

COMP130014 编译

# 第六讲：类型推导

徐辉

xuh@fudan.edu.cn



# 大纲

一、类型推导问题

二、标识符作用域

三、类型约束和求解

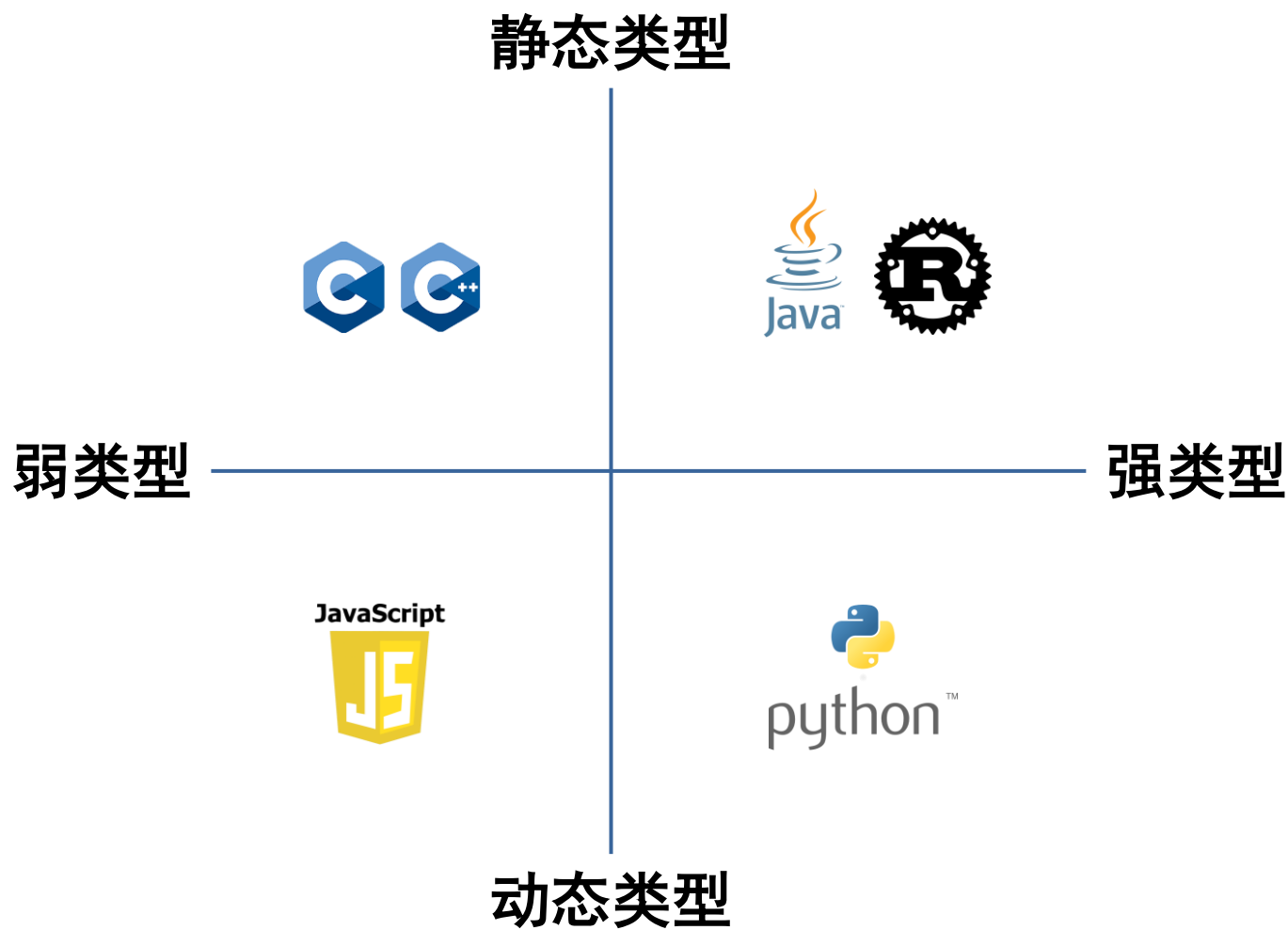
# 一、类型推导问题



# 类型系统基本概念

- 类型系统包括由类型和规则组成
- 类型：
  - 基础类型（Primitive Type）
    - 标量类型（Scalar Types）：bool、char、int、float
    - 复合类型（Compound Type）：数组、元组
  - 自定义类型：结构体、枚举
- 类型规则：
  - 类型推导和检查规则
  - 隐式类型转换

# 类型系统分类



# 动态类型 vs 静态类型

- 静态类型系统：编译时检查类型的一致性
- 动态类型系统：运行时检查类型的一致性

```
//python代码  
def foo(x):  
    if x == 1:  
        return "bingo!"  
    return x
```

```
//foo的类型是什么?  
print(foo(10))  
print(foo(1))  
print(foo(10) + foo(1))
```

```
#: python factorial.py  
10  
bingo!  
Traceback (most recent call last):  
  File "factorial.py", line 11, in <module>  
    print(foo(10) + foo(1))  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

# 强类型 vs 弱类型

- 强类型系统：一般不允许隐式类型转换
- 弱类型类型：自动隐式转换，灵活但易出错

//python代码

```
b = 1 + True;  
a = 1 + '2';  
c = '1' + True;
```

2

类型错误  
类型错误

//C代码

```
int a = 1 + true;  
int b = '1' + true;  
int c = 1 + '2';  
int d = 1 + "2";
```

2

50

51

4202501

//Javascript代码

```
1 + true;  
1 + '2';  
'1' + true;
```

2

'12'

'1true'

//Javascript代码

```
var a = 42;  
var b = "42";  
var c = [42];  
a === b;  
a == b;  
a == c;
```

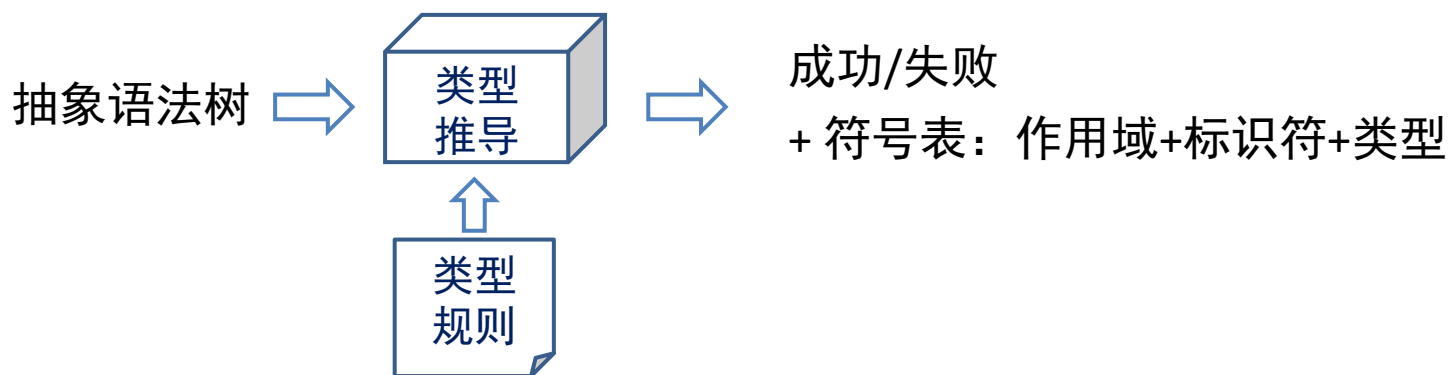
false

true

true

# 类型推导问题

- 已知源代码（抽象语法树）和类型规则
- 为所有标识符找到满足类型规则的唯一解
  - 如满足运算符、函数签名等要求
- 类型检查是类型推导问题的特例





# 类型推导思路

- 基于抽象语法树对标识符进行作用域识别和类型分析
  - 声明新标识符：确定作用域，建立索引
  - 使用标识符：确定索引，推导或检查类型

```
let g: int = 10;
fn fib(x: int) -> int {
  if (x <= 1) {
    ret x;
  }
  let a = fib(x - 1);
  let b = fib(x - 2);
  let r = a + b;
  ret r;
}
fn main() {
  let r = fib(10) + g;
}
```

标识符	作用域（粗）	索引	类型
g	global	0xd9c2	int
fib	global	0xd470	(int) → int
main	global	0xd318	(void) → void
x	fib	0xd398	int
a	fib	0xd5b0	int
b	fib	0xd2c2	int
r	fib	0x1234	int
r	main	0x82d0	int

## 二、标识符作用域

---

# 作用域分析（细粒度）

```
let g: int = 10;
fn fib(x: int) -> int { //scope fib: available {g, x}
    if (x <= 1) {
        ret x;
    }
    let a = fib(x - 1); //{ scope 1: available {g, x, a}
    let b = fib(x - 2); //{ scope 2: available {g, x, a, b}
    let r = a + b; //{ scope 3: available {g, x, a, b, r}
    ret r;
    // end scope 3 }
    // end scope 2 }
    // end scope 1 }
}

