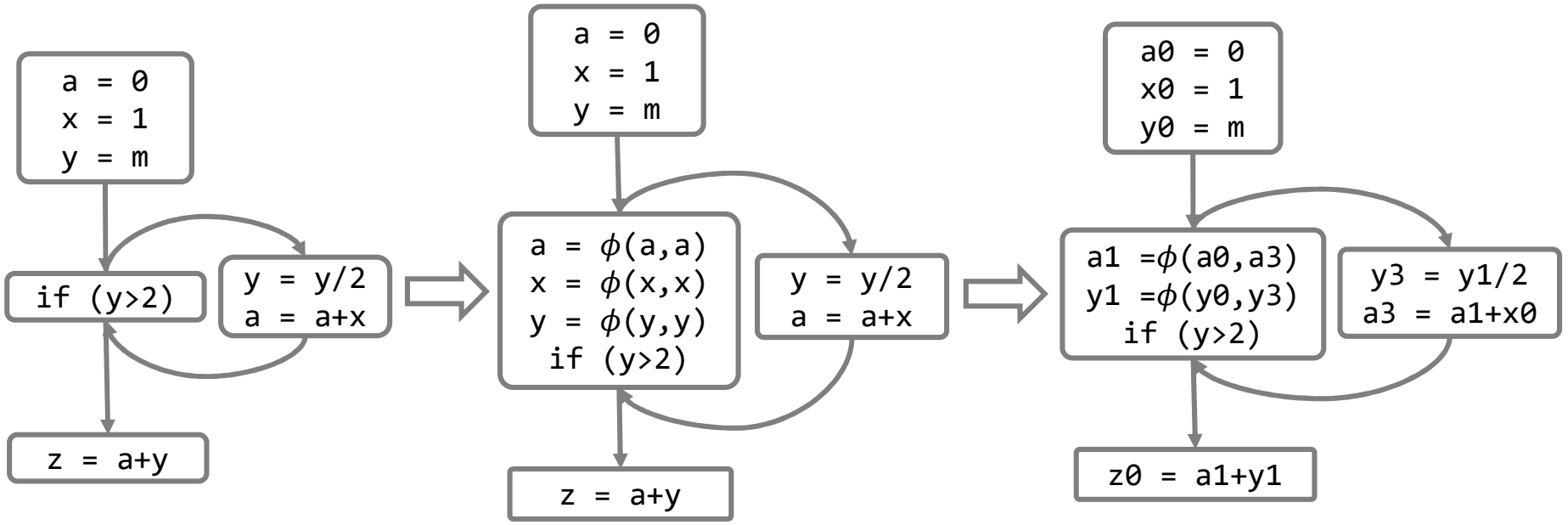


优化def-use关系

- 在merge节点上放置phi函数



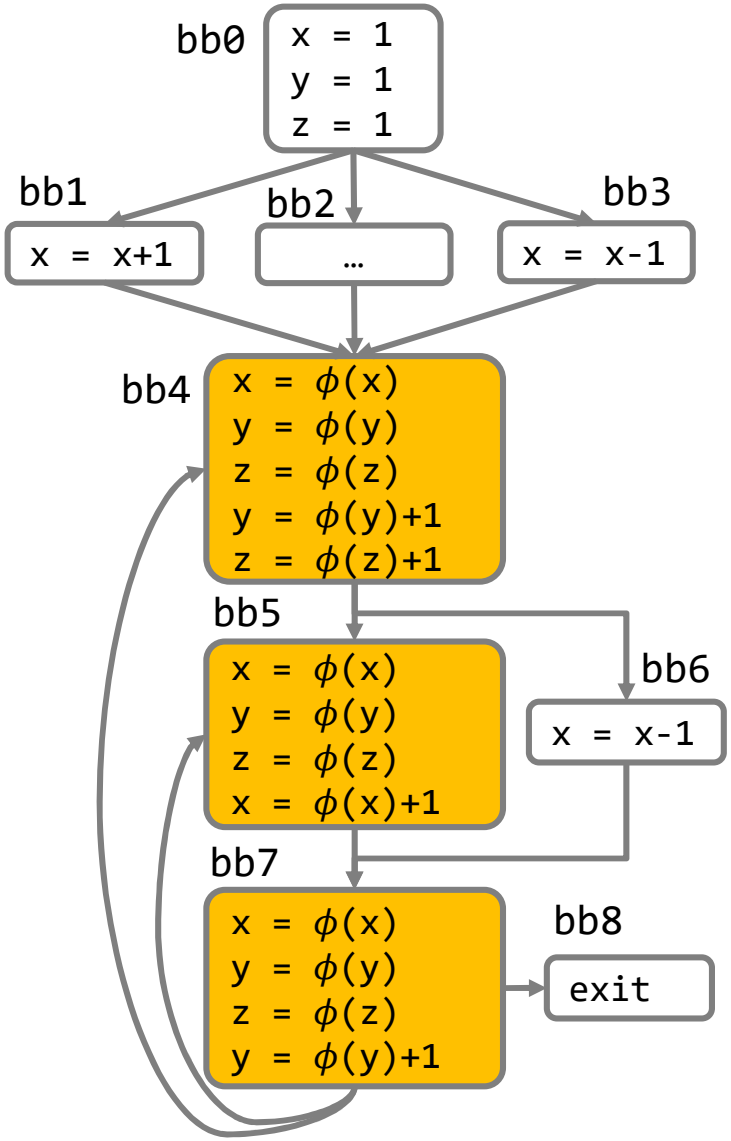
多个变量的情况


a = 0
x = 1
y = m
if (y>2)
y = y/2
a = a+x
z = a+y
a = 0
x = 1
y = m
a = ϕ(a,a)
x = ϕ(x,x)
y = ϕ(y,y)
if (y>2)
y = y/2
a = a+x
z = a+y
a0 = 0
x0 = 1
y0 = m
a1 =ϕ(a0,a3)
y1 =ϕ(y0,y3)
if (y>2)
y3 = y1/2
a3 = a1+x0
z0 = a1+y1

