PLT COMS 4115

Empath

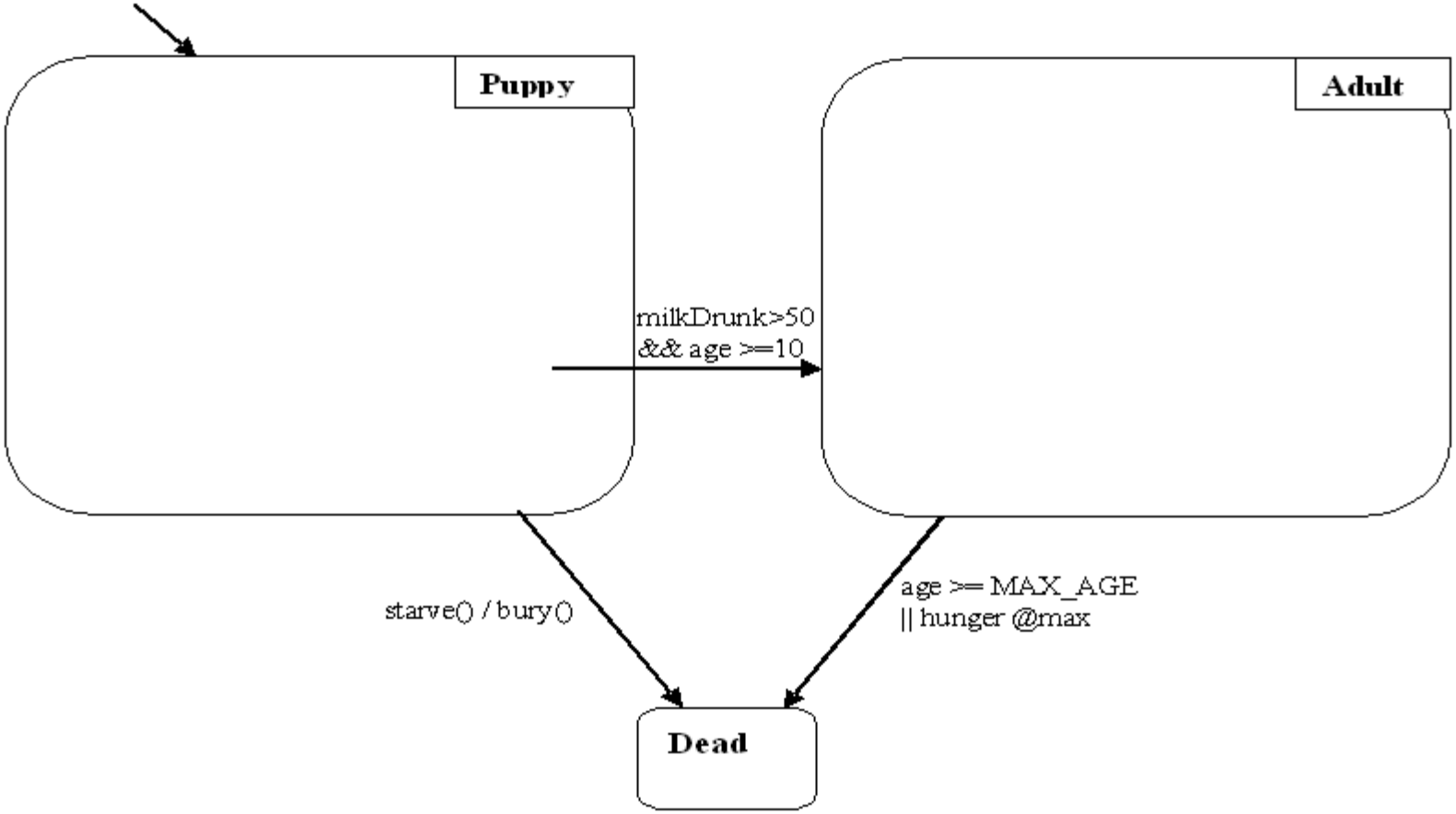
- Jeremy Posner
- Nalini Kartha
- Sampada Sonalkar
- William Mee

