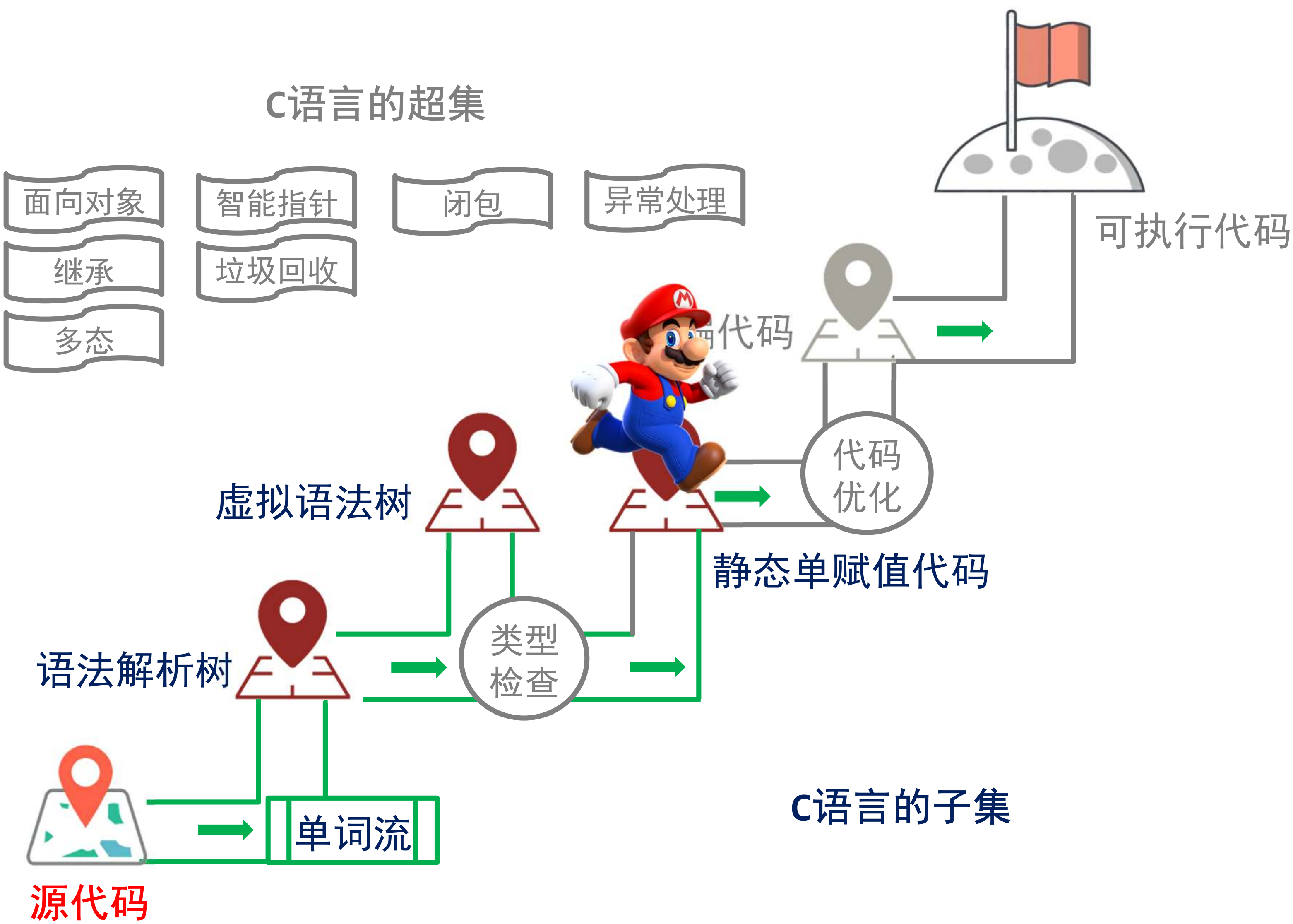
Lecture 4

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
类型
检查
语法解析树
单词流
源代码
C语言的子集

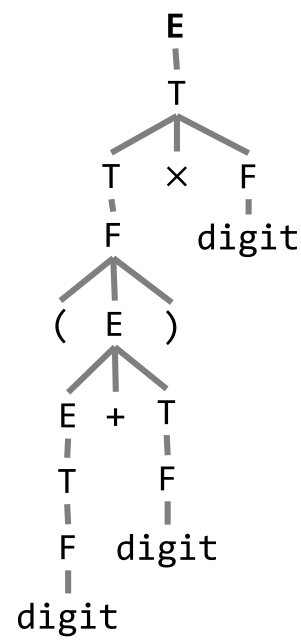
回顾：

- 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

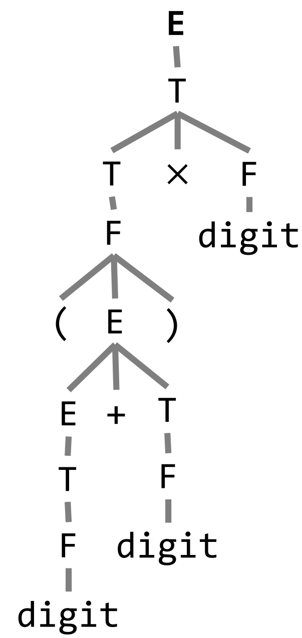
1) $E \rightarrow E + T$
2) $E \rightarrow T$
3) $T \rightarrow T \times F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow digit$



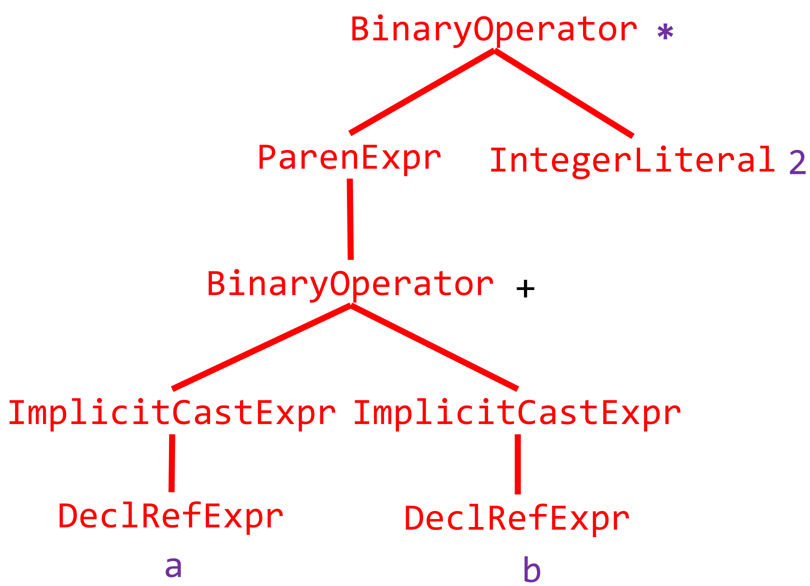
语法解析树

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



语法解析树



LLVM虚拟语法树

```
%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

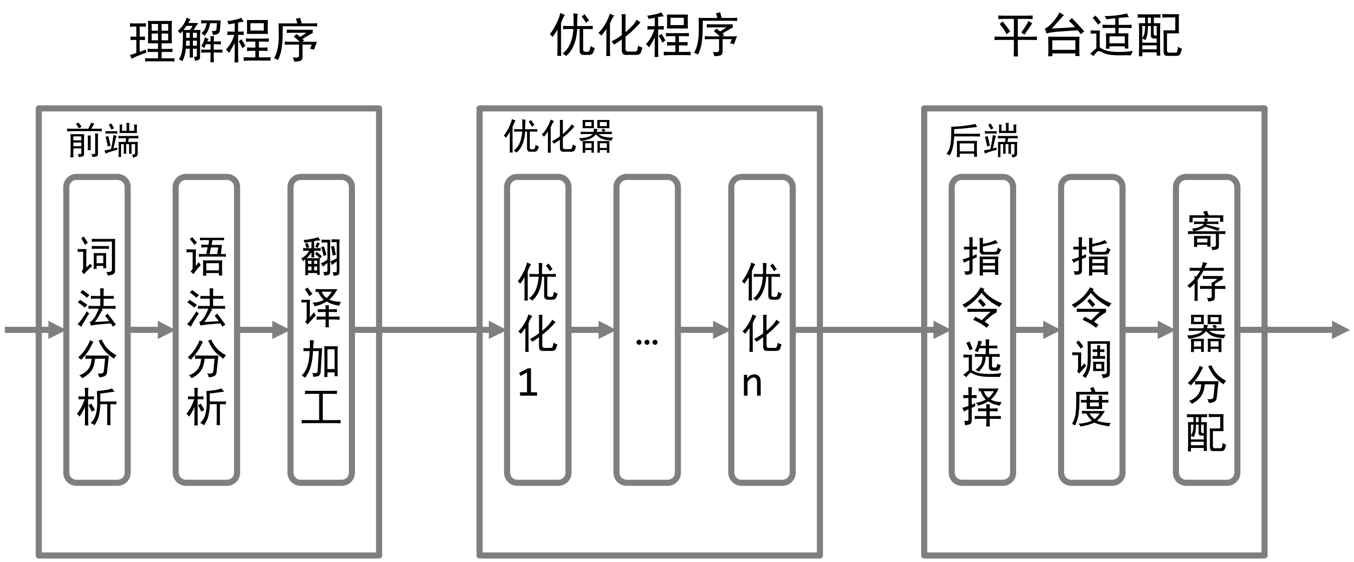
```c
int main(){
  int s = 1, r = 1;
  while (r < 100){
    if (s < r)
      s = s+r;
    else r = s+r;
  }
  return r;
}
```

⇨

```llvm
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



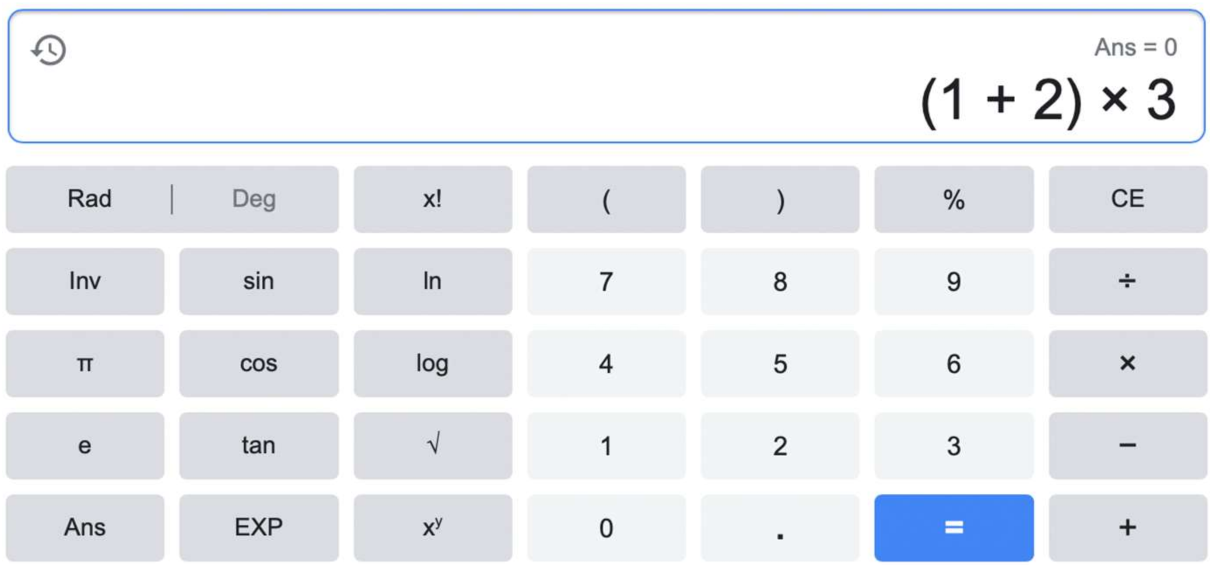
大纲

- 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

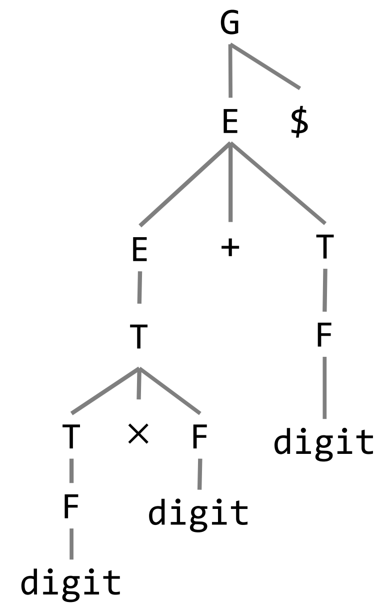
举例：计算器程序

- 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$

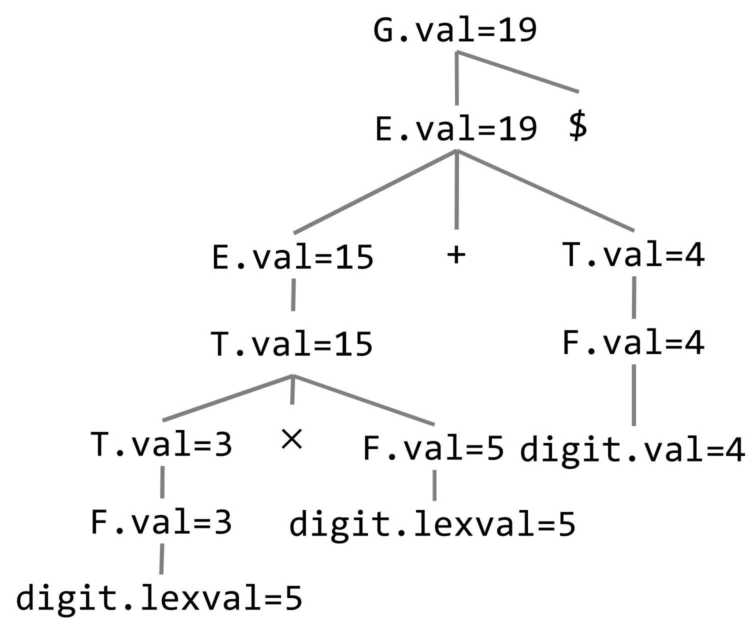
语法制导：Syntax-Directed Translation

- 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。

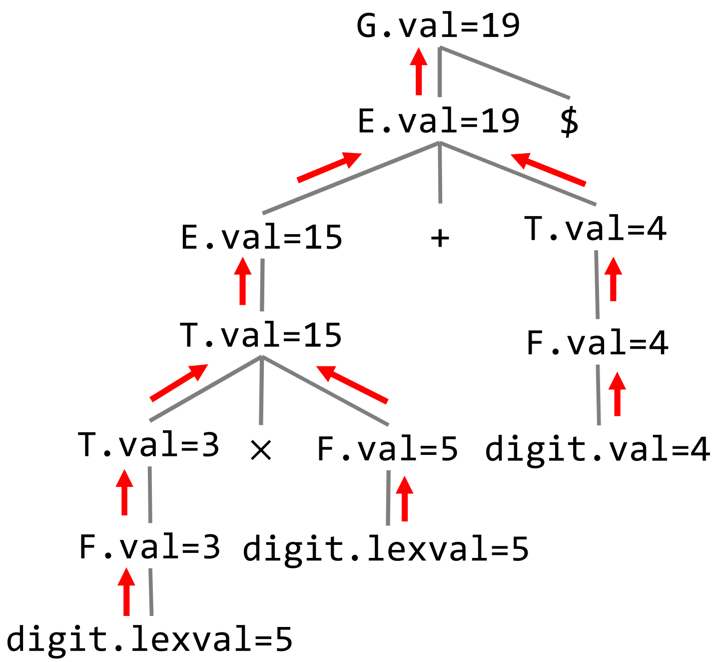
Production	**Semantic Rules**
1) $G \rightarrow E\ \$$	$G.val = E.val$
2) $E \rightarrow E_1 + \mathrm{T}$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 \times F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F = E.val$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

