Lecture 7

AST + Types

徐 辉

xuh@fudan.edu.cn

大纲

1. 抽象语法树
2. 类型检查
3. 类型推断

1. 抽象语法树

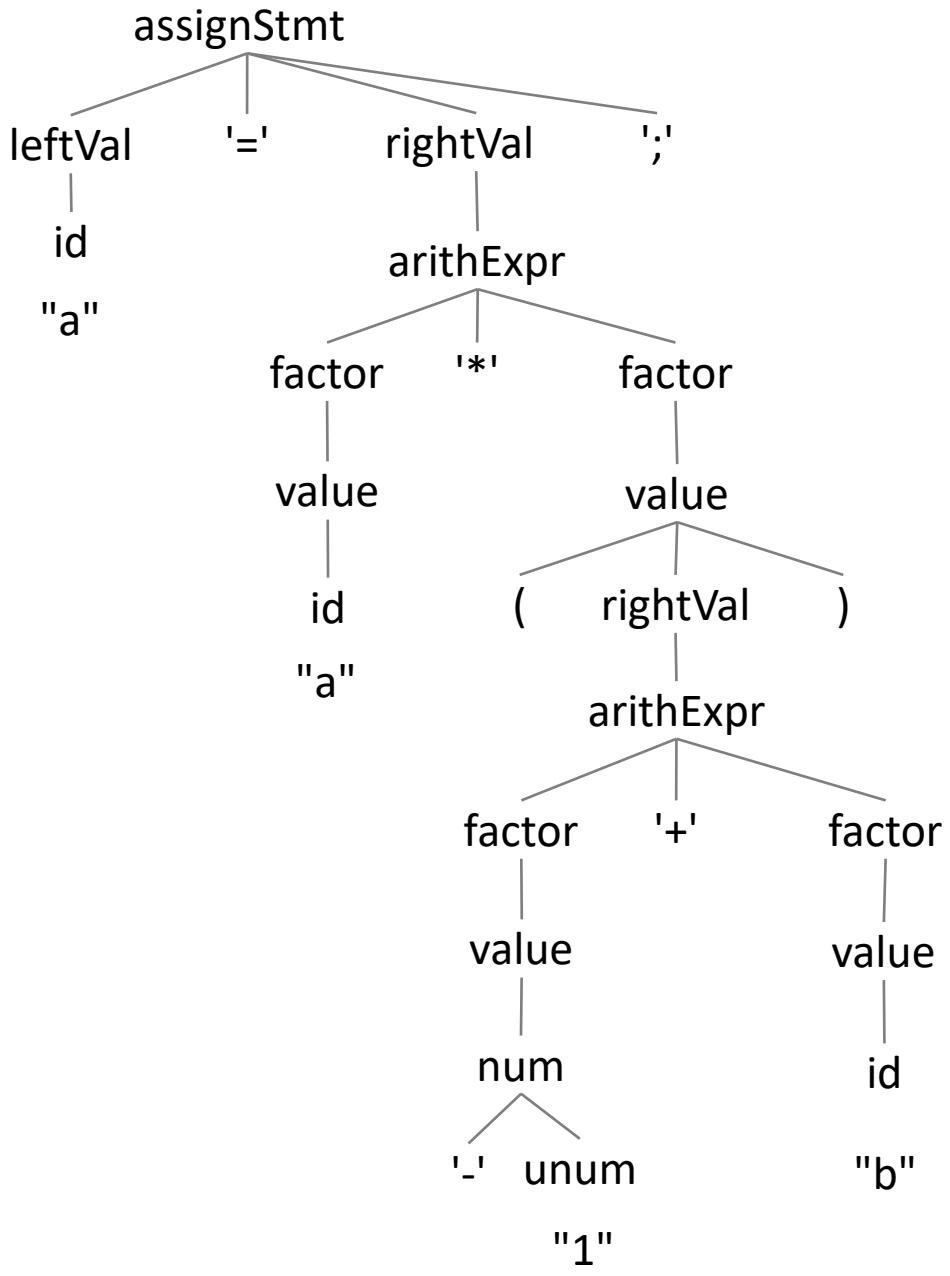
语法解析树回顾

- 画出下列代码的语法解析树?

```
a = a*(-1+b);
```

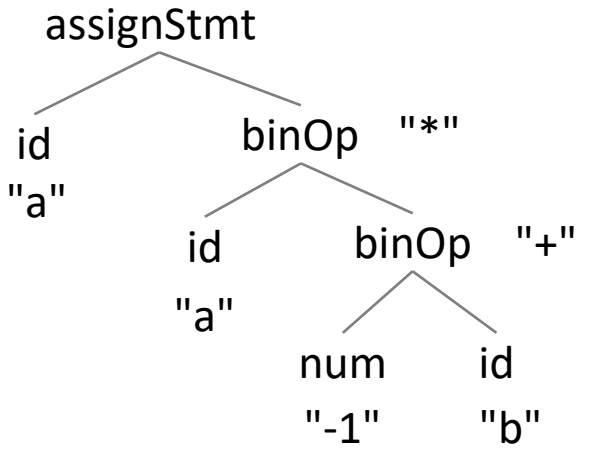
```
assignStmt  ⟼ leftVal '=' rightVal ';'
   leftVal  ⟼ id | deref | id '[' (num | id) ']' | id '.' id
  rightVal  ⟼ arithExpr | value
 arithExpr  ⟼ factor (('+' | '-') factor)*
    factor  ⟼ value (('*' | '/') value)*
     value  ⟼ num | id | fnCall | '(' rightVal ')'
     value  ⟼ id '.' id | id '[' (id | num) ']'
       num  ⟼ unum | ('-' unum)
```