Empath

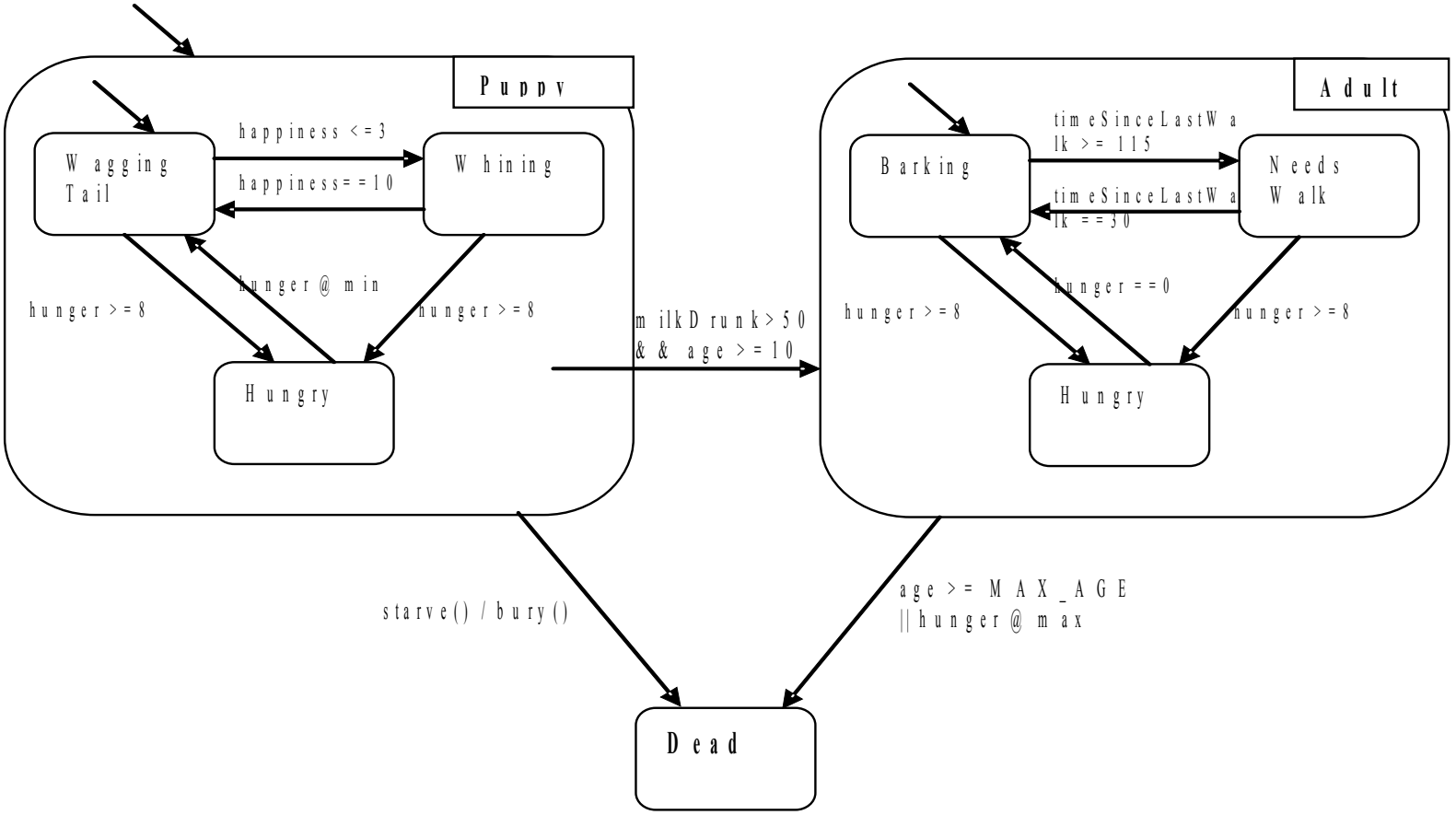
A language for modeling digital pets.