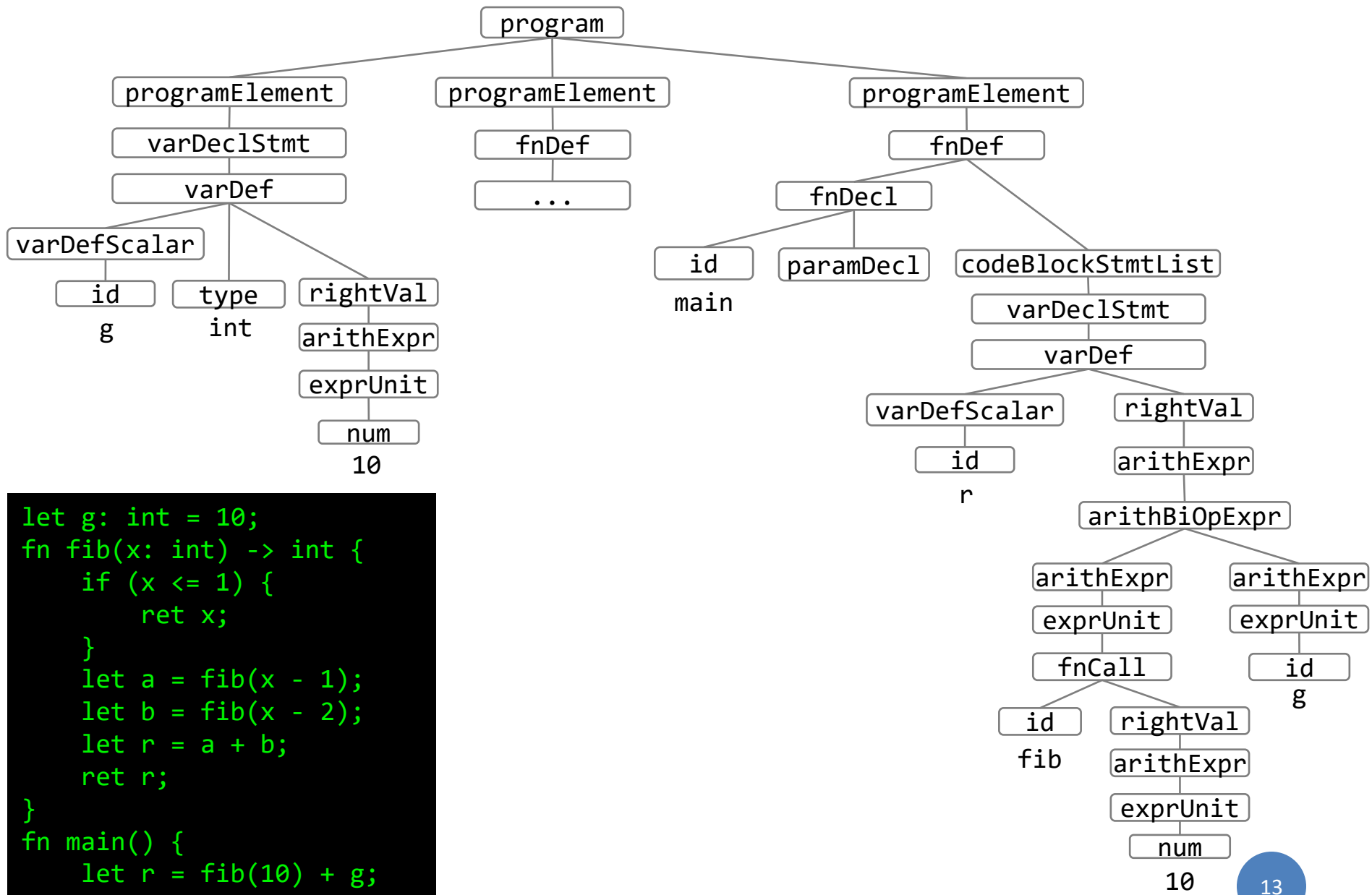
fn main() { //scope main
    let r = fib(10) + g;
}
```

Scope之间的偏序关系:  $\text{fib} > 1 > 2 > 3$

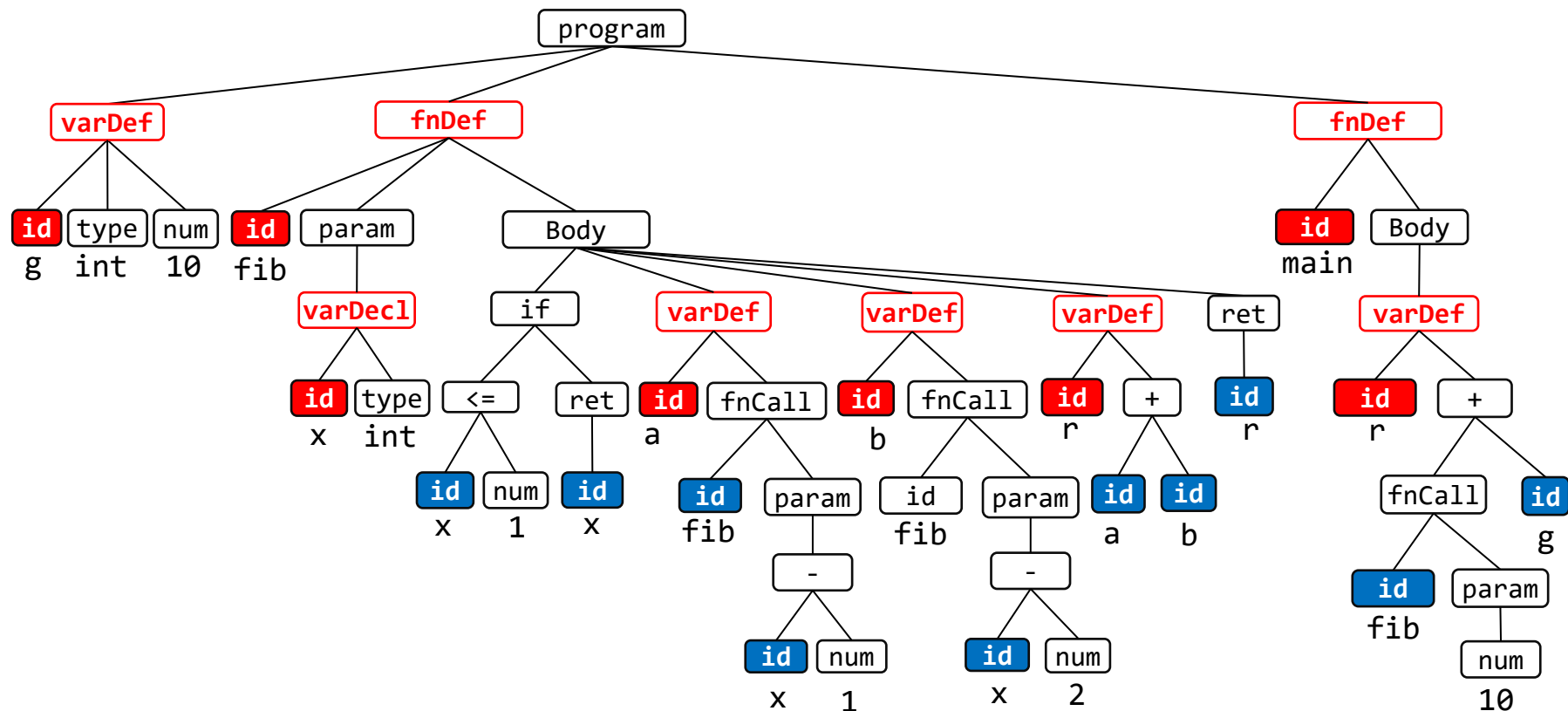
# 抽象语法树： Abstract Syntax Tree

- 具体语法：程序员实际写的代码
  - 语法解析树是对源代码的完整表示
- 抽象语法树：消除解析过程中的一些步骤或节点
  - 单一展开形式塌陷，如 $E1 \rightarrow E2 \rightarrow E3 \rightarrow \text{NUM}$
  - 去除括号等冗余信息
  - 避免在叶子结点使用运算符和保留字
- AST记录编译器的阶段性分析结果，会被持续编辑

# 示例：TeaPL编译器生成的语法树

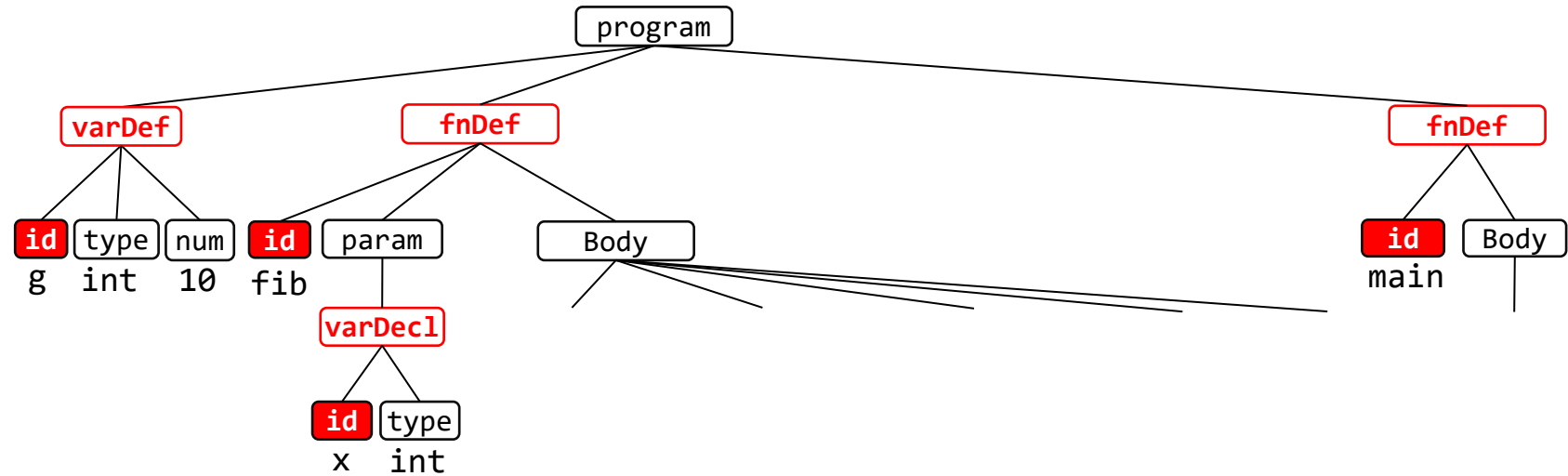


# 语法树化简=>问题抽象



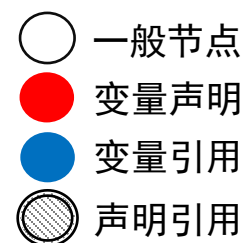