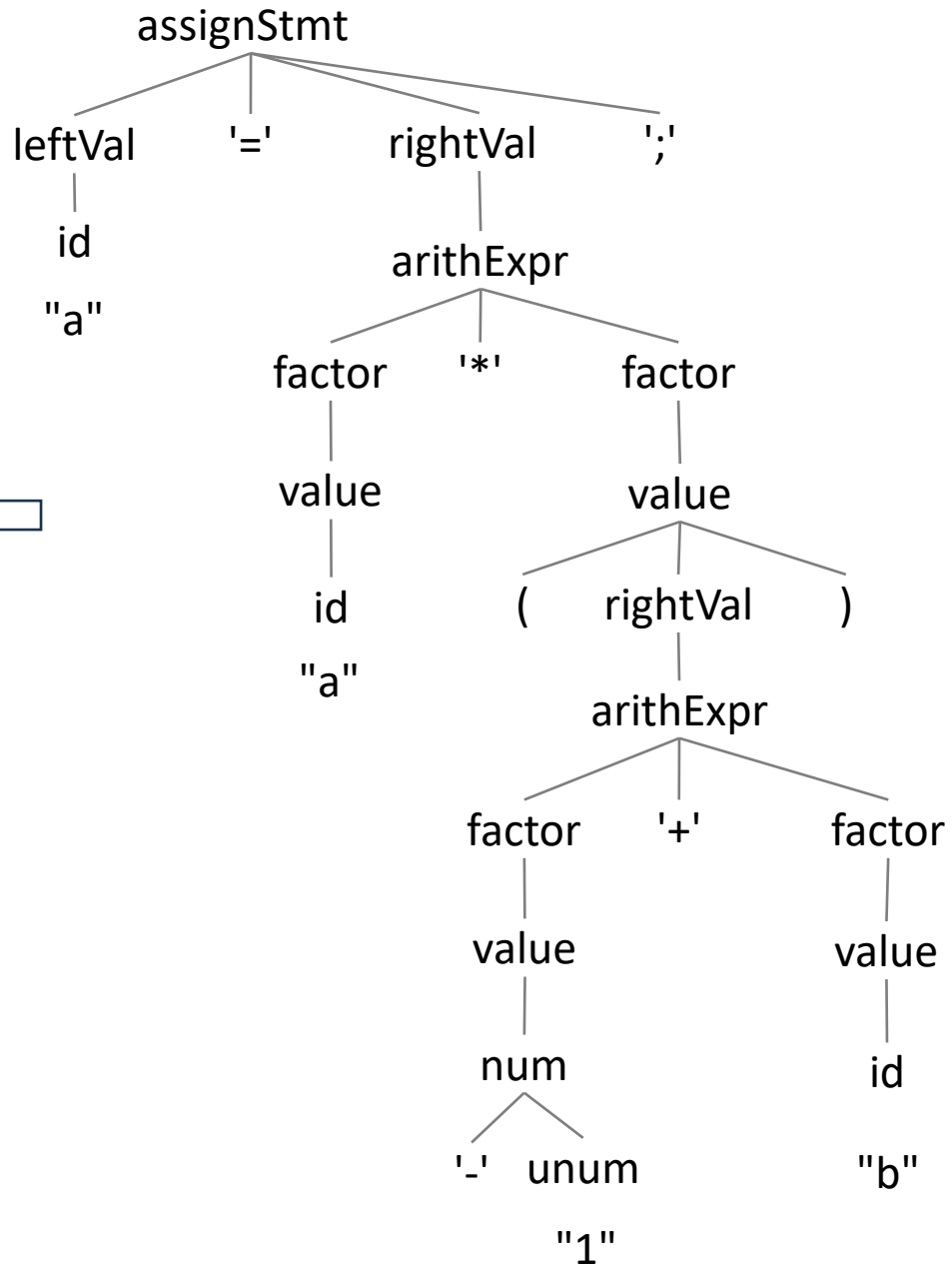
```
a = a*(-1+b);
```


assignStmt
leftVal
'='
rightVal
';'
id
"a"
arithExpr
factor
'*'
factor
value
value
id
"a"
(
rightVal
)
arithExpr
factor
'+'
factor
value
value
num
id
'-'
unum
"b"
"1"

语法解析树的问题

- 冗余节点较多
- 信息比较原始





抽象语法树：Abstract Syntax Tree

- 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
  - 单一展开形式塌陷，如factor->value->num->unum
  - 去掉括号等冗余信息
  - 运算符和关键字一般不作为叶子结点
- 可以被编译器后续编辑，记录上下文相关的信息

