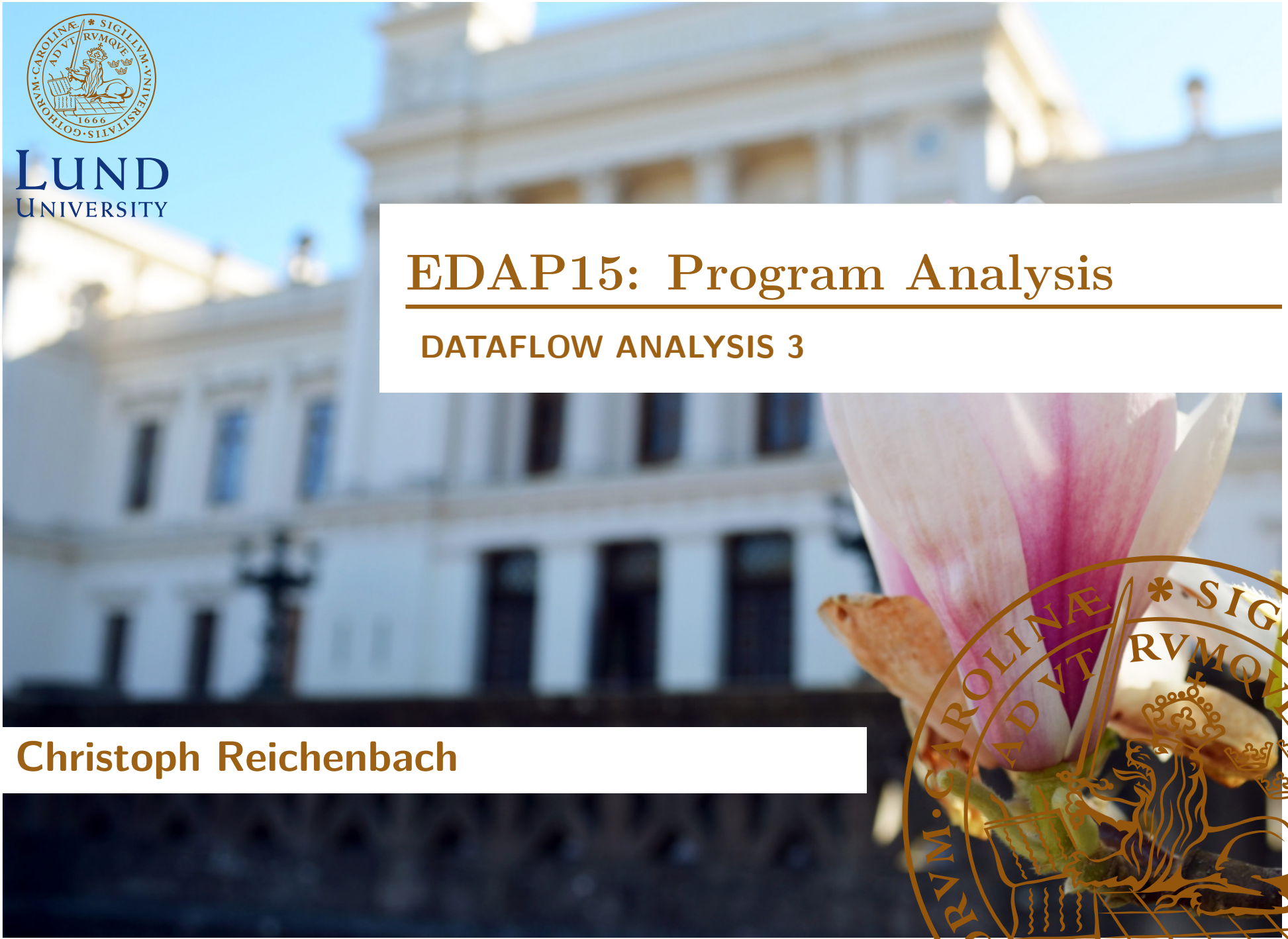


LUND UNIVERSITY

EDAP15: Program Analysis

DATAFLOW ANALYSIS 3

Christoph Reichenbach

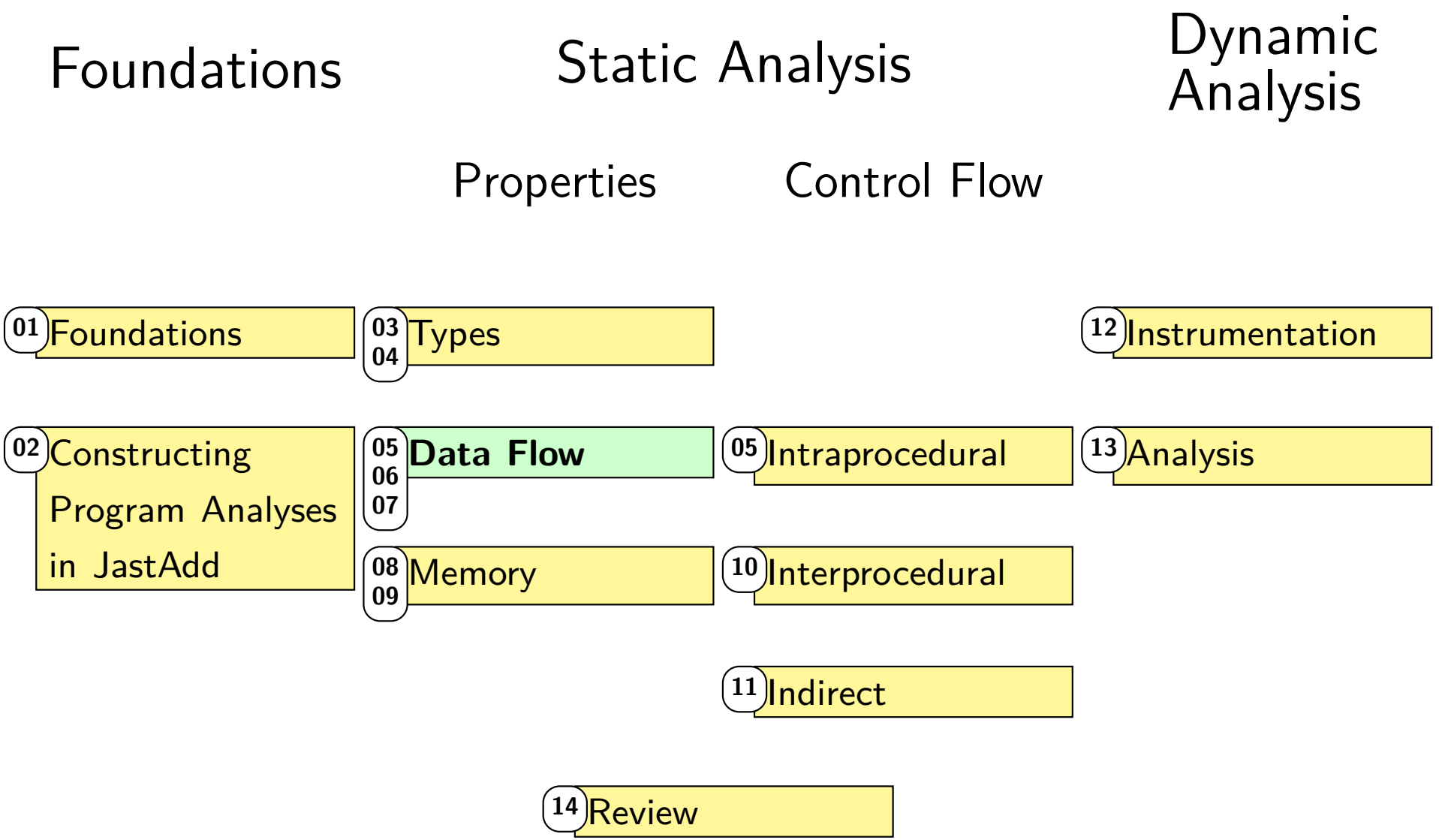
Welcome back!

Some Administrativa:

- Quiz deadlines: some slack if you missed a deadline
  - 8 days buffer (cumulative across all quizzes), not counting weekends

Questions?

Lecture Overview

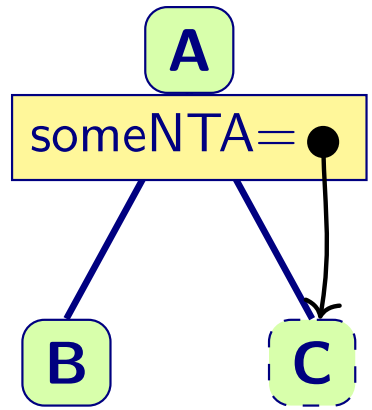


Non-Terminal Attributes

JastAdd

```
syn nta C AnyNode.someNTA() = new C(this);
```

- AST node as attribute
- Reifying implicit constructs (making them explicit in AST)
  - Built-in types
  - Built-in functions, constants
  - "Null Objects", handle missing declarations ($\Rightarrow$ EDAN65)



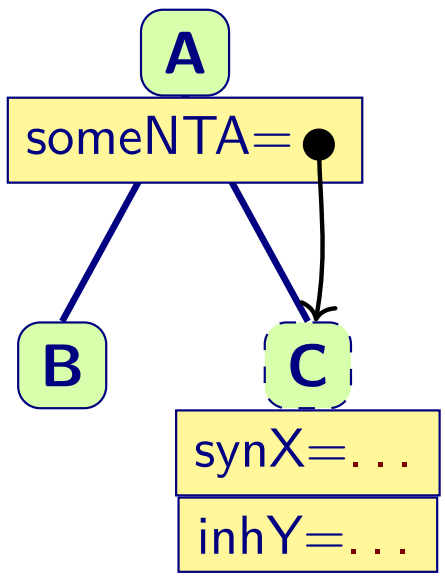
Beware

- NTAs must be **fresh** objects
- AST will be inconsistent if you re-use nodes
  JastAdd does not check for this!