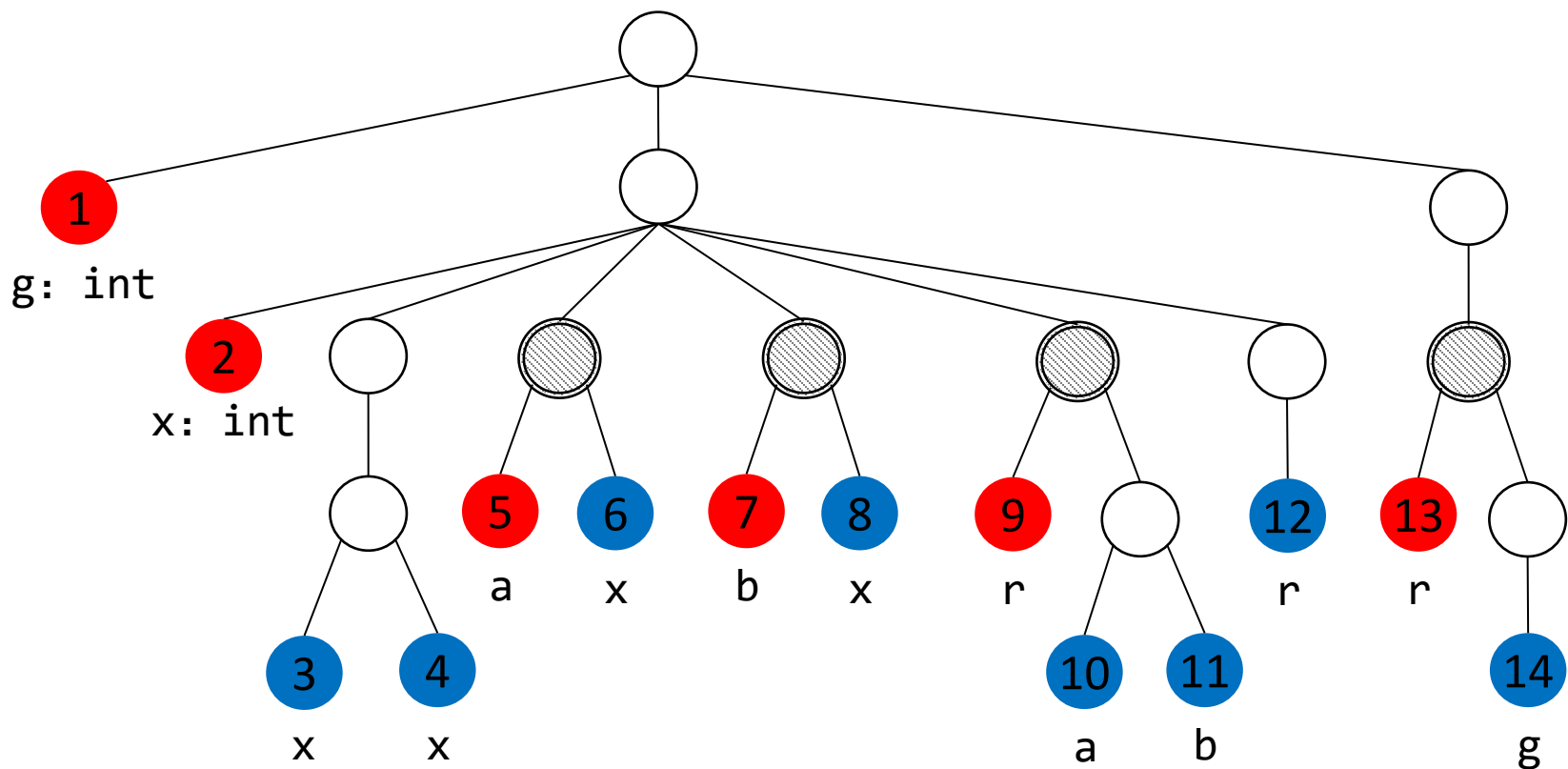
● 变量声明  
● 变量引用

# 全局标识符符号表



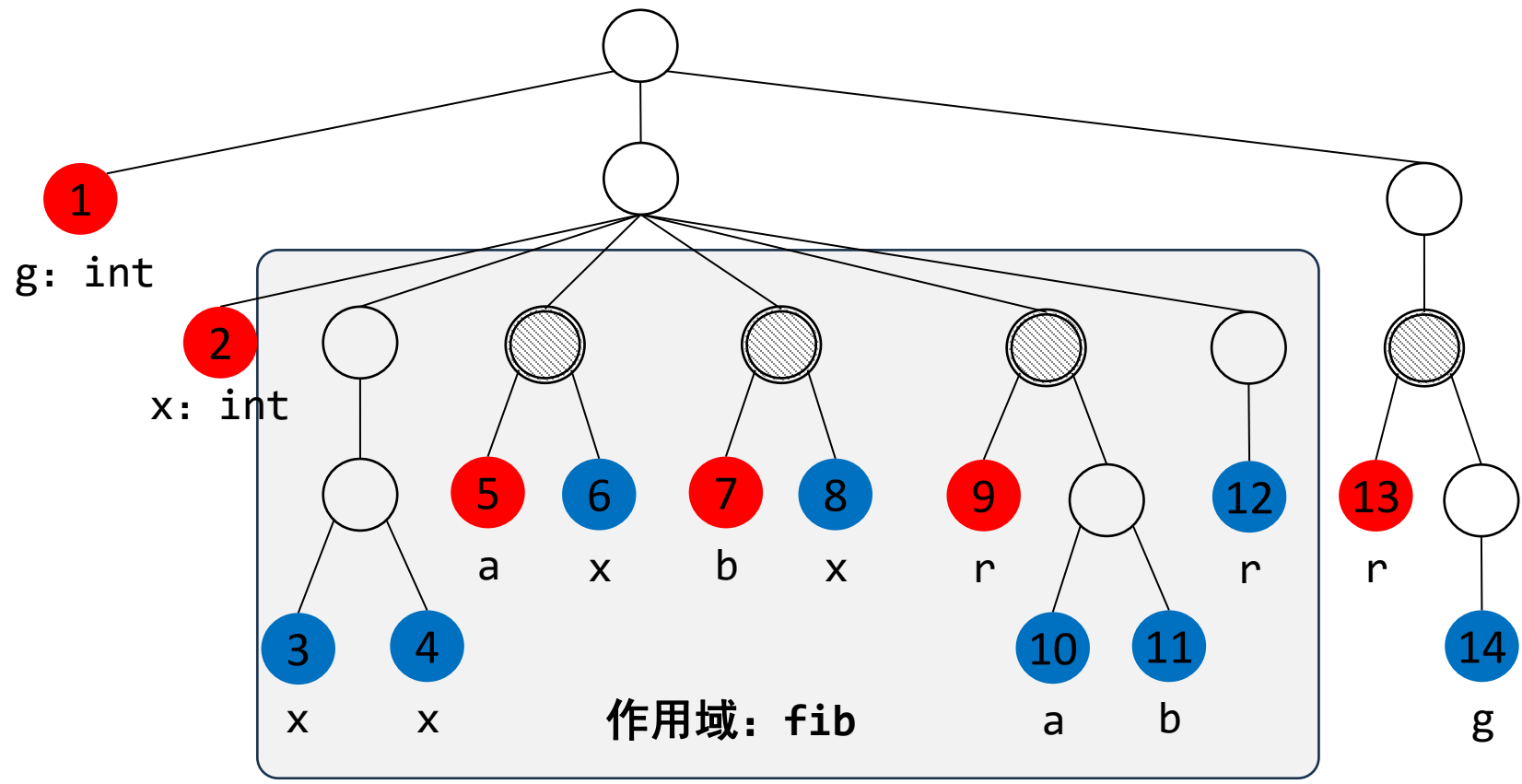
标识符	作用域	索引	类型
g	global	0xd9c2	int
fib	global	0xd470	(int) → int
main	global	0xd318	(void) → void

# 局部变量类型分析





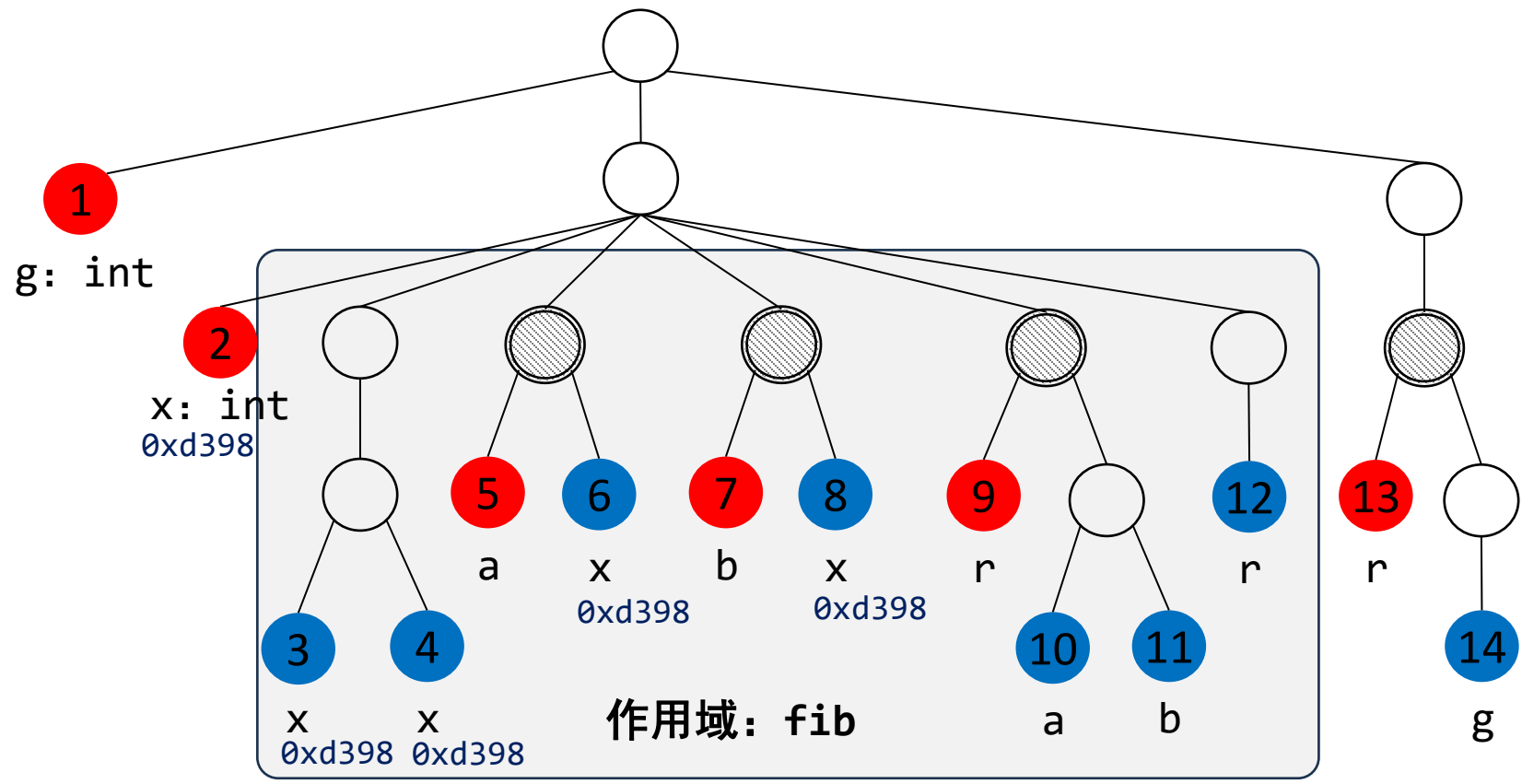
# 局部变量类型分析：x



标识符	作用域	索引	类型
x	fib	0xd398	int
a			
b			
r			

- 一般节点
- 变量声明
- 变量引用
- ⦿ 声明引用

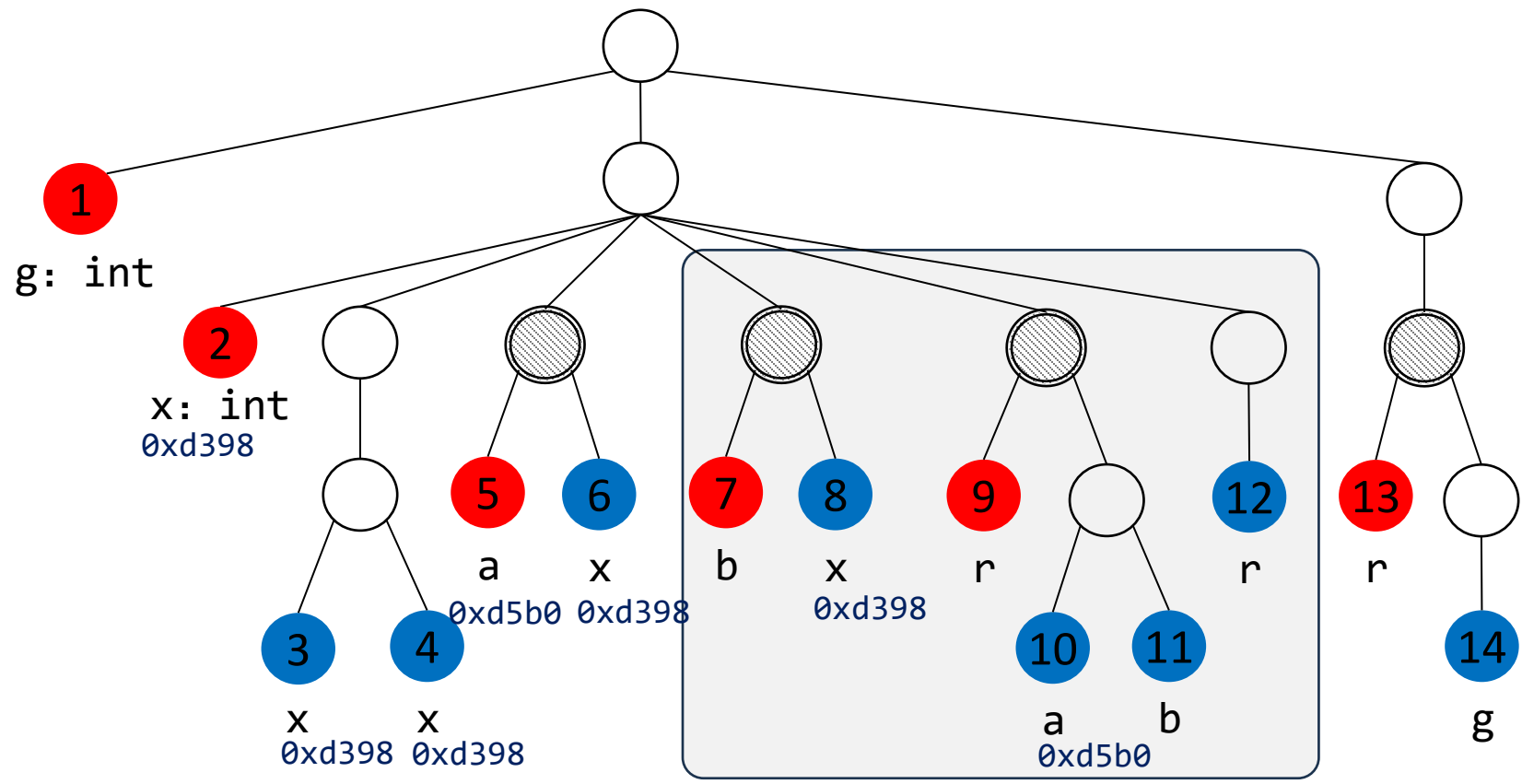
# 标识符索引化: x



标识符	作用域	索引	类型
x	fib	0xd398	int
a			
b			
r			

- 一般节点
- 变量声明
- 变量引用
- ⊘ 声明引用

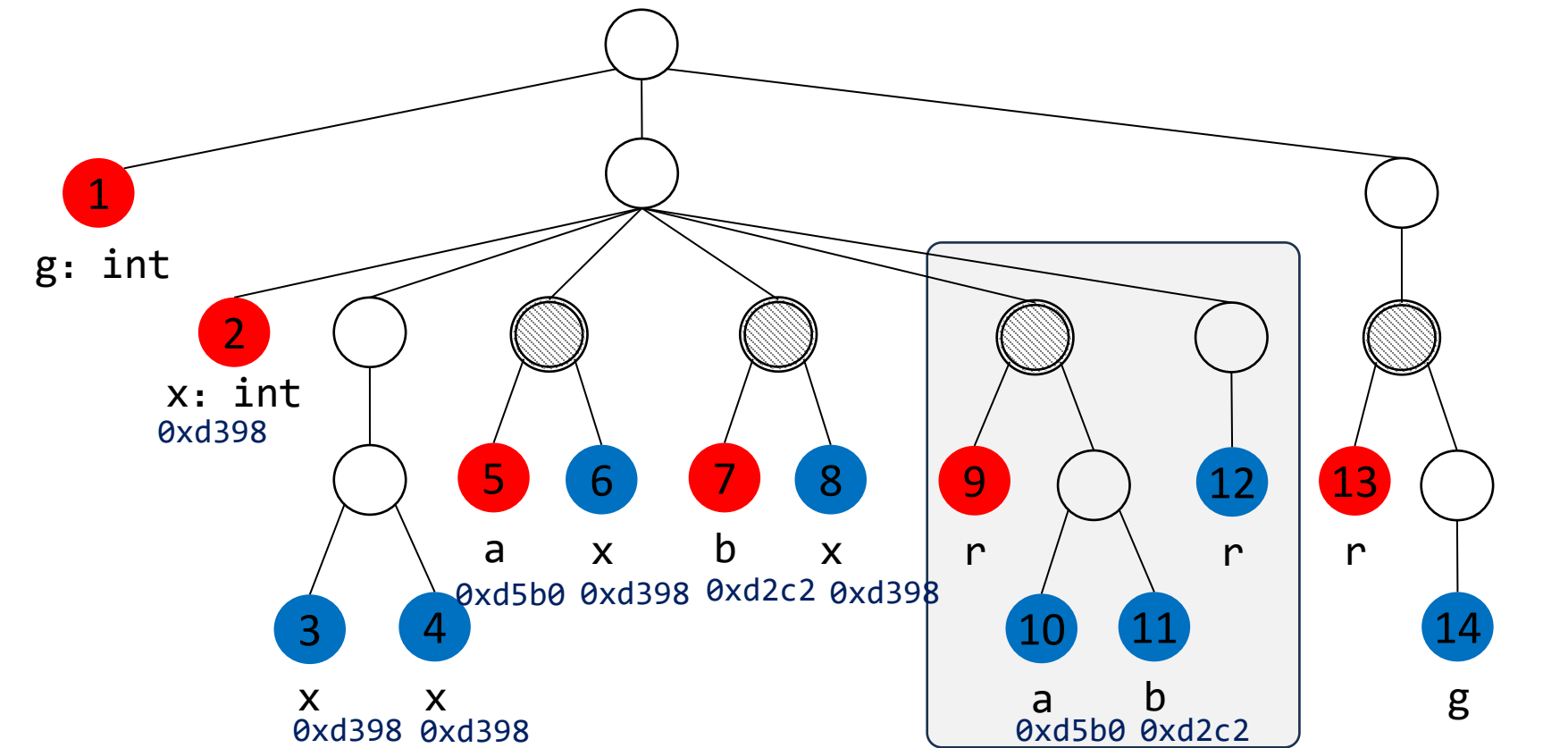