AST构造思路：语法制导（跳过语法解析树）

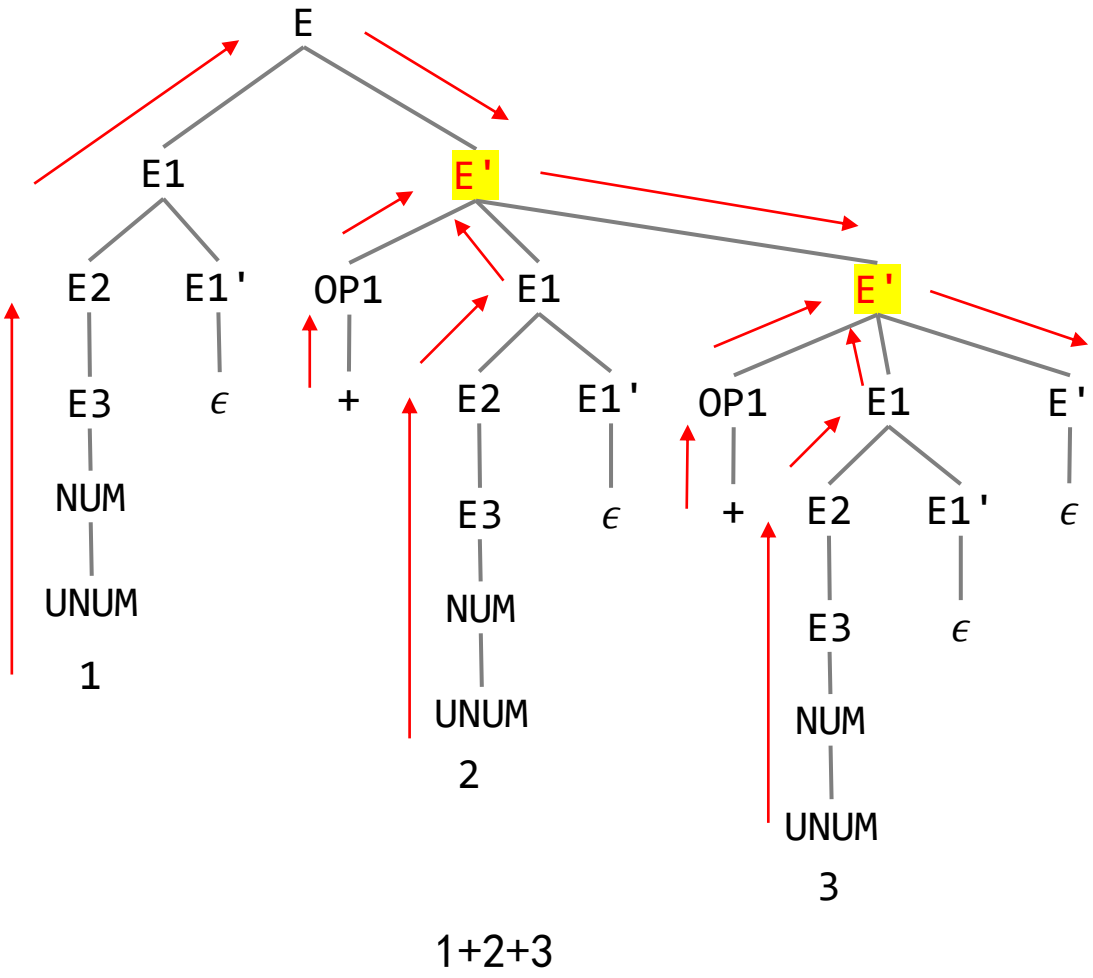
- S-attributed SDD：所有节点的属性语法都是根据其子节点的属性定义的

```
...
arithExpr
    : factor '+' factor { $$ = createBinOpNode($1, '+', $3); }
    | factor '-' factor { $$ = createBinOpNode($1, '-', $3); }
    | factor {$$ = $1;}
factor
    : value '*' value { $$ = createBinOpNode($1, '*', $3); }
    | value '/' value { $$ = createBinOpNode($1, '/', $3); }
    | value {$$ = $1;}
value
    : num             { $$ = createNumNode($1); }
    | id              { $$ = createIdNode($1);}
    | fnCall          { $$ = createFnCall($1);}
    | "(" arithExpr ")"      { $$ = $2; }
...
```

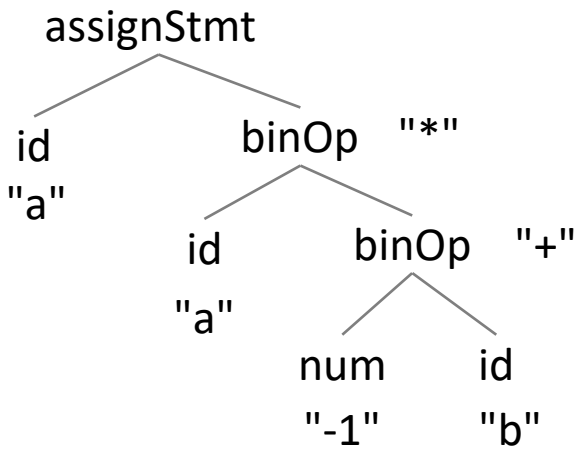
LL(1)的属性文法属于L-Attributed SDD

- L-attributed SDD：对于$A \to \beta_1..\beta_i..\beta_n$中的任意$\beta_i$来说，其属性语法依赖$A$或$\beta_1, \ldots, \beta_{i-1}$

```
[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>
```