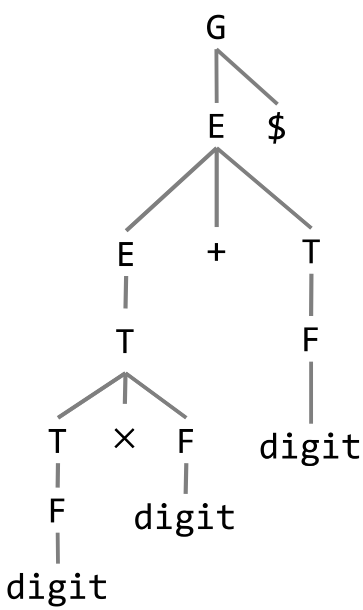
属性依赖关系图：

- 点：语法解析树上符号的属性
- 边：依赖关系

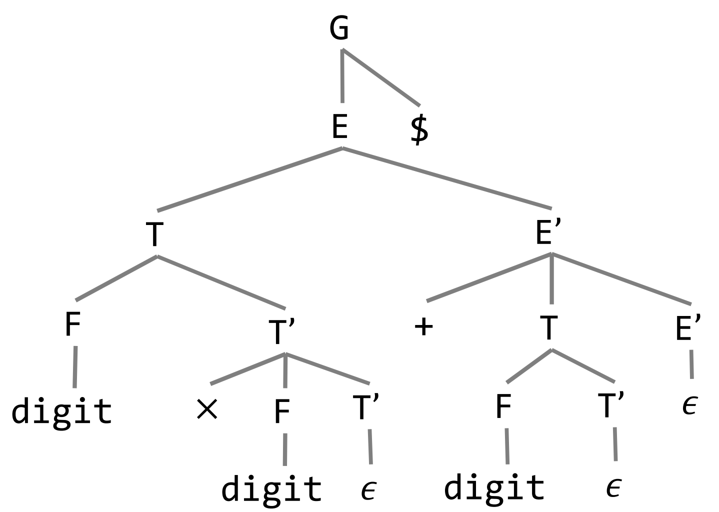
LL(1)语法如何处理

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) $F \rightarrow digit$

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

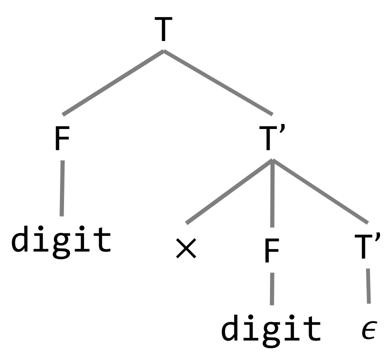


语法解析树

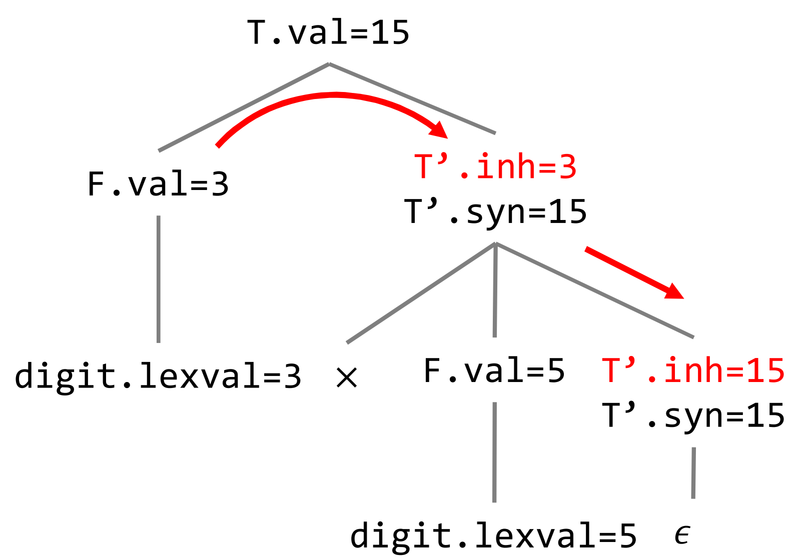
继承属性：Inherited Attaibute

- 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) $F \to digit$	$F.val = digit.lexval$



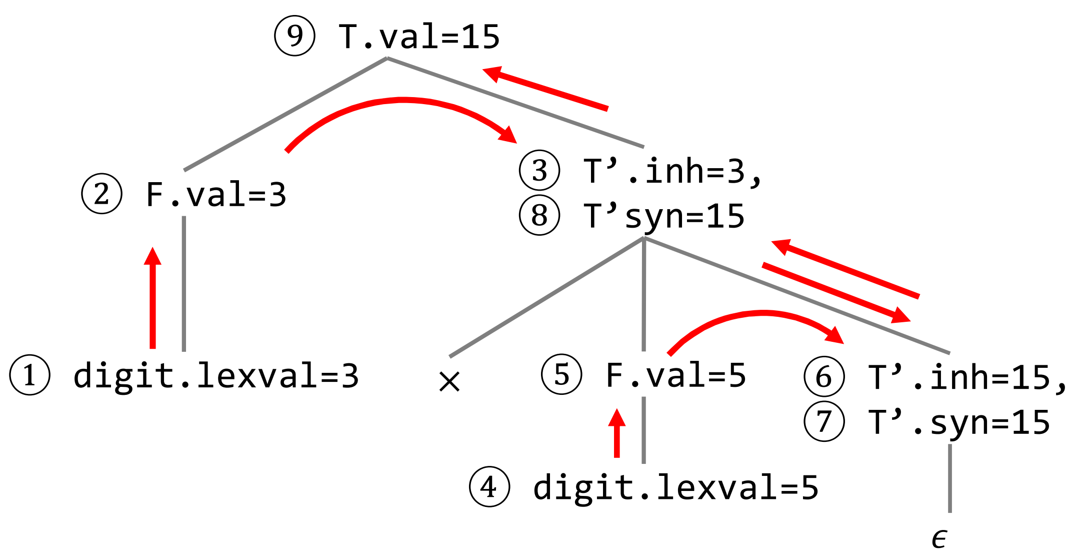
语法解析树：3×5



带注解的语法解析树：3×5

基于L-attributed SDD

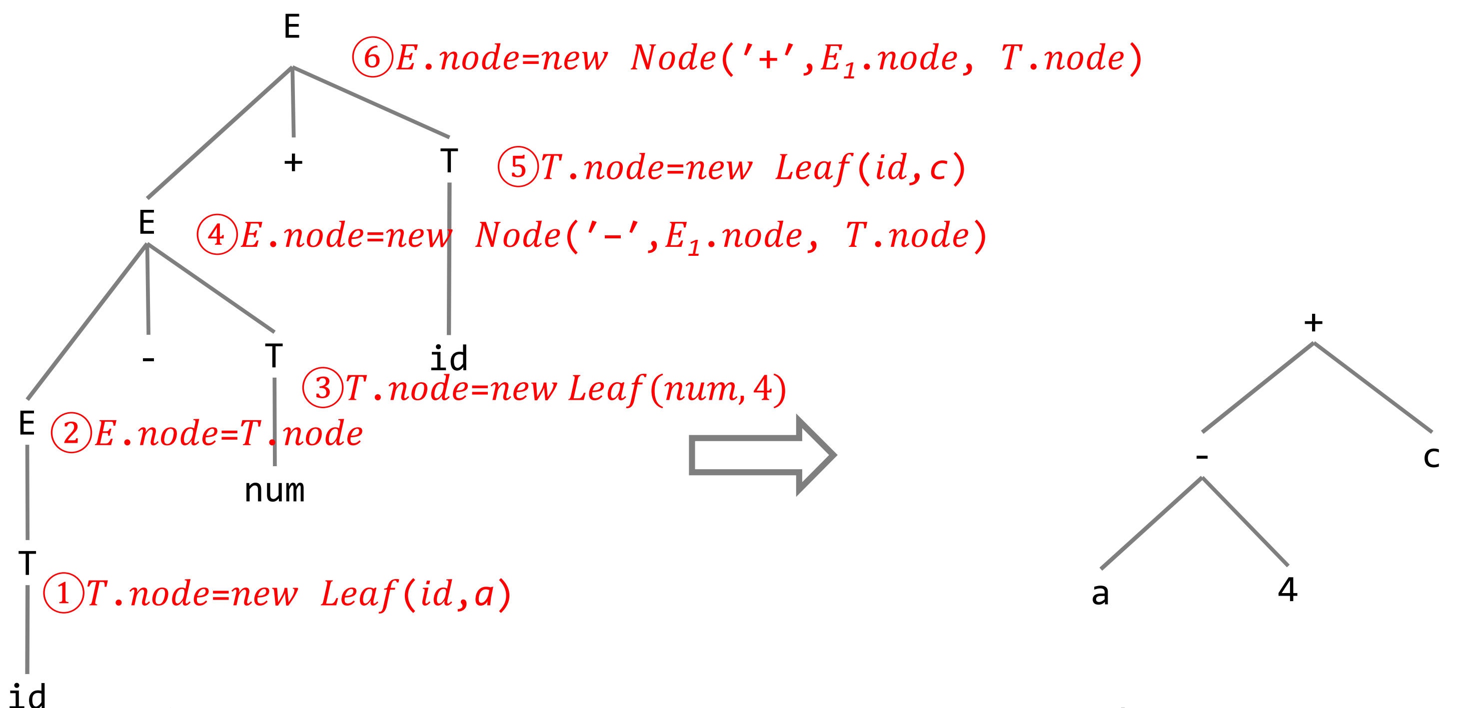
- 所有的属性都是合成属性，或对于$A \rightarrow \beta_1..\beta_i..\beta_n$中的任意$\beta_i$来说，其继承属性只依赖$A$或$\beta_1, \ldots, \beta_{i-1}$
- 适合自顶向下的解析算法，为什么？
  - 解析树构建采用“前序遍历”，最后访问右孩子节点；
  - L-Attributed SDD的继承属性计算依赖最后访问右孩子节点。