- Finite state machine based
- Compiled
- Static scoping

Dog Example


Puppy
Adult
milkDrunk>50
&& age >=10
starve() / bury()
age >= MAX_AGE
|| hunger @max
Dead

Dog – State Transition Diagram



Dog – The Program

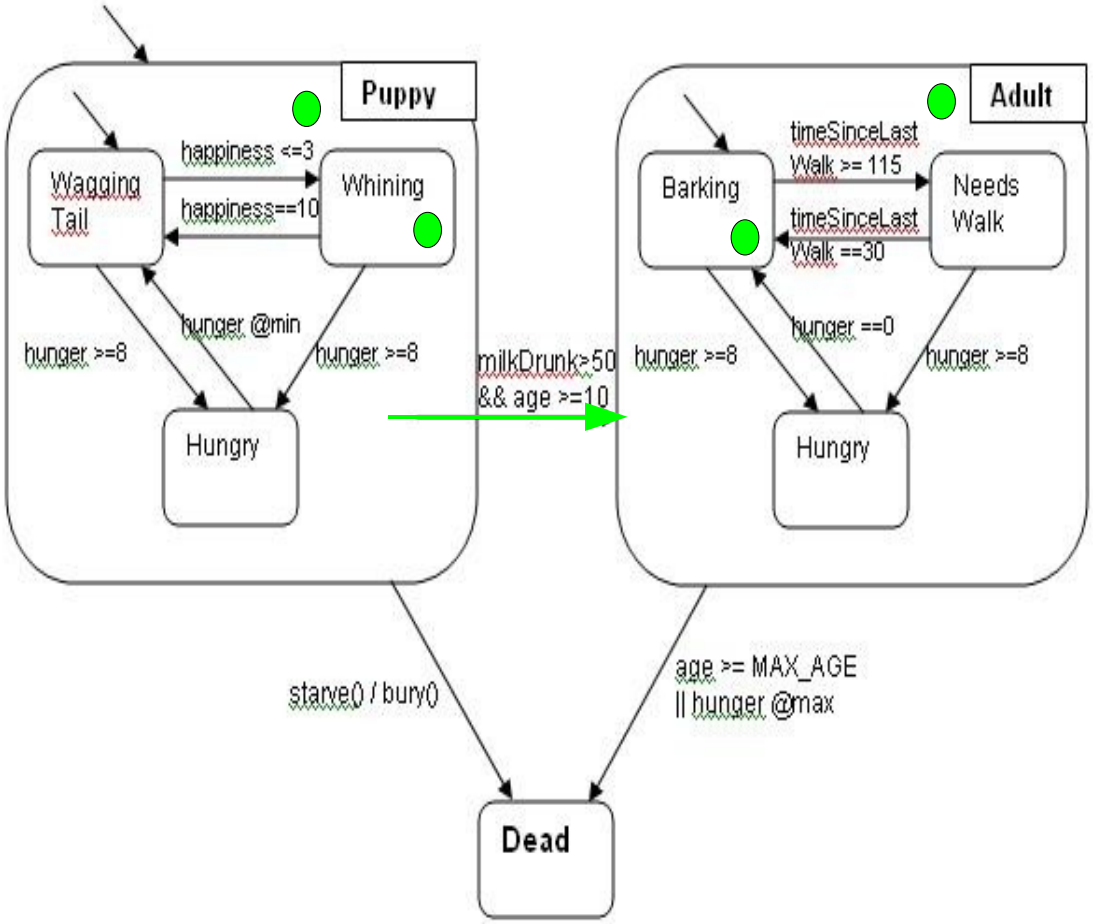
```
entity Dog label "mutt" {
          range [0:10] hunger = 5;      float age = 0.0;
          function void onClockTick() {  hunger++;          age+=0.2;  }
          event feed(int quantity)  {  hunger-=quantity;  }
          trigger starve() {
                    if (hunger>16)      return true;
                    else      return false;
          }
          function void bury() {  output("the dog has been buried");  }
          state init DogPuppy, DogAdult, DogDead;
          transition DogPuppy to DogDead if (starve()) / bury() ;
          DogPuppy label "puppy" {
                    range [0:100] milkDrunk=0;
                    .......
          }
}
```

