

Lecture 3.2

树形IR和类型检查

徐 辉

xuh@fudan.edu.cn



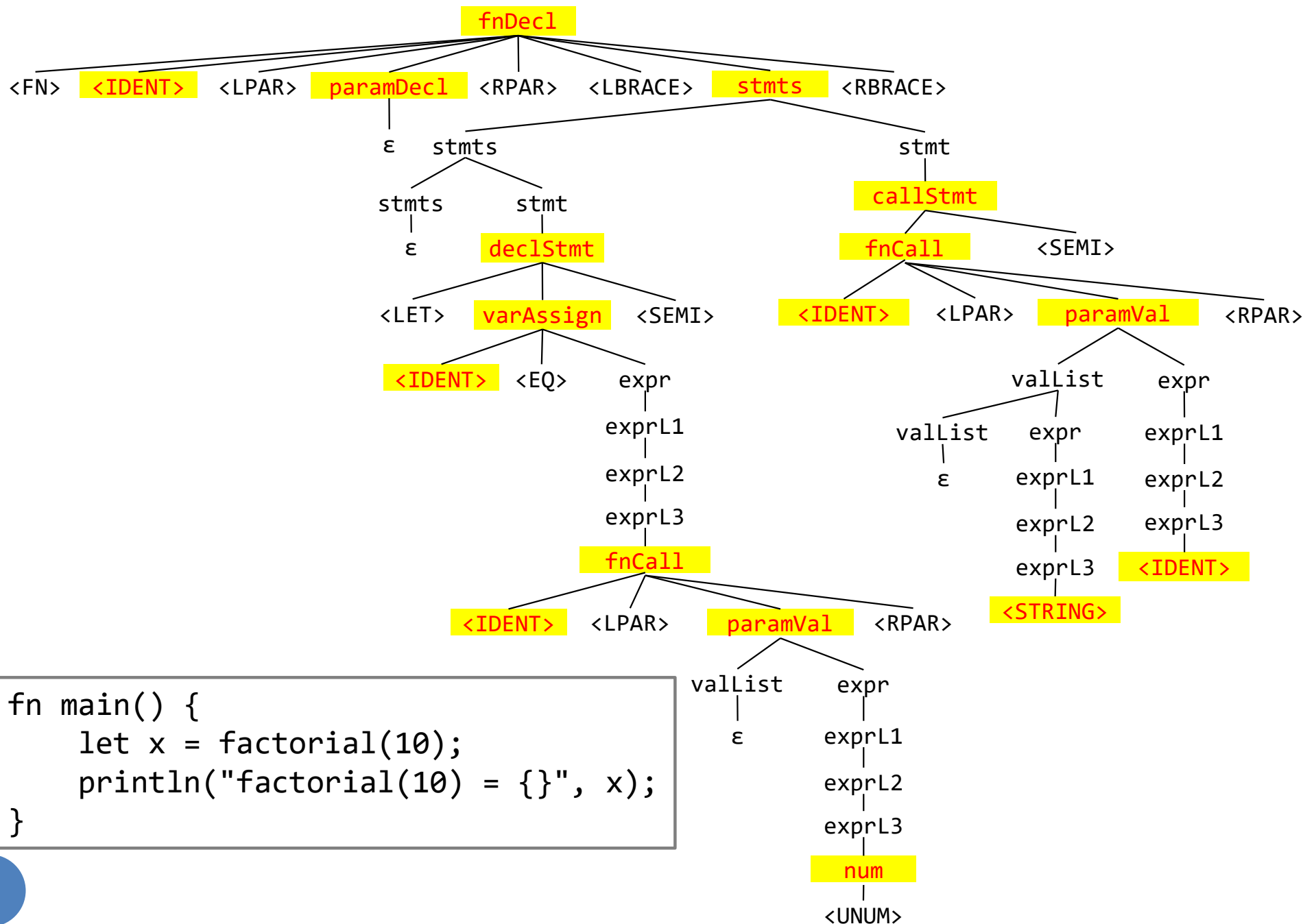
大纲

- 一、回顾和展望
- 二、抽象语法树
- 三、类型系统
- 四、类型检查

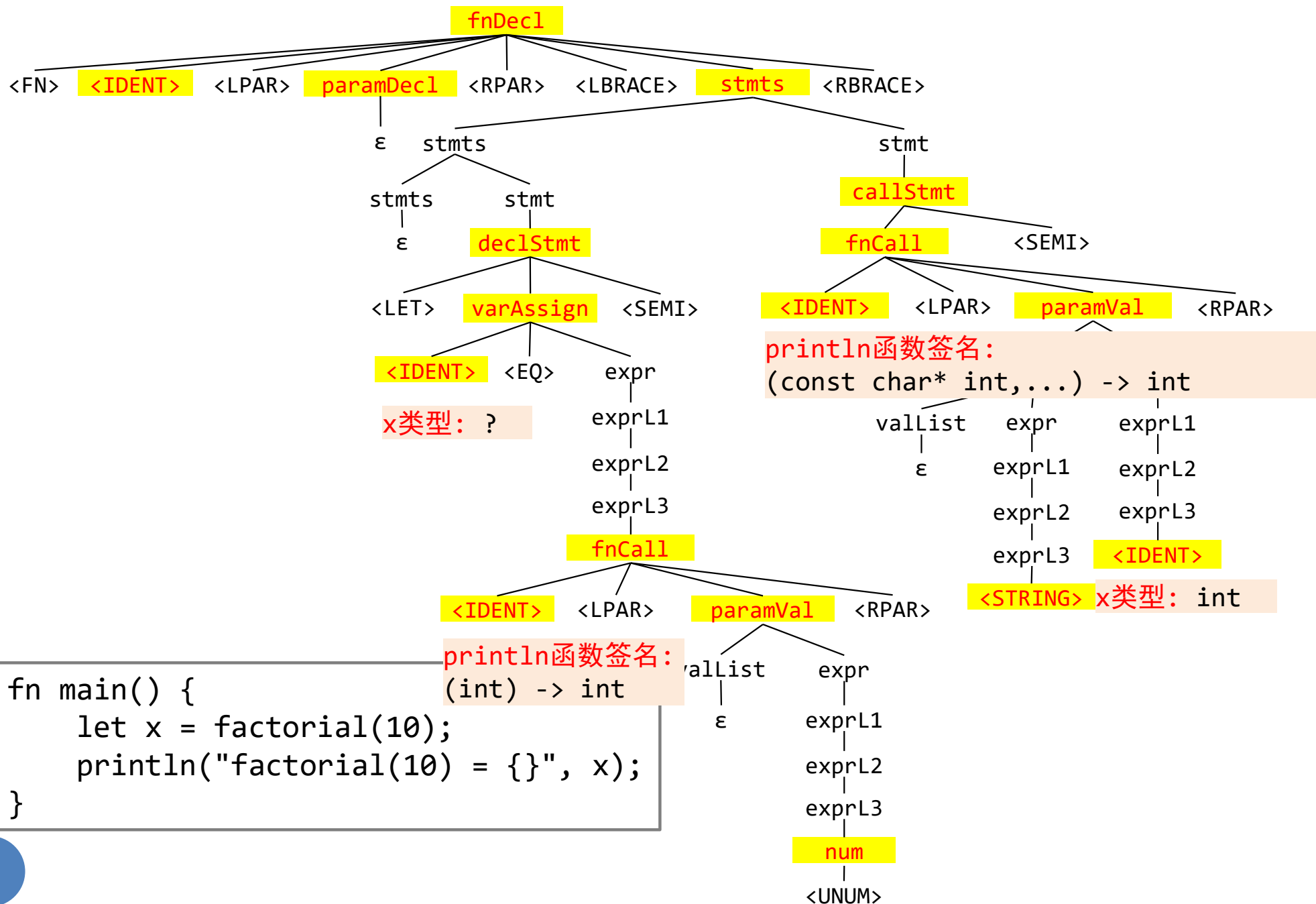
回顾：

- 前端解决了哪些问题？
 - 准确理解句子，生成语法解析树
- 语法解析树主要问题
 - 冗余信息较多
 - 可解析的程序未必正确

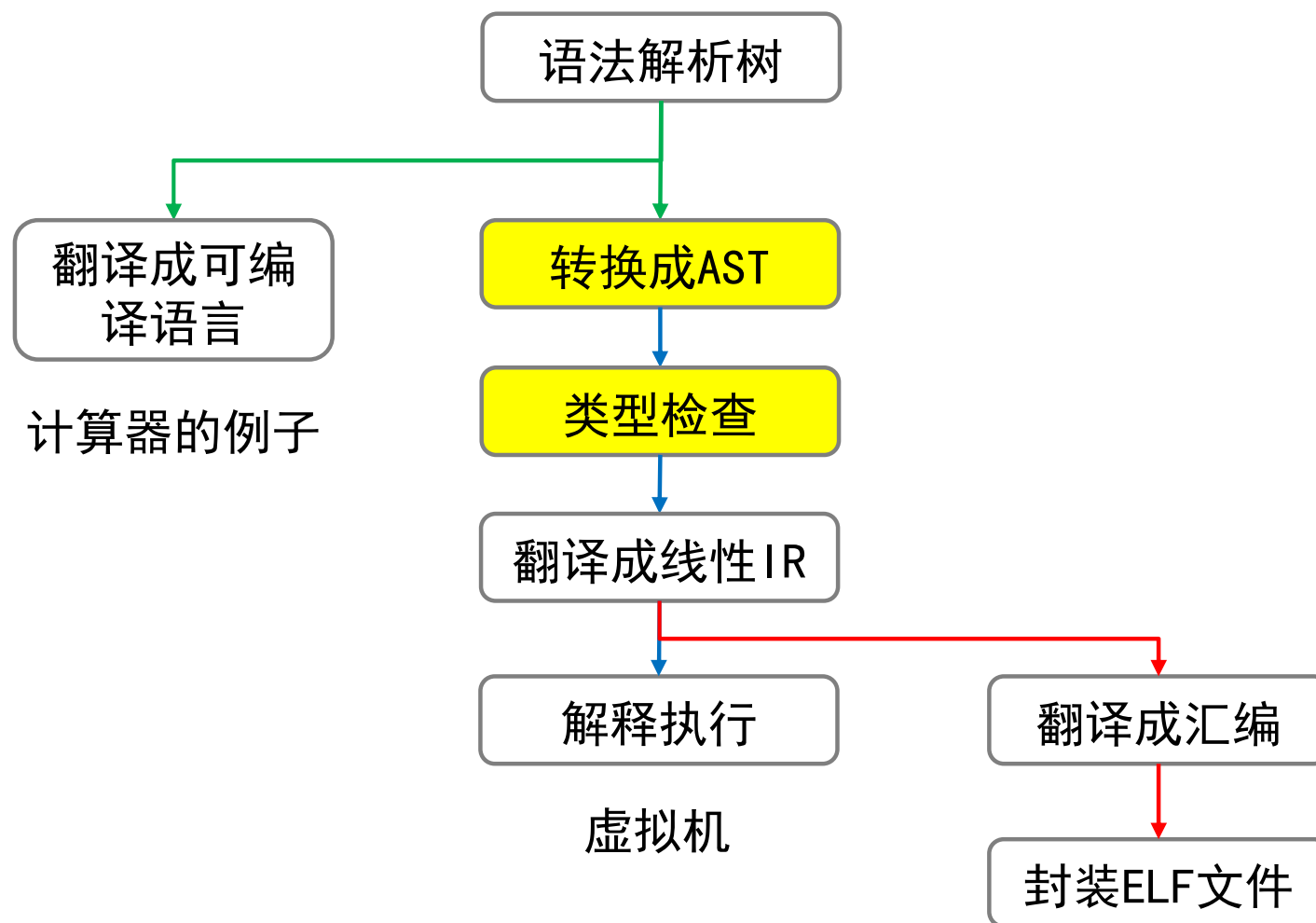
语法解析树冗余节点较多



类型问题



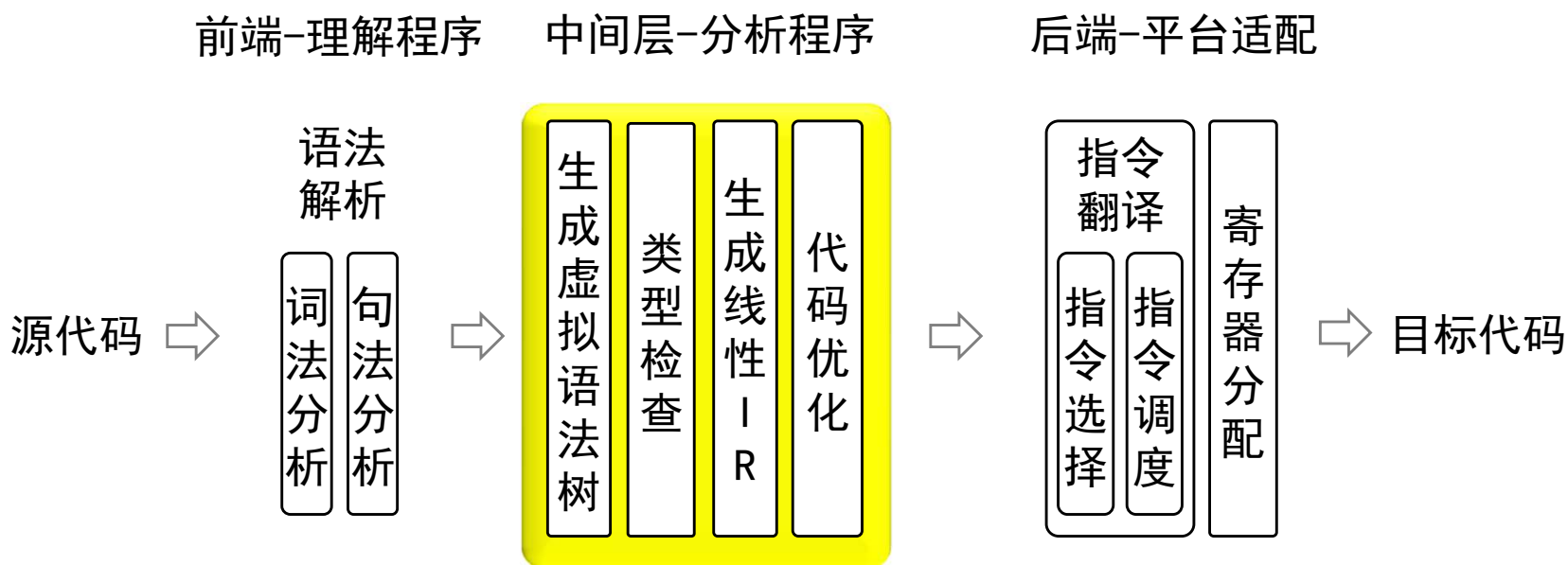
展望



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，中间代码格式相同
- 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、回顾和展望
- 二、抽象语法树
- 三、类型系统
- 四、类型检查

抽象语法树

- Concrete Syntax: 程序员实际写的代码
 - 解析源代码得到语法解析树，是对源代码的完整表示。
- Abstract Syntax: 编译器实际需要的内容
- 抽象语法树：消除推导过程中的一些步骤或节点
 - 运算符和关键字一般不是叶子结点
 - 单一展开形式塌陷，如 $E \rightarrow T \rightarrow F \rightarrow \text{digit}$
 - 去掉括号等冗余信息
- 可以被编译器后续编辑，记录上下文相关的信息