# 标识符索引化：a



标识符	作用域	索引	类型
x	fib	0xd398	int
a	fib:scope1	0xd5b0	
b			
r			

- 一般节点
- 变量声明
- 变量引用
- ⊘ 声明引用

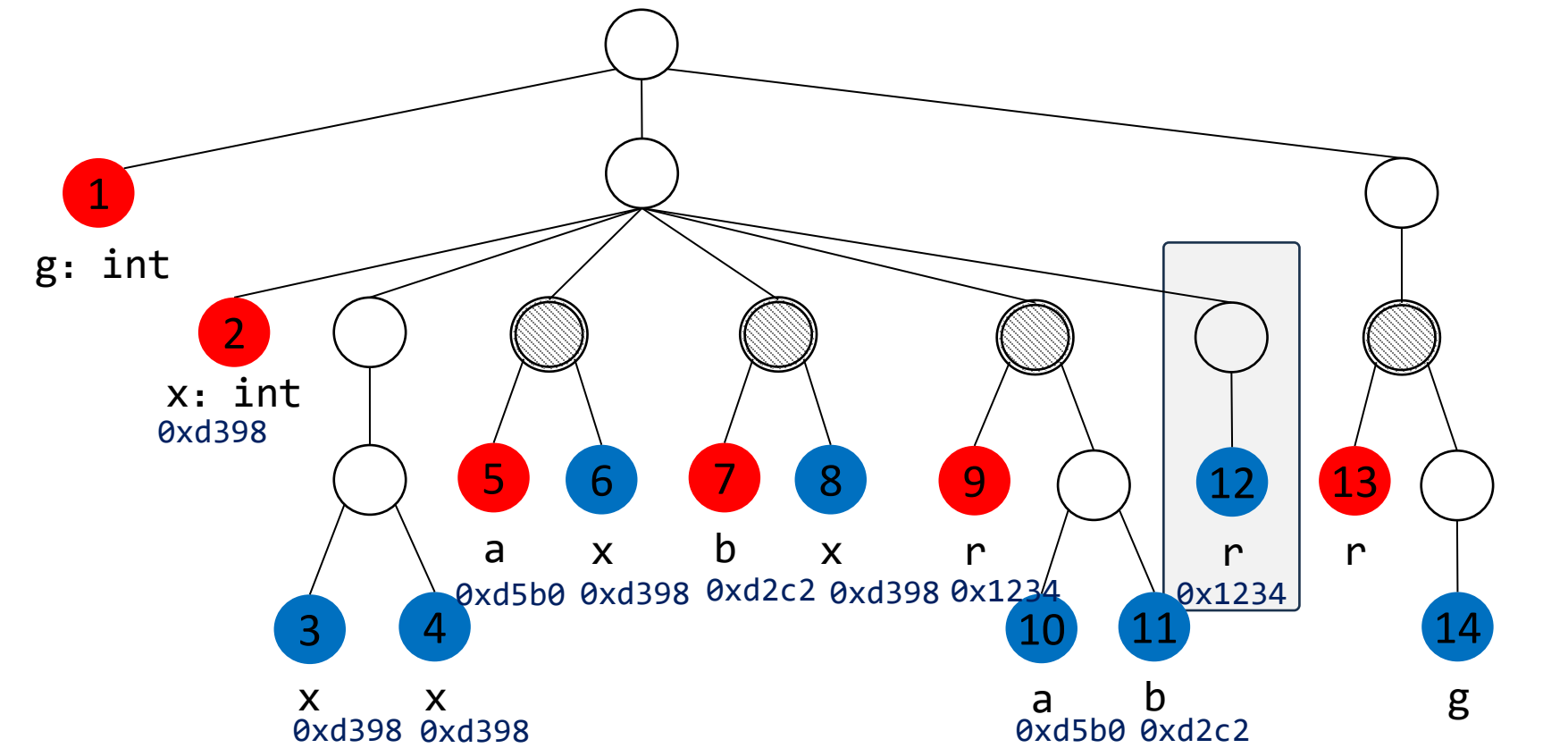
# 标识符索引化: b



标识符	作用域	索引	类型
x	fib	0xd398	int
a	fib:scope1	0xd5b0	
b	fib:scope2	0xd2c2	
r			

- 一般节点
- 变量声明
- 变量引用
- ⊘ 声明引用

# 标识符索引化: r



标识符	作用域	索引	类型
x	fib	0xd398	int
a	fib:scope1	0xd5b0	
b	fib:scope2	0xd2c2	
r	fib:scope3	0x1234	

- 一般节点
- 变量声明
- 变量引用
- ⦿ 声明引用

# 算法实现思路

- 动态维护两个哈希表记录标识符作用域和类型
  - 全局符号表
  - 局部符号表
- 遍历抽象语法树对标识符索引化
  - 标识符声明时将其加入符号表
  - 离开作用域时将其移出符号表

# 作用域导致的类型错误举例