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

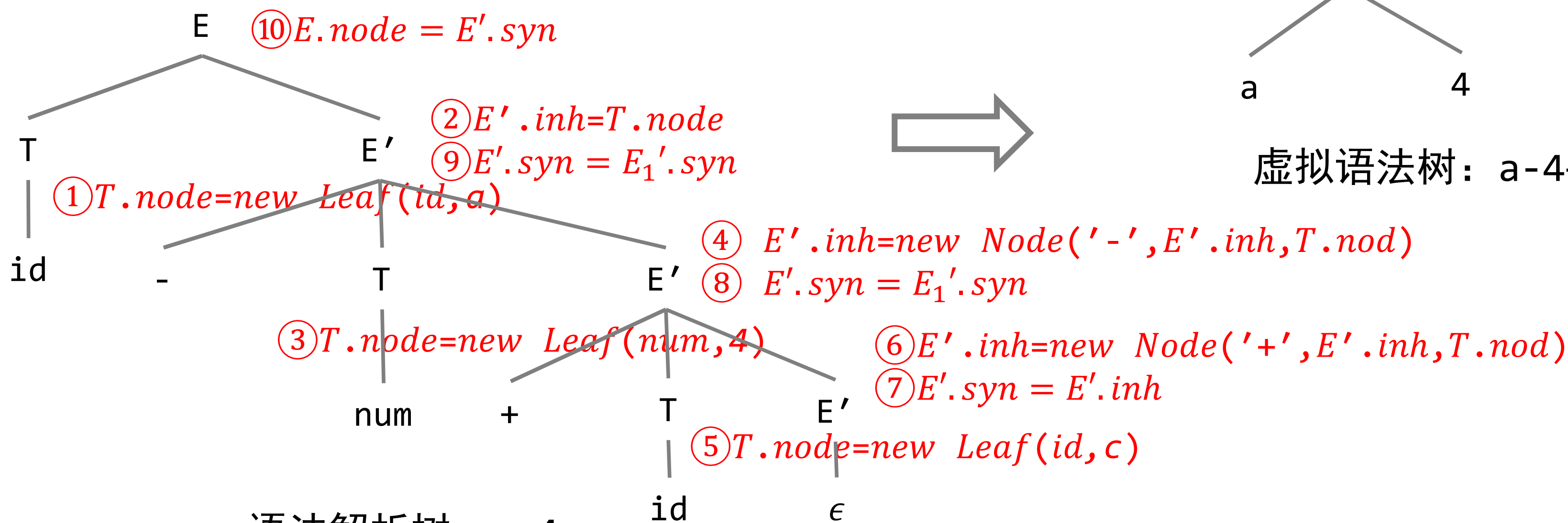
基于S-attaibuted SDD构建AST



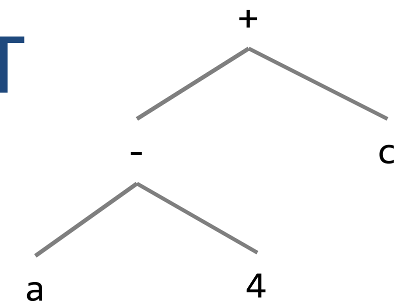
语法解析树：a-4+c　　　　虚拟语法树：a-4+c

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

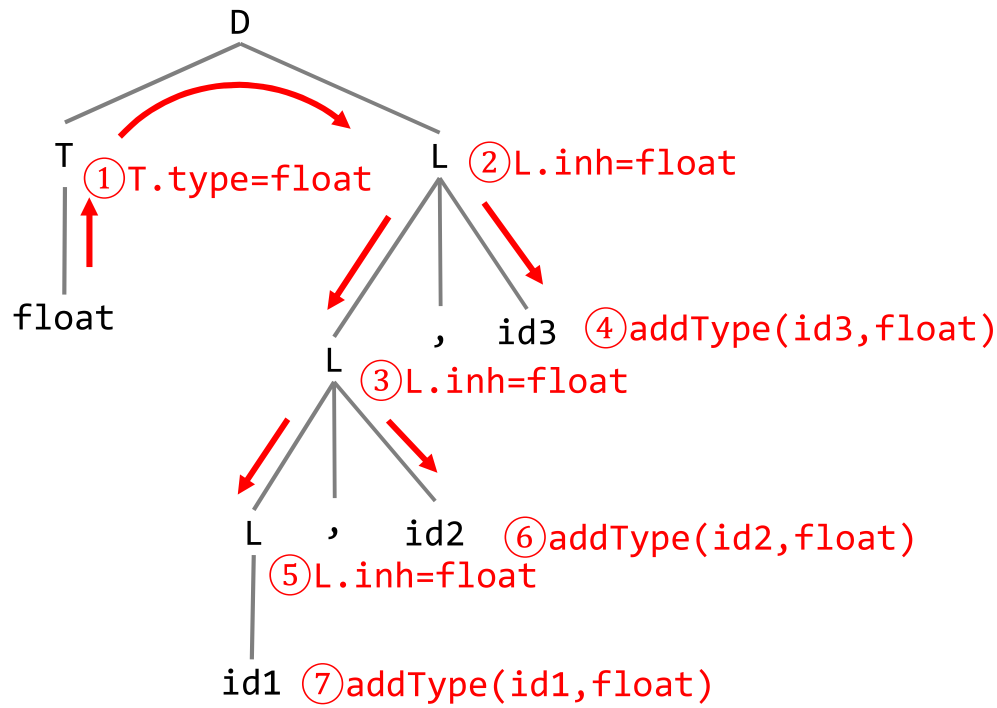


虚拟语法树：a-4+c

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

生成类型约束

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

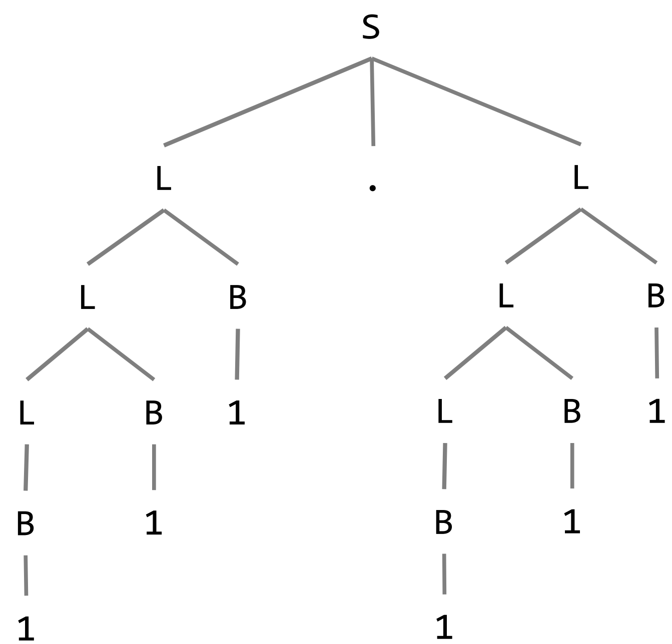


float id1, id2, id3

练习

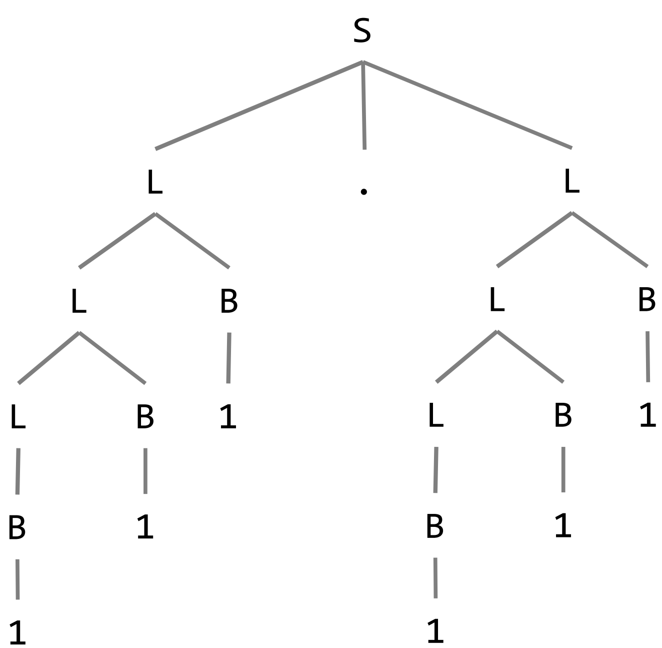
- 下列语法可以解析二进制数。
  1) 设计S-attaibuted SDD将其转化为十进制数；
  2) 设计L-attaibuted SDD将其转化为十进制数。
     1) 将整数/小数部分的信息传递给子树。
  - 例如：如101.101的对应的十进制数是5.625。

$$
\begin{array}{ll}
[1] & S \rightarrow L.L_1 \\
[2] & \quad\quad | \; L \\
[3] & L \rightarrow L_1 B \\
[4] & \quad\quad | \; B \\
[5] & B \rightarrow 0 \\
[6] & \quad\quad | \; 1
\end{array}
$$



参考答案：S-attaibuted SDD

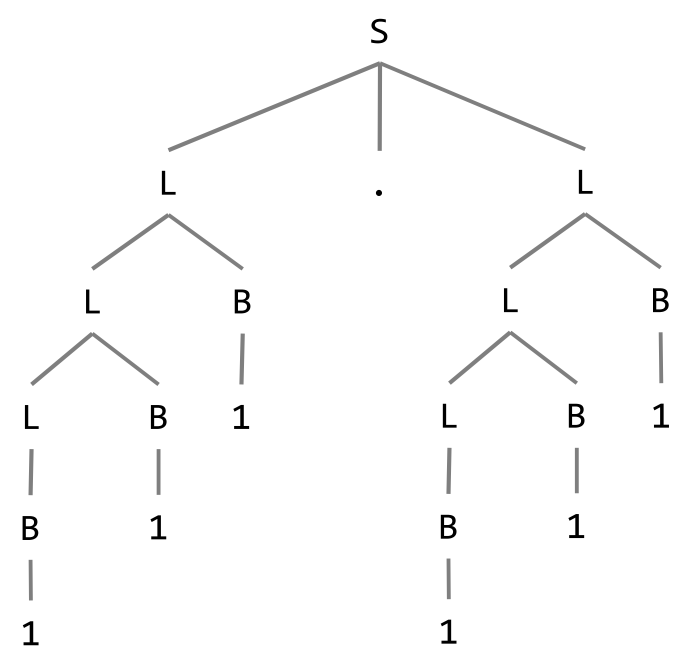
[1]	$S \rightarrow L.L_1$		$\{S.val = L.val + L_1.val/L_1.frac\}$
[2]		$\mid L$	$\{L.val = L.val\}$
[3]	$L \rightarrow L_1 B$		$\{L.val = L_1.val \times 2 + B.val; L.frac = L_1.frac \times 2\}$
[4]		$\mid B$	$\{L.val = B.val; L.frac = 2\}$
[5]	$B \rightarrow 0$		$\{B.val = 0\}$
[6]		$\mid 1$	$\{B.val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



- 主要问题：会冗余计算整数部分的$L.frac$。

参考答案：L-attaibuted SDD

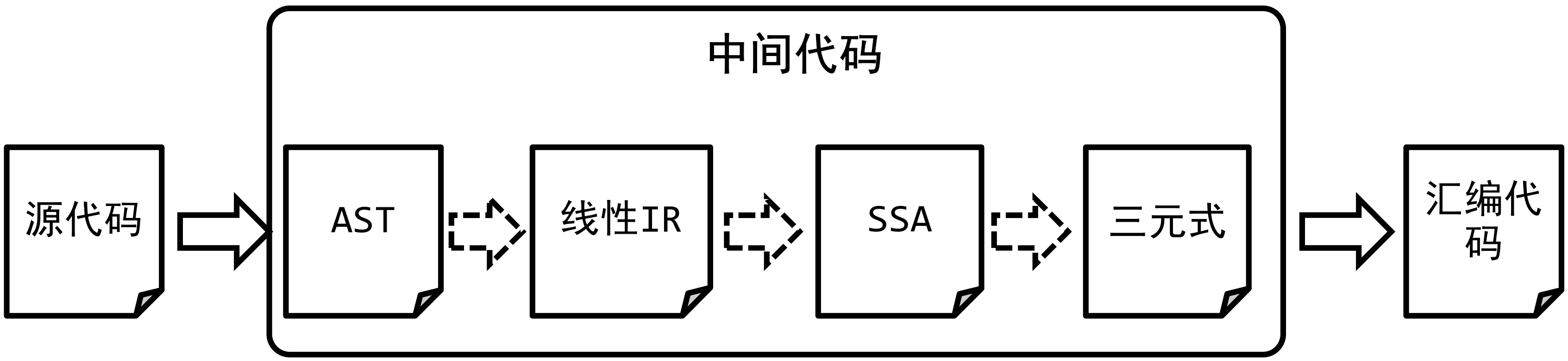
[1]	$S \rightarrow L.L_1$	$\{S.val = L.val + L_1.val; L.isFrac = false; L_1.isFrac = true\}$
[2]	$\mid L$	$\{S.val = L.val; L.isFrac = false\}$
[3]	$L \rightarrow L_1 B$	$\{ L_1.isFrac = L.isFrac; L.pos = L_1.pos + 1;$ $L.val = L_1.isFrac?\ L_1.val + B.val/2^{L_1.pos}: L_1.val * 2 + B.val$
[4]	$\mid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



二、中间代码生成

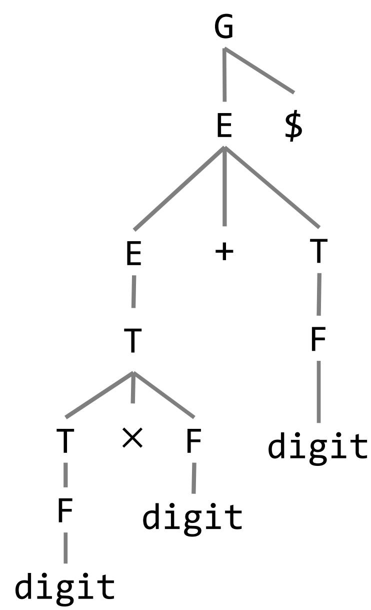
中间代码生成过程

- 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。
  - 简化了变量的def-use关系。

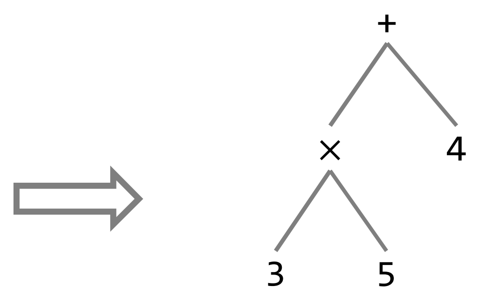


创建虚拟语法树AST

- 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



语法解析树：3×5+4$



虚拟语法树

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。

$$
\begin{array}{ll}
[1] & Expr \rightarrow Expr\ Bop\ Expr \\
[2] & \qquad\quad \mid num \\
[3] & \qquad\quad \mid (Expr) \\
[4] & Bop \rightarrow + \\
[5] & \qquad\quad \mid - \\
[6] & \qquad\quad \mid \times \\
[7] & \qquad\quad \mid \div
\end{array}
$$

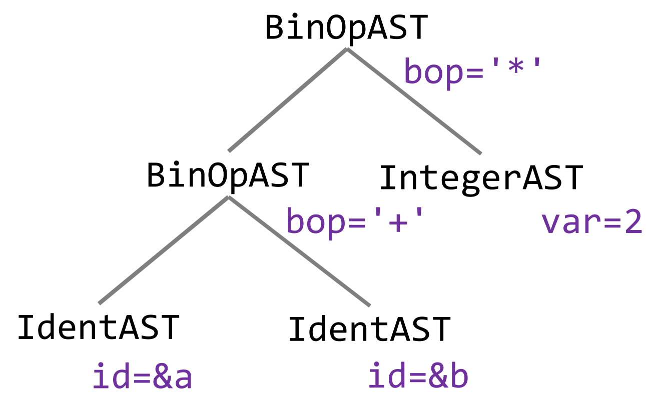
$$
\begin{array}{l}
< regex >::=< union > \mid < concat > \\
\qquad\qquad\quad \mid < closure > \mid < term > \\
< union >::=< regex > \text{"|"} < regex > \\
< concat >::=< regex >< regex > \\
< closure >::=< regex >* \\
< term >::=< group > \mid < alphanum > \\
< group >::= (< regex >)
\end{array}
$$

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

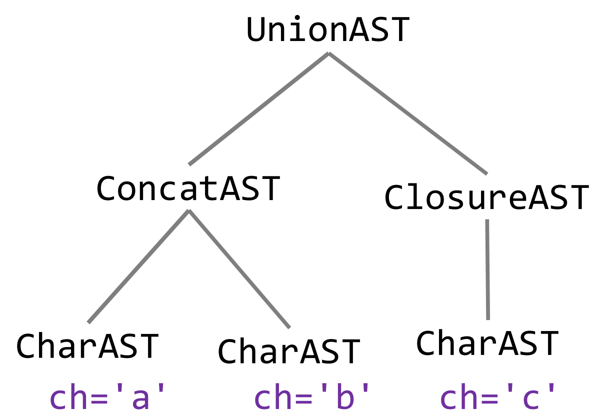


(a+b)*2虚拟语法树

更多AST的例子

```
class RegexAST{}
class ConcatAST : RegexAST{
    RegexAST * lhs, rhs;
}
class UnionAST : RegexAST{
    RegexAST * lhs, rhs;
}
class ClosureAST : RegexAST{
    RegexAST* reg;
}
class CharAST : RegexAST{
    char ch;
}
```

Regex的AST节点



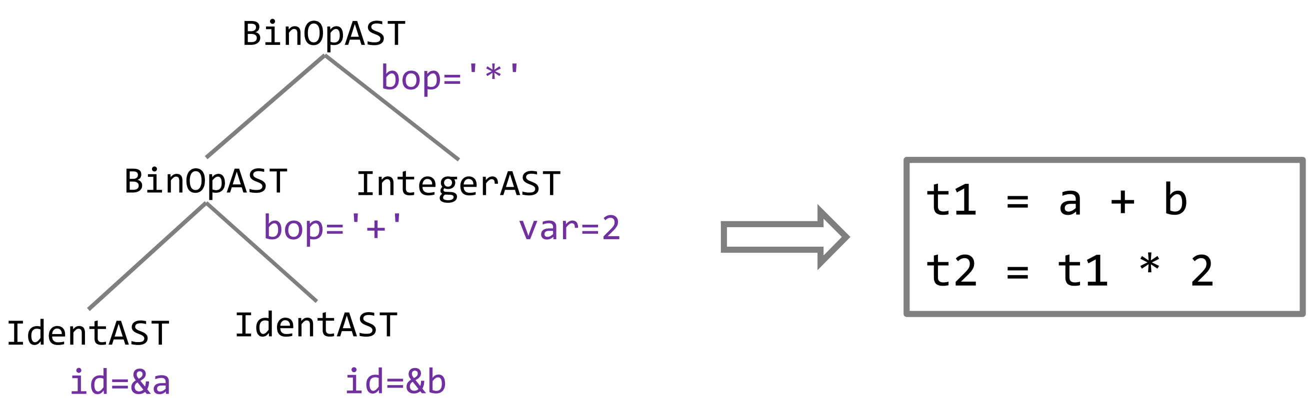
ab|c*虚拟语法树

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（跳过AST树）直接基于属性语法

CFG语法规则

$IfStmt \rightarrow if\ (cond)\ trueBB\ else\ falseBB$

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

属性语法

$trueLabel = newBBLabel()$

$falseLabel = newBBLabel()$

$trueBB.next = falseBB.next = IfStmt.next$

$$
\begin{aligned}
IfStmt.code = cond.code \mid\mid & trueLabel \mid\mid trueBB.code \\
\mid\mid & codegen('goto'\ trueBB.next) \\
\mid\mid & falseLabel \mid\mid falseBody.code \\
\mid\mid & codegen('goto'\ falseBB.next)
\end{aligned}
$$