抽象语法树构造的两种思路

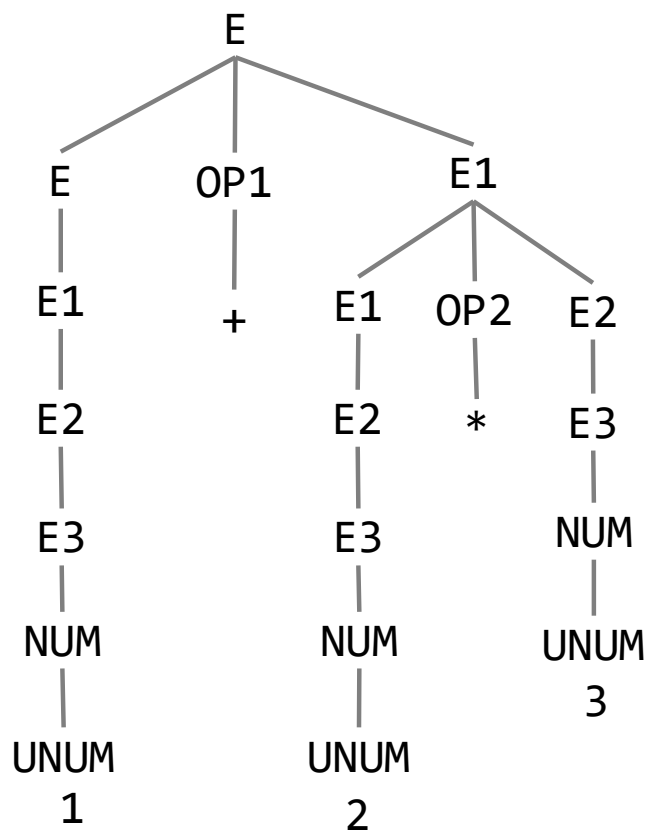
- 语法制导（经典方法），跳过语法解析树直接构造
- 遍历语法解析树

语法制导：Syntax-Directed Translation

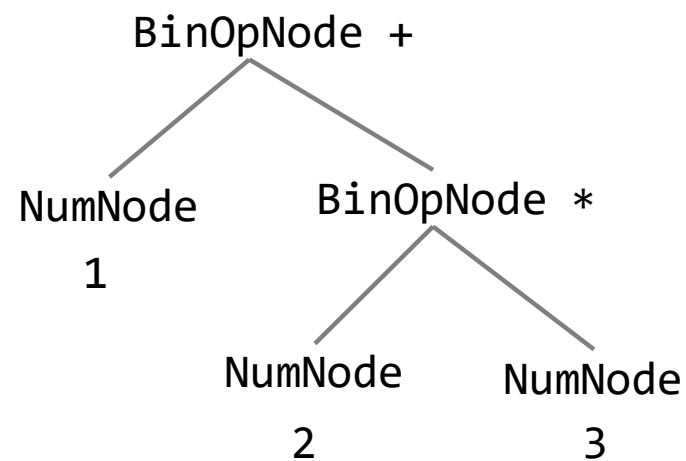
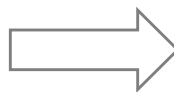
- 为每条上下文无关文法设置对应的属性文法
 - 使用Bison编写计算器的例子

```
input
: E { *expression = $1; }
E
: E OP1 E1 { $$ = createBinOpNode($1, $2, $3); }
| E1      { $$ = $1; }
E1
: E1 OP2 E2 { $$ = createBinOpNode($1, $2, $3); }
| E2      { $$ = $1; }
E2
: E3 OP3 E2 { $$ = createBinOpNode($1, $2, $3); }
| E3      { $$ = $1; }
E3
: NUM      { $$ = $1; }
| "(" E ")" { $$ = $2; }
NUM
: UNUM      { $$ = createNumNode($1); }
| "-" UNUM  { $$ = createNumNode(0-$2); }
OP1
: "+"      { $$ = setOp(ADD); }
| "-"      { $$ = setOp(SUB); }
OP2
: "*"      { $$ = setOp(MUL); }
| "/"      { $$ = setOp(DIV); }
OP3
: "^"      { $$ = setOp(EXP); }
```

树状结构举例



语法解析树



抽象语法树

依赖关系分析: S-atttributed SDD

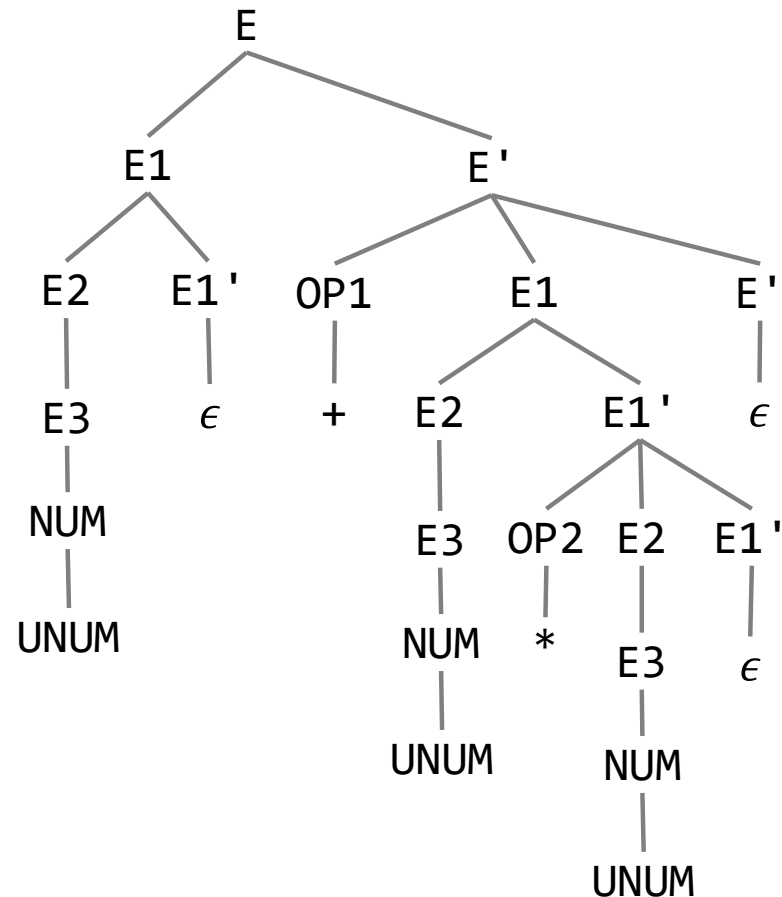
- 合成属性 (Synthesized Attribute) 节点的属性是根据其子节点的属性定义的
- S-atttributed SDD: 所有节点的属性语法都是合成属性

```
input
: E { *expression = $1; }
E
: E OP1 E1 { $$ = createBinOpNode($1, $2, $3); }
| E1      { $$ = $1; }
E1
: E1 OP2 E2 { $$ = createBinOpNode($1, $2, $3); }
| E2      { $$ = $1; }
E2
: E3 OP3 E2 { $$ = createBinOpNode($1, $2, $3); }
| E3      { $$ = $1; }
E3
: NUM      { $$ = $1; }
| "(" E ")" { $$ = $2; }
NUM
: UNUM      { $$ = createNumNode($1); }
| "-" UNUM  { $$ = createNumNode(0-$2); }
OP1
: "+"      { $$ = setOp(ADD); }
| "-"      { $$ = setOp(SUB); }
OP2
: "*"      { $$ = setOp(MUL); }
| "/"      { $$ = setOp(DIV); }
OP3
: "^"      { $$ = setOp(EXP); }
```

LL (1) 文法的依赖关系分析

- 如何编写属性文法？

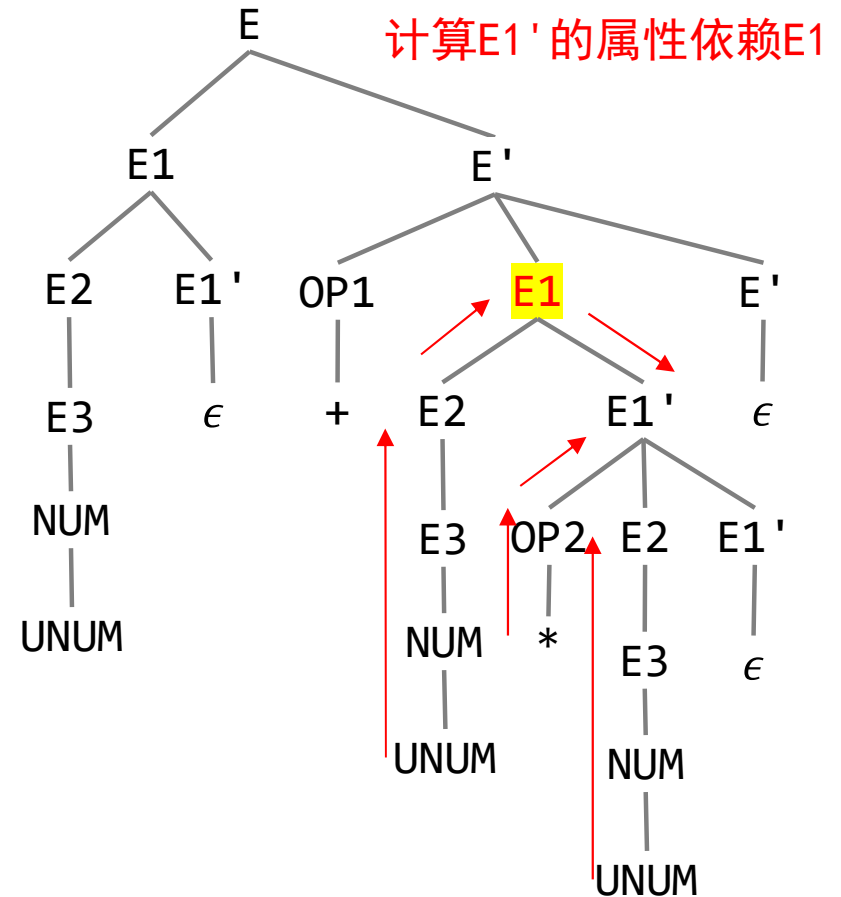
```
[1] E → E1 E'  
[2] E' → OP1 E1 E'  
[3]      | ε  
[4] E1 → E2 E1'  
[5] E1' → OP2 E2 E1'  
[6]      | ε  
[7] E2 → E3 OP3 E2  
[8]      | E3  
[9] E3 → NUM  
[10]      | <LPAR> E <RPAR>  
[11] NUM → <UNUM>  
[12]      | <SUB> <UNUM>  
[13] OP1 → <ADD>  
[14]      | <SUB>  
[15] OP2 → <MUL>  
[16]      | <DIV>  
[17] OP3 → <EXP>
```