Non-Terminal Attributes

JastAdd

```
syn nta C AnyNode.someNTA() = new C(this);
```



- NTA may have attributes
- Owner node must provide inherited attributes
  `A.someNTA().inhY() = ...`

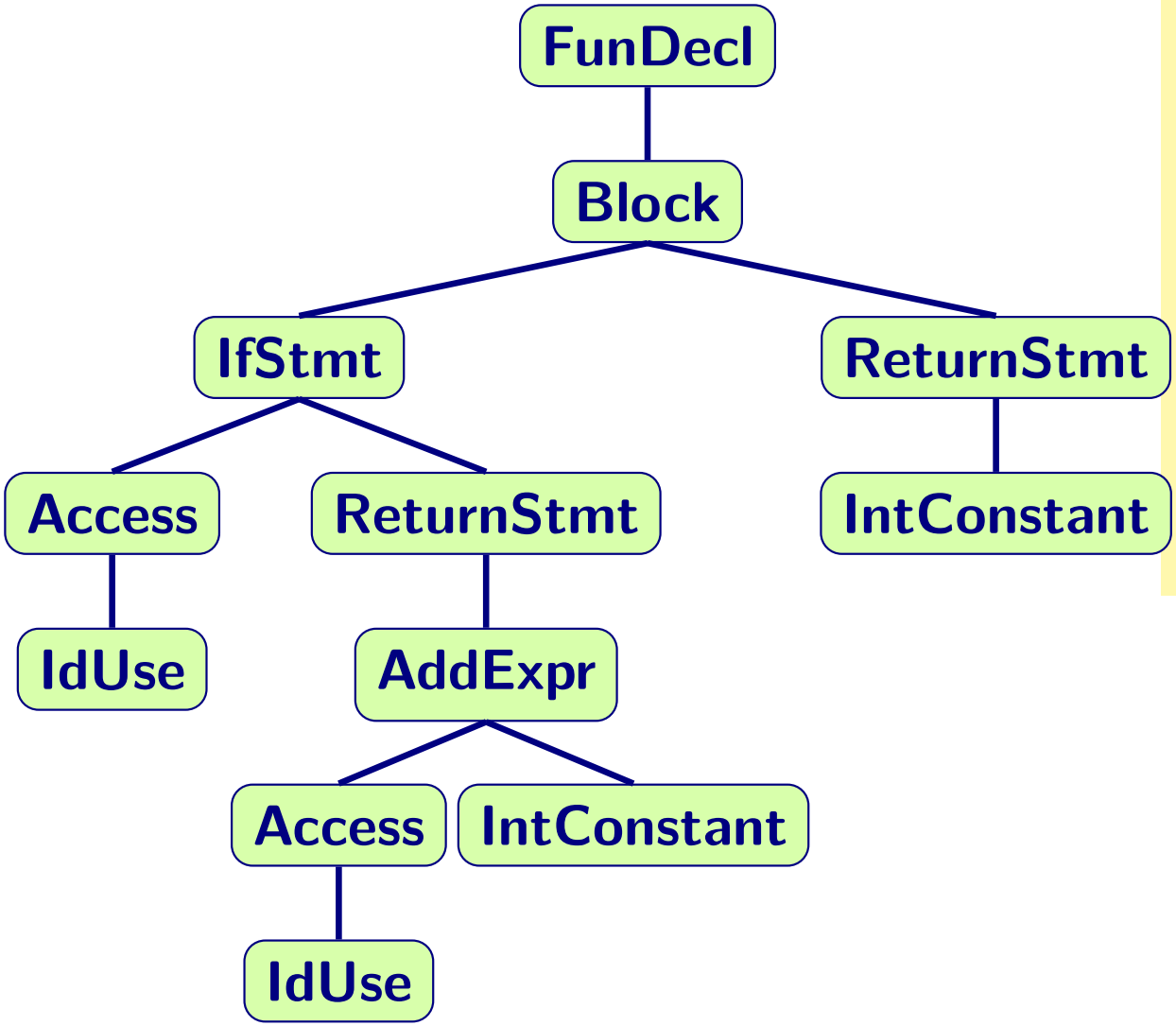
Summary

- Nonterminal Attributes (NTAs):
  - "Synthetic" AST node
  - Useful e.g. for CFG nodes that have no AST equivalent
  - Need to be *fresh*
  - Need to be owned by exactly one parent
- Function like normal AST nodes
  - Can define / inherit attributes
  - Can participate in collection attributes

Building CFGs

- **1.** CFG separate from AST
  - AST $\Rightarrow$ CFG translation
  - Often simplified
  - *Advantages*:
    - Fewer CFG node types
    - Analyses can communicate results by transforming CFG (Remove unreachable CFG nodes etc.)
    - *Common in compiler mid-ends/back-ends*
- **2. CFG is part of AST**
  - *Some* AST nodes are also CFG nodes
  - *Advantages*:
    - No translation needed
    - *Common in compiler front-ends and IDEs*
    - Teal: Uses JastAdd's IntraCFG framework