控制流语句：While/For

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

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

```
condBB:
    b = x < 100;
    i = i + 1;
    ifFalse b goto nextBB
trueBB:
    x = x + 2;
    goto condBB;
nextBB:
    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
- 一般在SSA之后才进行处理。

```
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

```
t1 = a + b;
t2 = t1 * 2;
```

三地址线性IR

op	arg1	arg2	result
+	a	b	t1
*	t1	2	t2

四元式

	op	arg1	arg2
(1)	+	a	b
(2)	*	(1)	2

三元式

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

两地址形式

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

实际汇编代码

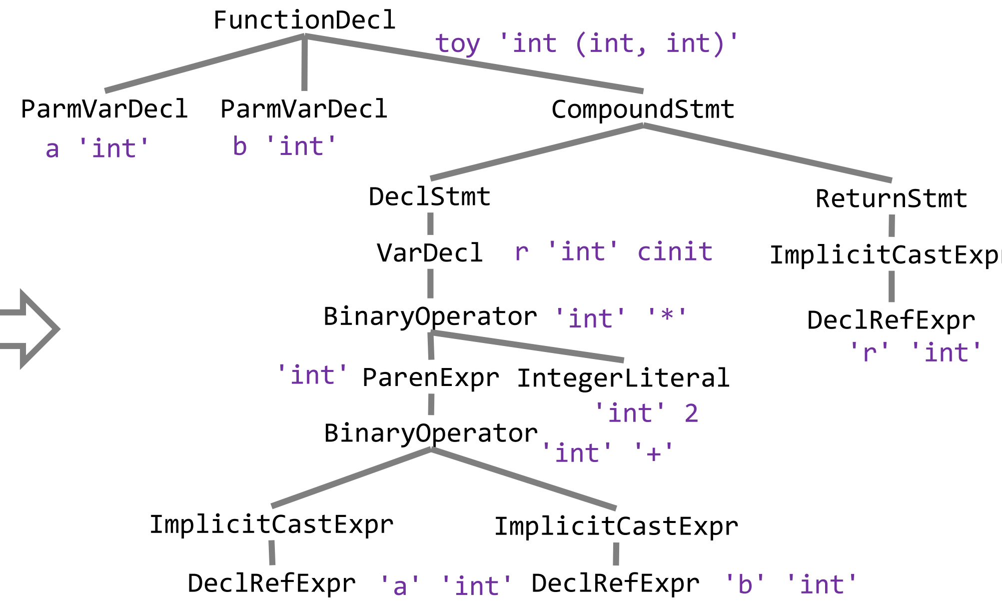
三、LLVM IR案例分析

AST

IR

LLVM输出AST

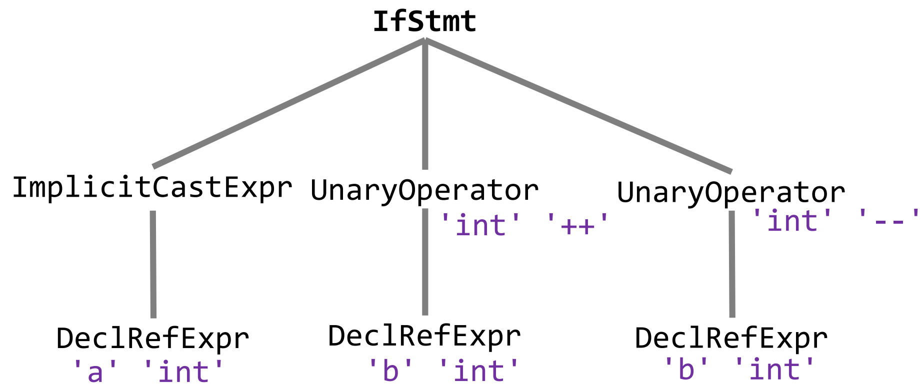
```
int toy(int a, int b){
  int r = (a + b) * 2;
  return r;
}
```




```
#:clang -Xclang -ast-dump -fsyntax-only expr.c
|-FunctionDecl 0x15403d0 <expr.c:3:1, line:6:1> line:3:5 used toy 'int (int, int)'
| |-ParmVarDecl 0x1540278 <col:9, col:13> col:13 used a 'int'
| |-ParmVarDecl 0x15402f8 <col:16, col:20> col:20 used b 'int'
| `-CompoundStmt 0x1540650 <col:22, line:6:1>
|   |-DeclStmt 0x15405f0 <line:4:3, col:22>
|   | `-VarDecl 0x1540498 <col:3, col:21> col:7 used r 'int' cinit
|   |   `-BinaryOperator 0x15405d0 <col:11, col:21> 'int' '*'
|   |     |-ParenExpr 0x1540590 <col:11, col:17> 'int'
|   |     | `-BinaryOperator 0x1540570 <col:12, col:16> 'int' '+'
|   |     |   |-ImplicitCastExpr 0x1540540 <col:12> 'int' <LValueToRValue>
|   |     |   | `-DeclRefExpr 0x1540500 <col:12> 'int' lvalue ParmVar 0x1540278 'a' 'int'
|   |     |   `-ImplicitCastExpr 0x1540558 <col:16> 'int' <LValueToRValue>
|   |     |     `-DeclRefExpr 0x1540520 <col:16> 'int' lvalue ParmVar 0x15402f8 'b' 'int'
|   |     `-IntegerLiteral 0x15405b0 <col:21> 'int' 2
|   `-ReturnStmt 0x1540640 <line:5:3, col:10>
|     `-ImplicitCastExpr 0x1540628 <col:10> 'int' <LValueToRValue>
|       `-DeclRefExpr 0x1540608 <col:10> 'int' lvalue Var 0x1540498 'r' 'int'
```

If-Else语句的AST

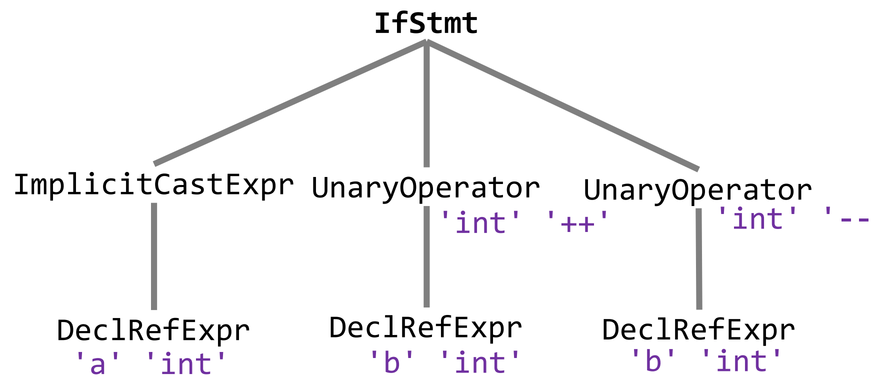
```c
int toy(int a, int b){
  if(a) b++;
  else b--;
  return b;
}
```




```
#:clang -Xclang -ast-dump -fsyntax-only ifelse.c
|-FunctionDecl 0x1fb04a0 <ifelse.c:3:1, line:7:1> line:3:5 used phib 'int (int, int)'
| |-ParmVarDecl 0x1fb0348 <col:10, col:14> col:14 used a 'int'
| |-ParmVarDecl 0x1fb03c8 <col:17, col:21> col:21 used b 'int'
| `-CompoundStmt 0x1fb0668 <col:23, line:7:1>
|   |-IfStmt 0x1fb05f8 <line:4:5, line:5:11> has_else
|   | |-ImplicitCastExpr 0x1fb0570 <line:4:8> 'int' <LValueToRValue>
|   | | `-DeclRefExpr 0x1fb0550 <col:8> 'int' lvalue ParmVar 0x1fb0348 'a' 'int'
|   | |-UnaryOperator 0x1fb05a8 <col:11, col:12> 'int' postfix '++'
|   | | `-DeclRefExpr 0x1fb0588 <col:11> 'int' lvalue ParmVar 0x1fb03c8 'b' 'int'
|   | `-UnaryOperator 0x1fb05e0 <line:5:10, col:11> 'int' postfix '--'
|   |   `-DeclRefExpr 0x1fb05c0 <col:10> 'int' lvalue ParmVar 0x1fb03c8 'b' 'int'
|   `-ReturnStmt 0x1fb0658 <line:6:5, col:12>
|     `-ImplicitCastExpr 0x1fb0640 <col:12> 'int' <LValueToRValue>
|       `-DeclRefExpr 0x1fb0620 <col:12> 'int' lvalue ParmVar 0x1fb03c8 'b' 'int'
```

If-Else语句的AST

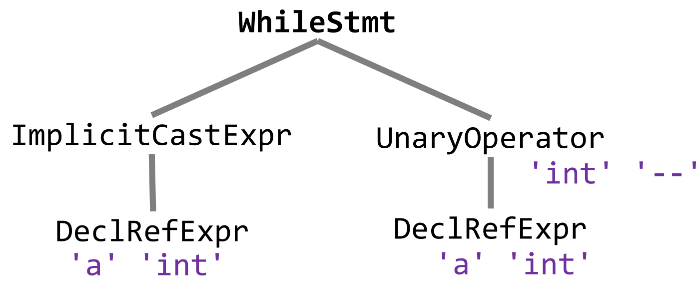
```
int toy(int a, int b){
  if(a) b++;
  else b--;
  return b;
}
```




```
#:clang -Xclang -ast-dump -fsyntax-only ifelse.c
|-FunctionDecl 0x1fb04a0 <ifelse.c:3:1, line:7:1> line:3:5 used phib 'int (int, int)'
| |-ParmVarDecl 0x1fb0348 <col:10, col:14> col:14 used a 'int'
| |-ParmVarDecl 0x1fb03c8 <col:17, col:21> col:21 used b 'int'
| `-CompoundStmt 0x1fb0668 <col:23, line:7:1>
|   |-IfStmt 0x1fb05f8 <line:4:5, line:5:11> has_else
|   | |-ImplicitCastExpr 0x1fb0570 <line:4:8> 'int' <LValueToRValue>
|   | | `-DeclRefExpr 0x1fb0550 <col:8> 'int' lvalue ParmVar 0x1fb0348 'a' 'int'
|   | |-UnaryOperator 0x1fb05a8 <col:11, col:12> 'int' postfix '++'
|   | | `-DeclRefExpr 0x1fb0588 <col:11> 'int' lvalue ParmVar 0x1fb03c8 'b' 'int'
|   | `-UnaryOperator 0x1fb05e0 <line:5:10, col:11> 'int' postfix '--'
|   |   `-DeclRefExpr 0x1fb05c0 <col:10> 'int' lvalue ParmVar 0x1fb03c8 'b' 'int'
|   `-ReturnStmt 0x1fb0658 <line:6:5, col:12>
|     `-ImplicitCastExpr 0x1fb0640 <col:12> 'int' <LValueToRValue>
|       `-DeclRefExpr 0x1fb0620 <col:12> 'int' lvalue ParmVar 0x1fb03c8 'b' 'int'
```

While语句的AST

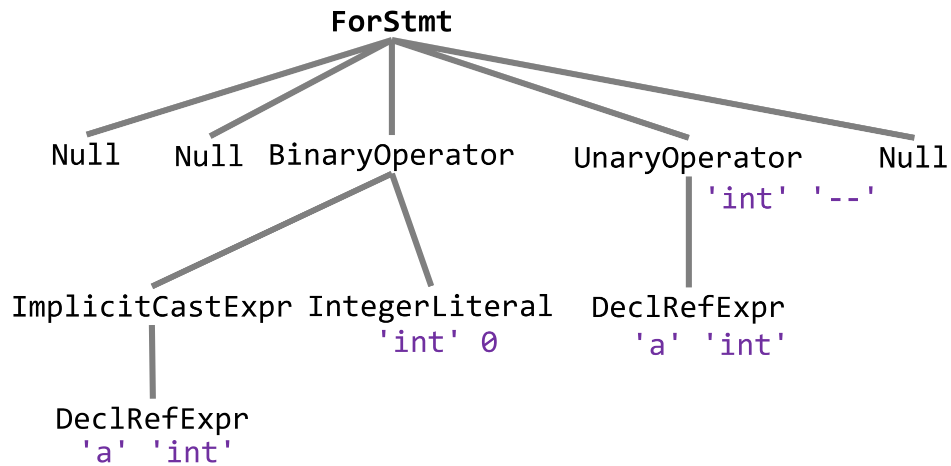
```c
int toywhile(int a){
    while(a)
        a--;
    return a;
}
```




```
#:clang -Xclang -ast-dump -fsyntax-only while.c
|-FunctionDecl 0x2263310 <while.c:3:1, line:7:1> line:3:5 toywhile 'int (int)'
| |-ParmVarDecl 0x2263278 <col:14, col:18> col:18 used a 'int'
| `-CompoundStmt 0x2263488 <col:20, line:7:1>
|   |-WhileStmt 0x2263428 <line:4:5, line:5:3>
|   | |-ImplicitCastExpr 0x22633d8 <line:4:11> 'int' <LValueToRValue>
|   | | `-DeclRefExpr 0x22633b8 <col:11> 'int' lvalue ParmVar 0x2263278 'a' 'int'
|   | `-UnaryOperator 0x2263410 <line:5:2, col:3> 'int' postfix '--'
|   |   `-DeclRefExpr 0x22633f0 <col:2> 'int' lvalue ParmVar 0x2263278 'a' 'int'
|   `-ReturnStmt 0x2263478 <line:6:5, col:12>
|     `-ImplicitCastExpr 0x2263460 <col:12> 'int' <LValueToRValue>
|       `-DeclRefExpr 0x2263440 <col:12> 'int' lvalue ParmVar 0x2263278 'a' 'int'
```