```
fn foo(n: int) -> int {  
  while (n>0) {  
    ...  
    n = n-1;  
  }  
  ret x;  
}
```

错误：变量x未声明

当前使用变量不在符号表中

```
fn foo(n: int) -> int {  
  while (n>0) {  
    let x: int;  
    ...  
    n = n-1;  
  }  
  ret x;  
}
```

声明变量x，但作用域太小

错误：变量x未声明

# 作用域导致的类型检查问题

```
fn foo(n: int) -> int {  
  let x: int;  
  if (n>0) {  
    let x: int;  
    ...  
    x = n-1;  
  }  
  ret x;  
}
```

声明变量x

错误：重复声明

当前变量名已经在局部变量表中

```
fn foo(n: int) -> int {  
  if (n>0) {  
    let x: int;  
    ...  
  }  
  else {  
    let x: int;  
    ...  
  }  
}
```


正确：声明变量x

正确：声明变量x




# TeaPL中的全局变量使用要求

```
let x: int = a + 5;  
let a: int = 5;
```




全局变量声明和在全局中的使用顺序有关

```
fn foo(n: int) -> int {  
  x = x + n;  
  ret x;  
}  
let x: int = 0;
```



全局变量声明和在函数中的使用顺序无关

```
let a: int = 203;  
let b: int = 713;  
  
fn foo(a: int) -> int {  
  let b: int = 10;  
  ret a + b;  
}
```



局部变量不能和全局变量重名

# TeaPL中的函数重名问题：是否允许重载/多态？

```
fn foo(x: int);  
fn foo() -> int;
```



TeaPL暂时不允许重载

支持方法：在全局符号表中增添项目即可

标识符	作用域	索引	类型
foo	global	0xadc2	(int) → void
foo	global	0xbc70	(void) → int

### 三、类型约束和求解

---

# TeaPL的类型系统

- 基础类型
  - 标量类型：int、bool
  - 复合类型：数组
  - 函数类型
- 自定义类型：使用struct定义
- 规则（静态类型系统）
  - 类型推导：为代码中的每个标识符和表达式确定类型
  - 类型检查：分析每个参数类型是否符合运算符或函数签名要求
  - 类型转换：允许隐式类型转换？

# 类型错误举例

```
fn foo(n: int) -> int {  
    ...  
}  
fn main() {  
    foo(foo);  
}
```

参数类型错误