初始思路

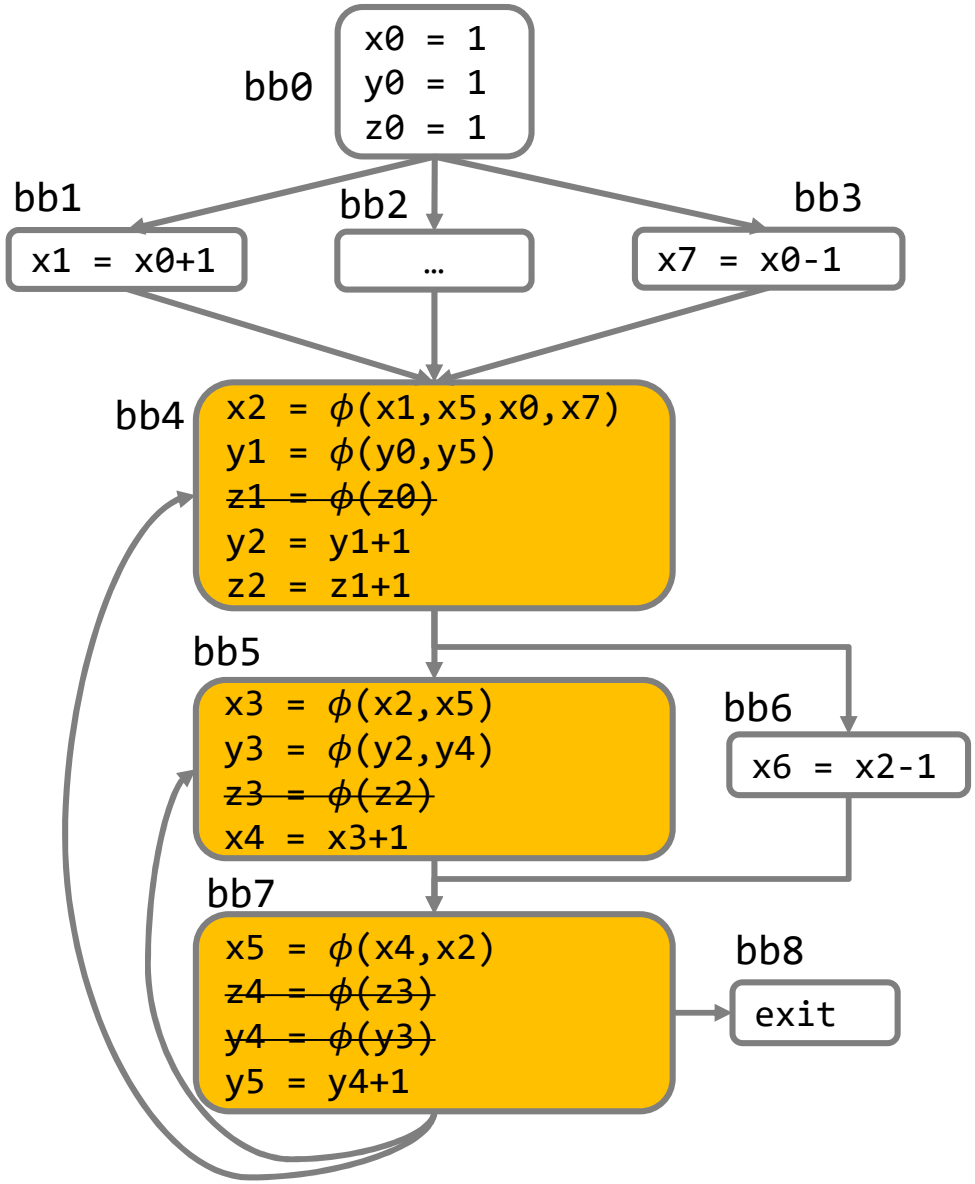
- 在入度≥2的块上放置phi节点
- 后续问题：
  - 对哪些变量使用phi函数？
  - 每个变量的ssa标识符是什么？
- 如何遍历控制流图分析标识符索引
  - 深度优先
  - 广度优先

遍历控制流图构建SSA



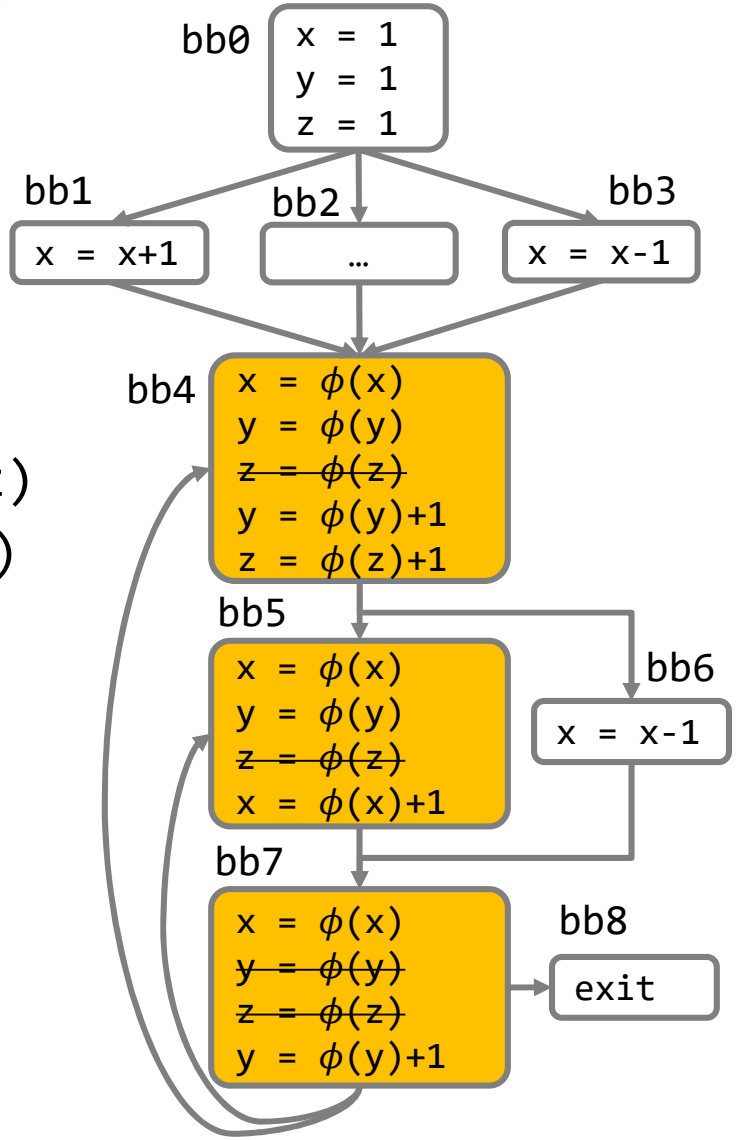
- DFS顺序：bb0->bb1->bb4->bb5->bb7->bb6->bb2->bb3
  - bb0：x0=1，y0=1，z0=1
  - bb1：x1=x0+1
  - bb4：x2=ϕ(x1)，y1=ϕ(y0)，z1=ϕ(z0)，y2=y1+1，z2=z1+1
  - bb5：x3=ϕ(x2)，y3=ϕ(y2)，z3=ϕ(z2)，x4=x3+1
  - bb7：x5=ϕ(x4)，y4=ϕ(y3)，z4=ϕ(z3)，y5=y4+1
  - 不能简单回退到bb4
    - 进一步更新bb4、bb5
- 复杂度$O(V+E)$
- 每个节点需要更新次数为入度