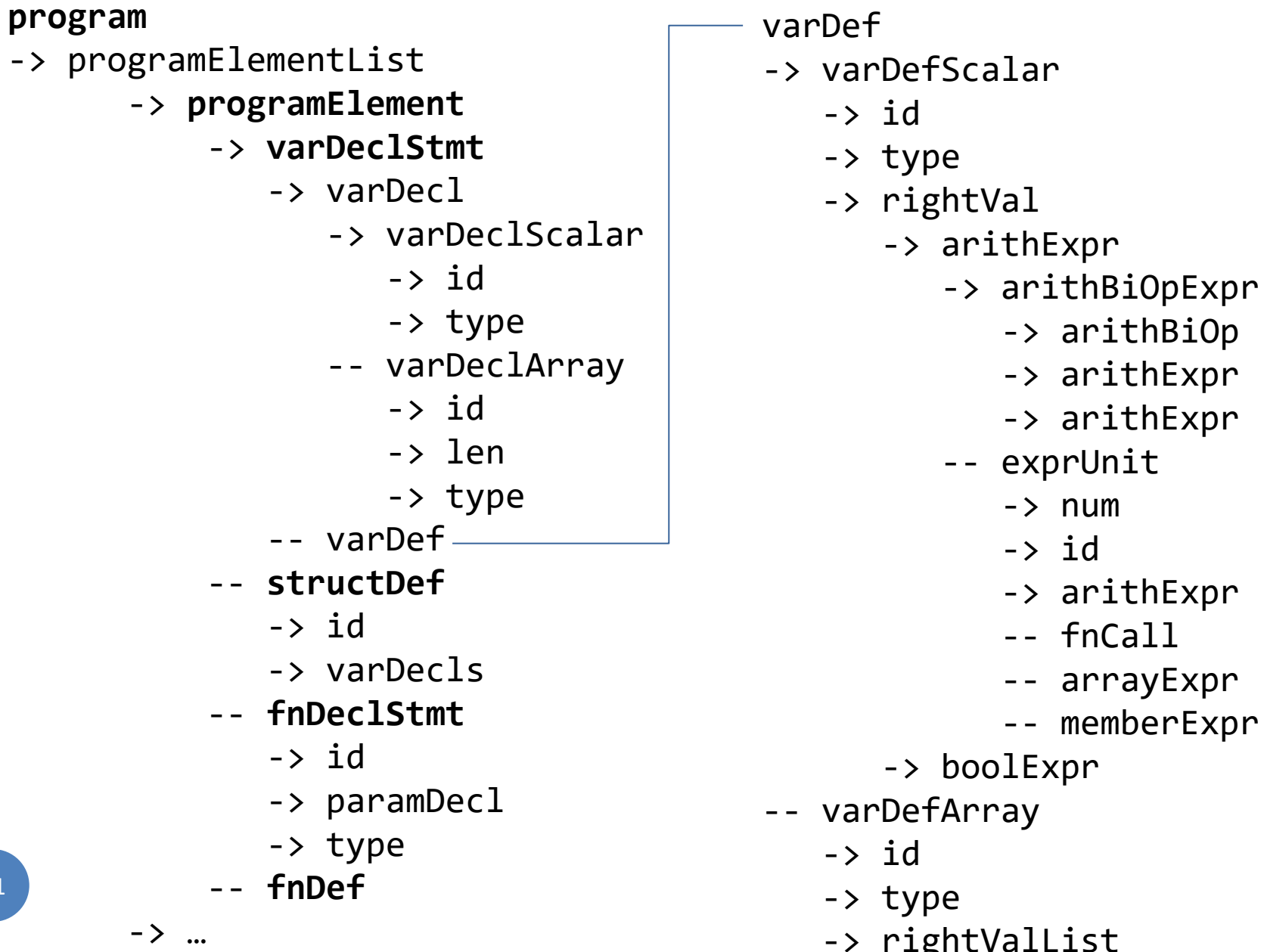
抽象语法树构造思路：遍历语法解析树




```
struct assignStmt {
    lv : *leftVal, //enumerate type
    rv : *rightVal //enumerate type
}
```

