East Carolina UNIVERSITY

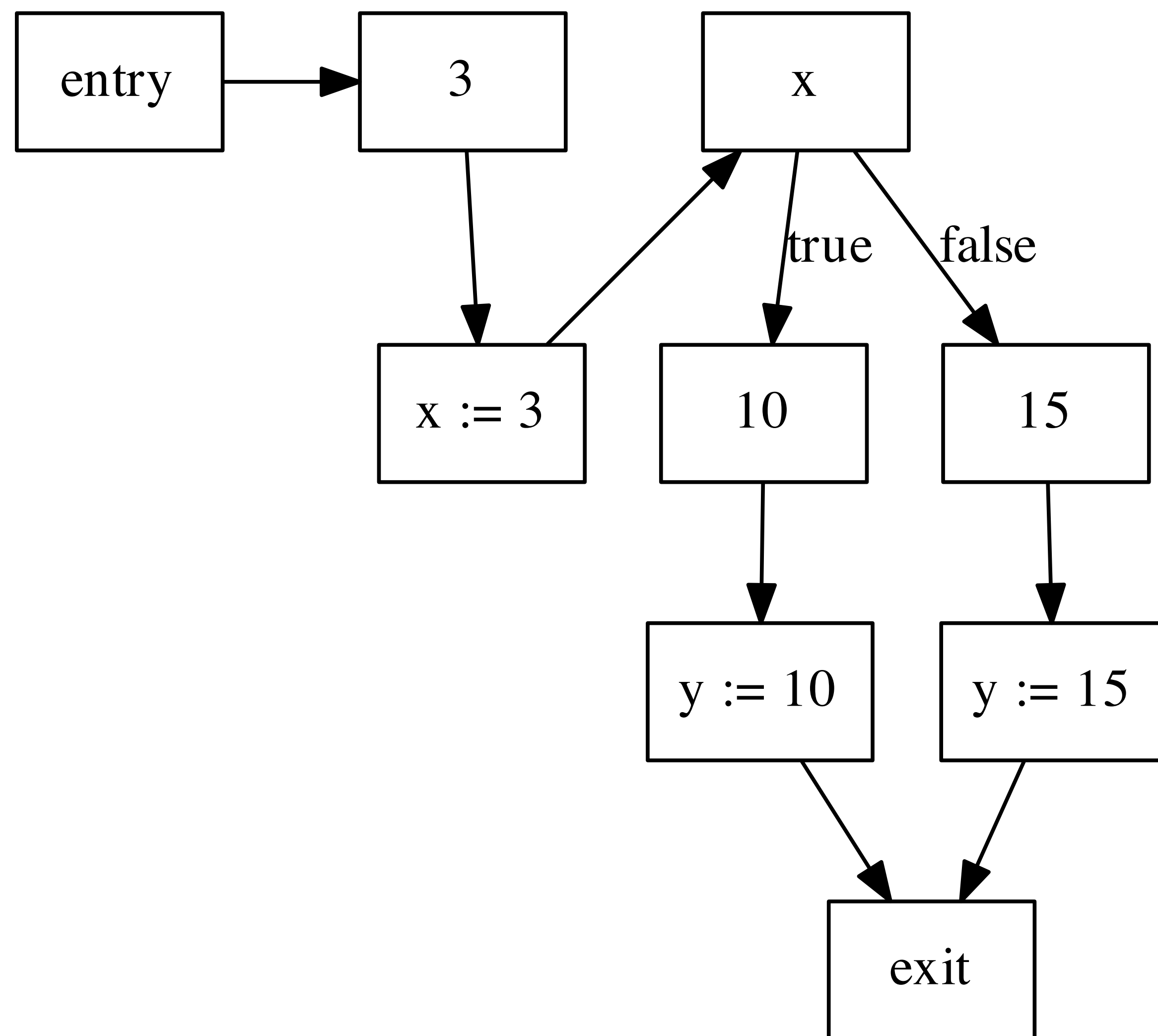
Streamlining Control-Flow Graph Construction with DCFlow