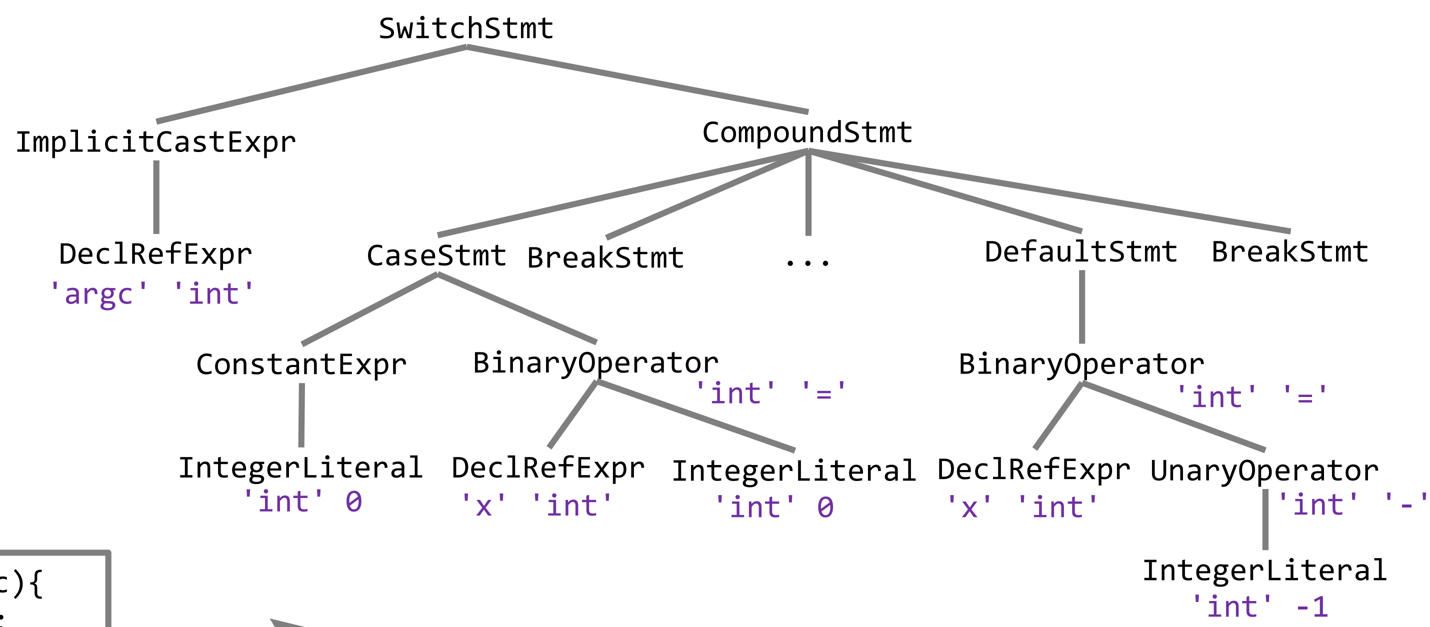
For语句的AST

```c
int toyfor(int a){
    for(; a>0; a--)
        ;
    return a;
}
```




```
#:clang -Xclang -ast-dump -fsyntax-only while.c
|-FunctionDecl 0x1108310 <for.c:3:1, line:7:1> line:3:5 toywhile 'int (int)'
| |-ParmVarDecl 0x1108278 <col:14, col:18> col:18 used a 'int'
| `-CompoundStmt 0x11084f0 <col:20, line:7:1>
|   |-ForStmt 0x1108470 <line:4:5, line:5:2>
|   | |-<<<NULL>>>
|   | |-<<<NULL>>>
|   | |-BinaryOperator 0x1108410 <line:4:11, col:13> 'int' '>'
|   | | |-ImplicitCastExpr 0x11083f8 <col:11> 'int' <LValueToRValue>
|   | | | `-DeclRefExpr 0x11083b8 <col:11> 'int' lvalue ParmVar 0x1108278 'a' 'int'
|   | | `-IntegerLiteral 0x11083d8 <col:13> 'int' 0
|   | |-UnaryOperator 0x1108450 <col:16, col:17> 'int' postfix '--'
|   | | `-DeclRefExpr 0x1108430 <col:16> 'int' lvalue ParmVar 0x1108278 'a' 'int'
|   | `-NullStmt 0x1108468 <line:5:2>
|   `-ReturnStmt 0x11084e0 <line:6:5, col:12>
|     `-ImplicitCastExpr 0x11084c8 <col:12> 'int' <LValueToRValue>
|       `-DeclRefExpr 0x11084a8 <col:12> 'int' lvalue ParmVar 0x1108278 'a' 'int'
```

Switch-Case语句的AST




```
switch(argc){
    case 0:
        x = 0;
        break;
    case 1:
        x = 1;
        break;
    case 2:
        x = 3;
        break;
    default:
        x = -1;
        break;
}
```

标识符：Identifiers

- 局部变量：%开头
- 全局变量：@开头
- 常量

```
int global_var = 1;

int toy(int a, int b){
  int r = (a + global_var) * 2;
  return r;
}
```

```
define dso_local i32 @ident(i32 %0) #0 {
  %2 = load i32, i32* @global_var, align 4
  %3 = add nsw i32 %0, %2
  %4 = mul nsw i32 %3, 2
  ret i32 %4
}
```

数据存取

- 内存分配：alloc
- 数据读取：load
- 数据存入：store

```
void ptr(int* a){
  *a = *a + 1;
}

int main(int argc, char** argv){
  int a = 1;
  ptr(&a);
}
```

```
define dso_local void @ptr(i32* %0) #0 {
  %2 = load i32, i32* %0, align 4
  %3 = add nsw i32 %2, 1
  store i32 %3, i32* %0, align 4
  ret void
}

define dso_local i32 @main(i32 %0, i8** %1) #0 {
  %3 = alloca i32, align 4
  store i32 1, i32* %3, align 4
  call void @ptr(i32* %3)
  ret i32 0
}
```

整数运算

- 二元整数运算
  - add
  - sub
  - mul
  - sdiv
  - udiv
  - urem
  - srem

```
void intarith(int x, int y){
  int a = x + y;
  int b = x - y;
  int c = x * y;
  int d = x / y;
  int e = x % y;
  int f = (unsigned int) x / (unsigned int) y;
  int g = (unsigned int) x % (unsigned int) y;
  ...
}
```

⇩

```
define dso_local void @intarith(i32 %0, i32 %1) #0 {
  %3 = add nsw i32 %0, %1
  %4 = sub nsw i32 %0, %1
  %5 = mul nsw i32 %0, %1
  %6 = sdiv i32 %0, %1
  %7 = srem i32 %0, %1
  %8 = udiv i32 %0, %1
  %9 = urem i32 %0, %1
  ...
}
```

- 如果x=5，y=-3，计算a、b、c、d、e、f、g
  - 2,8,-15,-1,2,0,5

位运算

- 二元位运算
  - and
  - or
  - xor
  - shl
  - ashr
  - lshr

```c
void bitops(int x, int y){
  int a = x & y;
  int b = x | y;
  int c = x ^ y;
  int d = x << 1;
  int e = x >> 1;
  int f = (unsigned int) x >> 1;
  ...
}
```

```llvm
define dso_local void @bitops(i32 %0, i32 %1) #0 {
  %3 = and i32 %0, %1
  %4 = or i32 %0, %1
  %5 = xor i32 %0, %1
  %6 = shl i32 %0, 1
  %7 = ashr i32 %0, 1
  %8 = lshr i32 %0, 1
  ...
}
```

- 如果x=-2，y=1，计算a、b、c、d、e、f
  - 00000000,ffffffff,ffffffff,fffffffc,ffffffff,7fffffff

浮点数运算

- 二元浮点数运算
  - fadd
  - fsub
  - fmul
  - fdiv
  - frem
- 一元运算
  - fneg

```
void farith(float x, float y){
  float a = -x + y;
  float b = x - y;
  float c = x * y;
  float d = x / y;
  ...
}
```

⇩

```
define dso_local void @farith(float %0, float %1) #0 {
  %3 = fneg float %0
  %4 = fadd float %3, %1
  %5 = fsub float %0, %1
  %6 = fmul float %0, %1
  %7 = fdiv float %0, %1
  ...
}
```

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可表示成01000011010010000000000000000000
  - $2^7 \times 1.5625 = 200$

$$0\underbrace{10000110}_{\text{exponent (8 bits)}}\underbrace{10010000000000000000000}_{\text{mantissa (23 bits)}}$$

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

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

根据实数计算浮点数

$$(11.25)_{10} = (?)_2$$

```
11/2 = 5 + 1
5/2  = 2 + 1
2/2  = 1 + 0
1/2  = 0 + 1
```

```
0.25 * 2 = 0.5 + 0
0.50 * 2 = 0.0 + 1
```

```
   1011.01
1+.01101      ⇨  exp = 3
```

```
01000001001101000000000000000000
```

练习

- 下列哪个小数可以使用浮点数精确表示？
  - 0.1, 0.2, 0.3, 0.4, 0.5

```
0.1 * 2 = 0.2 + 0     0.3 * 2 = 0.6 + 0     0.5 * 2 = 0 + 1
0.2 * 2 = 0.4 + 0     0.6 * 2 = 0.2 + 1
0.4 * 2 = 0.8 + 0     0.2 * 2 = 0.4 + 0
0.8 * 2 = 0.6 + 1     …
0.6 * 2 = 0.2 + 1
…
0001.100110011…       01.001100110011…      1.000…

0.1 = 001111011100110011…
0.2 = 001111100100110011…
0.3 = 001111101001100110…
0.4 = 001111101100110011…
0.4 = 001111110000000000…
```

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

```
void convert(int x){
  short a = x;
  long b = x;
  unsigned int c = (unsigned short) a;
  float d = x;
  float e = (unsigned int) x;
  int f = c;
  unsigned int g = d;
  double h = d;
  float i = h;
  void* j = b;
  int k = j;
  ...
}
```

⇩

```
define dso_local void @convert(i32 %0) #0 {
  %2 = trunc i32 %0 to i16
  %3 = sext i32 %0 to i64
  %4 = zext i16 %2 to i32
  %5 = sitofp i32 %0 to float
  %6 = uitofp i32 %0 to float
  %7 = fptoui float %5 to i32
  %8 = fpext float %5 to double
  %9 = fptrunc double %8 to float
  %10 = inttoptr i64 %3 to i8*
  %11 = ptrtoint i8* %10 to i32
  ...
}
```

指针操作

- getelementptr

```c
void array(int x){
  int a[2];
  a[1] = 99;
}
```

```llvm
define dso_local void @array(i32 %0) #0 {
  %2 = alloca [2 x i32], align 4
  %3 = getelementptr inbounds [2 x i32],
             [2 x i32]* %2, i64 0, i64 1
  store i32 99, i32* %3, align 4
  ret void
}
```

```c
struct mystruct_t{
  int i;
  float f;
};

void callstruct(int x){
  struct mystruct_t s;
  s.i = 1;
  int a = s.i;
}
```

```llvm
%struct.mystruct_t = type { i32, float }

define dso_local void @callstruct(i32 %0) #0 {
  %2 = alloca %struct.mystruct_t, align 4
  %3 = getelementptr inbounds %struct.mystruct_t,
             %struct.mystruct_t* %2, i32 0, i32 0
  store i32 1, i32* %3, align 4
  %4 = getelementptr inbounds %struct.mystruct_t,
             %struct.mystruct_t* %2, i32 0, i32 0
  %5 = load i32, i32* %4, align 4
  ret void
}
```

比较运算

- icmp
  - eq
  - ne
  - ugt
  - uge
  - ult
  - ule
  - sgt
  - sge
  - slt
  - sle
- fcmp

```
%0 = icmp eq i32 4, 5                    F
%1 = icmp ne float* @a, @a               F
%2 = icmp ult i16  4, 5                  T
%3 = icmp sgt i16  4, 5                  F
%4 = icmp ule i16 -4, 5                  F
%5 = icmp sge i16  4, 5                  F
```

控制流相关

- br：直接跳转、条件跳转
- switch
- indirectbr：goto

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

```
switch i32 %val, label %bbdefault [
  i32 0, label %bb1
  i32 1, label %bb2
  i32 2, label %bb3 ]
```

```
int findbr(int a){
  static const void *labels[]
        = {&&bb1, &&bb2};
  goto *labels[a];
bb1:
  return 1;
bb2:
  return 0;
}
```

⇨

```
define dso_local i32 @findbr(i32 %0) #0 {
  %2 = sext i32 %0 to i64
  %3 = getelementptr inbounds [2 x i8*],
       [2 x i8*]* @findbr.labels, i64 0, i64 %2
  %4 = load i8*, i8** %3, align 8
  br label %8

5:
  br label %7

6:
  br label %7

7:
  %.0 = phi i32 [ 1, %5 ], [ 0, %6 ]
  ret i32 %.0

8:
  %9 = phi i8* [ %4, %1 ]
  indirectbr i8* %9, [label %5, label %6]
}
```

数据流相关

- phi
- select

```
r = a>0? 1 : -1;
```

```
%2 = icmp sgt i32 %0, 0
%3 = zext i1 %2 to i64
%4 = select i1 %2, i32 1, i32 -1
```

函数调用和返回

- 函数调用：call
- 返回指令：ret

```
call void @ptr(i32* %3)
%2 = call i32 @test(i32 %1)
```

LLVM的SSA

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

```
int phib(int a, int b){
    if(a) b++;
    return b;
}
```

ir

```
define dso_local i32 @phib(i32 %0,
i32 %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = icmp ne i32 %5, 0
  br i1 %6, label %7, label %10

7:
  %8 = load i32, i32* %4, align 4
  %9 = add nsw i32 %8, 1
  store i32 %9, i32* %4, align 4
  br label %10

10:
  %11 = load i32, i32* %4, align 4
  ret i32 %11
}
```

mem2reg

```
define dso_local i32 @phib(i32 %0, i32
%1) #0 {
  %3 = icmp ne i32 %0, 0
  br i1 %3, label %4, label %6

4:
  %5 = add nsw i32 %1, 1
  br label %6

6:
  %.0 = phi i32 [ %5, %4 ], [ %1, %2 ]
  ret i32 %.0
}
```

If-Else语句的IR

```c
int ifelse(int a, int b){
    if(a) b++;
    else b--;
    return b;
}
```

```llvm
define dso_local i32 @ifelse(i32 %0, i32 %1) #0 {
  %3 = icmp ne i32 %0, 0
  br i1 %3, label %4, label %6

4:
  %5 = add nsw i32 %1, 1
  br label %8

6:
  %7 = add nsw i32 %1, -1
  br label %8

8:
  %.0 = phi i32 [ %5, %4 ], [ %7, %6 ]
  ret i32 %.0
}
```

While语句的IR

```
int whilefun(int a){
    while(a)
        a--;
    return a;
}
```

```
define dso_local i32 @whilefun(i32 %0) #0 {
  br label %2

2:
  %.0 = phi i32 [ %0, %1 ], [ %5, %4 ]
  %3 = icmp ne i32 %.0, 0
  br i1 %3, label %4, label %6

4:
  %5 = add nsw i32 %.0, -1
  br label %2

6:
  ret i32 %.0
}
```

For语句的IR

```c
int forfun(int a){
    for(; a>0; a--)
        ;
    return a;
}
```

```llvm
define dso_local i32 @toyfor(i32 %0) #0 {
  br label %2

2:
  %.0 = phi i32 [ %0, %1 ], [ %6, %5 ]
  %3 = icmp sgt i32 %.0, 0
  br i1 %3, label %4, label %7

4:
  br label %5

5:
  %6 = add nsw i32 %.0, -1
  br label %2

7:
  ret i32 %.0
}
```