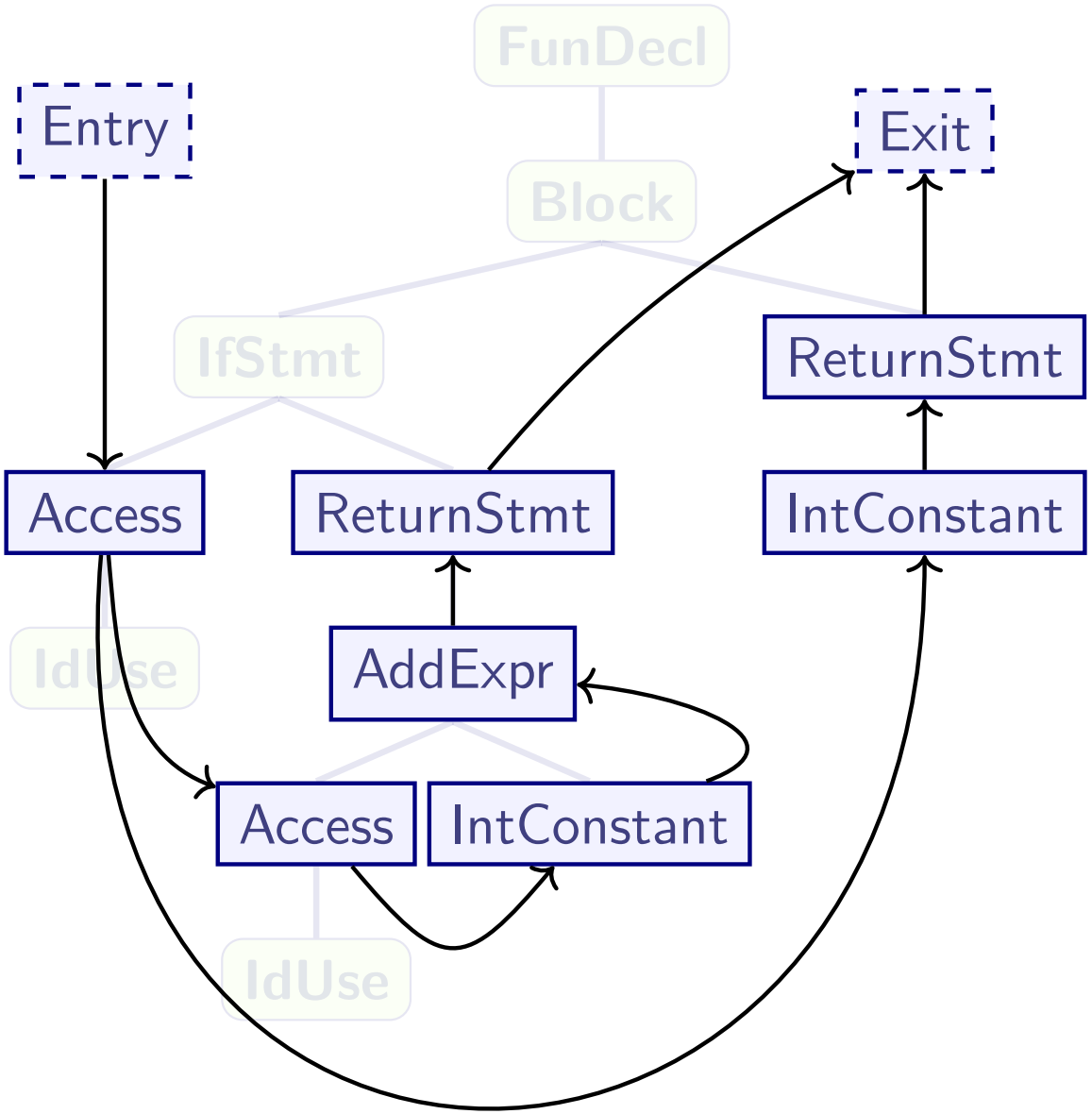
CFGs on the AST



Teal

```
fun f(x) = {
  if x {
    return x + 1;
  }
  return 0;
}
```

CFGs on the AST



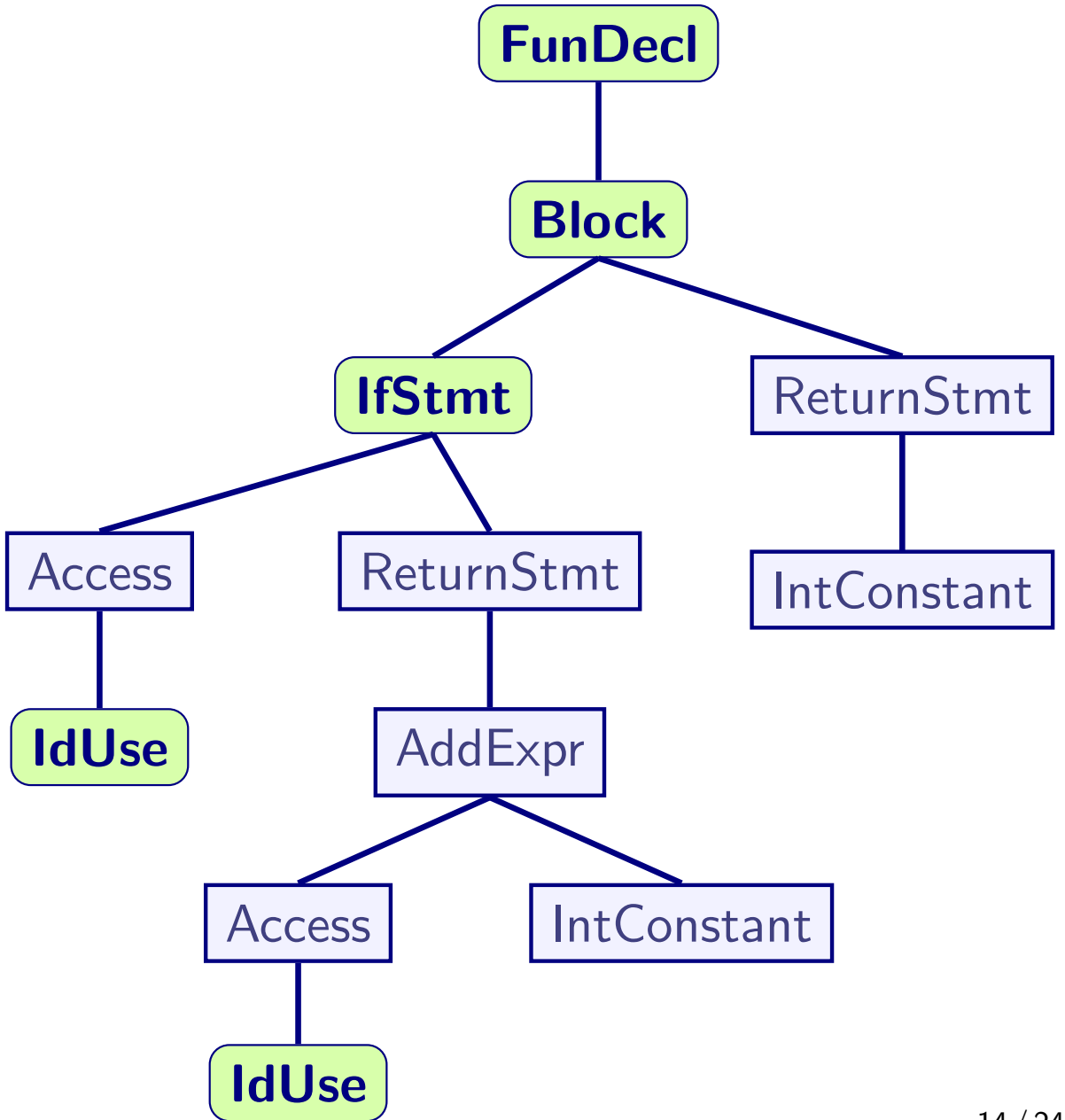
- Some **ASTNode**s are CFGNodes
  - For example, Teal:
    - All **Expr**, some **Stmt**
    - Special NTAs: **Entry**, **Exit**,...
- `CFGNode.succ()` attribute for CFG successors: N1 → N2