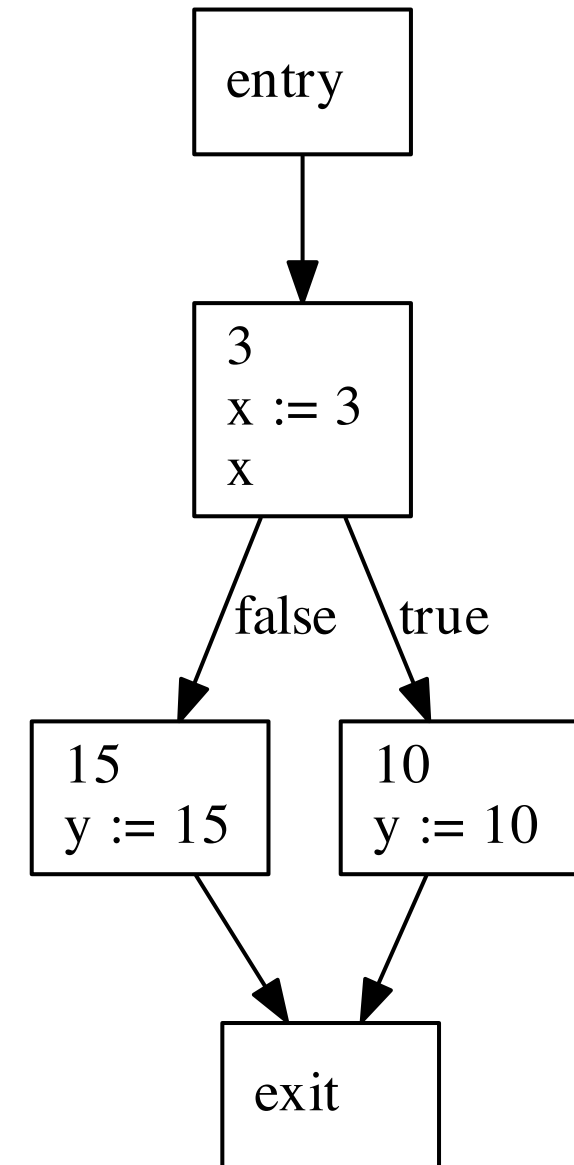
Mark Hills

7th International Conference on Software Language Engineering (SLE 2014)
September 15-16, 2014
Västerås, Sweden