语法解析树：1+2*3

L-Attributed SDD

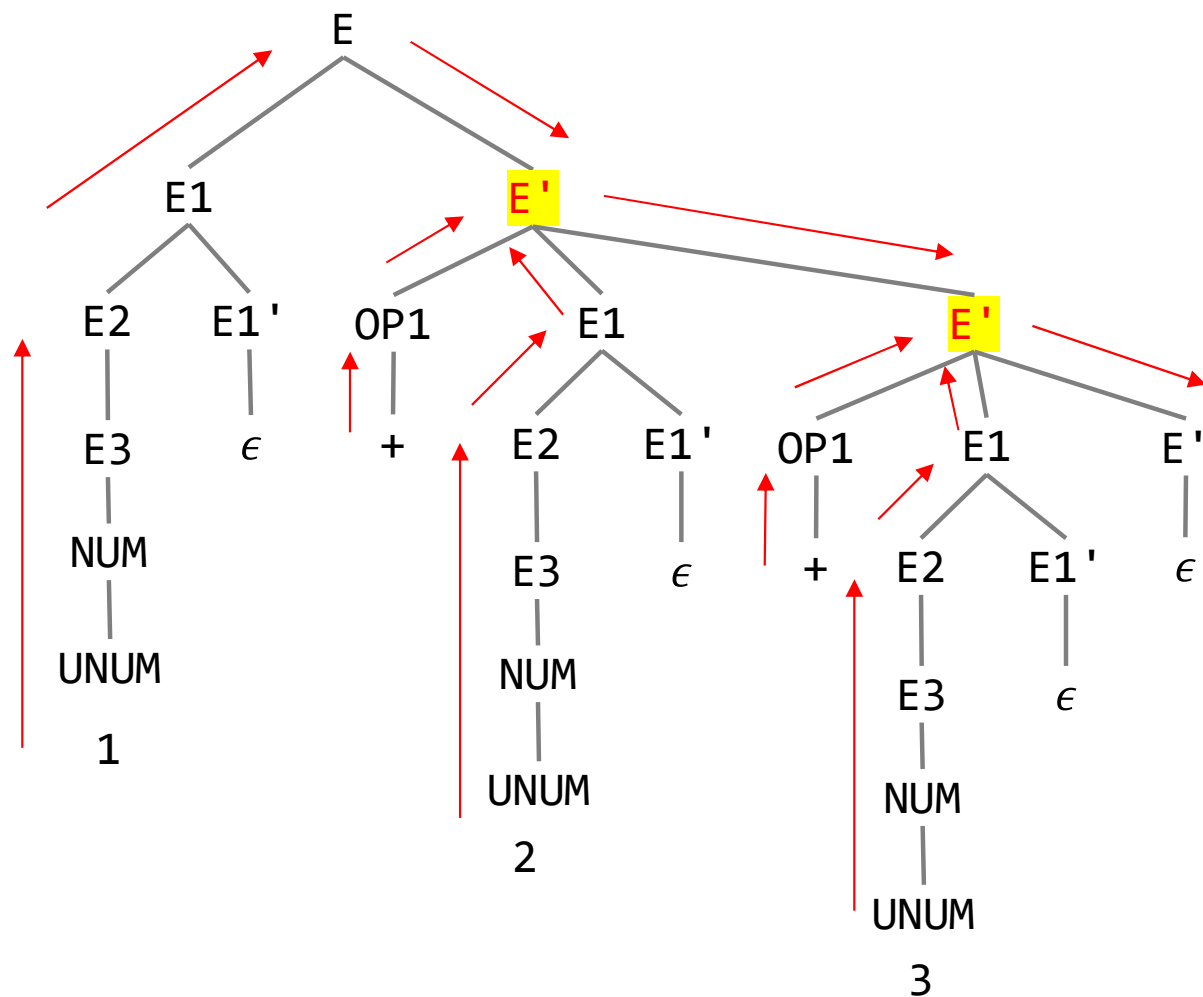
- 继承属性：节点 β_2 的属性是根据其父节点 ($A \rightarrow \beta_1\beta_2\beta_3$) 的生成式规则确定的
 - 属性计算依赖其父节点A，或兄弟节点 β_1 、 β_3
- L-Attributed SDD：都是合成属性，或对于 $A \rightarrow \beta_1.. \beta_i.. \beta_n$ 中的任意 β_i 来说，其继承属性只依赖A或 $\beta_1, \dots, \beta_{i-1}$



语法解析树：1+2*3

LL (1) 的属性文法可读性差

```
[1] E → E1 E'
[2] E' → OP1 E1 E'
[3]     | ε
[4] E1 → E2 E1'
[5] E1' → OP2 E2 E1'
[6]     | ε
[7] E2 → E3 OP3 E2
[8]     | E3
[9] E3 → NUM
[10]    | <LPAR> E <RPAR>
[11] NUM → <UNUM>
[12]    | <SUB> <UNUM>
[13] OP1 → <ADD>
[14]     | <SUB>
[15] OP2 → <MUL>
[16]     | <DIV>
[17] OP3 → <EXP>
```



大纲

- 一、回顾和展望
- 二、抽象语法树
- 三、类型系统
- 四、类型检查



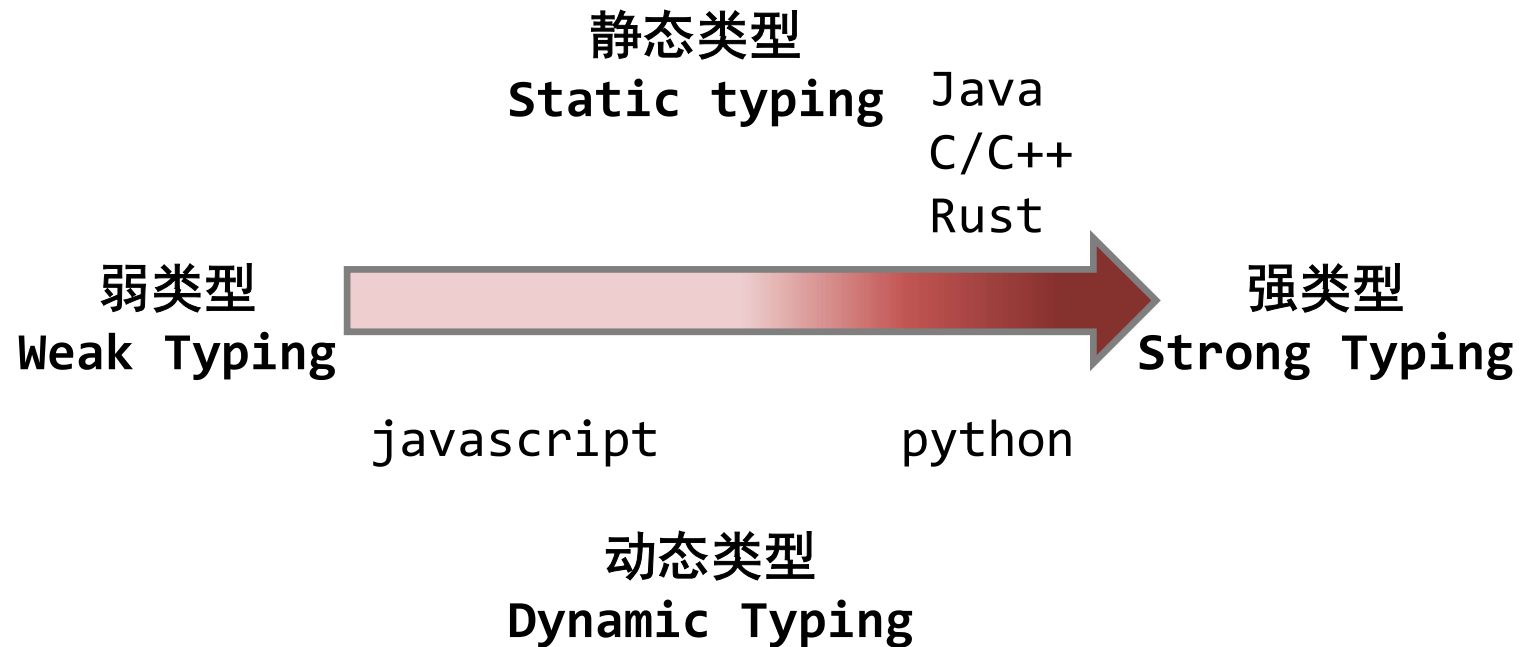
基本概念

- 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)
 - 复合类型 (Compound Type)
 - 自定义类型
- 类型规则：
 - 类型推断和检查规则
 - 是否允许隐式类型转换？

类型定义

- 标量类型
 - bool
 - char
 - int
 - long
 - float
 - double
 - const str
- 复合类型
 - 数组: array
 - 元组: tuple
- 自定义类型
 - struct
- 函数类型
 - fn

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，避免运行时错误
- 动态类型系统：运行时检查类型的一致性，一般不用显式定义变量类型

```
//python代码，foo的类型是什么？  
def foo(x):  
    if x == 1:  
        return "bingo!"  
    return x  
  
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'
```

强类型语言

- 优先确定变量和值的类型，一般不允许隐式类型转换
- 代表语言：Python、Java
 - C/C++?