Switch-Case语句的IR

```c
int x;
switch(argc){
case 0:
    x = 0;
    break;
case 1:
    x = 1;
    break;
case 2:
    x = 3;
    break;
default:
    x = -1;
    break;
}
int y = x * 2;
```

```llvm
 switch i32 %0, label %6 [
    i32 0, label %3
    i32 1, label %4
    i32 2, label %5
  ]

3:
  br label %7

4:
  br label %7

5:
  br label %7

6:
  br label %7

7:
  %.0 = phi i32 [ -1, %6 ], [ 3, %5 ],
                [ 1, %4 ], [ 0, %3 ]
  %8 = mul nsw i32 %.0, 2
```

更多LLVM IR指令（暂时用不到）

- 异常处理相关：
  - invoke
  - landingpad
  - catchpad
  - cleanuppad
  - catchret
  - catchswitch
  - cleanupret
  - resume
- 并发访问
  - fense
  - cmpxchg
  - atomicrmw
- 其它：
  - callbr
  - va_arg
  - unreachable
  - freeze
  - bitcast..to
  - addrspacecast..to
  - extractelement
  - insertelement
  - shufflevector
  - extractvalue
  - insertvalue

异常处理

```
void fthrow(int i) { if(i < 0) throw -1; }

int main(int argc, char** argv) {
    try{ fthrow(1); } catch (const int msg) { }
}
```

```
define dso_local void @_Z5entryv() #0 personality i8* bitcast (i32 (...)* @__gxx_personality_v0 to i8*) {
  invoke void @_Z6fthrowi(i32 1)
          to label %1 unwind label %2

1:br label %13

2:%3 = landingpad { i8*, i32 }
          catch i8* bitcast (i8** @_ZTIi to i8*)
  %4 = extractvalue { i8*, i32 } %3, 0
  %5 = extractvalue { i8*, i32 } %3, 1
  br label %6

6:%7 = call i32 @llvm.eh.typeid.for(i8* bitcast (i8** @_ZTIi to i8*)) #3
  %8 = icmp eq i32 %5, %7
  br i1 %8, label %9, label %14

9:%10 = call i8* @__cxa_begin_catch(i8* %4) #3
  %11 = bitcast i8* %10 to i32*
  %12 = load i32, i32* %11, align 4
  call void @__cxa_end_catch() #3
  br label %13

13:ret void

14: %15 = insertvalue { i8*, i32 } undef, i8* %4, 0
  %16 = insertvalue { i8*, i32 } %15, i32 %5, 1
  resume { i8*, i32 } %16
}
```

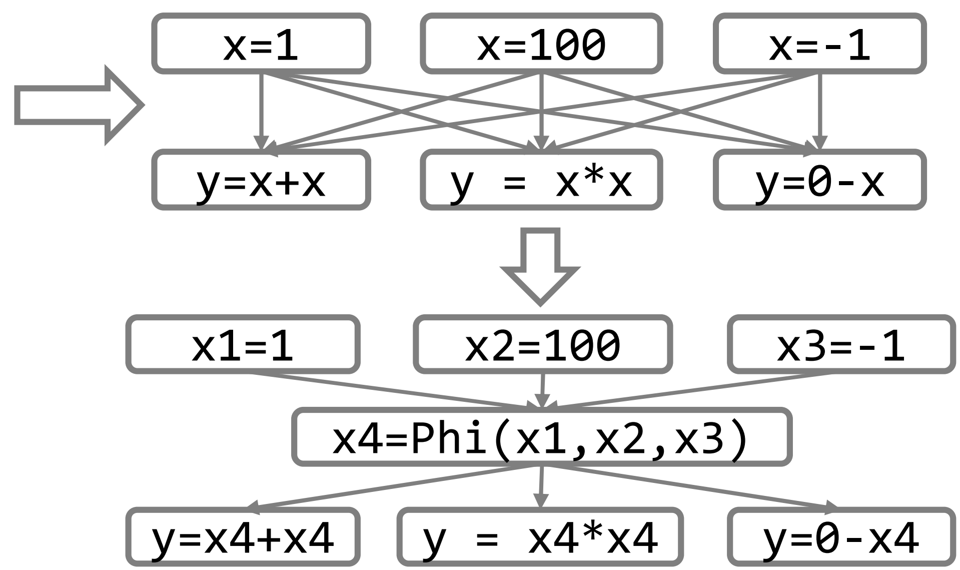
四、静态单赋值

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$O(n)$。

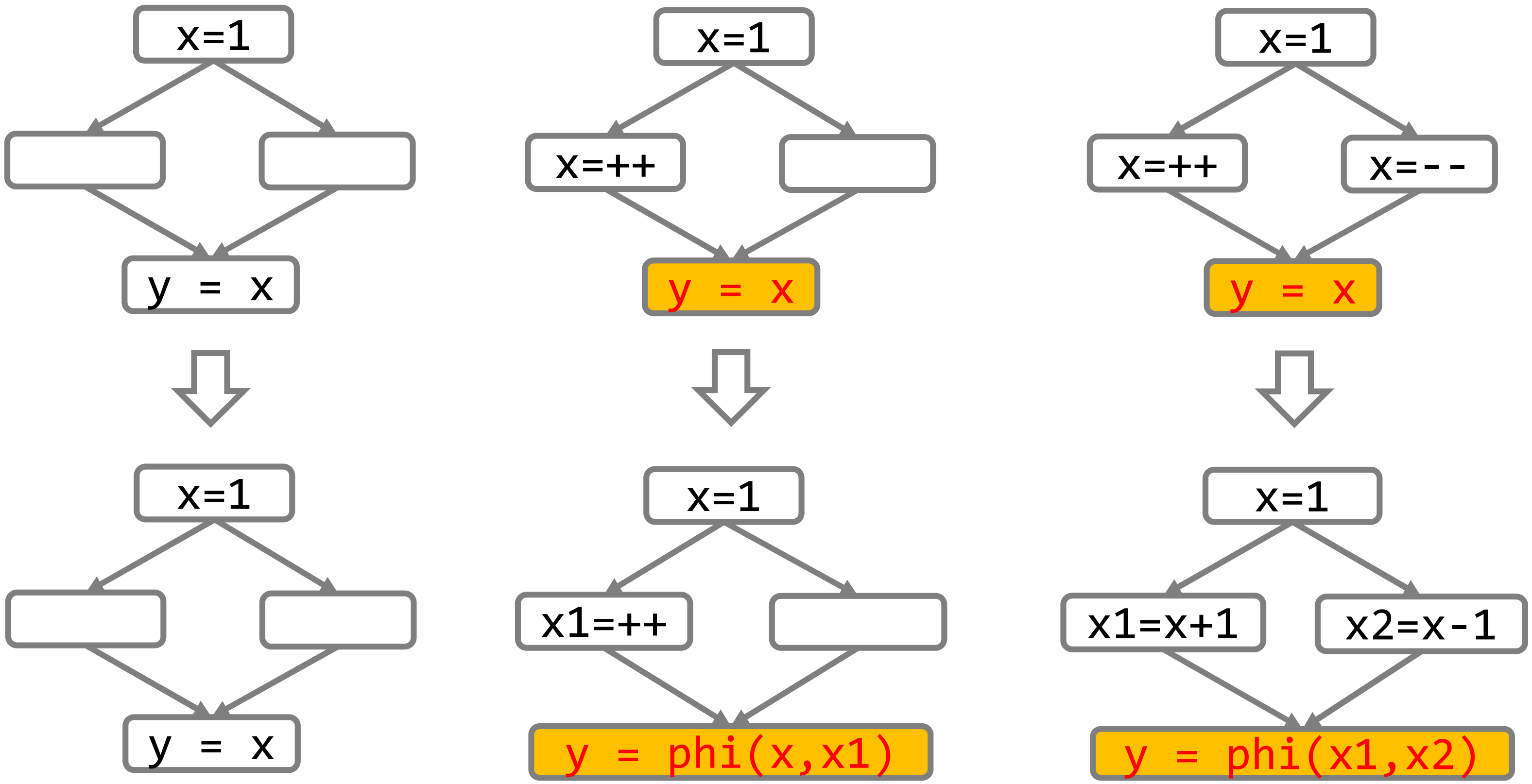
```
switch…
case 0: x = 1; break;
case 1: x = 100; break;
default: x = -1; break;
…
switch…
case 1: y = x+x; break;
case 2: y = x*x; break;
default: y = 0 – x; break;
```