结果


bb0
x0 = 1
y0 = 1
z0 = 1
bb1
x1 = x0+1
bb2
…
bb3
x7 = x0-1
bb4
x2 = ϕ(x1,x5,x0,x7)
y1 = ϕ(y0,y5)
z1 = ϕ(z0)
y2 = y1+1
z2 = z1+1
bb5
x3 = ϕ(x2,x5)
y3 = ϕ(y2,y4)
z3 = ϕ(z2)
x4 = x3+1
bb6
x6 = x2-1
bb7
x5 = ϕ(x4,x2)
z4 = ϕ(z3)
y4 = ϕ(y3)
y5 = y4+1
bb8
exit

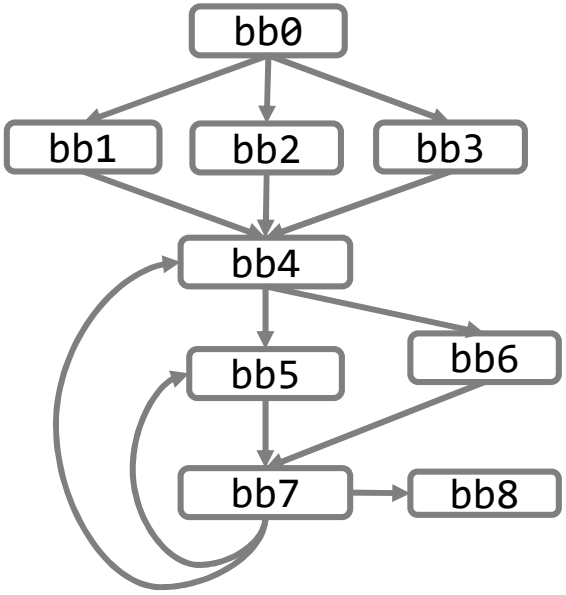
如何优化phi函数的设置？

- 初始思路主要存在的问题
  - 引入了非必要的phi函数
    - bb4、bb5、bb7中的def(z)
    - bb7中的def(y)
- 应如何优化？
  - bb0中的def(x)、def(y)、def(z)不会影响bb5中的use(x)、use(y)
  - 所有的bb0-bb5路径都会经过bb4
- 基于支配边界的方法
  - 仅放置必要的phi节点
  - 获取所有SSA中的变量定义



支配的基本概念

- 给定一个有向图$G(V,E)$与一个起点v_0，如果从v_0到某个点v_j均需要经过点v_i，则称v_i支配v_j或v_i是v_j 的一个支配点。
  - $v_i \in Dom(v_j)$
- 如果$v_i \neq v_j$，则称v_i严格支配v_j。

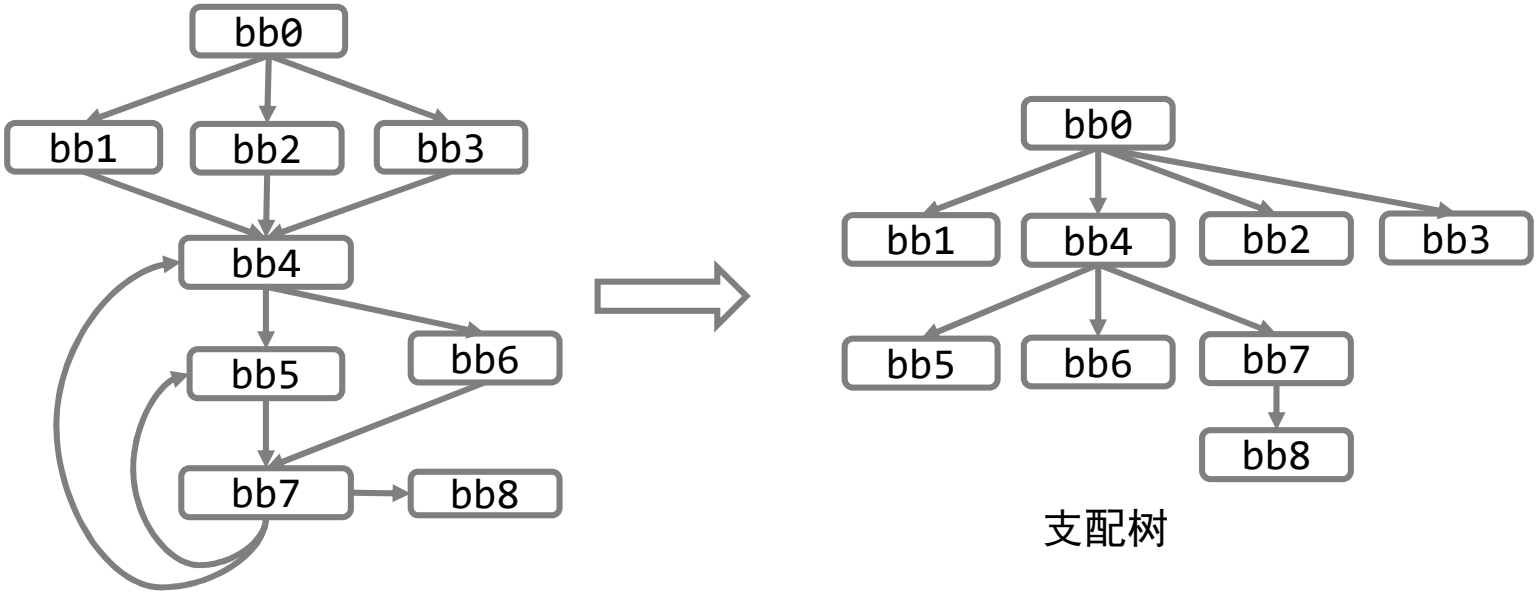


控制流图

$$Dom(bb_0) = \{bb_0\}$$
$$Dom(bb_1) = \{bb_0, bb_1\}$$
$$Dom(bb_2) = \{bb_1, bb_2\}$$
$$Dom(bb_3) = \{bb_0, bb_3\}$$
$$Dom(bb_4) = \{bb_0, bb_4\}$$
$$Dom(bb_5) = \{bb_0, bb_4, bb_5\}$$
$$Dom(bb_6) = \{bb_0, bb_4, bb_6\}$$
$$Dom(bb_7) = \{bb_0, bb_4\}$$
$$Dom(bb_8) = \{bb_0, bb_7\}$$