Teal

```
fun f(x) = {
  if x {
    return x + 1;
  }
  return 0;
}
```

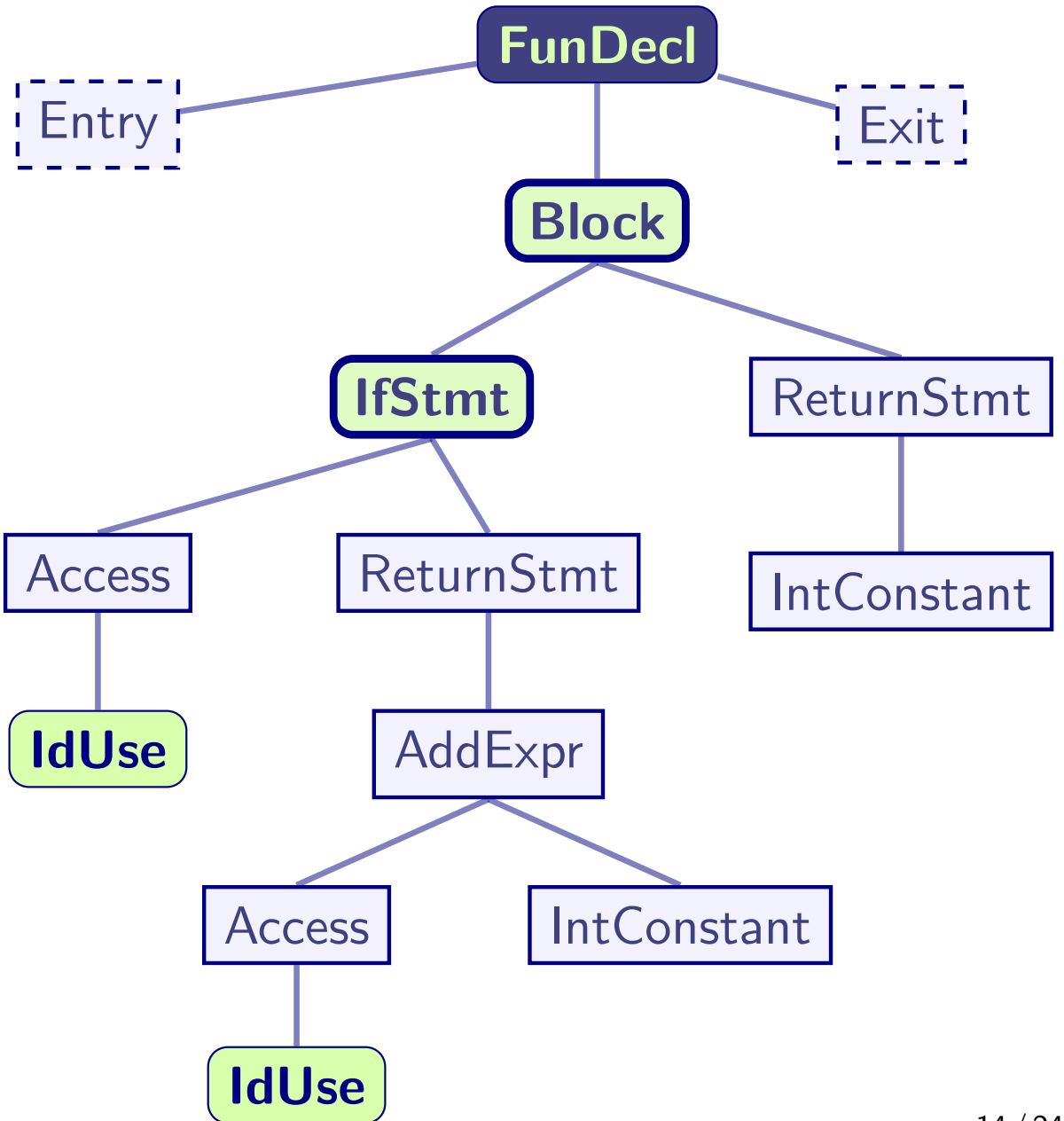
Constructing CFGs on the AST (1/2)