http://www.rascal-mpl.org

Say you need a control flow graph…

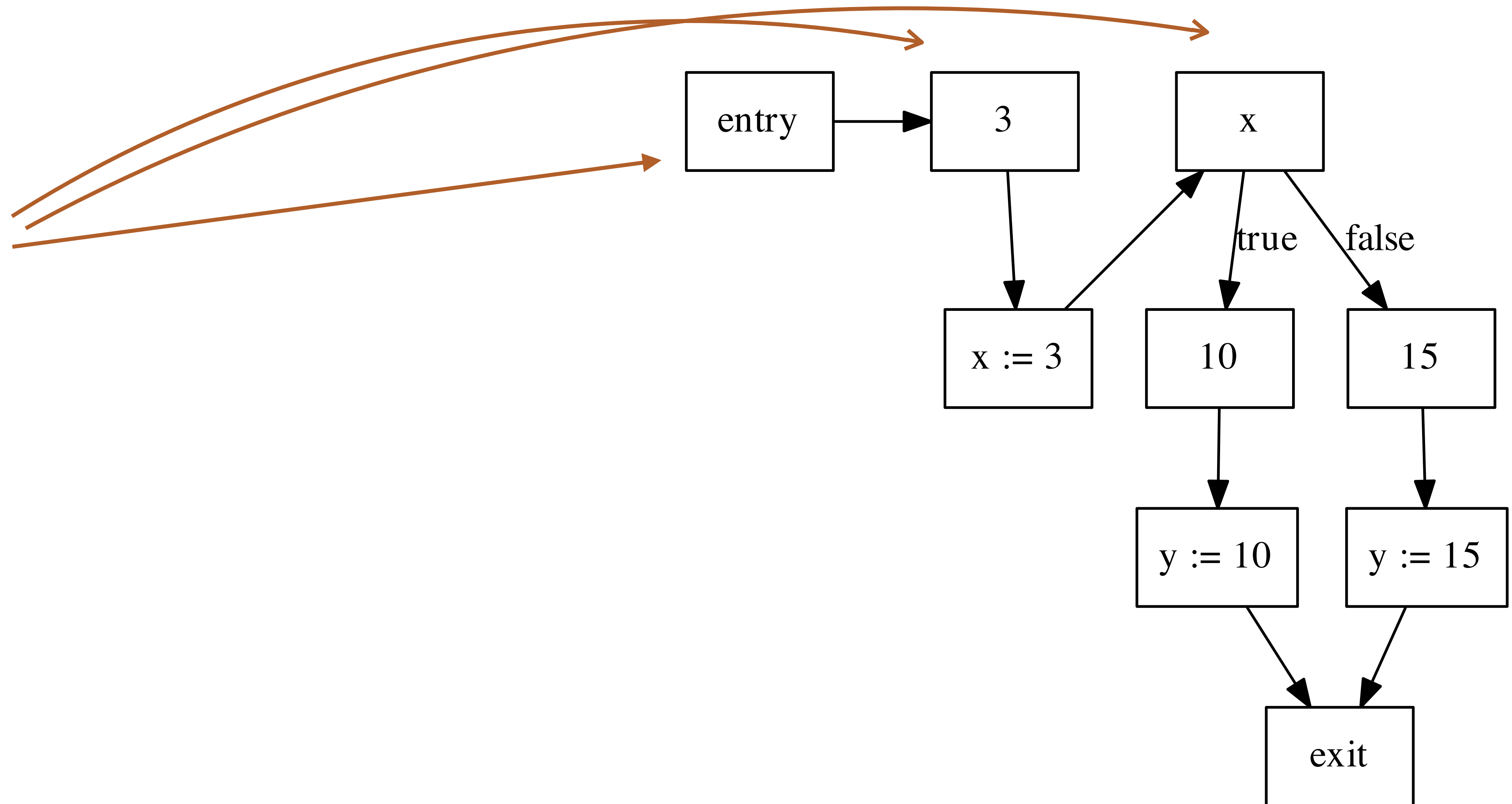




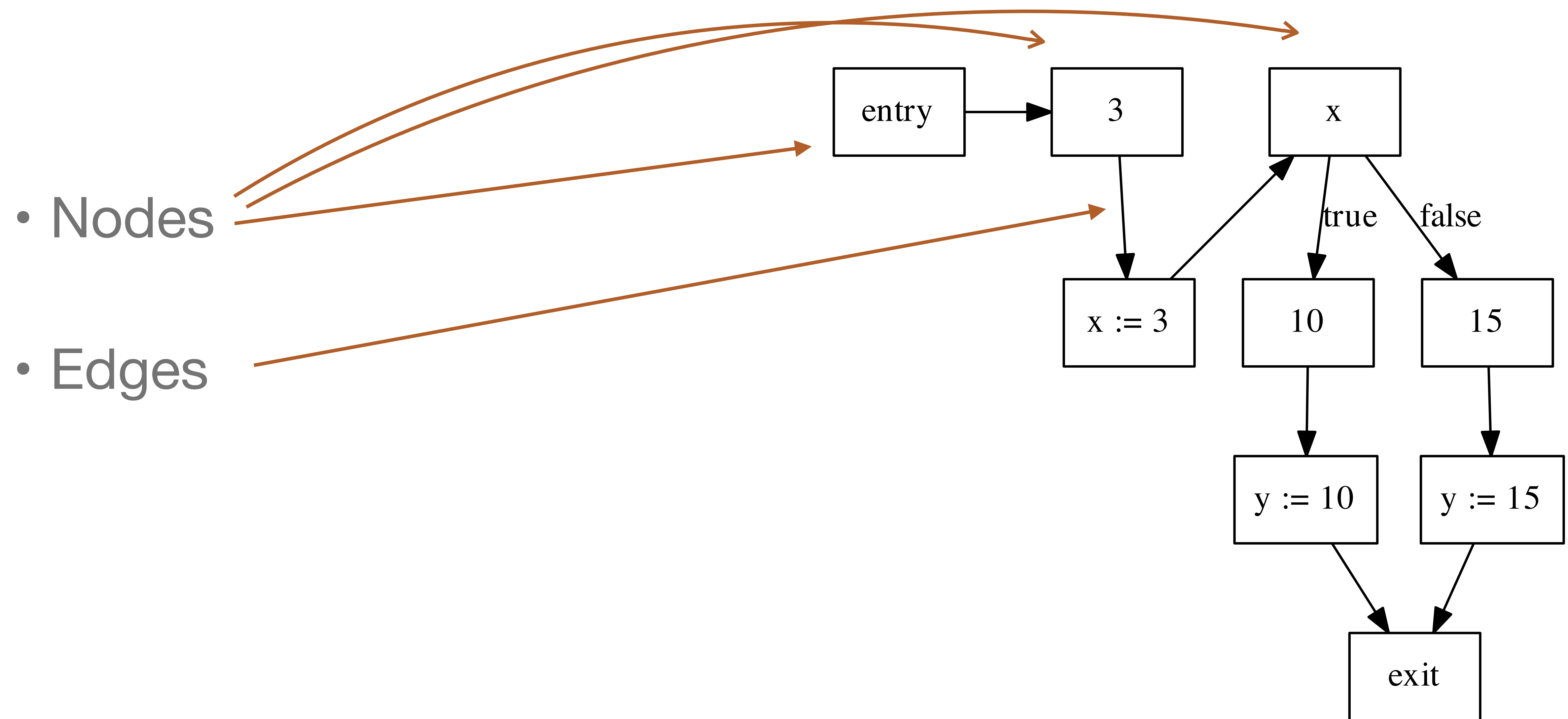
East Carolina University

First, define the basics!