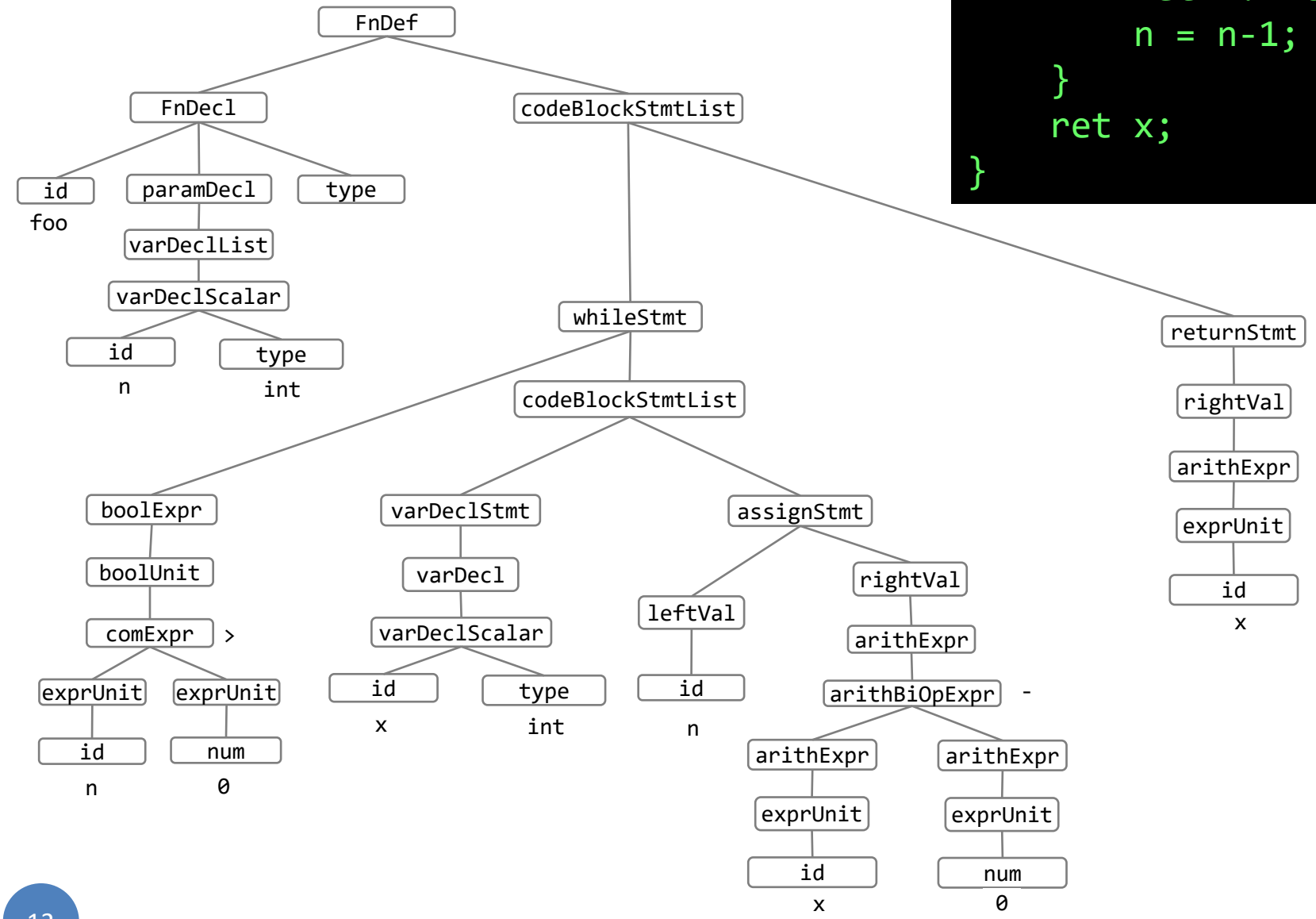
```
struct binOp {
    op : OP, //enumerate type:+,-,*,/
    op1 : *operand, //enumerate type
    op2 : *operand //enumerate type
}
```

实验项目中采用的树型结构



应用举例

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



2. 类型检查

TeaPL的类型系统

- 类型：
  - 基础类型（Primitive Type）
    - 标量类型（Scalar Types）：int、bool
    - 复合类型（Compound Type）：数组
    - 函数类型
  - 自定义类型：使用struct定义
- 规则（静态类型系统）
  - 类型推断：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？（类型强弱）

类型动静：编译时检查类型的一致性

- 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'
```

类型强弱：是否允许隐式类型转换？

```
//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

```
//javascript代码
var a = 42;
var b = "42";
var c = [42];
a === b;
a == b;
a == c;
```

false
true
true

类型检查问题（假设类型都已确定）

- 已知类型系统中运算符的类型定义或函数签名
- 分析代码中的变量、常量是否满足类型约束
- 基本思路：基于作用域对AST上对标识符做区分和检验
  - 声明新标识符：确定作用域
  - 使用标识符：确定类型