//python代码

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

TypeError

2

TypeError

//C代码

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

51

4202501

2

50

4202503

弱类型语言

- 类型会发生隐式转换，可能会造成意想不到的错误
- 代表语言：JavaScript

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

*ToString(number) + string
number + ToNumber(boolean)
string + ToString(boolean)*

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

*false
true
true*

```
if (a == 1 && a == 2) {  
    alert('A bug!');  
}
```

```
var i = 1;  
Number.prototype.valueOf = function() {  
    return i++;  
};
```

```
var a = new Number(1);
```

Javascript Equality Game

- <https://eqeq.js.org/>

How well do you know `==` in JavaScript?

[Restart](#) [Help](#)

Language
English

About

Test your mettle against what's considered a textbook example of a confusing language design flaw – JavaScript's loose equality operator.

Flag all cells where the values are loosely equal according to `==`. The cells that are strictly equal are already revealed.

The table is diagonally symmetrical, so only one side needs to be flagged.

Wrong guesses count against the final score:

$$\text{wrongness} = \frac{\text{flags} - \text{hits} + \text{misses}}{\text{max hits}}$$

Score
▶ 000 ✓ 000 ✗ 000

Results
Pending... 🤔
[Show Results](#) [Twitter](#)

| | true | false | 1 | 0 | -1 | "true" | "false" | "1" | "0" | "-1" | " " | null | undefined | Infinity | -Infinity | [] | {} | [[[]]] | [0] | [1] | NaN |
|-----------|------|-------|---|---|----|--------|---------|-----|-----|------|-----|------|-----------|----------|-----------|----|----|--------|-----|-----|-----|
| true | ✓ | | | | | | | | | | | | | | | | | | | | |
| false | | ✓ | | | | | | | | | | | | | | | | | | | |
| 1 | | | ✓ | | | | | | | | | | | | | | | | | | |
| 0 | | | | ✓ | | | | | | | | | | | | | | | | | |
| -1 | | | | | ✓ | | | | | | | | | | | | | | | | |
| "true" | | | | | | ✓ | | | | | | | | | | | | | | | |
| "false" | | | | | | | ✓ | | | | | | | | | | | | | | |
| "1" | | | | | | | | ✓ | | | | | | | | | | | | | |
| "0" | | | | | | | | | ✓ | | | | | | | | | | | | |
| "-1" | | | | | | | | | | ✓ | | | | | | | | | | | |
| " " | | | | | | | | | | | ✓ | | | | | | | | | | |
| null | | | | | | | | | | | | ✓ | | | | | | | | | |
| undefined | | | | | | | | | | | | | ✓ | | | | | | | | |
| Infinity | | | | | | | | | | | | | | ✓ | | | | | | | |
| -Infinity | | | | | | | | | | | | | | | ✓ | | | | | | |
| [] | | | | | | | | | | | | | | | | | | | | | |
| {} | | | | | | | | | | | | | | | | | | | | | |
| [[[]]] | | | | | | | | | | | | | | | | | | | | | |
| [0] | | | | | | | | | | | | | | | | | | | | | |
| [1] | | | | | | | | | | | | | | | | | | | | | |
| NaN | | | | | | | | | | | | | | | | | | | | | |

大纲

- 一、回顾和展望
- 二、抽象语法树
- 三、类型系统
- 四、类型检查

如何实现类型安全？

- 类型推断（type inference）：为代码中的每个标识符和表达式确定类型
 - 如果有解，则说明可类型（typeable）
- 类型检查（type checking）：分析每一个参数类型是否与运算符或函数签名要求一致
 - 需要提前确定变量类型
- 类型转换（type coercion）：如果类型检查不通过则判断是否可进行隐式转换