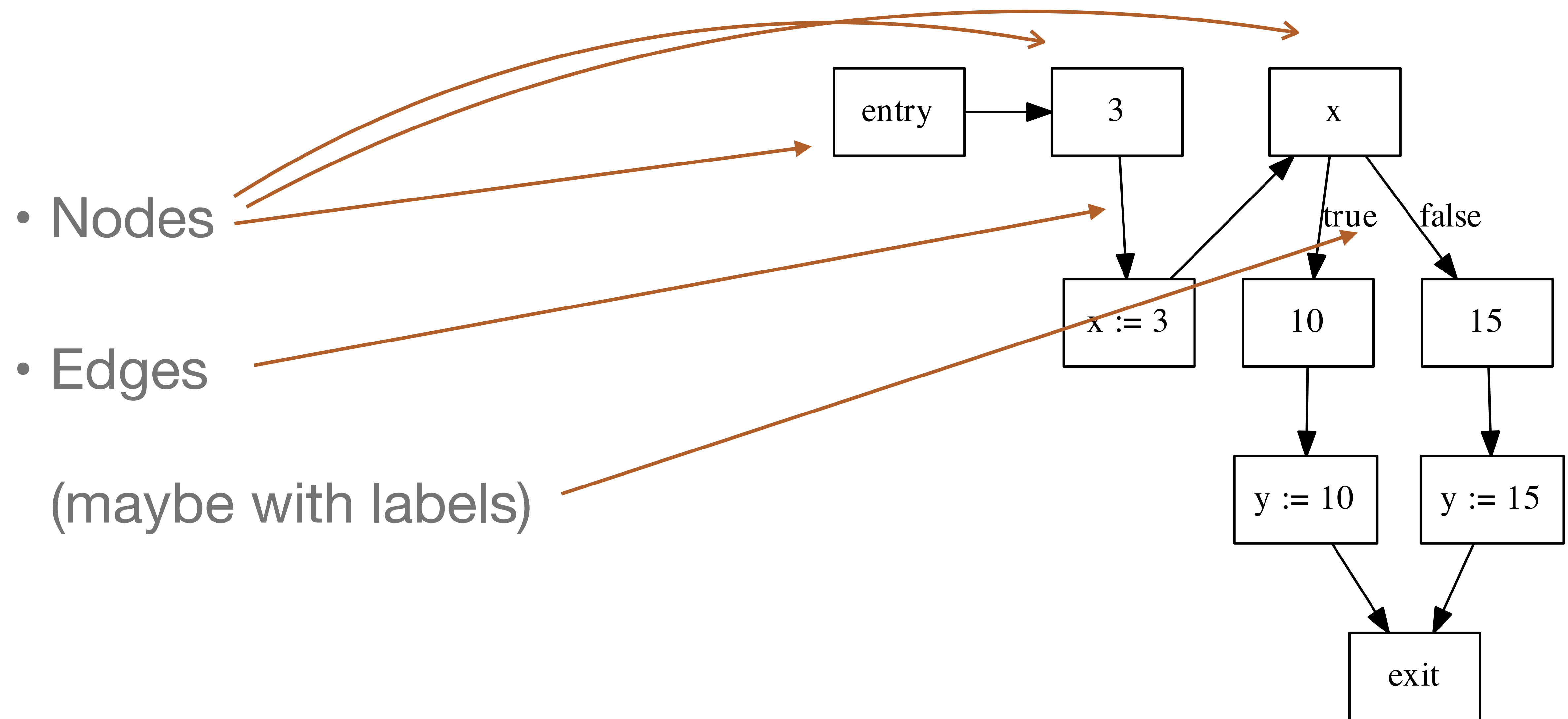
- Nodes



First, define the basics!