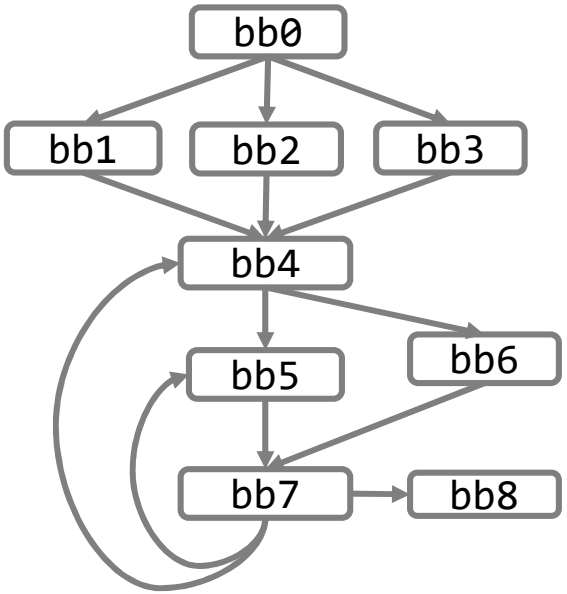
支配树的基本概念

- 所有v_j的严格支配点中与v_j最接近的点成为v_j的最近支配点。
  - $Idom(v_j) = v_i$，v_j的其它严格支配点均严格支配v_i。
- 连接接所有的最近支配关系，形成一棵支配树。
  - 根节点外的每一点均存在唯一的最近支配点。



支配边界Dominance Frontier

- v_i的支配边界是所有满足条件的v_j的集合
  - v_i支配v_j的一个前序节点
  - v_i并不严格支配v_j

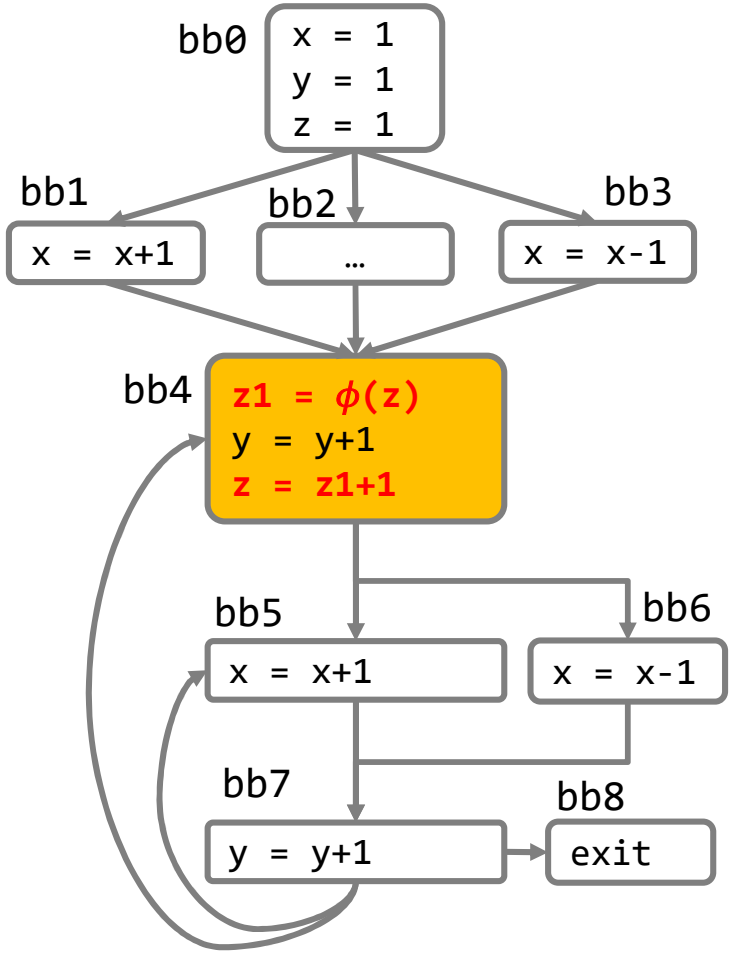


控制流图

$$DF(bb_0) = \{\}$$
$$DF(bb_1) = \{bb_4\}$$
$$DF(bb_2) = \{bb_4\}$$
$$DF(bb_3) = \{bb_4\}$$
$$DF(bb_4) = \{bb_4\}$$
$$DF(bb_5) = \{bb_7\}$$
$$DF(bb_6) = \{bb_7\}$$
$$DF(bb_7) = \{bb_4, bb_5\}$$
$$DF(bb_8) = \{\}$$