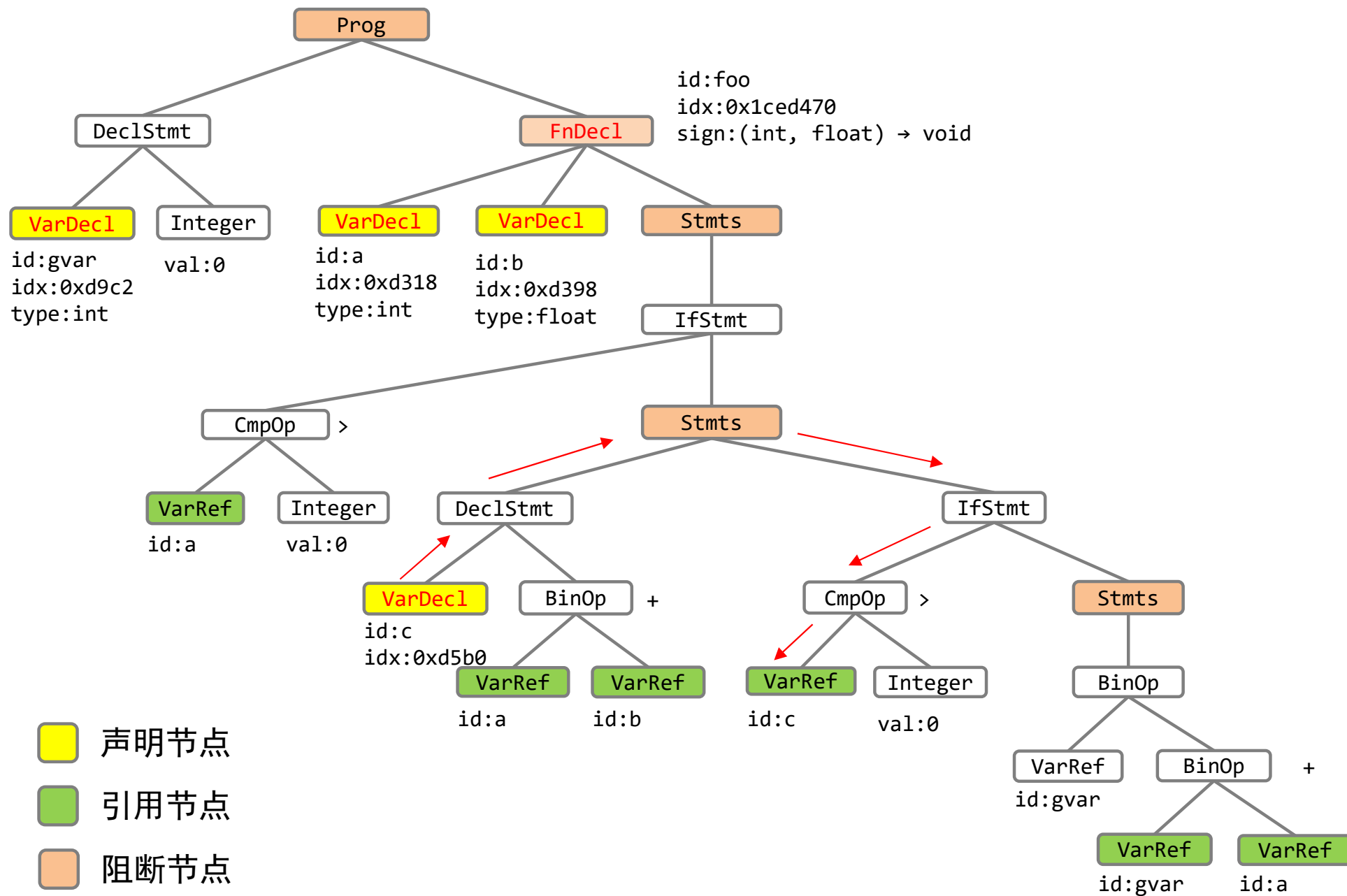
确定标识符的作用域

- 在AST上对标识符做区分：
 - 声明新标识符：确定作用域
 - 使用标识符

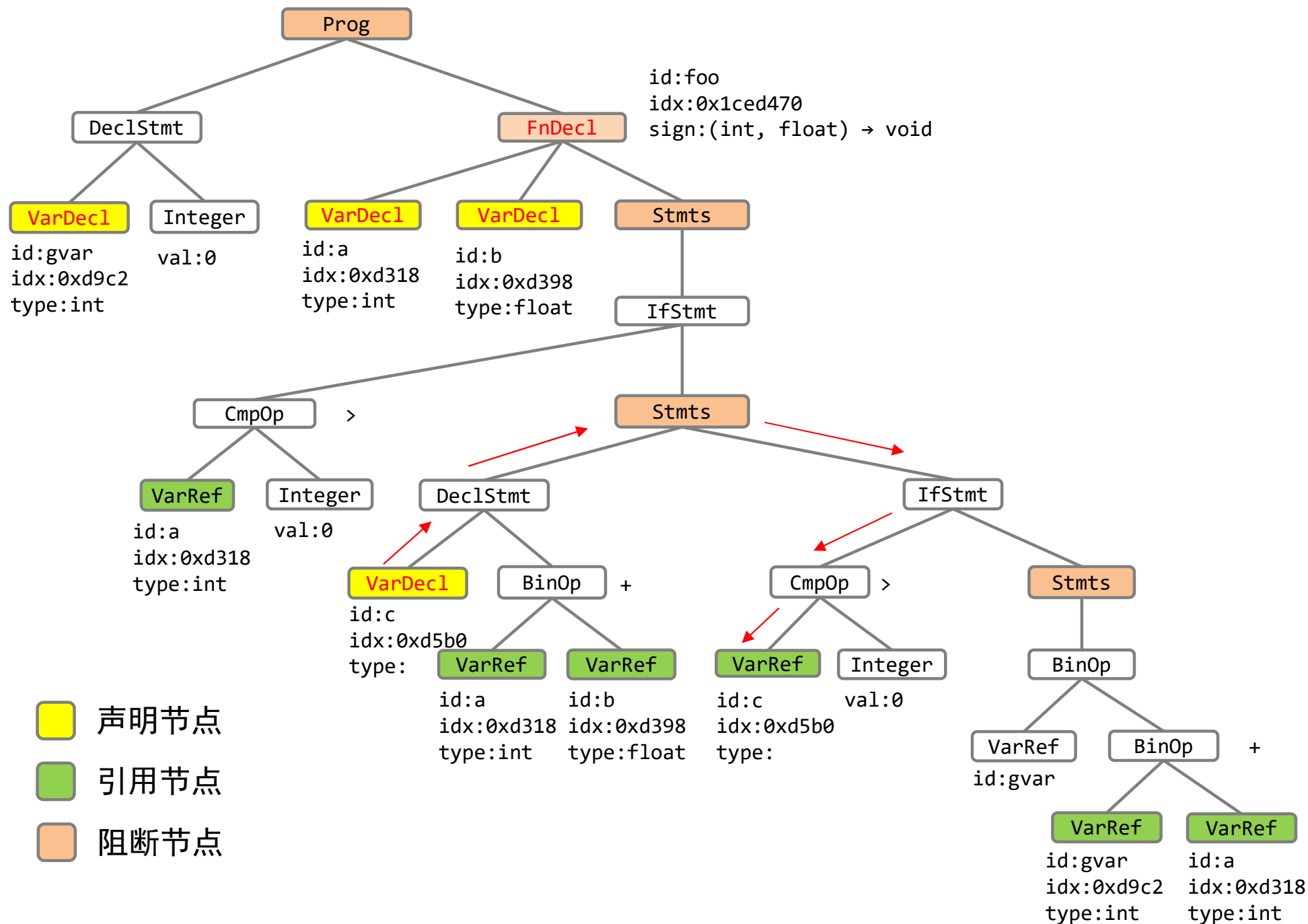
```
let gvar:int = 0;
fn foo(int a, float b){
    if(a > 0) {
        let c = a + b;
        if(c > 0){
            gvar = gvar + a;
        }
    }
}
```

| 标识符 | 作用域 | 索引 | 类型 |
|------|-------------|--------|--------------------|
| gvar | global | 0xd9c2 | int |
| foo | global | 0xd470 | (int,float) → void |
| a | foo | 0xd318 | int |
| b | foo | 0xd398 | float |
| c | foo:if-true | 0xd5b0 | |
| | | | |

遍历AST确定标识符作用域



遍历AST确定标识符作用域



如何推断类型？

- 为不同的语法制定相应的推断规则

```
let a = 0;  
let a = 0.0;  
let a = '1';  
let a = "1";
```

0的类型是int => a的类型是int
0.0的类型是float => a的类型是float
'1'的类型是char => a的类型是char
"1"的类型是str => a的类型是str

```
let a;  
let b:int = 0;  
a = &b;  
a = b + 1;  
a = b + 1.0;
```

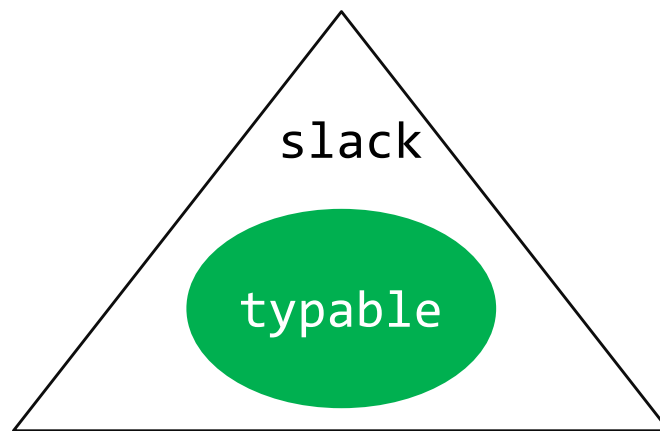
b的类型是int => &int
+ (int int)-> int => a的类型是int
+ (int float)-> float => a的类型是float

```
let a;  
if (a) {  
}
```

if条件类型为boolean => a的类型是？

类型推断

- Damas-Hindley-Milner类型推断方法
 - 基于约束求解的方法；
 - ML、Haskell、Ocaml等语言中使用
- 使用保守的推断策略
 - 根据抽象语法树获得类型约束；
 - 如果可类型，则不应出现运行时错误；
 - 有些程序可能被错误拒绝 (slack/false positive)



类型约束

- 基于AST生成类型约束：
 - 约束一般都为等价关系
 - 通过合一算法 (unification algorithm) 求解
- 类型变量：
 - 变量X的类型变量用 $[[X]]$ 表示
 - 所有的变量标识符都是唯一的
 - 表达式E（非标识符的）的类型变量用 $[[E]]$ 表示
 - E为一个AST节点

类型约束生成规则举例

程序语句 (AST节点)

NUM

E1 op E2

V = E

E1 == E2

if(E){S}

if(E){S1}else{S2}

while(E){S}

F(E1,...,En)

&V

*E

*V = E

V = *E

类型约束

$\llbracket \text{NUM} \rrbracket = \text{i32/u32/float/double}$

$\llbracket E1 \rrbracket = \llbracket E2 \rrbracket = \llbracket E1 \text{ op } E2 \rrbracket$

$\llbracket V \rrbracket = \llbracket E \rrbracket$

$\llbracket E1 \rrbracket = \llbracket E2 \rrbracket, \llbracket E1 == E2 \rrbracket = \text{bool}$

$\llbracket E \rrbracket = \text{bool}$

$\llbracket E \rrbracket = \text{bool}$

$\llbracket E \rrbracket = \text{bool}$

$\llbracket F \rrbracket = (\llbracket E1 \rrbracket \dots \llbracket En \rrbracket) \rightarrow \llbracket E(E1 \dots En) \rrbracket$