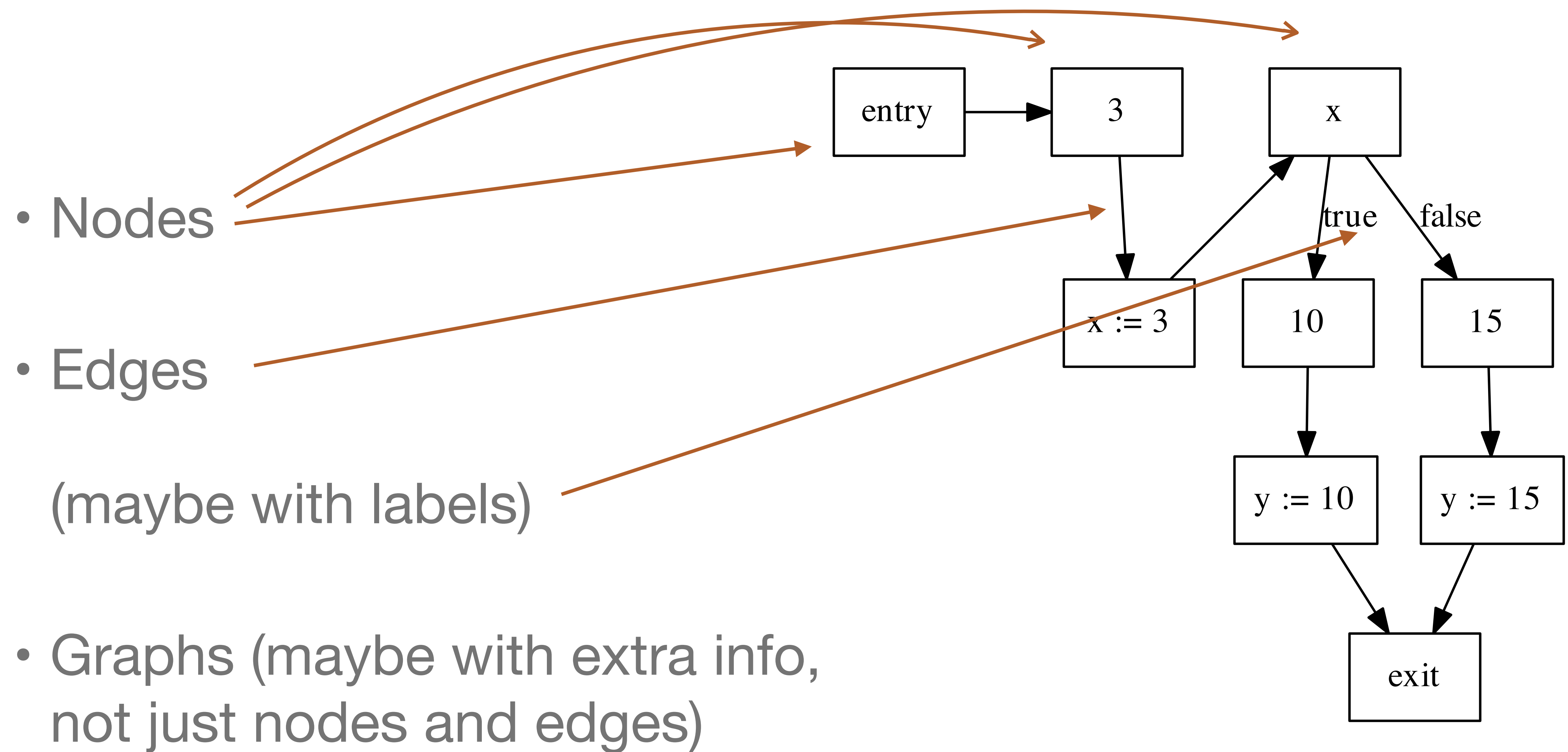


First, define the basics!



East Carolina University

First, define the basics!



CFG basics, for Pico (example from Rascal library)

- Four types of nodes: entry, exit, choice, and statement nodes
- Implicit edges based on graph over nodes, no labels
- Explicit tracking of entry and exit nodes

```
public data CFNode
   = entry(loc location)
   | exit()
   | choice(loc location, EXP exp)
   | statement(loc location, STATEMENT stat);

alias CFGraph = tuple[set[CFNode] entry, Graph[CFNode] graph, set[CFNode] exit];
```

Second, define the control flow

- Flow based on semantics of each construct

- May have a lot of boilerplate code

```
CFGraph cflowStat(s:asgStat(PicoId Id, EXP Exp)) {
   S = statement(s@location, s);
   return <{S}, {}, {S}>;
}

CFGraph cflowStat(ifElseStat(EXP Exp, list[STATEMENT] Stats1, list[STATEMENT] Stats2)){
   CF1 = cflowStats(Stats1); CF2 = cflowStats(Stats2);
   E = {choice(Exp@location, Exp)};
   return < E, (E * CF1.entry) + (E * CF2.entry) + CF1.graph + CF2.graph,
      CF1.exit + CF2.exit >;
}
```

East Carolina UNIVERSITY

Third (optional), do something useful with it!

```
public rel[stat, def] reachingDefinitions(
         rel[stat,var] DEFS, rel[stat,stat] PRED)
{
   set[stat] STATEMENT = carrier(PRED);
   rel[stat,def] DEF  = definition(DEFS);
   rel[stat,def] KILL = kill(DEFS);

   rel[stat,def] IN = {};
   rel[stat,def] OUT = DEF;

   solve (IN, OUT) {
            IN =  {<S, D> | int S <- STATEMENT,
            stat P <- predecessors(PRED,S), def D <- OUT[P]};
         OUT = {<S, D> | int S <- STATEMENT,
            def D <- DEF[S] + (IN[S] - KILL[S])};
   };
   return IN;
}
```

East Carolina UNIVERSITY

What if we want to work with another language?

- *May* be able to reuse base CFG definition (but maybe not)
- Cannot reuse flow definition (unless CFG def is the same and feature has identical semantics)
- Cannot easily reuse analysis (since CFG definition and semantics differ)

East Carolina UNIVERSITY

What if we want to work with another language?

- *May* be able to reuse base CFG definition (but maybe not)
- Cannot reuse flow definition (unless CFG def is the same and feature has identical semantics)
- Cannot easily reuse analysis (since CFG definition and semantics differ)

  So, we write the entire thing over again
  (and again, and again…)

East Carolina University

Motivating Questions

- Can we create a DSL to make this process easier?
  - Provide uniform definition of control-flow graphs
  - Generate CFG extraction code, including all the boilerplate
  - Provide a declarative definition that *should* be easier to keep up to date as the language evolves
- What shouldn't the DSL do?
  - Don't try to do everything, give an "escape hatch" into Rascal