Special Constructs

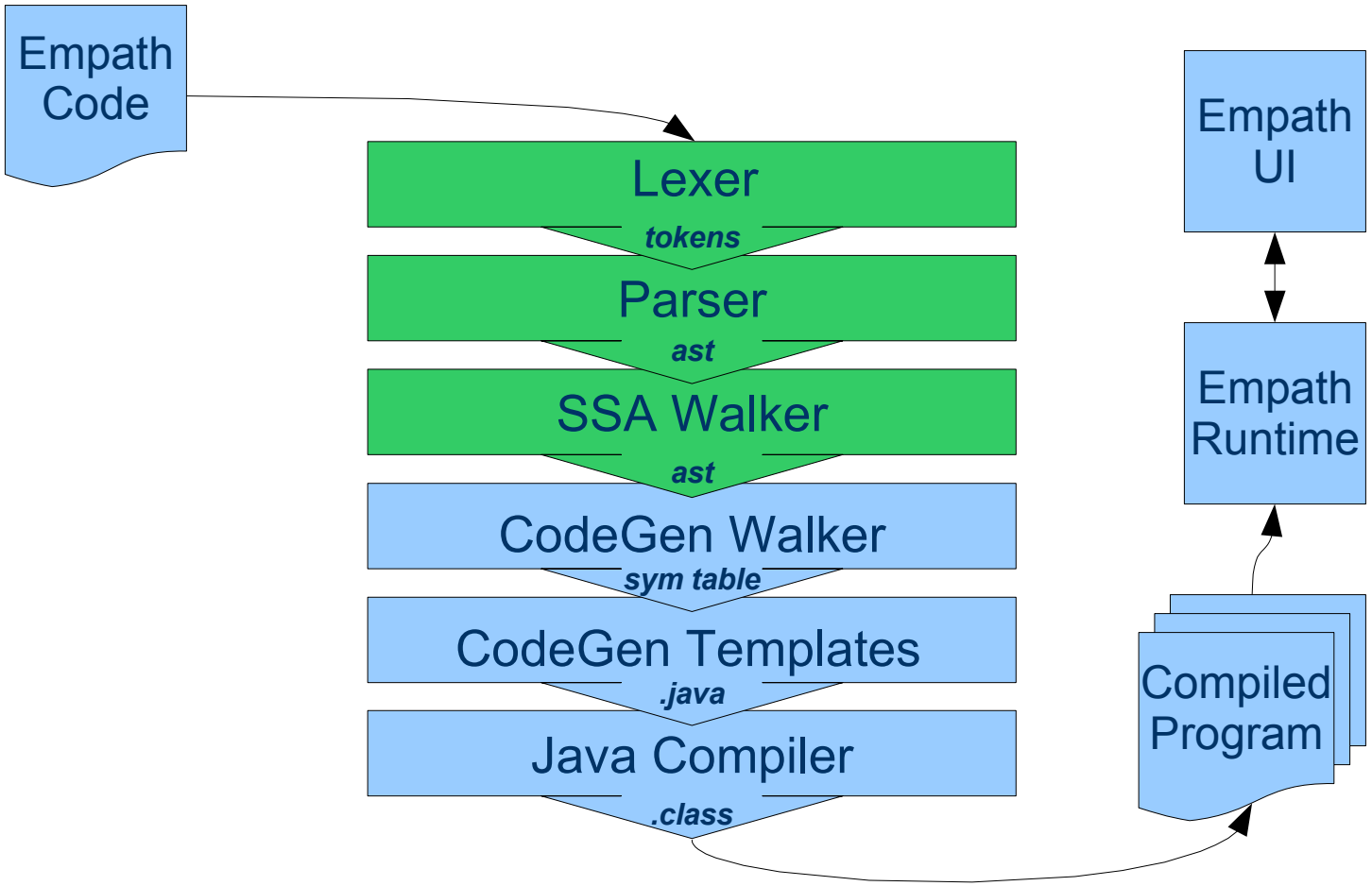
- states & transitions
- range datatype & @max, @min operators
- trigger functions
- event functions
- onEntry & onExit
- onClockTick & tick keyword