```
fn foo(n: int) -> int {  
    if (n <= 1) {  
        ret n;  
    }  
    let x:int = foo(n - 1);  
}
```

缺少返回语句

# 类型推导/检查思路

Damas–Hindley–Milner方法

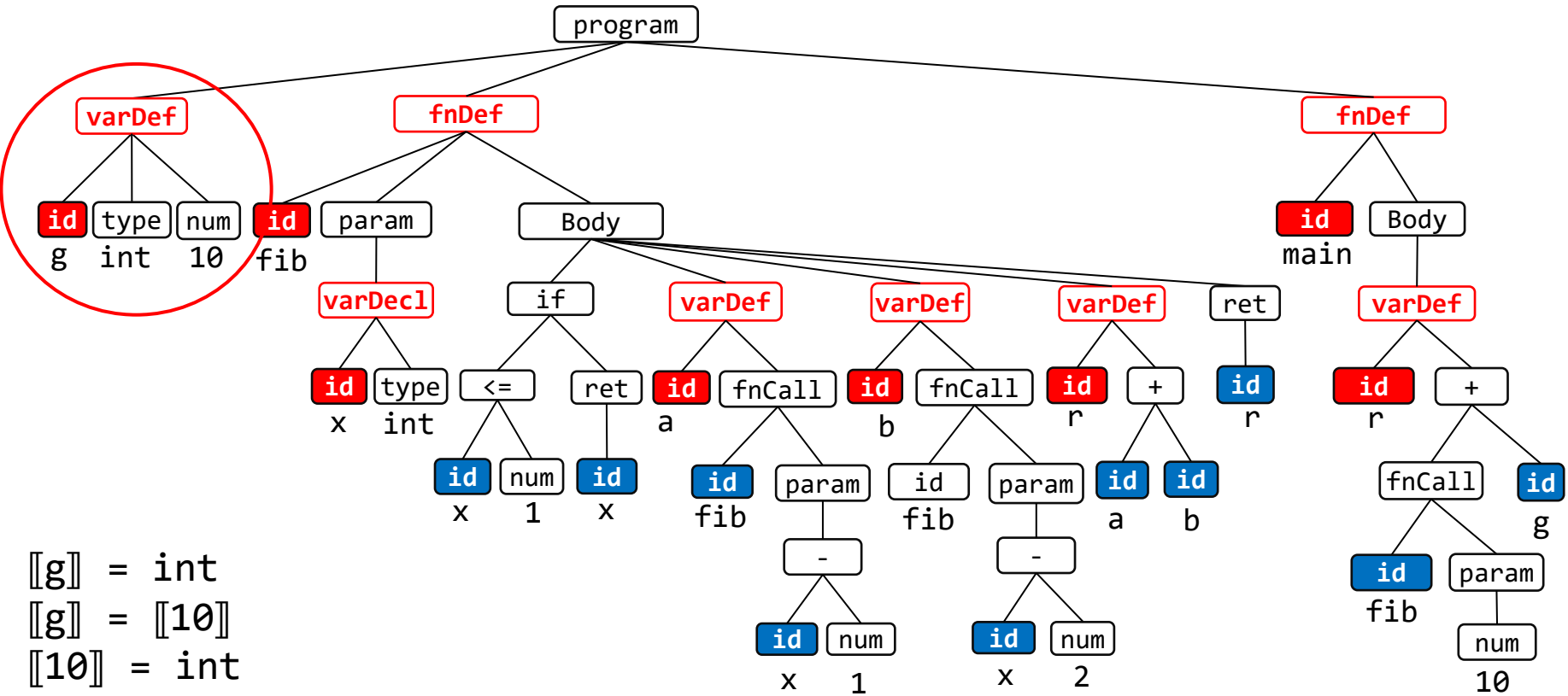
- 根据符号表确定变量类型
- 基于类型规则提取类型约束
  - 类型表示：用 $\llbracket X \rrbracket$ 表示变量 $X$ 的类型
  - 约束提取：一般都为等价关系，如果支持子类型和范型除外
- 约束求解

# 类型规则设计

- 为不同的语法制定相应的推断规则

代码示例	代码模式	约束
<code>a: int</code>	$X: Ty$	$\llbracket X \rrbracket = Ty$
<code>0</code>	$N$	$\llbracket N \rrbracket = \text{int}$
<code>a = b;</code>	$X = Y$	$\llbracket X \rrbracket = \llbracket Y \rrbracket$
<code>a + b;</code>	$X \text{ bop } Y$	$\llbracket X \rrbracket = \llbracket Y \rrbracket = \llbracket X+Y \rrbracket$
<code>c = a + b;</code>	$Z = X \text{ bop } Y$	$\llbracket X \rrbracket = \llbracket Y \rrbracket = \llbracket X+Y \rrbracket = \llbracket Z \rrbracket$

# 示例：约束提取



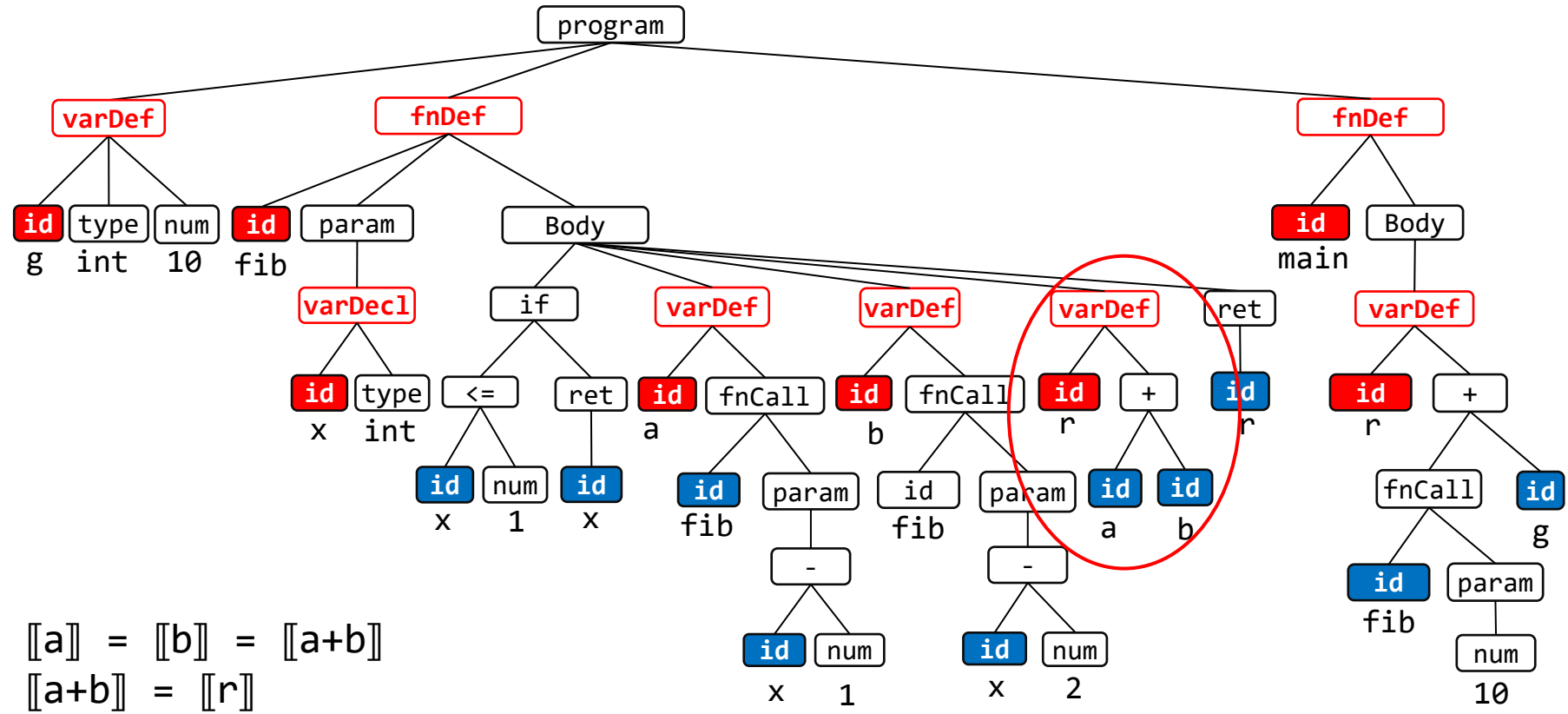
`[[g]] = int`  
`[[g]] = [[10]]`  
`[[10]] = int`

标识符	作用域	索引	类型
g	global	0xd9c2	int
fib	global	0xd470	(int) → int
main	global	0xd318	(void) → void

● 变量声明  
● 变量引用



# 示例：约束提取



标识符	作用域	索引	类型
x	fib	0xd398	int
a	fib:scope1	0xd5b0	
b	fib:scope2	0xd2c2	
r	fib:scope3	0x1234	

● 变量声明  
● 变量引用

# 更多类型规则

## 代码示例

```
a > b
```

```
a && b
```

```
if(a){...}
```

```
while(a){...}
```

## 代码模式

```
X rop Y
```

```
X lop Y
```

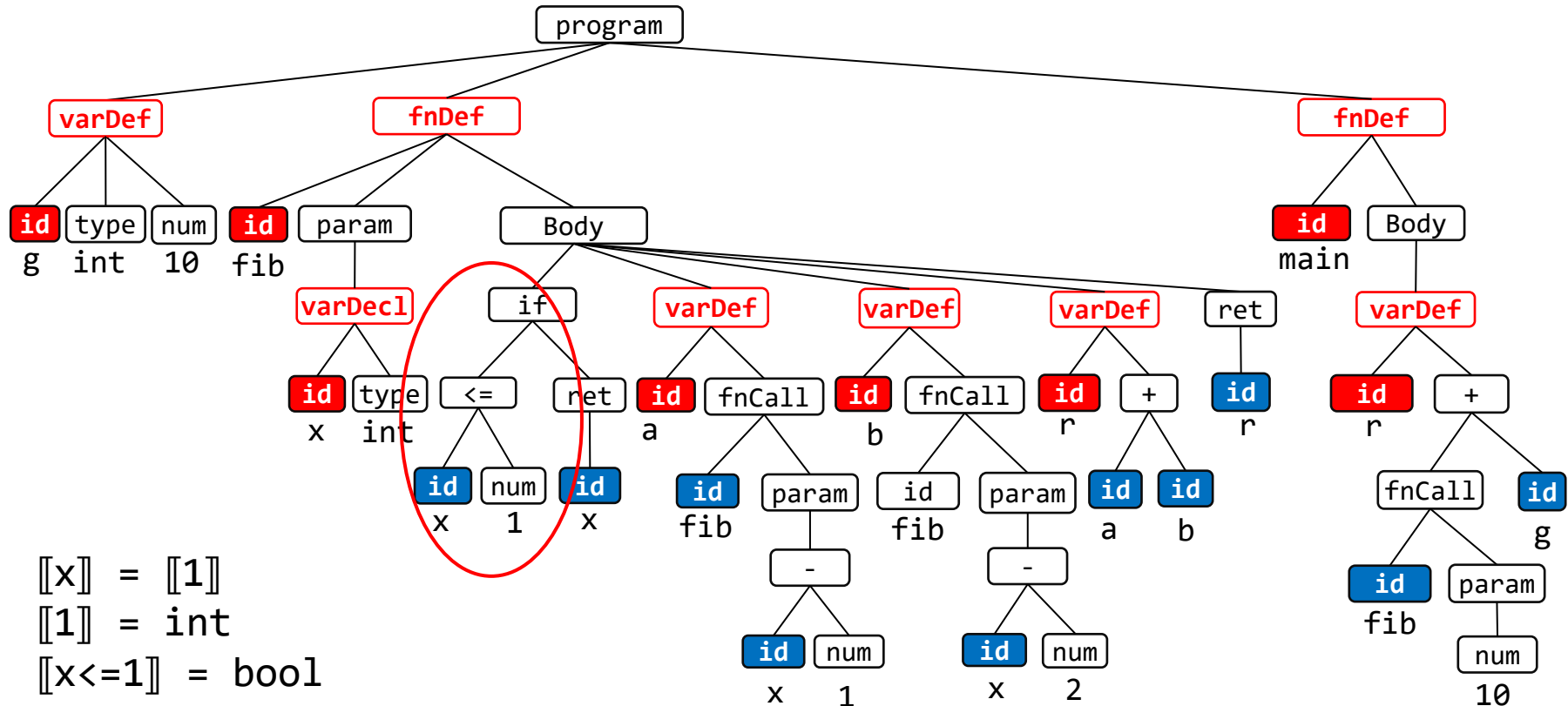
```
if(X)
```

```
while(X)
```

## 约束

$$\llbracket X \rrbracket = \llbracket Y \rrbracket, \llbracket X \text{ rop } Y \rrbracket = \text{bool}$$
$$\llbracket X \rrbracket = \llbracket Y \rrbracket = \llbracket X \text{ lop } Y \rrbracket = \text{bool}$$
$$\llbracket X \rrbracket = \text{bool}$$
$$\llbracket X \rrbracket = \text{bool}$$

## 示例：约束提取



标识符	作用域	索引	类型
x	fib	0xd398	int
a	fib:scope1	0xd5b0	
b	fib:scope2	0xd2c2	
r	fib:scope3	0x1234	

- 变量声明
- 变量引用

# 更多类型规则

代码示例