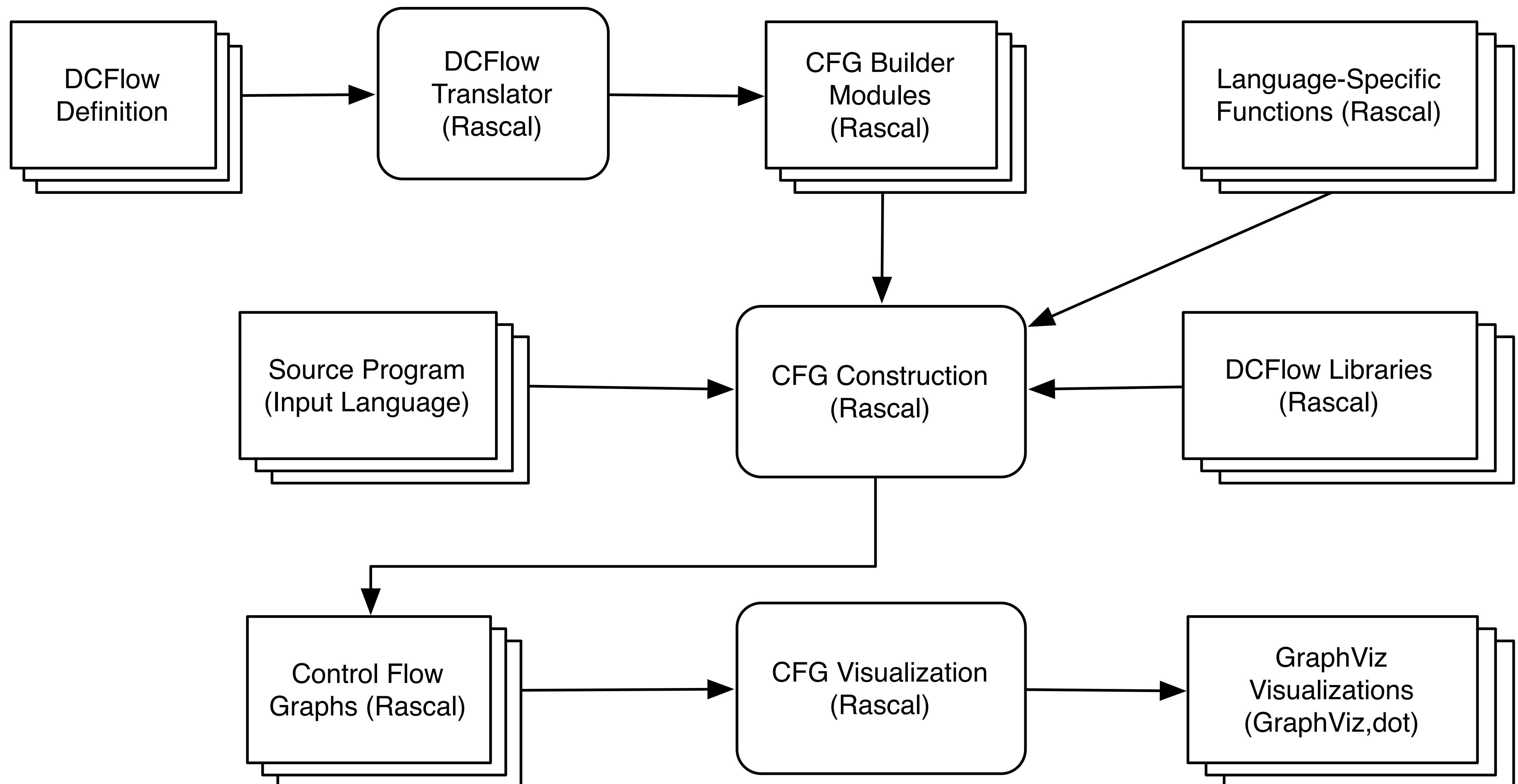
East Carolina UNIVERSITY

DCFlow: Declarative Control Flow

- Declarative DSL for defining control flow rules
- Generates Rascal code to build intraprocedural control flow graphs with reusable library of CFG concepts
- Provides basic visualization to allow graphs to be rendered in GraphViz dot
- Provides *ignore* mechanism to indicate which language constructs we are *not* trying to define
- IDE provides basic checking to aid user (with more coming)