Execution Semantics

- Transitions in outermost FSM evaluated first
- Preemptive



Architecture



Static Semantic Analysis

- Type checking
- Declaration of variables, functions, and states
- Multiple initial states
- Consistency of function calls
- Function definitions
- Consistency of statements and expressions

```
function void dummy(int a,
state egg, chicken;
state int b, string c) {egg;
        b++;a;
}
function void test() {
        int x = 10;
        float y = 2.3;
        string z = null;
        dummy(x ,z , y );
}
```

Static Semantic Analysis

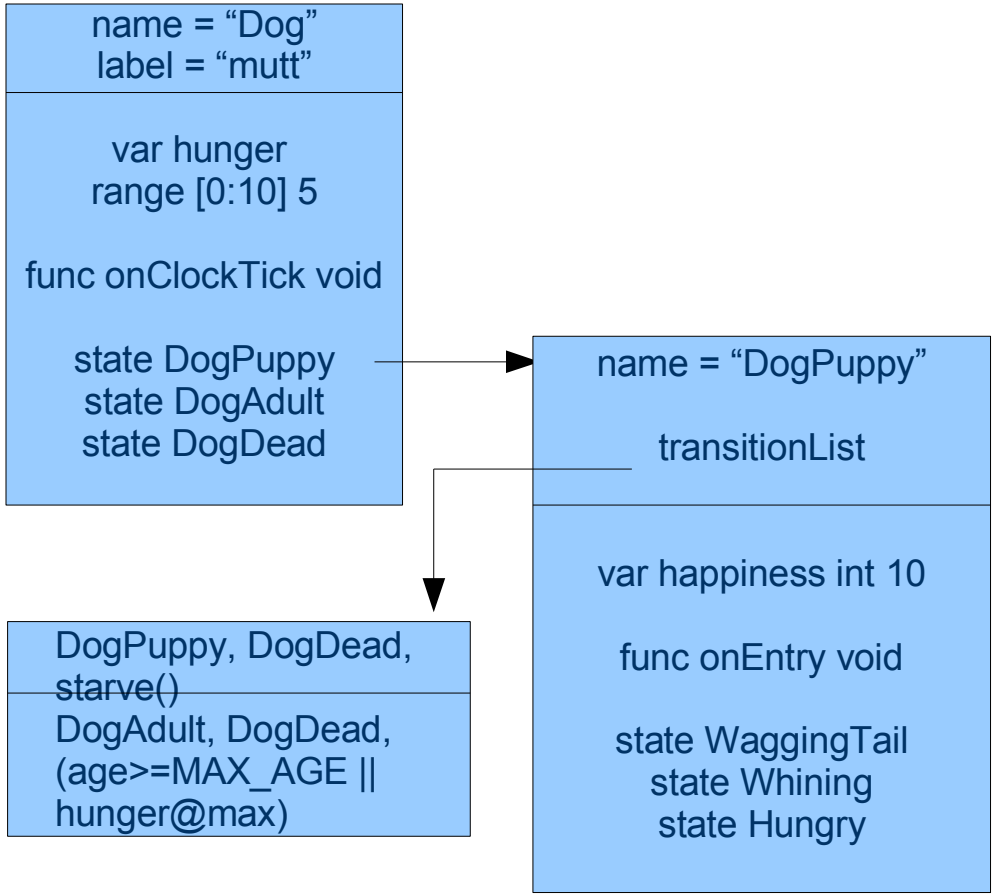
- Restriction on triggers
- Transition definition
  - –fromState, toState
  - –condition
  - –action

```
entity cycle {
   int age;

   state small;
   state big;

   transition x to y if (age+5);
}
```