- Categorise AST nodes by role in CFG
  - CFGNode: part of CFG



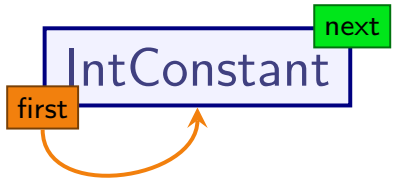
Constructing CFGs on the AST (1/2)

- Categorise AST nodes by role in CFG
  - CFGNode: part of CFG
  - **CFGRoot**: start/end of CFG with Entry / Exit NTAs (e.g., FunDecl)
  - **CFGSupport**: not part of CFG but influence CFG edges

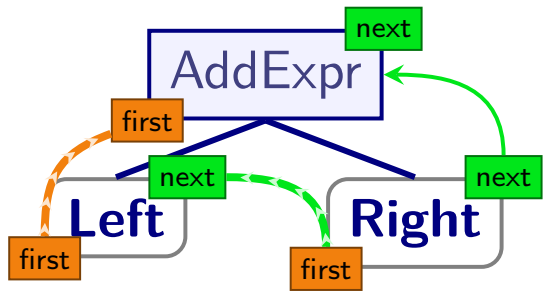


Constructing CFGs on the AST (2/2)

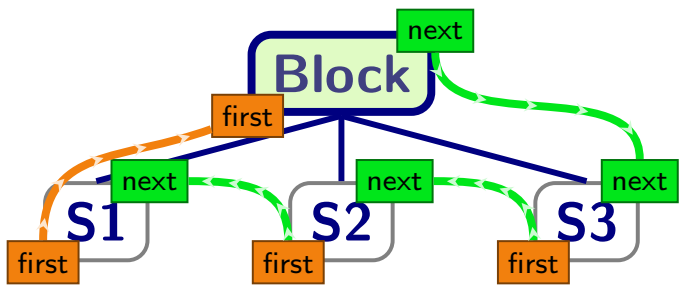
```
interface CFGNode extends CFGSupport;
  → syn Set CFGSupport.firstNodes()
  → inh Set CFGSupport.nextNodes()
```



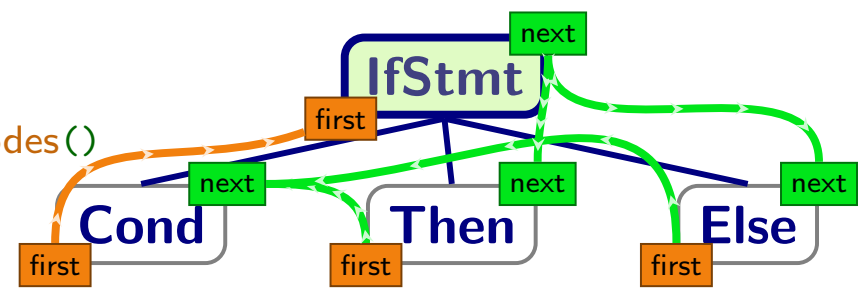

```
AddExpr.firstNodes() = getLeft().firstNodes()
AddExpr.getLeft().nextNodes() = getRight().firstNodes()
AddExpr.getRight().nextNodes() = new Set({this})
```




```
BlockStmt.firstNodes() = getStmt(0).firstNodes()
BlockStmt.getStmt(i).nextNodes() =
          if (i<size):  getStmt(i+1).firstNodes()
          else:  this.nextNodes()
```