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

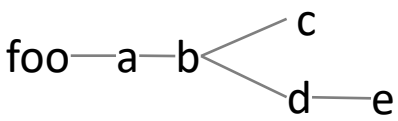
标识符	作用域（粗）	索引	类型
g	global	0xd9c2	int
fib	global	0xd470	(int) → int
main	global	0xd318	(void) → void
n	fib	0xd398	int
x	fib	0xd5b0	int
y	fib	0xd2c2	int

作用域分析

```
let g:int = 10;
fn foo(n: int) -> int {
    let a:int;
    {
        let b:int;
        {
            let c:int;
        }
        let d:int;
        {
            let e:int;
        }
    }
}
```

```
let g:int = 10;
fn foo(n: int) -> int { //scope foo
    let a:int; { //scope a
        { //scope 2
            let b:int; { //scope b
                { //scope 4
                    let c:int; { //scope c
                    }
                }
                let d:int; { //scope d
                    {//scope 7
                        let e:int; { //scope e
                        }
                    }
                }
            }
        }
    }
}
```

Partial order of scopes: foo > a > b > { c, d > e }



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 {
        return n;
    }
    let x:int = foo(n - 1);
}
```

缺少返回语句

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

未定义

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

未定义

重复声明问题

```
let x:int;
{
    let x:int;
}
```

```
let x:int;
{
    let x:char;
}
```

```
fn foo(x:int);