```
foo(a, b);
```

代码模式

$F(X, Y)$

约束

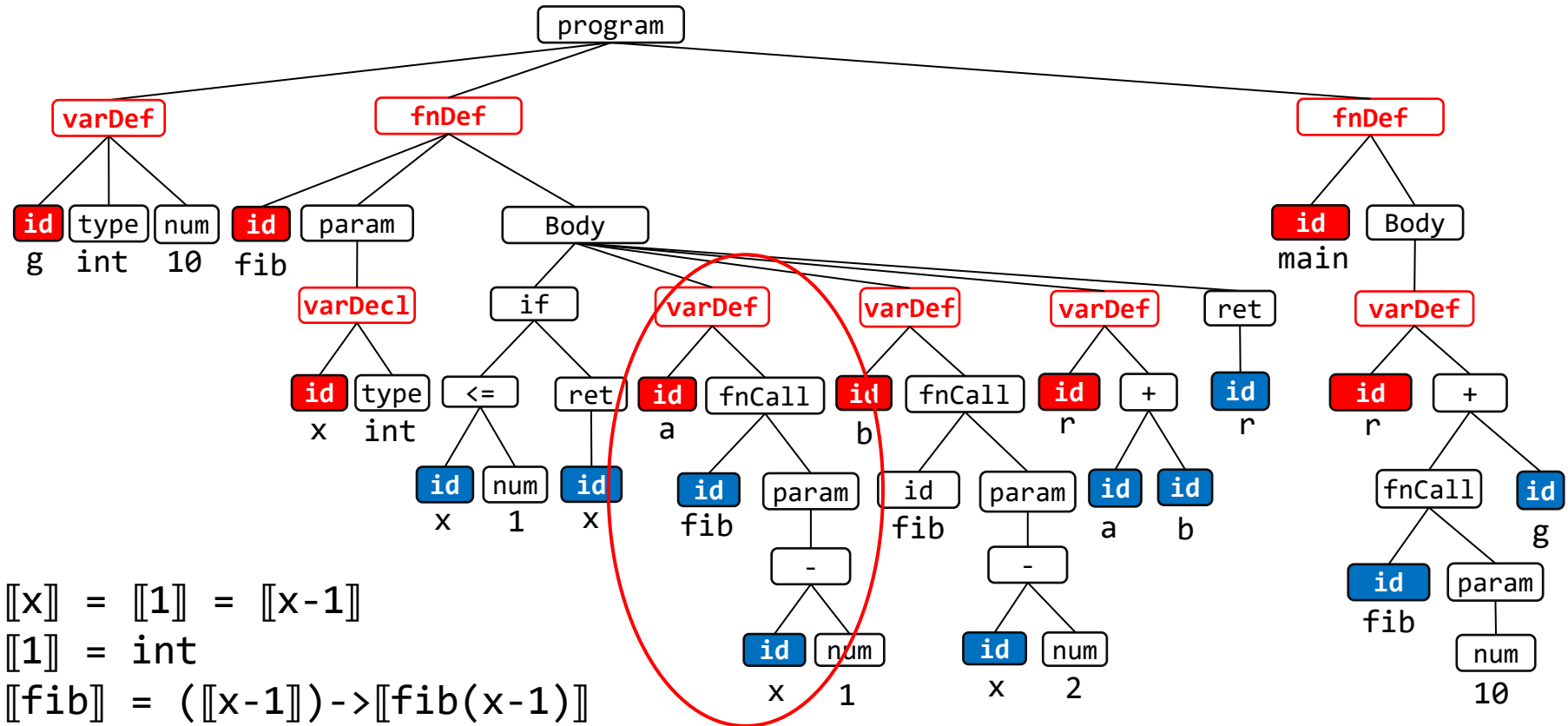
$\llbracket F \rrbracket = (\llbracket X \rrbracket, \llbracket Y \rrbracket) \rightarrow \llbracket F(X, Y) \rrbracket$

```
ret a;
```

$F(X) \rightarrow T_y \{$   
     $\dots$   
    ret  $Y$ ;  
 $\}$

$\llbracket F \rrbracket = (\llbracket X \rrbracket) \rightarrow T_y, \llbracket Y \rrbracket = T_y$

# 示例：约束提取

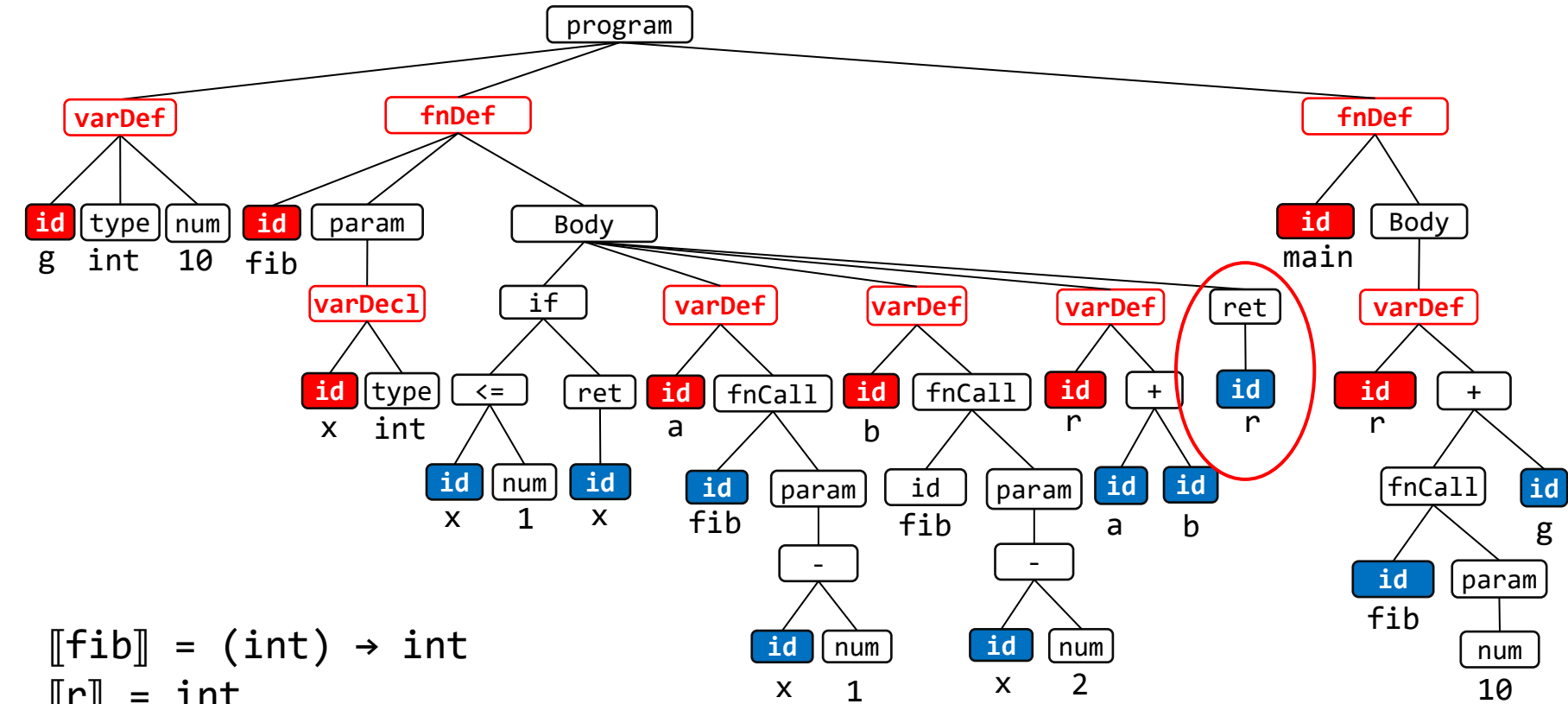


$\llbracket x \rrbracket = \llbracket 1 \rrbracket = \llbracket x-1 \rrbracket$   
 $\llbracket 1 \rrbracket = \text{int}$   
 $\llbracket \text{fib} \rrbracket = (\llbracket x-1 \rrbracket) \rightarrow \llbracket \text{fib}(x-1) \rrbracket$   
 $\llbracket a \rrbracket = \llbracket \text{fib}(x-1) \rrbracket$

标识符	作用域	索引	类型
g	global	0xd9c2	int
fib	global	0xd470	(int) → int
main	global	0xd318	(void) → void

● 变量声明  
● 变量引用

# 示例：约束提取



标识符	作用域	索引	类型
g	global	0xd9c2	int
fib	global	0xd470	(int) → int
main	global	0xd318	(void) → void

● 变量声明  
● 变量引用

# 约束提取结果

## 全局变量:

```
[[g]] = int  
[[g]] = [[10]]  
[[10]] = int
```

## main函数:

```
[[main]] = (void)->void
```

```
[[fib]] = ([[10]])->[[fib(10)]]  
[[fib(10)]] = [[g]] = [[r_main]]
```

## fib函数:

```
[[fib]] = ([[x]])->int  
[[r_fib]] = int  
[[x]] = int
```

```
[[x]] = [[1]]  
[[1]] = int  
[[x<=1]] = bool  
[[x]] = int
```

```
[[x]] = [[1]] = [[x-1]]  
[[1]] = int  
[[fib]] = ([[x-1]])->[[fib(x-1)]]  
[[a]] = [[fib(x-1)]]
```

```
[[x]] = [[2]] = [[x-2]]  
[[2]] = int  
[[fib]] = ([[x-2]])->[[fib(x-2)]]  
[[b]] = [[fib(x-2)]]
```

```
[[a]] = [[b]] = [[a+b]] = [[r_fib]]
```

# 基于并查集方法求解

- 维护不存在相交关系的集合，支持查找和联合两种操
  - Find(x): 返回包含变量x的集合
  - Union(x, y): 联合包含x和y的两个集合

```
while(getPair()!=NULL){  
    [p,q] = readPair(p,q);  
    pset = find(p);  
    qset = find(q);  
    if(pset == qset)  
        continue;  
    else union(p,q);  
}
```



# 应用并查集方法求解

```
[[g]] = int  
[[g]] = [[10]]  
[[10]] = int
```

S1:{int, [[10]], [[g]]}

```
[[fib]] = ([[x]])->int  
[[r_fib]] = int  
[[x]] = int
```

S1:{int, [[10]], [[g]], [[x]], [[r\_fib]]}

S2:{{[[fib]], ([[x]])->int, }}

```
[[x]] = [[1]]  
[[1]] = int  
[[x<=1]] = bool  
[[x]] = int
```

S1:{int, [[10]], [[1]], [[g]], [[x]], [[r\_fib]]}

S3:{bool, [[x<=1]]}

```
[[x]] = [[1]] = [[x-1]]  
[[1]] = int  
[[fib]] = ([[x-1]])->[[fib(x-1)]]  
[[a]] = [[fib(x-1)]]
```

S1:{int, [[10]], [[1]], [[g]], [[x]], [[r\_fib]],  
[[x-1]]}

S2:{{[[fib]], ([[x]])->int, ([[x-1]])->[[fib(x-1)]] }}

S4:{{[[a]], [[fib(x-1)]]}

# 应用并查集方法求解

$\llbracket x \rrbracket = \llbracket 2 \rrbracket = \llbracket x-2 \rrbracket$   
 $\llbracket 2 \rrbracket = \text{int}$   
 $\llbracket \text{fib} \rrbracket = (\llbracket x-2 \rrbracket) \rightarrow \llbracket \text{fib}(x-2) \rrbracket$   
 $\llbracket b \rrbracket = \llbracket \text{fib}(x-2) \rrbracket$

$S1: \{\text{int}, \llbracket 10 \rrbracket, \llbracket 1 \rrbracket, \llbracket 2 \rrbracket, \llbracket g \rrbracket, \llbracket x \rrbracket, \llbracket r\_fib \rrbracket, \llbracket x-1 \rrbracket, \llbracket x-2 \rrbracket\}$

$S2: \{\llbracket \text{fib} \rrbracket, (\llbracket x \rrbracket) \rightarrow \text{int}, (\llbracket x-1 \rrbracket) \rightarrow \llbracket \text{fib}(x-1) \rrbracket, (\llbracket x-2 \rrbracket) \rightarrow \llbracket \text{fib}(x-2) \rrbracket\}$

$S4: \{\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket \text{fib}(x-1) \rrbracket, \llbracket \text{fib}(x-2) \rrbracket\}$

$\llbracket a \rrbracket = \llbracket b \rrbracket = \llbracket a+b \rrbracket = \llbracket r\_fib \rrbracket$

$S1 + S4:$   
 $\{\text{int}, \llbracket 10 \rrbracket, \llbracket 1 \rrbracket, \llbracket 2 \rrbracket, \llbracket g \rrbracket, \llbracket x \rrbracket, \llbracket r\_fib \rrbracket, \llbracket x-1 \rrbracket, \llbracket x-2 \rrbracket, \llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket \text{fib}(x-1) \rrbracket, \llbracket \text{fib}(x-2) \rrbracket\}$

$\llbracket \text{main} \rrbracket = (\text{void}) \rightarrow \text{void}$

$S5: \{(\text{void}) \rightarrow \text{void}, \llbracket \text{main} \rrbracket\}$

$\llbracket \text{fib} \rrbracket = (\llbracket 10 \rrbracket) \rightarrow \llbracket \text{fib}(10) \rrbracket$   
 $\llbracket \text{fib}(10) \rrbracket = \llbracket g \rrbracket = \llbracket r\_main \rrbracket$

$S2: \{\llbracket \text{fib} \rrbracket, (\llbracket x \rrbracket) \rightarrow \text{int}, (\llbracket x-1 \rrbracket) \rightarrow \llbracket \text{fib}(x-1) \rrbracket, (\llbracket x-2 \rrbracket) \rightarrow \llbracket \text{fib}(x-2) \rrbracket, (\llbracket 10 \rrbracket) \rightarrow \llbracket \text{fib}(10) \rrbracket\}$

$S1 + S4:$   
 $\{\text{int}, \llbracket 10 \rrbracket, \llbracket 1 \rrbracket, \llbracket 2 \rrbracket, \llbracket g \rrbracket, \llbracket x \rrbracket, \llbracket r\_fib \rrbracket, \llbracket r\_main \rrbracket, \llbracket x-1 \rrbracket, \llbracket x-2 \rrbracket, \llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket \text{fib}(10) \rrbracket, \llbracket \text{fib}(x-1) \rrbracket, \llbracket \text{fib}(x-2) \rrbracket\}$

# 最终解

S1-4:{int, [[10]], [[1]], [[2]], [[g]], [[x]], [[r\_fib]], [[r\_main]], [[x-1]], [[x-2]],  
[[a]], [[b]], [[fib(10)]], [[fib(x-1)]], [[fib(x-2)]]}

S2:{{[[fib]], ([[x]])->int, ([[x-1]])->[[fib(x-1)]], ([[x-2]])->[[fib(x-2)]],  
([[10]])->[[fib(10)]]} }

S3:{bool, [[x<=1]]}

S5:{(void)->void, [[main]]}



[[g]] = int

[[a]] = int

[[b]] = int

[[r\_fib]] = int

[[r\_main]] = int

[[x<=1]] = bool

[[fib]] = (int) → int

[[main]] = (void)->void

[[fib(T3)]] = int

# 更多类型规则：数组

代码示例

```
{0, 1}
```

```
{0; 1}
```

```
let a[10]: int
```

```
b = a[i];
```

```
a[i] = b;
```

代码模式

```
{M, N}
```

```
{M; N}
```

```
X[I]: Ty
```

```
Y = X[Z]
```

```
X[Z] = Y
```

约束

```
[[{M, N}]] = &int
```

```
[[{M; N}]] = &int
```

```
[[X]] = &Ty
```

```
[[Z]] = int, [[Y]] = [[*X]], [[X]] = &[[*X]]
```

```
[[Z]] = int, [[X]] = &[[Y]]
```

# 更多类型规则：结构体

## 代码示例

```
struct Foo {  
    a: int,  
    b: int,  
}
```

```
foo.a = d;
```

## 代码模式

```
struct ST {  
    A: Ty1,  
    B: Ty2,  
}
```

```
X.A = Y
```

## 约束

$$\llbracket ST \rrbracket = \llbracket A, B \rrbracket = (Ty1, Ty2)$$
$$\llbracket X.A \rrbracket = \llbracket Y \rrbracket, \llbracket X.A, \_ \rrbracket = \llbracket X \rrbracket$$

# 类型推断可能存在的问题

- 解不唯一的情况：优先选择哪些类型？
- 无解的情况：是否允许隐式类型转换？
- 如何判断类型是否等价：名字相同 vs 结构相同

```
struct Pos { x:int, y:int, }  
struct Loc { x:int, y:int, }
```

# 递归问题

// TeaPL 代码

```
fn fac(n: int) -> int {  
    if (n == 0) {  
        ret 1;  
    } else {  
        let r = n * fac(n-1);  
        ret r;  
    }  
}
```

$\llbracket \text{fac} \rrbracket = (\text{int}) \rightarrow \text{int}$



// TeaPL 代码

```
struct List {  
    v: int,  
    next List,  
}
```



$\llbracket \text{List} \rrbracket = \phi = (\text{int}, \phi)$

// C 代码

```
struct List {  
    int data;  
    struct List* next;  
};
```



$\llbracket \text{List} \rrbracket = \phi = (\text{int}, \&\phi)$   
 $= (\text{int}, \text{usize})$

# 练习：类型检查

- 应用类型检查方法分析实验中的测试用例
- 链接： [https://github.com/hxuhack/compiler\\_project/blob/24f-assignment2/src/tests](https://github.com/hxuhack/compiler_project/blob/24f-assignment2/src/tests)



# 思考

- 假如TeaPL函数声明可缺省类型，设计方法分析下列代码中的类型信息

```
fn f(n) {  
    if (n == 0) {  
        ret 1;  
    } else {  
        let r = n * f(n-1);  
        ret r;  
    }  
}
```

```
fn f(f, n) {  
    if (n == 0) {  
        ret 1;  
    } else {  
        let r = n * f(f, n-1);  
        ret r;  
    }  
}
```