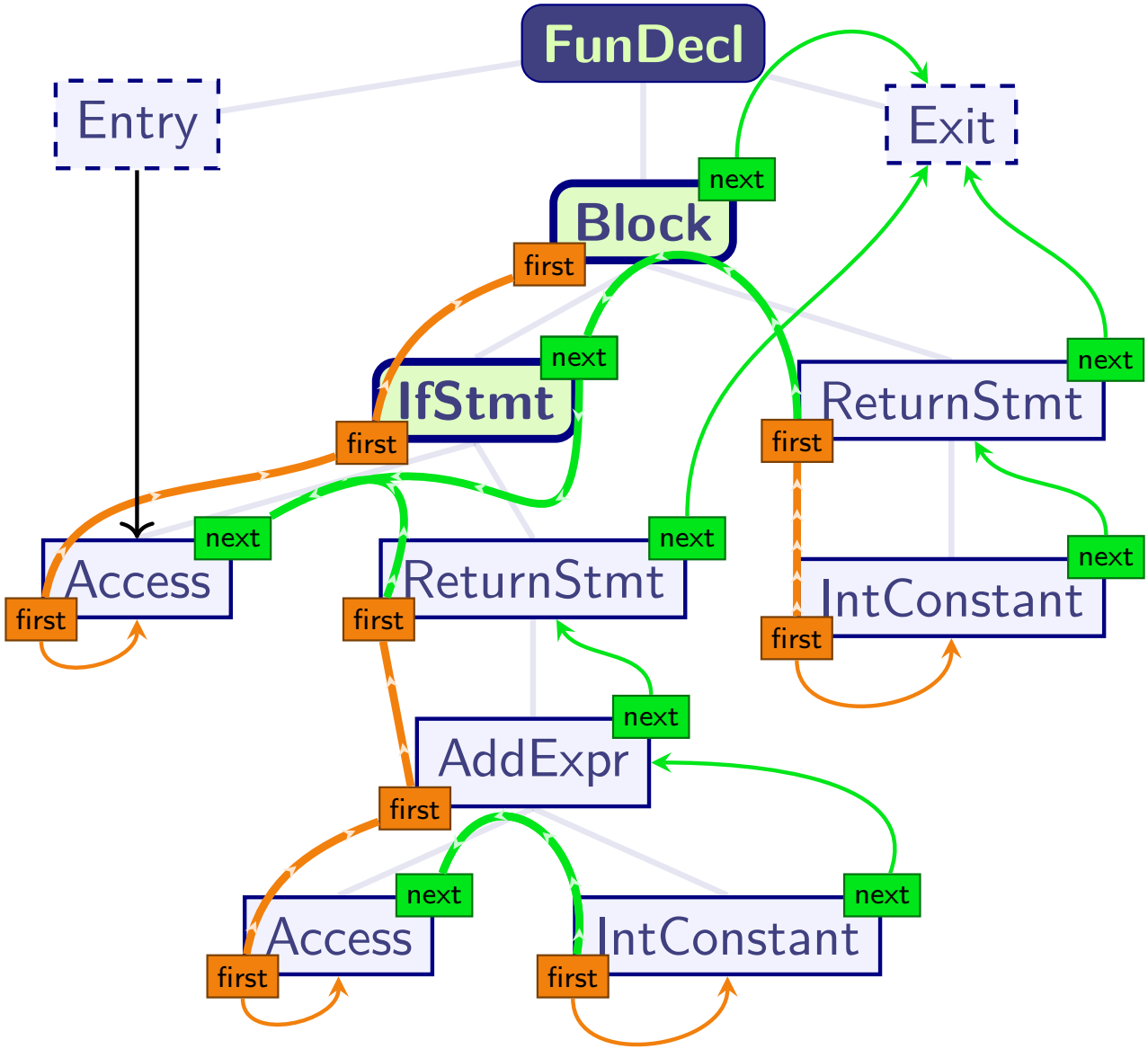

```
IfStmt.firstNodes() = getCond().firstNodes()
IfStmt.getCond().nextNodes() =
       getThen().firstNodes() ∪ getElse().firstNodes()
IfStmt.getThen().nextNodes() = this.nextNodes()
IfStmt.getElse().nextNodes() = this.nextNodes()
```



Constructing CFGs on the AST (1/2)

- Categorise AST nodes by role in CFG
  - CFGNode: part of CFG
  - **CFGRoot**: start/end of CFG with Entry / Exit NTAs (e.g., FunDecl)
  - **CFGSupport**: not part of CFG but influence CFG edges
  - **ASTNode**s: can ignore
- Construct edges
  - For each subtree: first CFGNodes in subtree?
  - For each CFGNode: next CFGNodes after self?

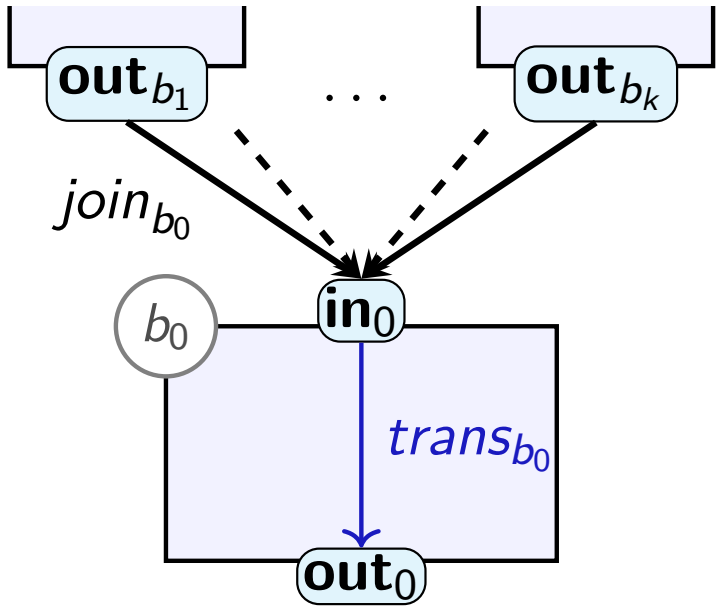
⟶ `succ()`
⟶ `firstNodes()`
⟶ `nextNodes()`



Summary

- CFG can be separate or overlaid on AST
- Teal uses an overlay CFG
- CFGNodes are:
  - ASTNodes that participate in CFG
  - Some NTAs:
    - **Entry**: Subprogram start
    - **Exit**: Subprogram end
    - Others can be useful e.g. for exception handling
- Constructing CFG with IntraCFG:
  - firstNodes: For this *subtree*, which CFGNodes execute first? *synthesised attribute*
  - nextNodes: For this *CFGNode*, which CFGNodes execute next? *inherited attribute*