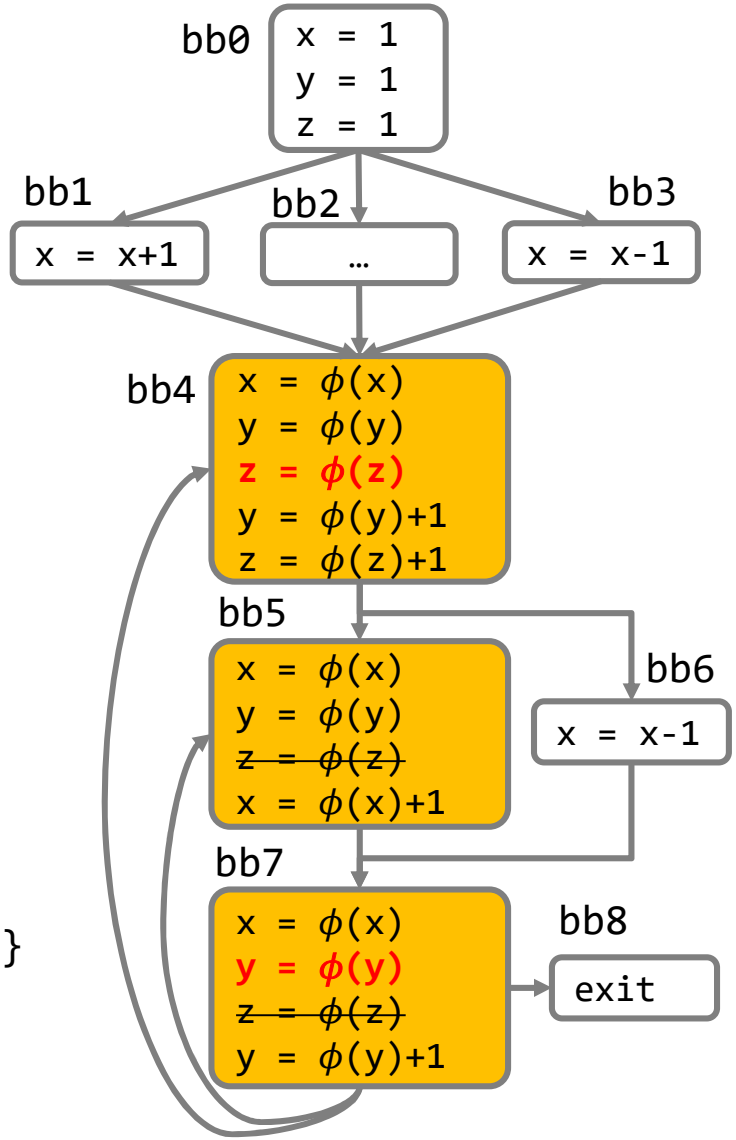
利用支配边界计算def

- 初始化：枚举所有变量的def-sites
  - def-sites(x) = {bb0,bb1,bb3,bb5,bb6}
  - def-sites(y) = {bb0,bb4,bb7}
  - def-sites(z) = {bb0,bb4}
- 为每个变量在bb_j增加phi节点：
  - $bb_i \in$ def-sites(x)
  - $bb_j \in DF(bb_i)$
- 以变量z为例：
  - $bb_0 \in$ def-sites(z)
    - $DF(bb_0) = \{\}$
  - $bb_4 \in$ def-sites(z)
    - $DF(bb_4) = \{bb_4\}$
    - 在bb_4增加phi函数的def(z)



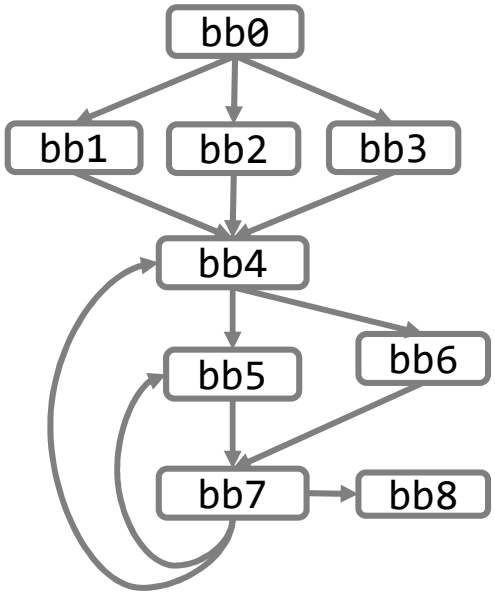
支配边界也不完美

- 依然存在冗余：
  - bb4中的ϕ(z)
  - bb7中的ϕ(y)
- 以变量y为例：
  - $bb_0 \in$ def-sites(y)
    - DF(bb_0) = {}
  - $bb_4 \in$ def-sites(y)
    - DF(bb_4) = {bb_4}
    - 在bb_4增加phi函数的def(y)
  - bb7 $\in$ def-sites(y)
    - DF(bb7) = {bb4,bb5}
    - 在bb5增加phi函数的def(y)
    - def-sites(y) = {bb0,bb4,bb7,bb5}
  - bb5 $\in$ def-sites(y)
    - DF(bb5) = {bb7}



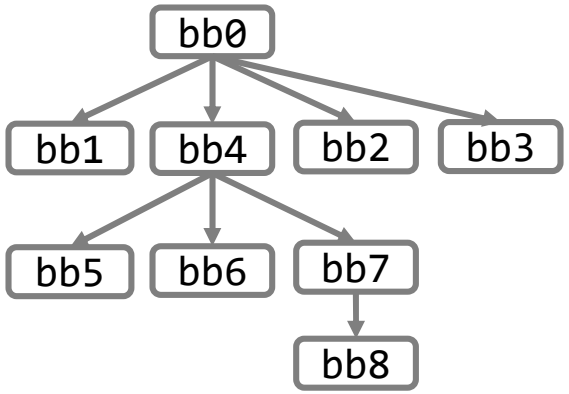
如何构建支配树：主要思路

$$Dom(v) = \begin{cases} \{v\}, & if\ v = v_0 \\ \{v\} \cup \left(\bigcap_{p \in pred(v)} Dom(p) \right), & if\ v \neq v_0 \end{cases}$$



控制流图

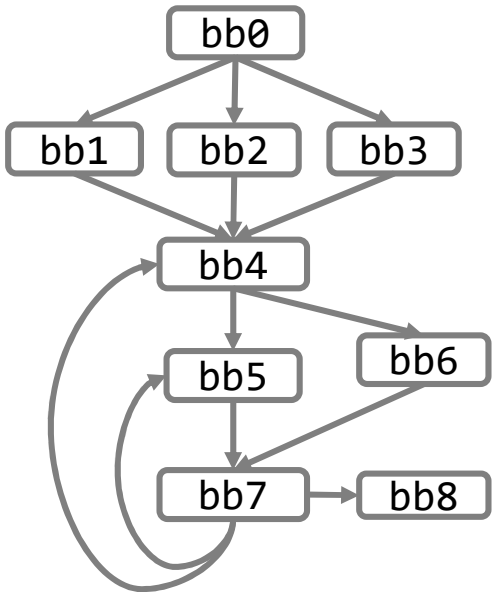
$$Dom(bb_0) = \{bb_0\}$$
$$Dom(bb_1) = \{bb_0, bb_1\}$$
$$Dom(bb_2) = \{bb_1, bb_2\}$$
$$Dom(bb_3) = \{bb_0, bb_3\}$$
$$Dom(bb_4) = \{bb_0, bb_4\}$$
$$Dom(bb_5) = \{bb_0, bb_4, bb_5\}$$
$$Dom(bb_6) = \{bb_0, bb_4, bb_6\}$$
$$Dom(bb_7) = \{bb_0, bb_4\}$$
$$Dom(bb_8) = \{bb_0, bb_7\}$$