fn foo() -> int;
```

```
{
    let x:int;
}
let x:int;
```

```
{
    let x:int;
}
{
    let x:int;
}
```

声明顺序问题

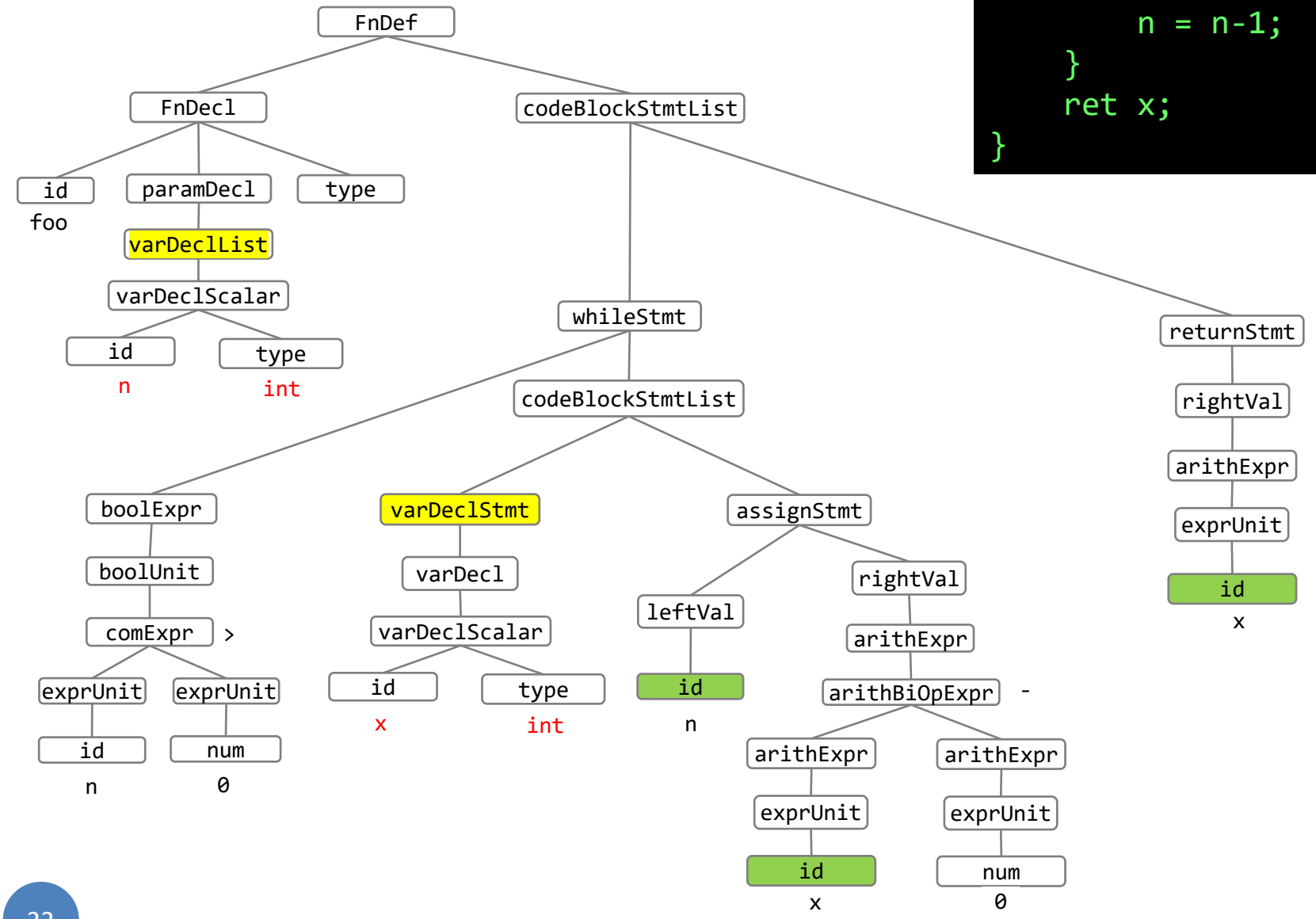
- 全局变量和函数不关注声明顺序

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

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

基于AST进行类型检查

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



检查算法思路

- 动态维护两个变量声明表（HashMap）记录变量作用域和类型
  - 全局变量表
  - 局部变量表
    - 声明时加入
    - 跳出作用域时移除
- 检查规则：
  - 重复声明：当前申明变量名称已经在局部变量表中
  - 未定义变量：当前使用变量不在局部变量表中
  - 类型错误：当前使用变量类型要求与局部变量表中的类型不匹配

类型检查的其它问题

- 如何判断类型是否等价?
  - 名字相同
  - 结构相同：Pos vs Loc
- 如果不通过是否允许类型转换?
  - 允许：插入转换节点ImplicitCast