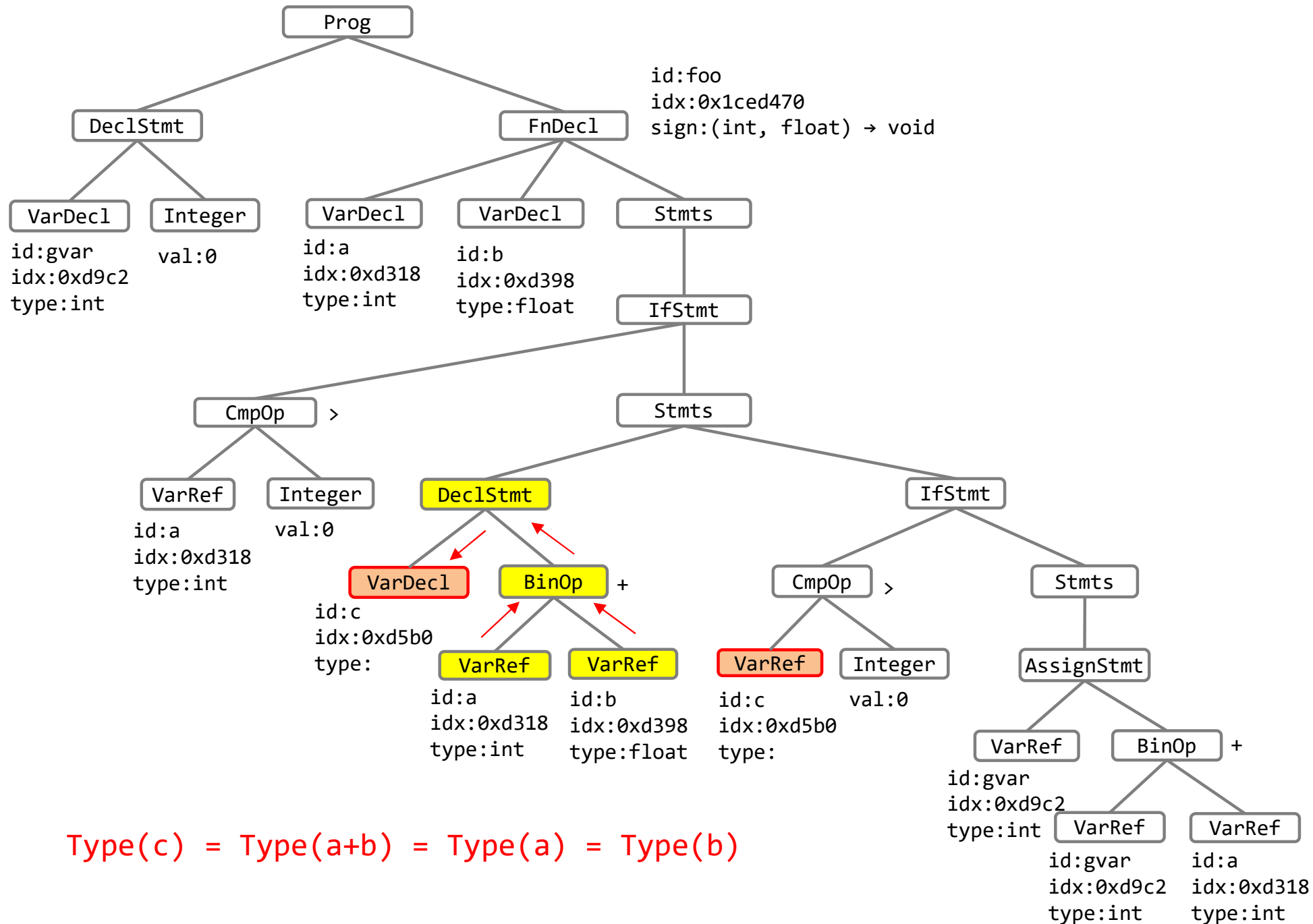
$\llbracket \&V \rrbracket = \&\llbracket V \rrbracket$

$\llbracket E \rrbracket = \&\llbracket *E \rrbracket$

$\llbracket V \rrbracket = \&\llbracket E \rrbracket$

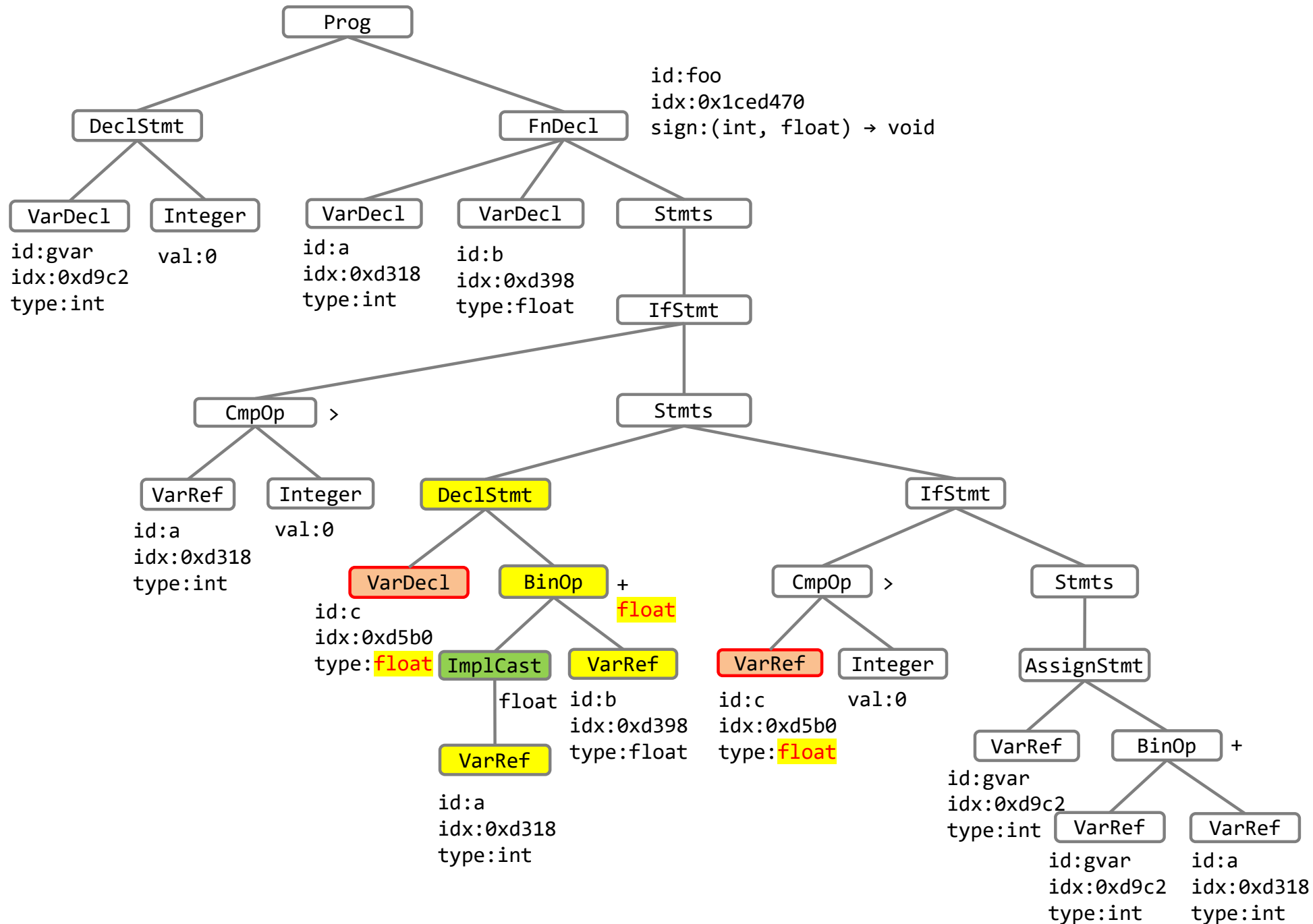
$\llbracket V \rrbracket = \llbracket *E \rrbracket, \llbracket E \rrbracket = \&\llbracket *E \rrbracket$

类型推断



$$\text{Type}(c) = \text{Type}(a+b) = \text{Type}(a) = \text{Type}(b)$$

不匹配的如何处理？隐式转换？



可能存在多个可行解

- 假设允许函数类型推断（python的例子）

程序语句（AST节点）

```
F(P1, ..., Pn){  
    ...  
    return E;  
}  
F(E1, ..., En)
```

类型约束

$$\llbracket F \rrbracket = (\llbracket P1 \rrbracket \dots \llbracket Pn \rrbracket) \rightarrow \llbracket E \rrbracket$$
$$\llbracket F \rrbracket = (\llbracket E1 \rrbracket \dots \llbracket En \rrbracket) \rightarrow \llbracket F(E1 \dots En) \rrbracket$$

```
foo(x) {  
    return *x;  
}
```

$$\begin{aligned}\llbracket \text{foo} \rrbracket &= (\llbracket x \rrbracket) \rightarrow \llbracket *x \rrbracket \\ \llbracket x \rrbracket &= \&\llbracket *x \rrbracket\end{aligned}$$

解约束：

$$\begin{aligned}\llbracket x \rrbracket &= \text{int} \\ &\quad | \ \&\text{int} \\ &\quad | \ \&\&\text{int} \\ &\quad \dots\end{aligned}$$

```
foo(x,y) {  
    *x = y;  
}
```

$$\begin{aligned}\llbracket \text{foo} \rrbracket &= (\llbracket x \rrbracket \llbracket y \rrbracket) \rightarrow \text{void} \\ \llbracket x \rrbracket &= \&\llbracket y \rrbracket\end{aligned}$$
$$\begin{aligned}\llbracket x \rrbracket &= \&\text{int} \\ \llbracket y \rrbracket &= \text{int} \\ &\dots\end{aligned}$$