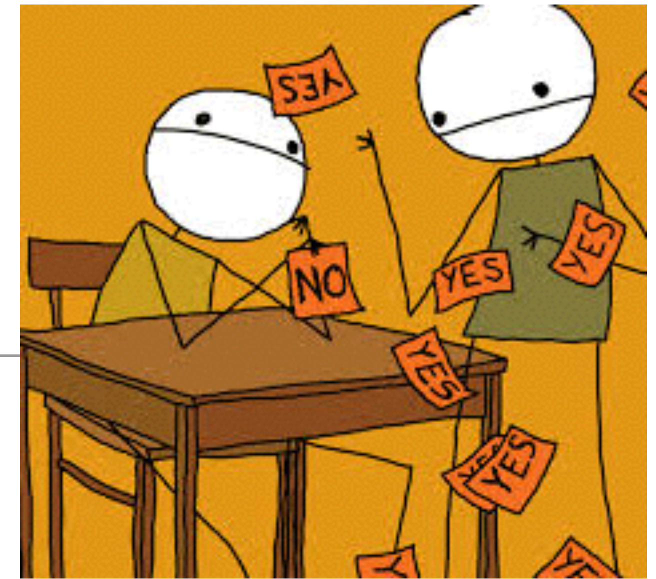
East Carolina UNIVERSITY

DCFlow Architecture



East Carolina UNIVERSITY

Design Decisions



- Focus on abstract syntax trees (should *almost* work on Rascal concrete syntax, but there are some differences)
- Leverage reified types for generation and checking
- Try to ensure added features are general — don't want to add something just because PHP or Java needs it
- Make sure generated code is understandable — it should look close to what you would write yourself

The basics: entries, exits, and basic flow

```
module Pico
ast demo::lang::Pico::Abstract;
import lang::pico::CFGBase;
context PROGRAM::program;
astType PROGRAM;

rule PROGRAM::program = entry(stats), exit(stats);
rule EXP::add = entry(left) --> right --> exit(self);
rule EXP::sub = entry(left) --> right --> exit(self);
rule EXP::conc = entry(left) --> right --> exit(self);
rule EXP::id = entry(self), exit(self);
rule EXP::strCon = entry(self), exit(self);
rule EXP::natCon = entry(self), exit(self);
rule STATEMENT::asgStat = entry(exp) --> exit(self);
```

East Carolina UNIVERSITY

Reified types in Rascal

```
public data EXP =
       id(PicoId name)
     | natCon(int iVal)
     | strCon(str sVal)
     | add(EXP left, EXP right)
     | sub(EXP left, EXP right)
     | conc(EXP left, EXP right)
     ;
```

```
cons(
   label(
     "add",
     adt(
       "EXP",
       [])),
   [
     label(
       "left",
       adt(
         "EXP",
         [])),
     label(
       "right",
       adt(
         "EXP",
         []))
   ],
   [],
   (),
   {}),
```

East Carolina UNIVERSITY

Introducing some useful shorthands

```
module Pico
ast demo::lang::Pico::Abstract;
import lang::pico::CFGBase;
context PROGRAM::program;

rule PROGRAM::program = ^$stats;
~~rule PROGRAM::program = entry(stats), exit(stats);~~
rule EXP::add EXP::sub EXP::conc = ^left -> right -> $self;
~~rule EXP::add = entry(left) --> right --> exit(self);~~
rule EXP::id EXP::strCon EXP::natCon = ^$self;
rule STATEMENT::asgStat = ^exp -> $self;
~~rule STATEMENT::asgStat = entry(exp) --> exit(self);~~
```

East Carolina University

Creating nodes and labeling edges

```
rule STATEMENT::ifElseStat = ^exp,
  exp -conditionTrue-> exit(thenpart,exp),
  exp -conditionFalse-> exit(elsepart,exp);
```

```
rule STATEMENT::whileStat =
  ^$exp -conditionTrue-> body -backedge-> exp,
  exp -conditionFalse-> create(footer);
```

East Carolina UNIVERSITY

Structured and unstructured jumps

```
context Script::script ClassItem::method Stmt::function;
structured target \break \continue;
unstructured target goto;

rule Stmt::goto = jump(\label),^$self;

rule Stmt::\while = create(footer),
           jumpTarget(cond,\continue),
           jumpTarget(footer,\break),
           ^$cond -conditionTrue-> body -backedge-> cond,
           cond -conditionFalse-> footer;

rule Stmt::\break = entry(breakExpr,self) --> $self,
           jump(breakExpr,\break);
rule Stmt::\continue = entry(continueExpr,$self) --> self,
           jump(continueExpr,\continue);
```

How does the generated code look?

```
tuple[FlowEdges,LabelState] internalFlow(EXP item:add(EXP left,EXP right),
   LabelState ls)
{
    FlowEdges edges = { };
    < edges, ls > = addEdges(edges, ls, left);
    < edges, ls > = addEdges(edges, ls, right);
    for(exlab <- exit(left,ls)) {
      < edges, ls > = linkItemsLabelLabel(edges, ls, exlab, entry(right,ls) );
   }
    for(exlab <- exit(right,ls)) {
      < edges, ls > = linkItemsLabelLabel(edges, ls, exlab, item@lab );
   }
    return < edges, ls >;
}
```

East Carolina UNIVERSITY

Evaluation: Comparing to Custom Tools

- Hand-built CFG extractor for Pico: 45 lines of code, 11 for headers and declarations, 34 for extraction rules
- DC-Flow Pico definition: 4 header lines, 6 rules — but generates 408 lines of Rascal
- PHP: 1583 lines of Rascal for hand-built, currently 85 lines of code (fewer rules, some of these are for rules on multiple lines), generates 2714 lines of Rascal, plus around 230 lines for parts written directly in Rascal

Evaluation: Comparing to DeFacto

- Very similar required effort for small languages
- Hard to tell for larger languages with more involved control flow, need to resurrect DeFacto to perform comparison
- DeFacto works over concrete syntax, DCFlow works over abstract syntax
- DeFacto is more general fact extraction language, DCFlow is more focused on control flow, has more tailored syntax, has access to Rascal features

East Carolina UNIVERSITY

Evaluation: Can we use it in an analysis?