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

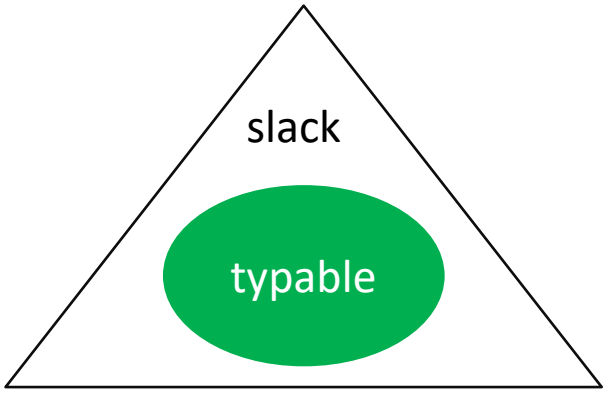
练习：对照实验文件定义TeaPL的类型检查规则

- 代码形式：x + y => x, y 是整型
- AST形式：exprUnit => 整形
- ...

3. 类型推断

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）



如何推断类型？

- 为不同的语法制定相应的推断规则

```
let a = 0;
```

X = I　　⟦X⟧ = int

```
let a;
let b:int;
a = b;
```

X = Y　　⟦X⟧ = ⟦Y⟧

```
let a = 1>0;
if(a);
```

if(X)　　⟦X⟧ = bool

```
let a;
let b;
let c = foo(a, b)
```

F(X,Y)　　⟦F⟧ = (⟦X⟧, ⟦Y⟧)->⟦F(X,Y)⟧

基于AST生成类型约束并求解

- 前置要求：确定变量作用域
- 类型表示：用⟦X⟧表示变量X的类型
- 约束提取：一般都为等价关系
- 约束求解：通过unification algorithm

练习：对照实验文件定义TeaPL的类型推导规则

- 类型检查和类型推导的顺序？