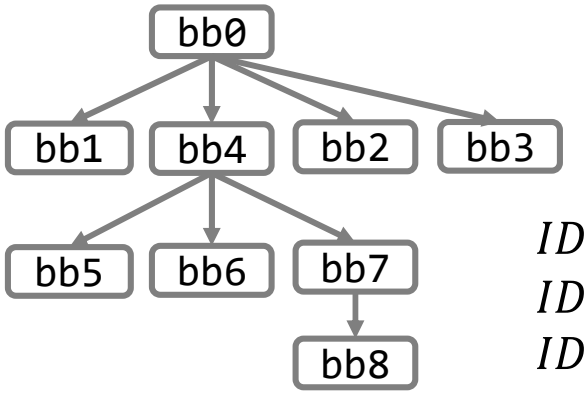
支配树

如何求支配边界：主要思路

- 什么节点会成为支配边界？
  - 入度>1
- 节点v是谁的支配边界？
  - v的所有前序节点，非支配节点：$Pred(v) - IDom(v)$
  - 所有前序节点的直接支配节点$\cup_{v_p \in Pred(v)-IDom(v)} IDom(v_p)$
  - 迭代下去直到遇到v的直接支配节点$IDom(v)$



控制流图



支配树

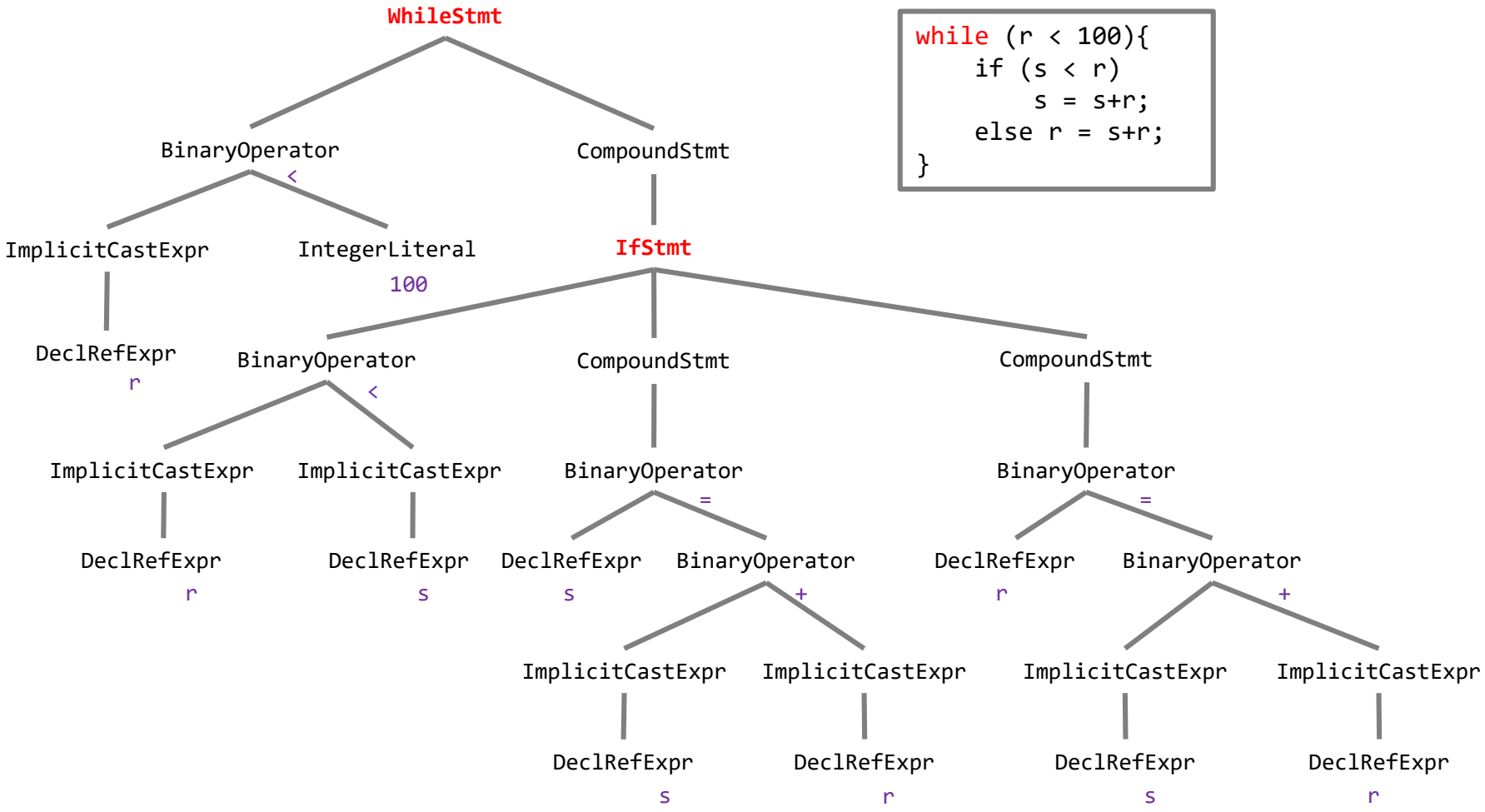
$$IDF(bb_4) = \{bb_1, bb_2, bb_3, bb_7, bb_4\}$$
$$IDF(bb_5) = \{bb_7\}$$
$$IDF(bb_7) = \{bb_5, bb_6\}$$

四、LLVM IR案例分析

AST

IR

While语句




```
while (r < 100){
    if (s < r)
        s = s+r;
    else r = s+r;
}
```

LLVM IR代码示例

```
int fibonacci(int n){
  int a = 0, b = 1;
  int t, r;
  for(int i = 0; i<n; i++){
    t = a + b;
    r = r + t;
    a = b;
    b = t;
  }
  return r;
}
```

```
define dso_local i32 @fibonacci(i32 %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  store i32 0, i32* %3, align 4
  store i32 1, i32* %4, align 4
  store i32 0, i32* %7, align 4
  br label %8
8: ; preds = %21, %1
  %9 = load i32, i32* %7, align 4
  %10 = load i32, i32* %2, align 4
  %11 = icmp slt i32 %9, %10
  br i1 %11, label %12, label %24
12:     ; preds = %8
  %13 = load i32, i32* %3, align 4
  %14 = load i32, i32* %4, align 4
  %15 = add nsw i32 %13, %14
  store i32 %15, i32* %5, align 4
  %16 = load i32, i32* %6, align 4
  %17 = load i32, i32* %5, align 4
  %18 = add nsw i32 %16, %17
  store i32 %18, i32* %6, align 4
  %19 = load i32, i32* %4, align 4
  store i32 %19, i32* %3, align 4
  %20 = load i32, i32* %5, align 4
  store i32 %20, i32* %4, align 4
  br label %21
21:     ; preds = %12
  %22 = load i32, i32* %7, align 4
  %23 = add nsw i32 %22, 1
  store i32 %23, i32* %7, align 4
  br label %8
24:     ; preds = %8
  %25 = load i32, i32* %6, align 4
  ret i32 %25
}
```