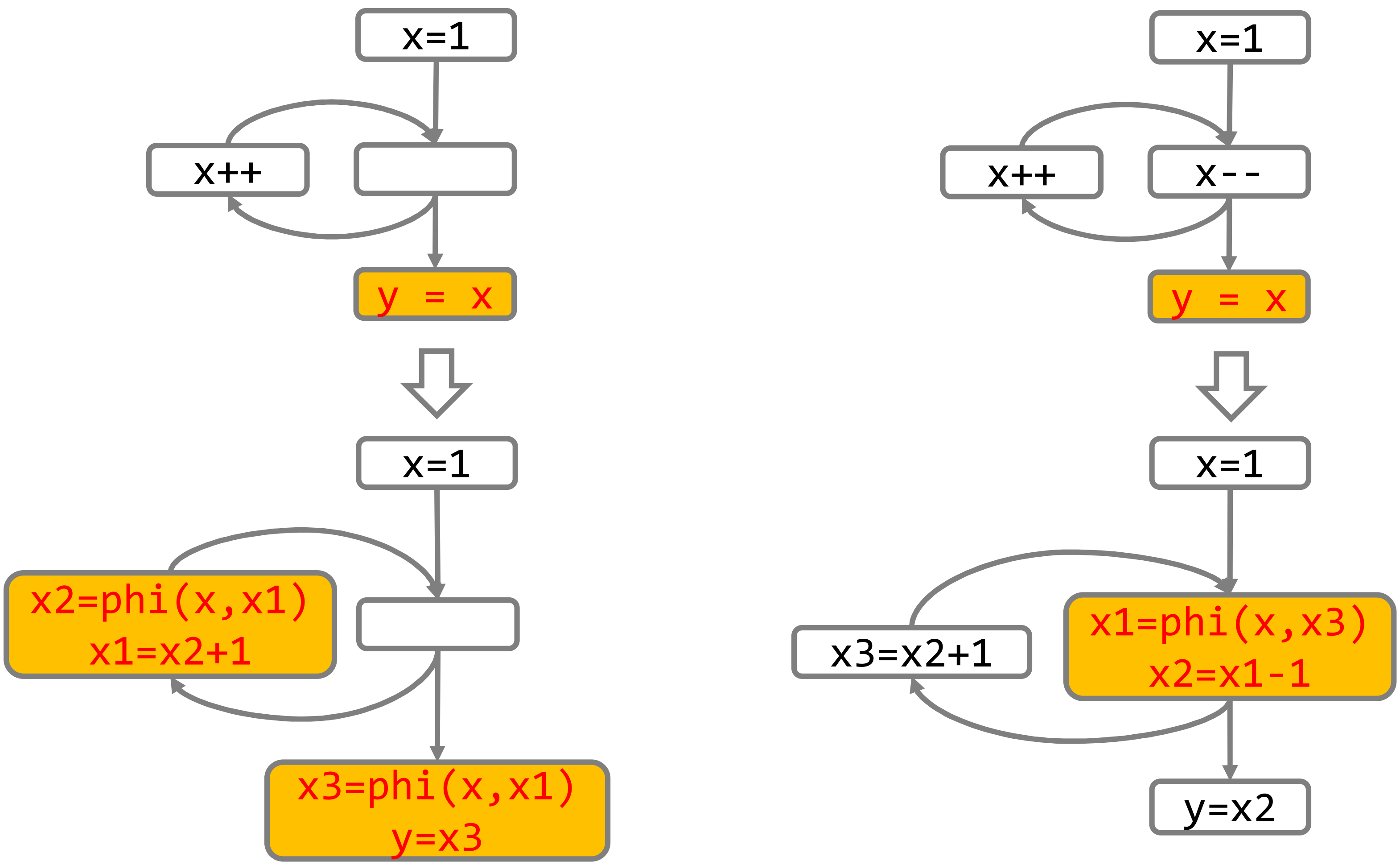
SSA的特点和构建思路

- 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？

哪种情况需要使用phi函数？

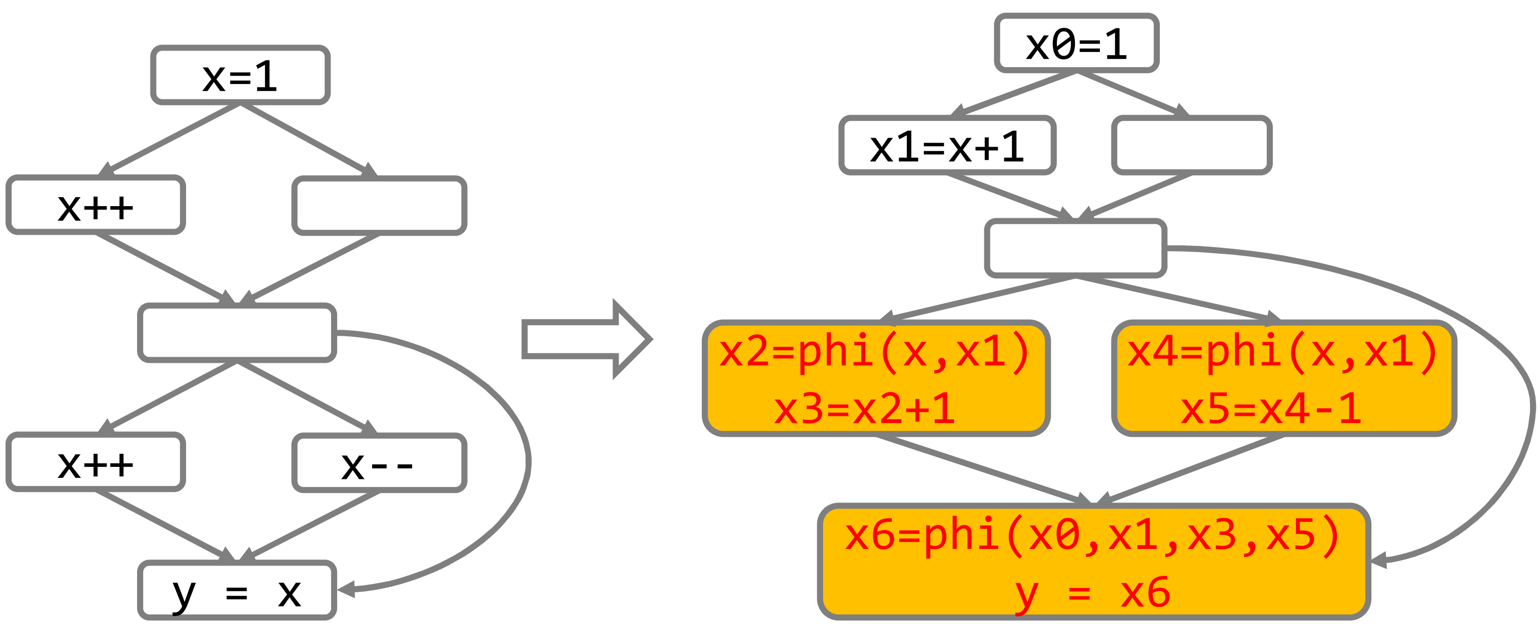


哪些节点需要使用phi函数？



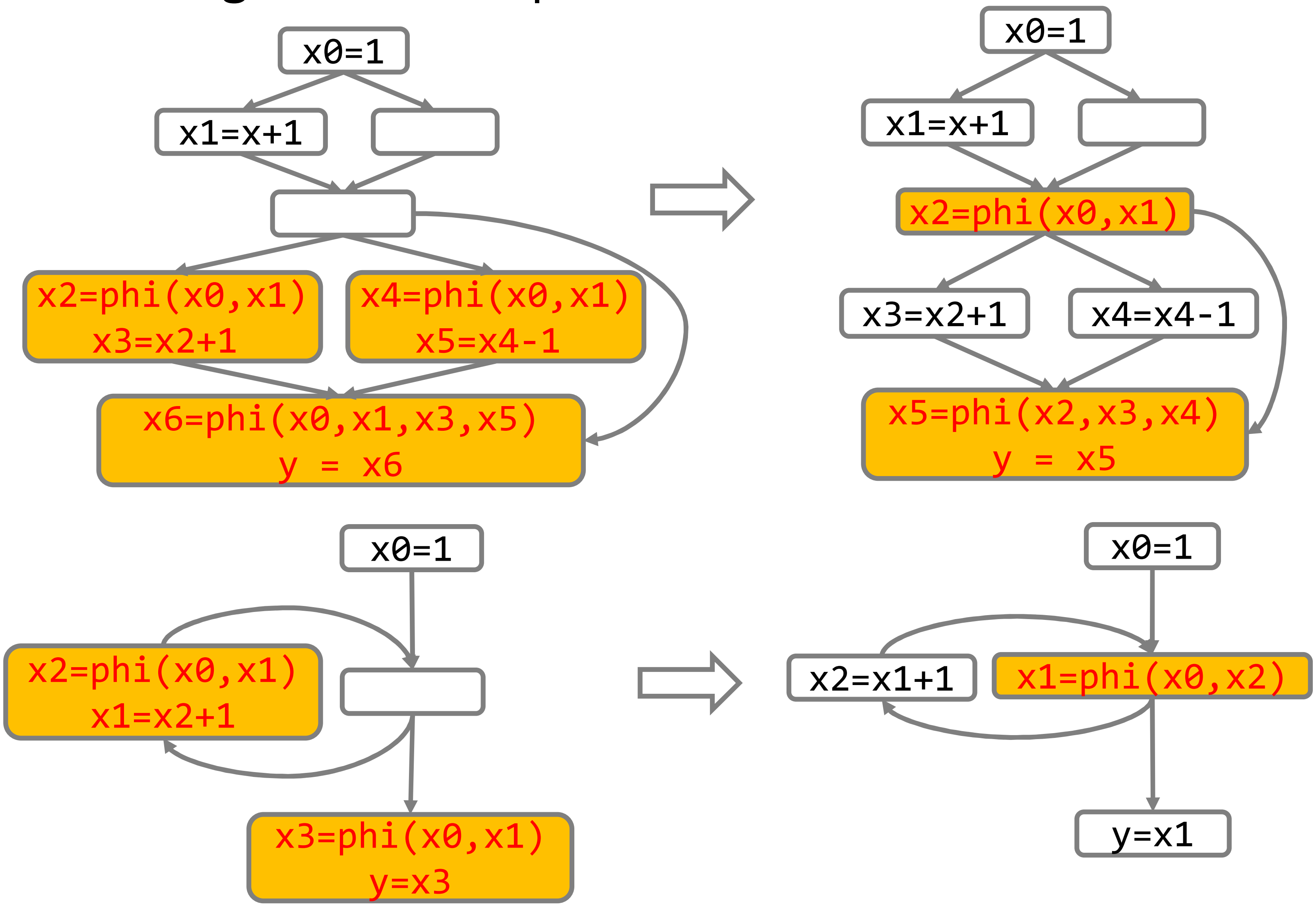
需要放置phi函数的条件

- 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

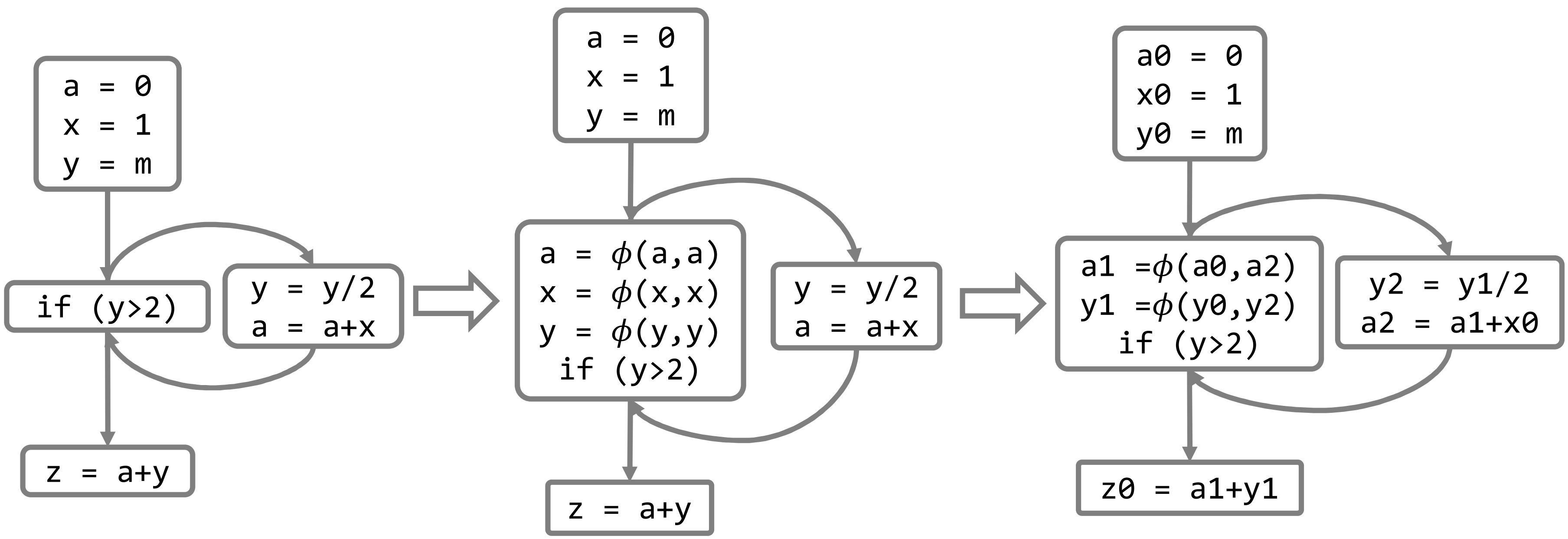


优化def-use关系

- 在merge节点上放置phi函数



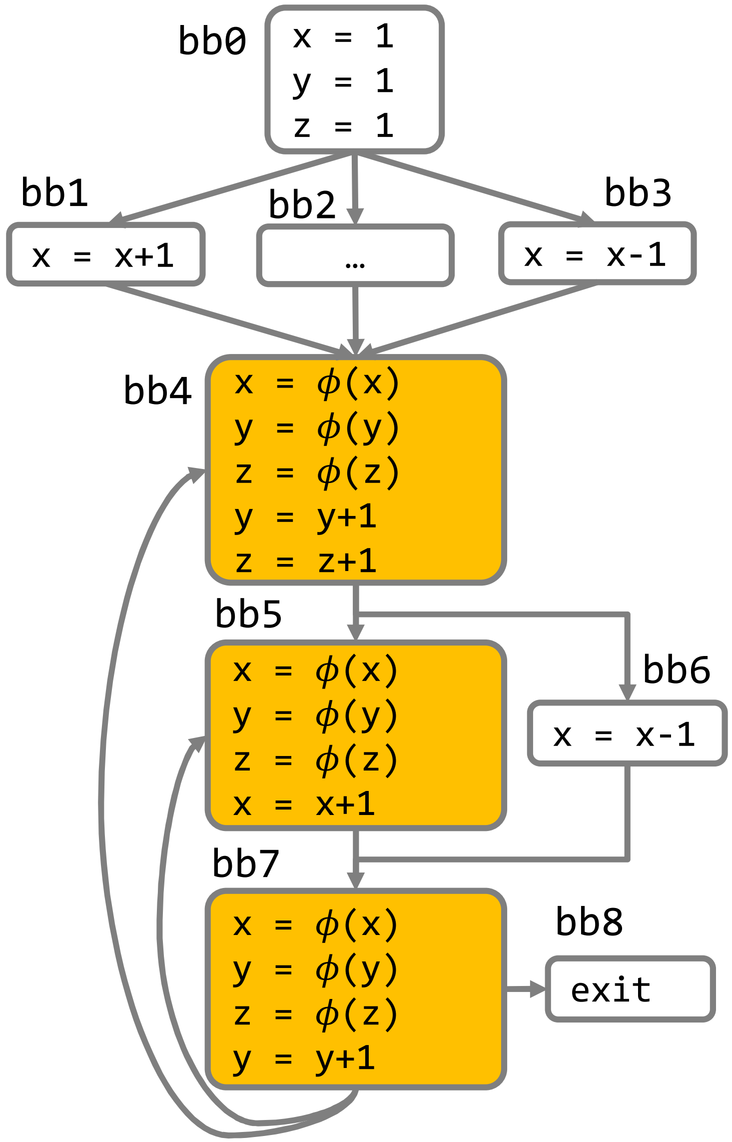
多个变量的情况


a = 0
x = 1
y = m
if (y>2)
y = y/2
a = a+x
z = a+y
a = 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,a2)
y1 =ϕ(y0,y2)
if (y>2)
y2 = y1/2
a2 = a1+x0
z0 = a1+y1

初始思路

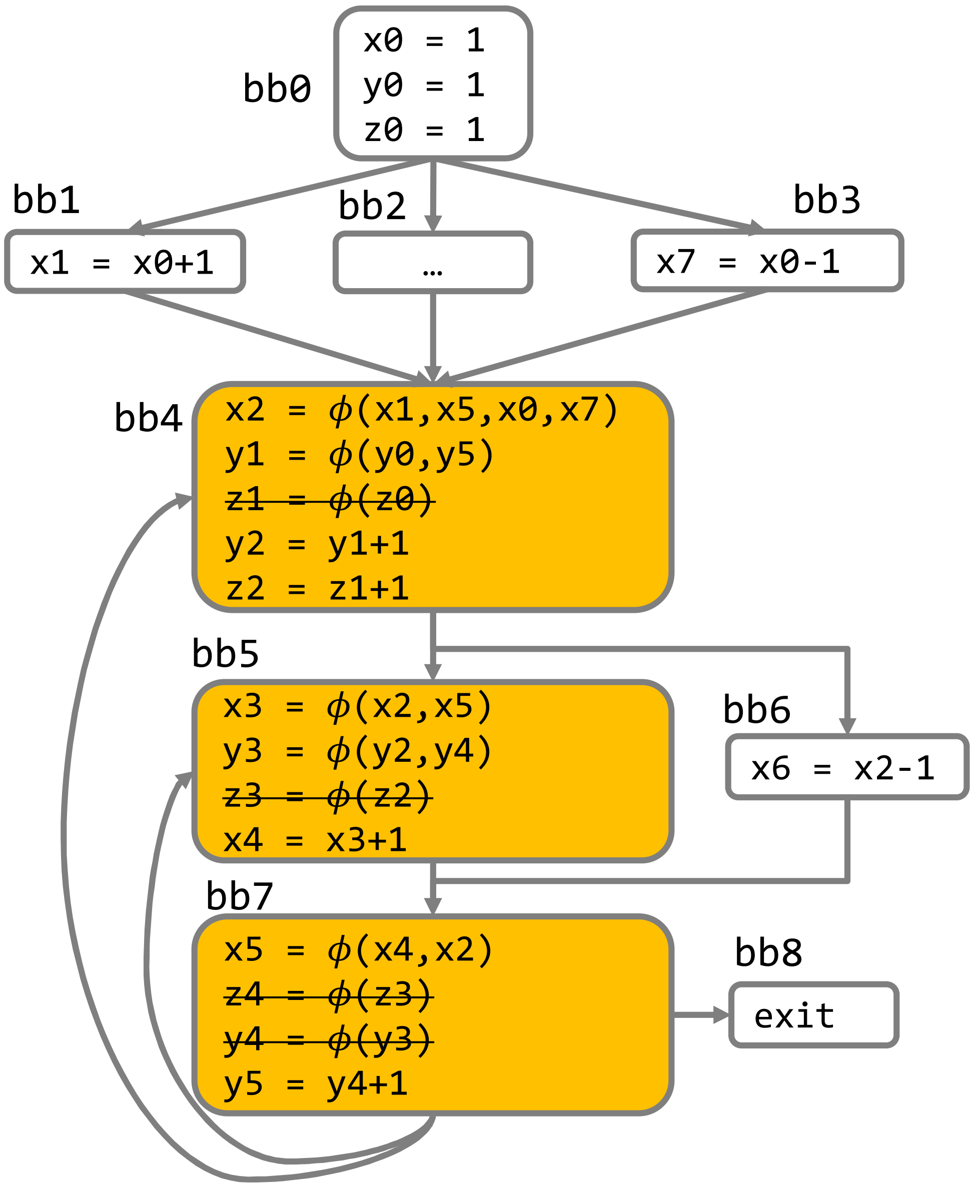
- 在入度≥2的块上放置phi节点
- 后续问题：
  - 对哪些变量使用phi函数？
  - 每个变量的ssa标识符是什么？
- 如何遍历控制流图分析标识符索引
  - 深度优先
  - 广度优先

遍历控制流图构建SSA



- DFS顺序：bb0->bb1->bb4->bb5->bb7->bb8->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
  - bb8：
  - 不能简单回退到bb4->bb6
    - 需要更新bb4、bb5
- 开销：
  - 每个节点需要更新次数为其入度
  - $\forall bb_i \to bb_j \in CFG, \text{Update}(bb_j)$