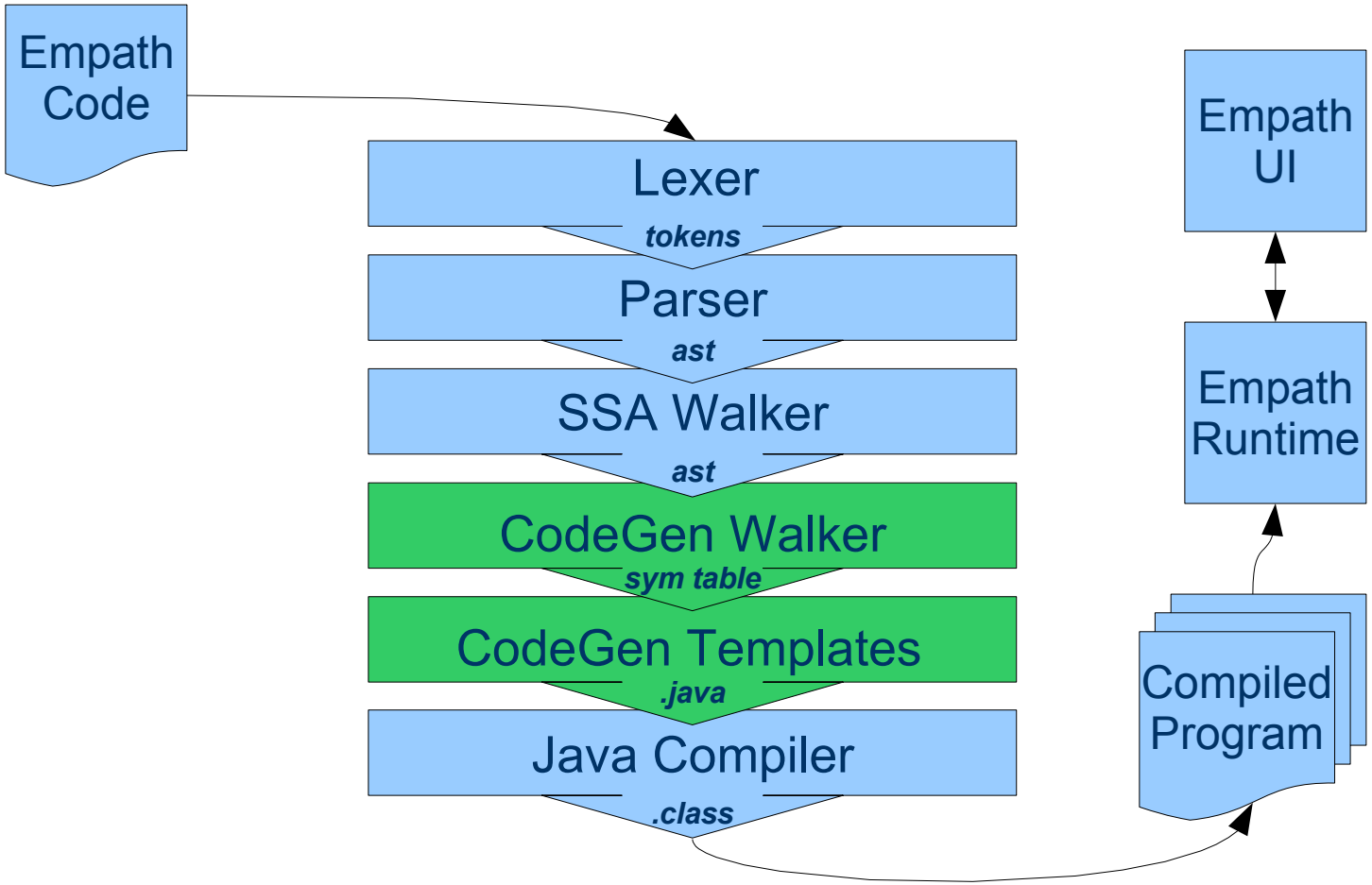
```
entity cycle {
   int age;

   trigger1() {
      age += 1;
      age %= 10;
      x = (age > 5);
      return x;
    }
  }
```