标识符：Identifiers

- 三类标识符
  - Named values：如%foo, @bar,
    - 代码中的标识符
    - 遵循正则表达式[%@][-a-zA-Z$._][-a-zA-Z$._0-9]*
    - 为什么必须要有前缀？避免和保留字冲突
  - Unamed values：如%1,
    - 临时创建的变量，只可一次赋值
    - 标号从%0开始递增
  - 常量

示例1

```
%0 = add i32 %X, %X
%1 = add i32 %0, %0
%foo = add i32 %1, %1
```

示例2

```
%foo = call i32 @someFunction ()
%foo = add i32 %foo, 1
```

```
%foo = call i32 @someFunction ()
%foo2 = add i32 %foo, 1
```

数据存取

- 主要操作：
  - 内存分配：alloc
  - 数据读取：load
  - 数据存入：store
  - 并发访问
    - fense
    - cmpxchg
    - atomicrmw

```
%ptr = alloca i32
store i32 3, i32* %ptr
%val = load i32, i32* %ptr
```

运算指令

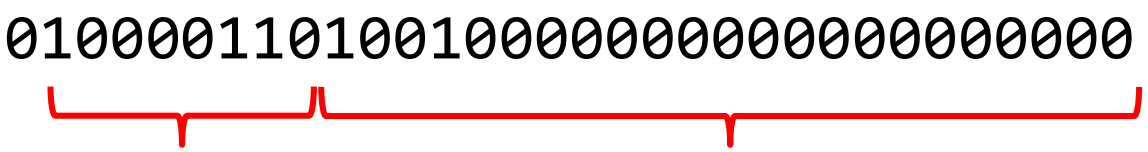
- 二元整数运算：
  - add/sub/mul/sdiv/udiv/urem/srem
  - sdiv/udiv分别对应操作数是否有符号
- 二元位运算
  - and/or/xor
  - shl/lshr/ashr
- 二元浮点数运算：
  - fadd/fsub/fmul/fdiv/frem
- 一元运算
  - fneg

```
1) <result> = add i32 4, %var
2) <result> = fadd float 4.0, %var
3) <result> = and i32 4, 8
4) <result> = or i32 4, 8
5) <result> = xor i32 4, 8
6) <result> = shl i32 1, 10
7) <result> = lshr i8 -2, 1
8) <result> = ashr i32 -2, 1
9) <result> = fneg float %val
```

```
int: %var + 4
float: %var + 4
0
12
12
1024
0x7F
-1
-%var
```

浮点数运算需要单独的指令

- 浮点数表示比较独特：IEEE-754标准
- 计算方式：$mantissa \times (2^{exp} - 127)$
  - 如200克表示成01000011010010000000000000000000
  - $2^7 \times 1.5625 = 200$



exponent (8 bits)

$2^7 + 2^2 + 2^1 - 127$
$= 7$

mantissa (23 bits)

$1 + 2^{-1} + 2^{-4}$
$= 1.5625$

类型转换

- trunc..to
- zext..to
- setx..to
- fptrunc..to
- fpext..to
- fptoui..to
- fptosi..to
- uitofp..to
- sitofp..to
- ptrtoint..to
- inttoptr..to
- bitcast..to
- addrspacecast..to

```
1) %X = trunc i32 257 to i8
2) %X = zext i32 257 to i64
3) %X = sext i8  -1 to i16
4) %X = fptrunc double 2.0 to float
5) %X = fpext float 3.125 to double
6) %X = fptoui double 123.0 to i32
7) fptosi double -123.0 to i32
8) %X = uitofp i32 257 to float
9) %X = sitofp i32 257 to float

10)%X = inttoptr i32 255 to i32*
11)%X = bitcast i8 255 to i8
12)%X = addrspacecast i32* %x to i32
   addrspace(1)*
```

```
i8:1
i64:257
i16:65535
float:2.0
double:3.125000e+00
i32:123
i32:-123
float:257.0
float:257.0
```

数组和结构体

- 数组操作：
  - extractelement/insertelement/shufflevector
- 结构体操作：
  - extractvalue/insertvalue
- 获取指针：getelementptr

```
1) <result> = extractelement <4 x i32> %vec, i32 0
2) <result> = insertelement <4 x i32> %vec, i32 1, i32 0
3) <result> = extractvalue {i32, float} %agg, 0
4) %agg2 = insertvalue {i32, float} %agg1, float %val, 1
5) %ptrs = getelementptr double, double* %B, <8 x i32> %C
```

```
i32
<4 x i32>
i32
```

比较运算

- **icmp**
  - eq: equal
  - ne: not equal
  - ugt: unsigned greater than
  - uge: unsigned greater or eq
  - ult: unsigned less than
  - ule: unsigned less or eq
  - sgt: signed greater than
  - sge: signed greater or eq
  - slt: signed less than
  - sle: signed less or eq
- **fcmp**

```
1) <result> = icmp eq i32 4, 5          F
2) <result> = icmp ne float* %X, %X     F
3) <result> = icmp ult i16  4, 5        T
4) <result> = icmp sgt i16  4, 5        F
5) <result> = icmp ule i16 -4, 5        F
6) <result> = icmp sge i16  4, 5        F
```

跳转指令

- 跳转指令：br
- 条件跳转：switch
- 间接跳转：indirectbr
- 其它：
  - resume
  - catchswitch

```
ret i32 5
ret { i32, i8 } { i32 4, i8 2 }
```

```
%cond = icmp eq i32 %a, %b
br i1 %cond, label %BB1, label %BB2
```

```
switch i32 %val, label %otherwise [ i32 0, label %onzero
  i32 1, label %onone
  i32 2, label %ontwo ]
```

```
indirectbr i8* %Addr,
[ label %bb1, label %bb2, label %bb3 ]
```