Implementing Data Flow Analysis



JastAdd

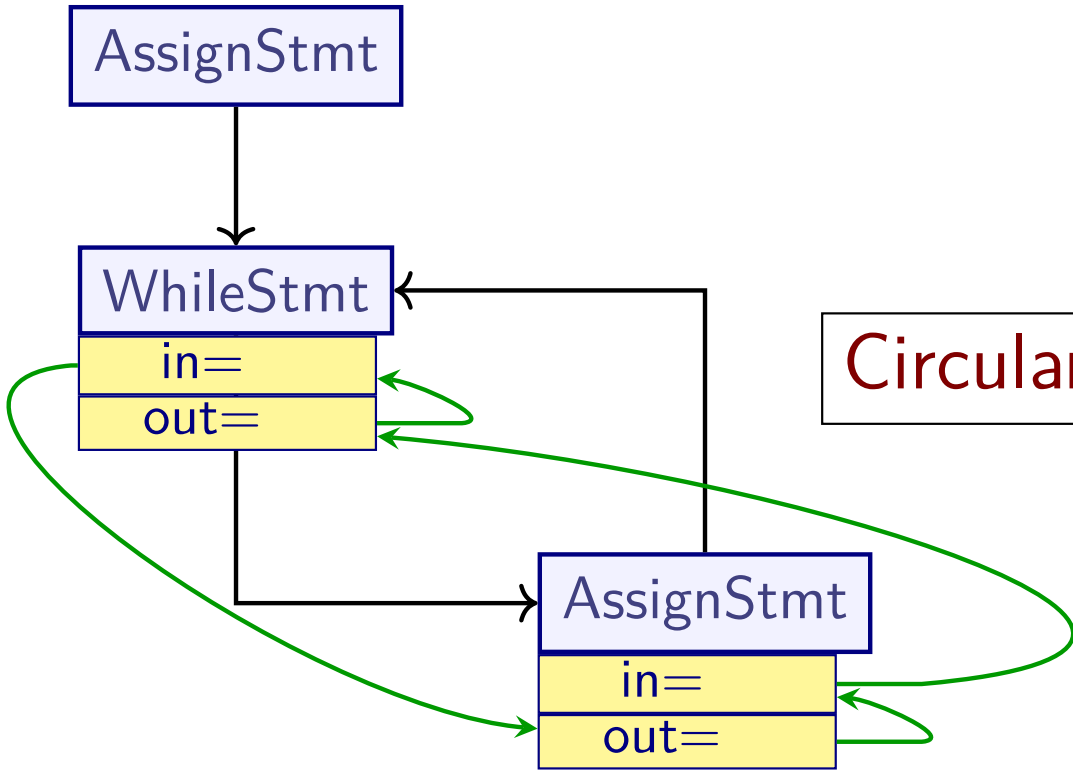
```
syn Lattice CFGNode.in() {
  Lattice r = ⊥;
  for (CFGNode b:  pred()) {
    r = r ⊔ b.out();
  }
  return r;
}
syn Lattice CFGNode.out() =
  trans();
```

JastAdd

```
// Default:  trans() = in()
syn Lattice CFGNode.trans() = in();

// Specialised transfer function
syn Lattice AssignStmt.trans() = ...
```

Fixpoints and Reference Attributes



Circular Dependency

This solution is *not well-defined*

Fixpoints from Circular Attributes

JastAdd

```
syn Lattice CFGNode.out() circular [Lattice.bottom()];
```

Lattice

$\bot$

- **Circular Attributes**
  - JastAdd allows circular dependency *if explicitly declared*
  - `CFGNode.out()` can now recursively call itself

    $v_1 = \bot$ = `CFGNode.out()` at iteration 1
    v_2 = `CFGNode.out()` at iteration 2
    ...
    v_k = `CFGNode.out()` at iteration k
  - Iterates until v_{k-1}`.equals(`v_k`)`

Beware

- Your lattice must have a correct `equals()` method
- You must be in a monotone framework

*JastAdd can **not** check for this!*

Implementation Strategy

- Definitions for analysis a on lattice $\mathcal{L}$:

Attribute	Forward	Backward
$\mathcal{L}$ a`In()`	$\bigsqcup\{p.a\texttt{Out()} \mid p \in \text{pred}()\}$	a`Transfer()`
$\mathcal{L}$ a`Transfer()`	$\mathit{trans}(a\texttt{In()})$	$\mathit{trans}(a\texttt{Out()})$
$\mathcal{L}$ a`Out()`:	a`Transfer()`	$\bigsqcup\{p.a\texttt{In()} \mid p \in \text{succ}()\}$

- Not necessary to declare all attributes as circular
  - Only one attribute in each cycle need be circular

. . . but can declare circular if unsure

Combining with Expressions

- Data flow analysis resolves *flow-sensitive* information
- May still combine with simple synthesised attributes
  Example: Constant propagation / folding
  - computing `Expr.constantValue()`
  - `Access.constantValue()` $\Leftarrow$ `constantOut()`
  - but `AddExpr.constantValue()` will not typically use `constantOut()` / `constantIn()` directly
  - uses `getLeft().constantValue()` / `getRight().constantValue()`

Summary

- Attributes that depend on themselves:
  Usually $\Longrightarrow$ *AG not well-defined*
- **Circular** attributes are exception
  - JastAdd suppresses recursion check
  - Repeated evaluation
  - Evaluation stops once current result `.equals()` last result
- It is up to attribute definition to guarantee termination!
  - Monotone framework
  - *Finite lattice height*
    - (Or *widening*, later today)

Summary and Outlook

- Summary:
  - Non-Terminal Attributes
  - Building CFGs
  - Circular Attributes
- Next up: Analysing the Heap

`https://cs.lth.se/EDAP15`