递归问题（可解）

```
factorial(n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}  
factorial(10);
```

约束:

$[[\text{factorial}]] = ([n]) \rightarrow [1]$

$[[\text{factorial}]] = ([n]) \rightarrow [(n * \text{factorial}(n-1))]$

$[n] = [0]$

$[n] = [(\text{factorial } (n-1))] = [(n * (\text{factorial}(n-1)))]$

$[n] = [1] = [n-1]$

解约束:

$[n] = \text{int}$

$[[\text{factorial}]] = (\text{int}) \rightarrow \text{int}$

递归问题（不可解）

```
factorial(f, n) {  
  if (n == 0)  
    return 1;  
  else  
    return n * f(f, n-1);  
}  
factorial(10, factorial);
```

$[[\text{factorial}]] = ([f], [n]) \rightarrow [1] = ([f], [n]) \rightarrow [n * f(f, n-1)]$

$[n] = [0] = [n == 0] = \text{int}$

$[n] = [f(f, n-1)] = [n * f(f, n-1)] = \text{int}$

$[x] = ([f], [n-1]) \rightarrow [f(f, n-1)]$

$[n] = [1] = [n-1] = \text{int}$

$[[\text{factorial}]] = ([10], [\text{factorial}]) \rightarrow [\text{factorial}(10, \text{factorial})]$

使用 ϕ 来标记Regular Type

$[[\text{factorial}]] = \phi = (\text{int}, \phi) \rightarrow \text{int}$

结构体递归定义

```
//Java代码  
class List<T> {  
    T value;  
    List<T> next;  
}
```

$[[\text{List}<T>]] = \phi = (T, \phi)$

```
//C代码  
struct List{  
    int data;  
    struct List next;  
};
```

$[[\text{List}]] = \phi = (\text{int}, \phi)$

不可解




```
//C代码  
struct List{  
    int data;  
    struct List *next;  
};
```




更多例子：C++中的递归函数？

```
//C++代码
auto factorial(int i) {
    if(i == 1)
        return i;
    else
        return factorial(i-1)*i;
}
```



```
//C++代码
auto factorial(int i) {
    return (i == 1) ? i : factorial(i-1)*i;
}
```



```
#: clang++ autofunc.cpp
autofunc.cpp:12:25: error: function 'factorial' with deduced return
type cannot be used before it is defined
    return (i == 1) ? i : factorial(i-1)*i;
```


类型检查问题

- 已知类型系统中运算符的类型定义或函数签名
- 分析当前语句中的变量、常量是否满足类型约束

程序语句 (AST节点)

NUM

E1 op E2

V = E

E1 == E2

if(E){S}

if(E){S1}else{S2}

while(E){S}

FnCall(E1,...,En)

&V

*E

*V = E

V = *E

类型约束

$\llbracket \text{NUM} \rrbracket = \text{i32/u32/float/double}$

$\llbracket E1 \rrbracket = \llbracket E2 \rrbracket = \llbracket E1 \text{ op } E2 \rrbracket$

$\llbracket V \rrbracket = \llbracket E \rrbracket$

$\llbracket E1 \rrbracket = \llbracket E2 \rrbracket, \llbracket E1 == E2 \rrbracket = \text{bool}$

$\llbracket E \rrbracket = \text{bool}$

$\llbracket E \rrbracket = \text{bool}$

$\llbracket E \rrbracket = \text{bool}$

$\llbracket E \rrbracket = (\llbracket E1 \rrbracket \dots \llbracket En \rrbracket) \rightarrow \llbracket E(E1 \dots En) \rrbracket$

$\llbracket \&V \rrbracket = \&\llbracket V \rrbracket$

$\llbracket E \rrbracket = \&\llbracket *E \rrbracket$

$\llbracket V \rrbracket = \&\llbracket E \rrbracket$

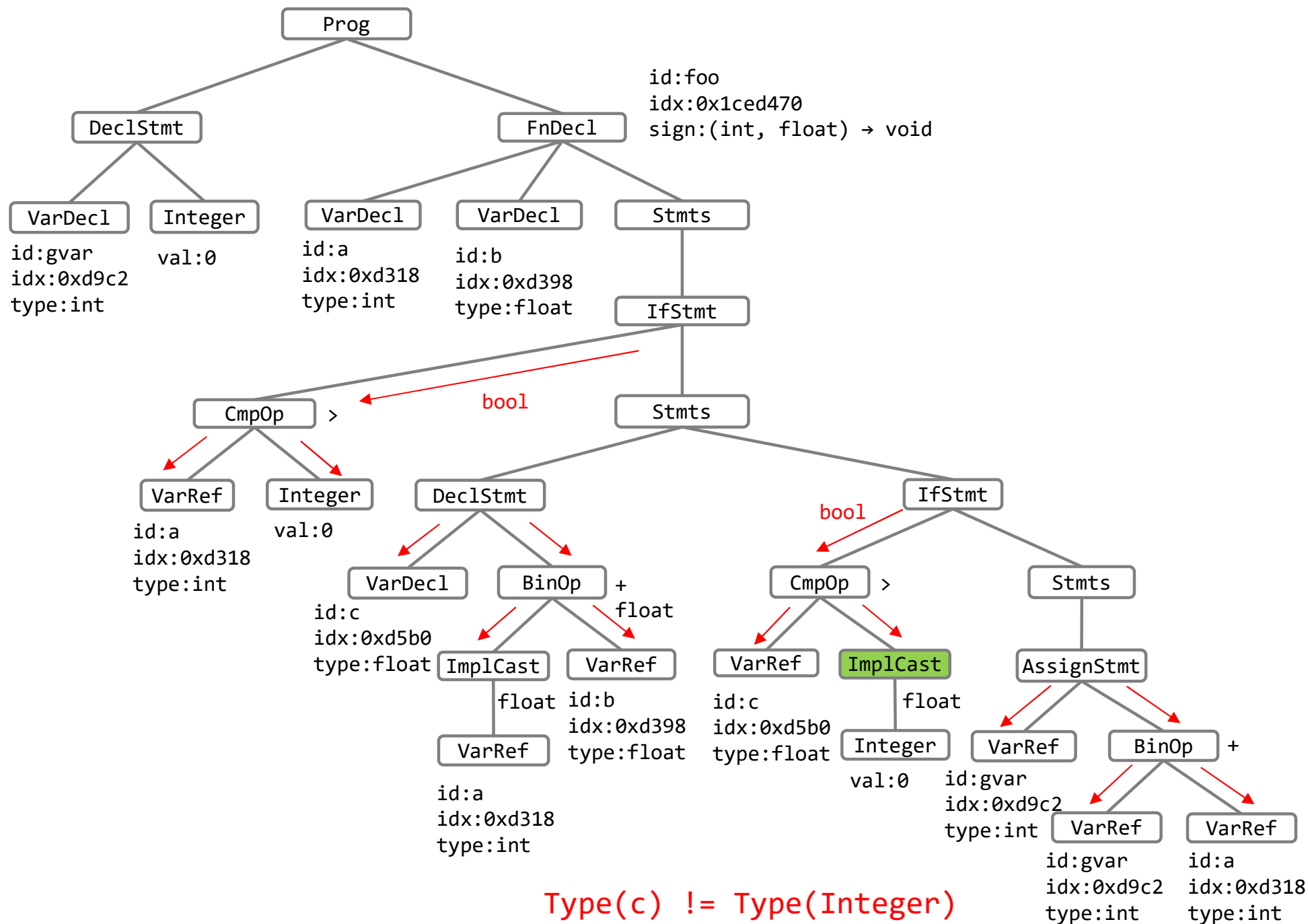
$\llbracket V \rrbracket = \llbracket *E \rrbracket, \llbracket E \rrbracket = \&\llbracket *E \rrbracket$

如何判断类型是否等价?

- 名字相同
- 结构相同
 - MyString vs String

```
struct MyString {  
    char* val;  
    len n;  
}  
struct String {  
    char* val;  
    len n;  
}
```

类型错误处理：隐式转换



思考：如何实现Variadic function?

- 可变参数的函数？如C语言的printf等函数

```
//c语言程序  
int printf(const char *format,...);
```

```
//c语言程序  
int sum(int num,...)  
{  
    va_list ap;  
    int sum = 0;  
    va_start(ap,num);  
    for(int i=0; i<num; i++){  
        sum += va_arg(ap,int);  
    }  
    va_end(ap);  
    return sum;  
}
```

总结

- 抽象语法树与语法解析树的区别
- 抽象语法树的构造方法
 - 树遍历
 - 属性语法和语法制导
- 类型系统设计方法
- 类型推断、类型检查和隐式转换