函数调用和返回

- 函数调用：call
  - 支持cleanup的函数调用：invoke
  - 其它：callbr
- 返回指令：ret
- 异常处理时相关的返回指令
  - catchret
  - cleanupret

```
%retval = call i32 @test(i32 %argc)
%retval = call i32 (i8*, ...)* @printf(i8* %msg, i32 12, i8 42)
```

```
%retval = invoke i32 @Test(i32 15) to label %Continue
          unwind label %TestCleanup
```

其它指令

- phi
- select
- va_arg
- 异常处理相关：
  - landingpad
  - catchpad
  - cleanuppad
- unreachable
- freeze

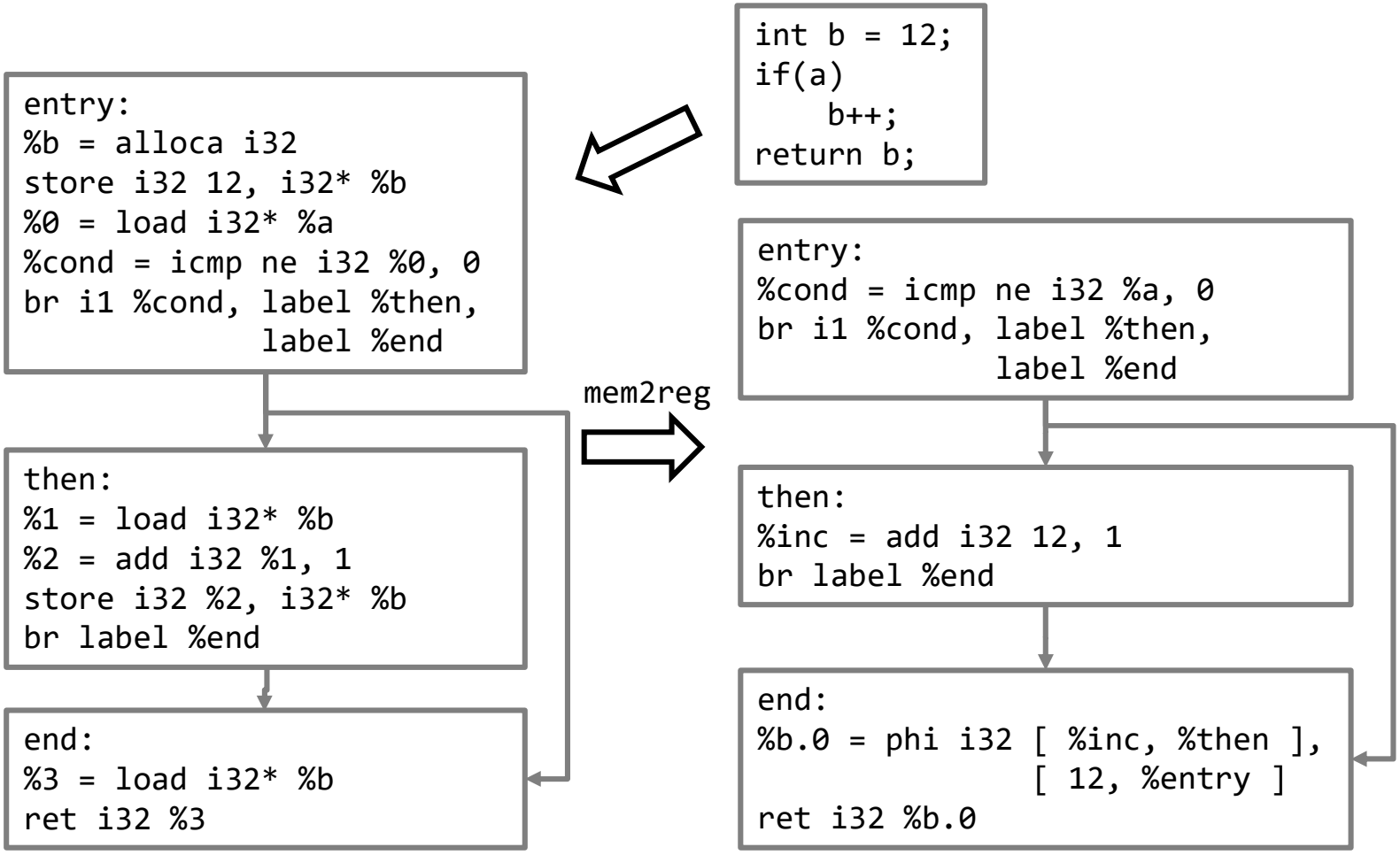
```
Loop:
%indvar = phi i32 [ 0, %LoopHeader ], [ %nextindvar, %Loop ]
%nextindvar = add i32 %indvar, 1
br label %Loop
```

```
%X = select i1 true, i8 17, i8 42
```

```
%struct.va_list = type { i8* }

define i32 @test(i32 %X, ...) {
  ; Initialize variable argument processing
  %ap = alloca %struct.va_list
  %ap2 = bitcast %struct.va_list* %ap to i8*
  call void @llvm.va_start(i8* %ap2)

  ; Read a single integer argument
  %tmp = va_arg i8* %ap2, i32

  ; Demonstrate usage of llvm.va_copy and llvm.va_end
  %aq = alloca i8*
  %aq2 = bitcast i8** %aq to i8*
  call void @llvm.va_copy(i8* %aq2, i8* %ap2)
  call void @llvm.va_end(i8* %aq2)

  ; Stop processing of arguments.
  call void @llvm.va_end(i8* %ap2)
  ret i32 %tmp
}
```

SSA和Phi指令

- 初始IR非严格SSA
  - 大量使用store/load
  - mem2reg pass负责转换



Select

- 一些指令集无需控制流解决

```
entry:
%tobool = icmp ne i32 %a, 0
%0 = select i1 %tobool, i32 13, i32 12
ret i32 %0
```

x86

```
testl %edi, %edi
setne %al
movzbl %al, %eax
orl $12, %eax
ret
```

ARM

```
mov r1, r0
mov r0, #12
cmp r1, #0
movne r0, #13
mov pc, lr
```

x86

```
cmplwi 0, 3, 0
beq 0, .LBB0_2
li 3, 13
blr
.LBB0_2:
li 3, 12
blr
```

参考资料

- 《编译原理（第2版）》
  - 第五章：Syntax-Directed Translation;
  - 第六章：Intermidiate-Code Generation
- 《编译器设计（第2版）》
  - 第四章：上下文相关分析。
  - 第五章：线性IR
- https://llvm.org/docs/LangRef.html