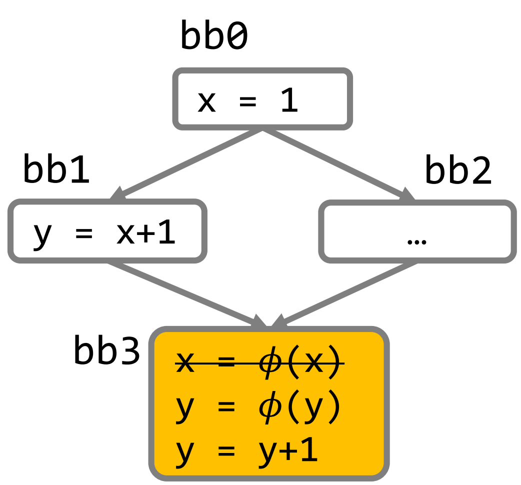
结果



- 引入了非必要的phi函数
  - bb4、bb5、bb7中的def(z)
  - bb7中的def(y)
- 效率低

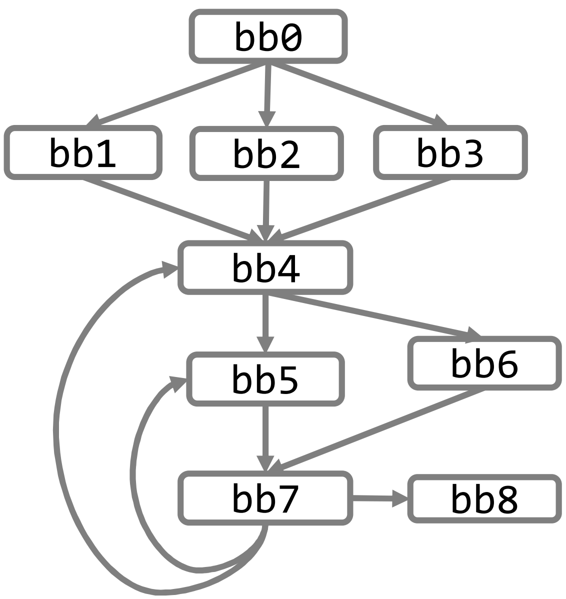
如何优化phi函数的设置？

- 如果bb1和bb2中都没有def(x)，bb3不需要phi(x)，可直接使用bb0中的def(x)。
- 如果bb1中有def(y)，bb3中很可能需要phi(y)，
  - 有可能是false positive。
- bb0支配bb2，bb1和bb2的支配边界都是bb3



支配的基本概念

- 给定有向图$G(V,E)$与起点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。

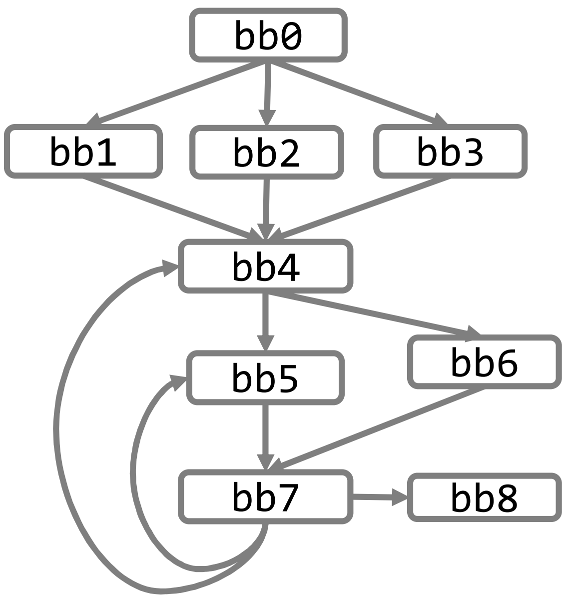


控制流图

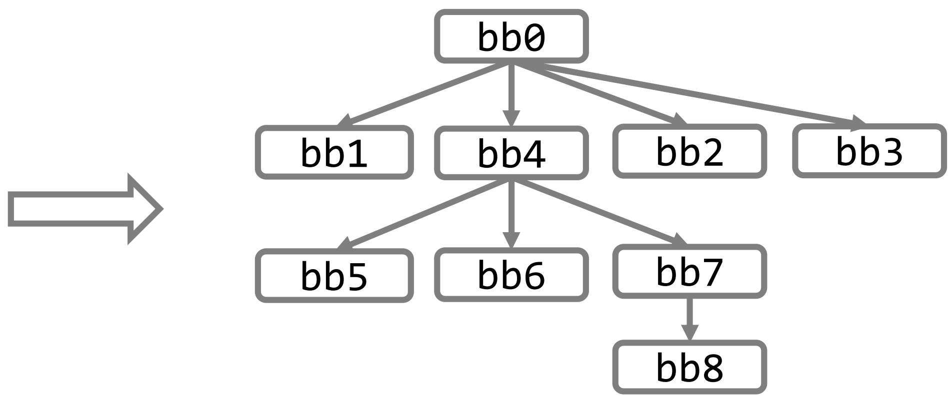
$$
\begin{aligned}
Dom(bb_0) &= \{bb_0\} \\
Dom(bb_1) &= \{bb_0, bb_1\} \\
Dom(bb_2) &= \{bb_1, bb_2\} \\
Dom(bb_3) &= \{bb_0, bb_3\} \\
Dom(bb_4) &= \{bb_0, bb_4\} \\
Dom(bb_5) &= \{bb_0, bb_4, bb_5\} \\
Dom(bb_6) &= \{bb_0, bb_4, bb_6\} \\
Dom(bb_7) &= \{bb_0, bb_4\} \\
Dom(bb_8) &= \{bb_0, bb_7\}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

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。



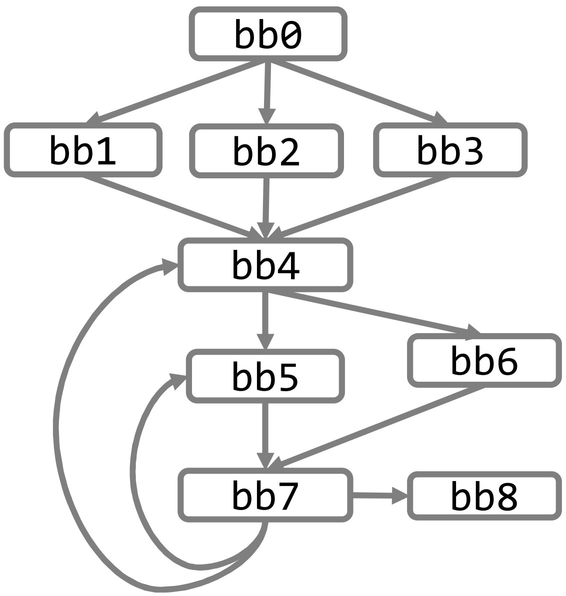
控制流图



支配树

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

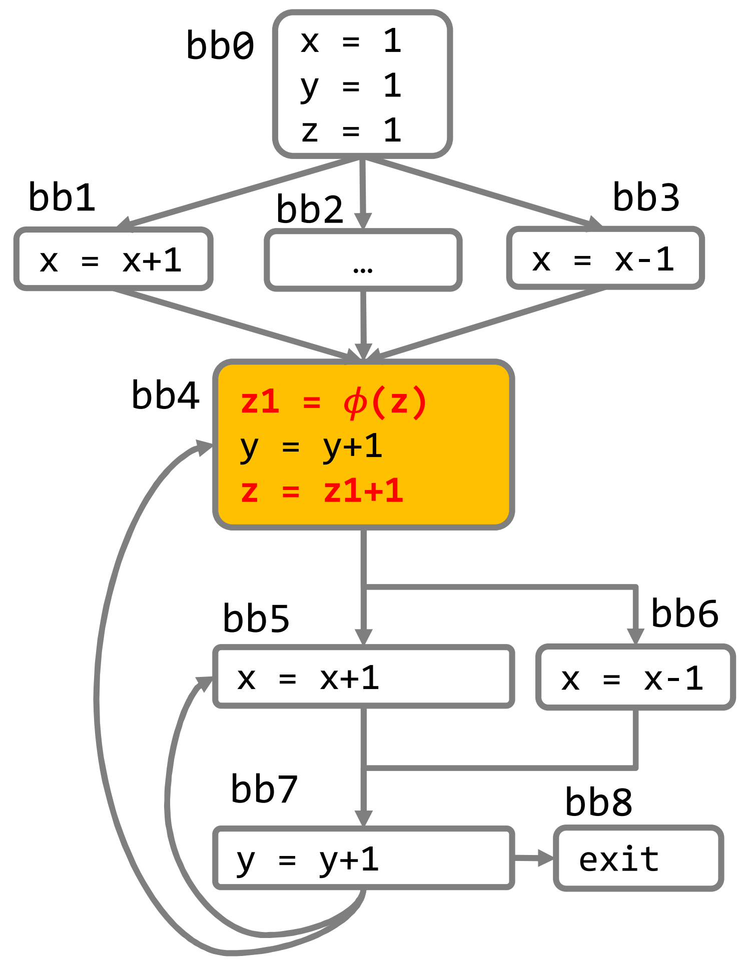


控制流图

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

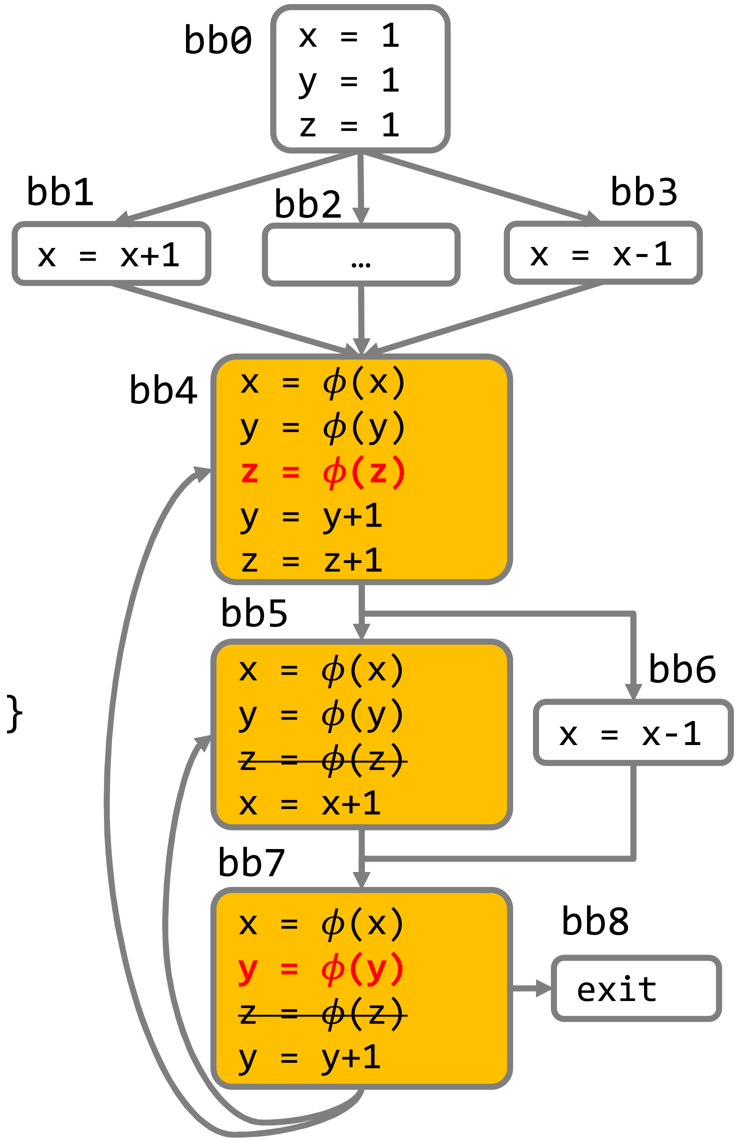
利用支配边界计算def

- 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)



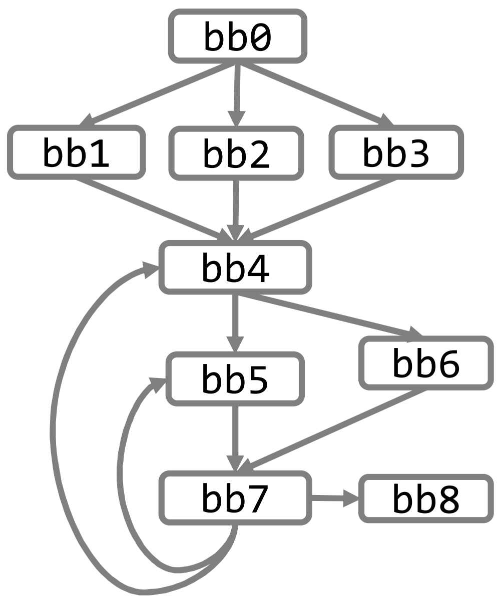
支配边界也不完美

- 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 ∈ 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 ∈ def-sites(y)
    - DF(bb5) = {bb7}
- 依然存在冗余：
  - bb4中的ϕ(z)
  - bb7中的ϕ(y)



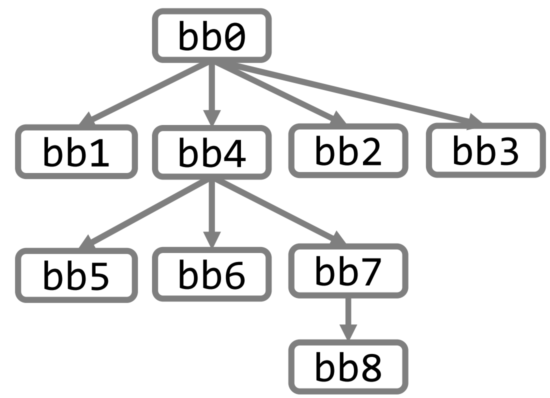
如何构建支配树：主要思路

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



控制流图

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

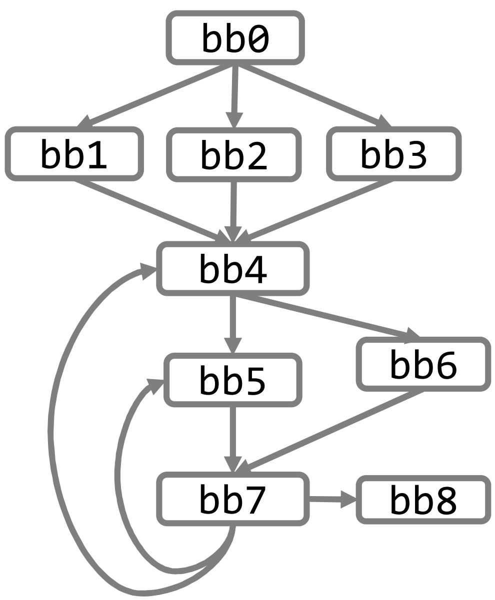


支配树

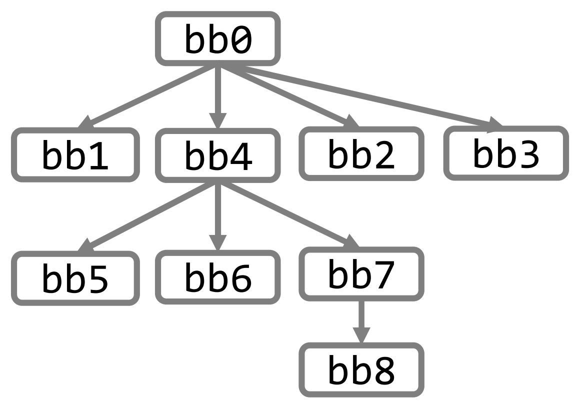
如何求支配边界：主要思路

- 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$IDom(v)$

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



控制流图



支配树

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