- Implemented reaching defs analysis to show this works in theory (code given in paper)
- Should be able to evaluate soon in PHP to verify it works in practice

East Carolina University

Current limitations

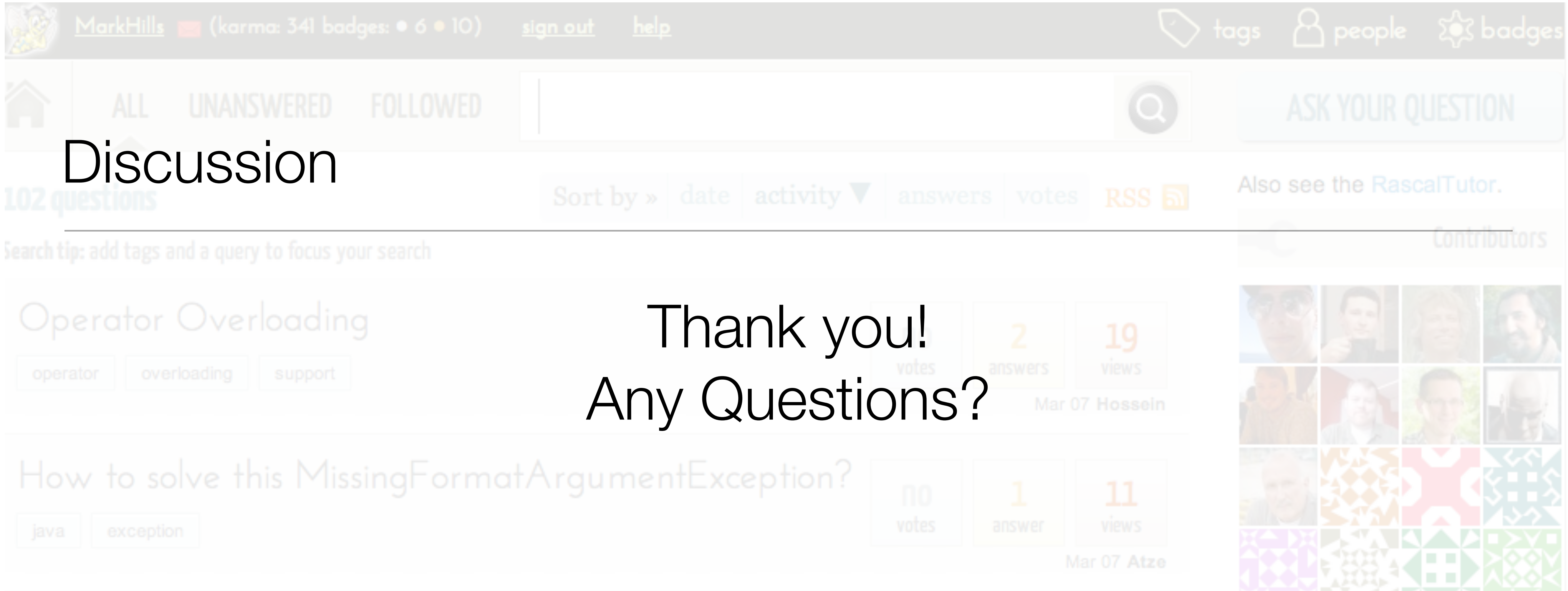
- Exceptions
  - Adding try/catch is the easy part
  - Adding exception flow edges in a generic way is harder
- No current support for interprocedural control flow graphs
- Current implementation needs more convenience functions to be easily usable as a black box

East Carolina UNIVERSITY

Future work

- Add support for exceptions (if this can be added generically)
- Merge generated CFG code into PHP AiR framework
- Integrate DC-Flow generation with Rascal Resources framework
- Add improved support for visualization

Discussion



Thank you!
Any Questions?

- Rascal: http://www.rascal-mpl.org
- Me: http://www.cs.ecu.edu/hillsma

East Carolina University

Related work

- “Extensible intraprocedural flow analysis at the abstract syntax tree level”, Söderberg, Ekman, Hedin, Magnusson
    - Uses attribute grammars to represent control flow
    - Reference attributes represent edges
    - Collection attributes represent inverse relations (e.g., pred)
    - Higher-order attributes allow building new AST nodes (e.g., entry and exit)

East Carolina UNIVERSITY

Related work

- Spoofax: NaBL, language for incremental type checking
- DHAL and variants for data flow analysis
- Related conceptually — use domain-specific languages for specific analysis-related tasks