Symbol Table

- Single namespace
- Hierarchy
  - Function local var
  - State definition



Code Generation and Runtime



Generated Code

- Java's object structure – A square peg for a round hole.
  - Transitions need to morph source to target
  - The entity itself should never change
- The generated code falls into two categories:
  - Entity
    - Contains code to populate State Tree
  - State Classes
    - One for each state
    - Each State Class extends the parent State Class

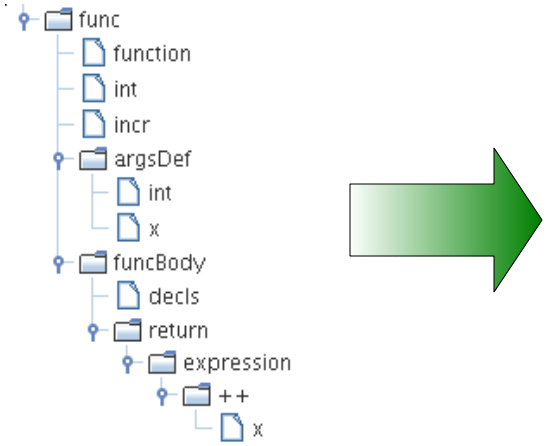
Code Generation

Combination of two approaches:

- Second “code generation” tree walk
- Template language produces .java files

Code Generation Walker

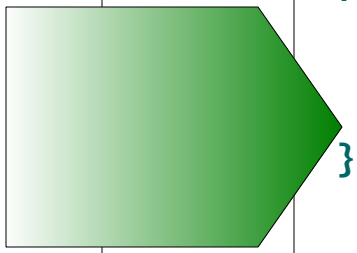
- Second “collapsing” walk of AST
- Enhancement of symbol table
- Empath functions become strings of Java code
- Important transformations



“int incr(int x) {return x++;}”

Code Generation Walker Transformations

```
function void onClockTick(){
  if (tick 2) {
    hunger++;
    age += 0.2;
  } else {
    happiness--;
  }
}
```



```
public void onClockTick(int
tick) {
  if (((tick%2)==0)) {
    hunger.incrementValue();
    age += 0.2;
  } else {
    happiness.decrementValue();
  }
  super.onClockTick(tick);
}
```

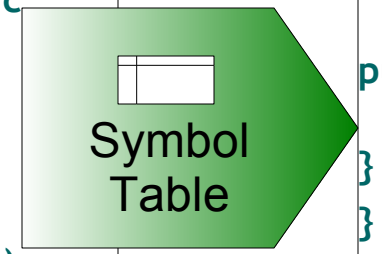
Code Generation Templates

- Templates used to generate .java files
- Uses the *String Template* project
- Target code easy to generate (by design)
- Just two templates
- Populates templates from symbol table

Code Generation 2: Template Language

```
public class $class$ extends
$superclass$ {

$variable:{protected static
$it$;}; separator="\n"$

public $class$ () {
  $if(state_label)$
  setLabel("$state_label$");
  $endif$
}
}
```

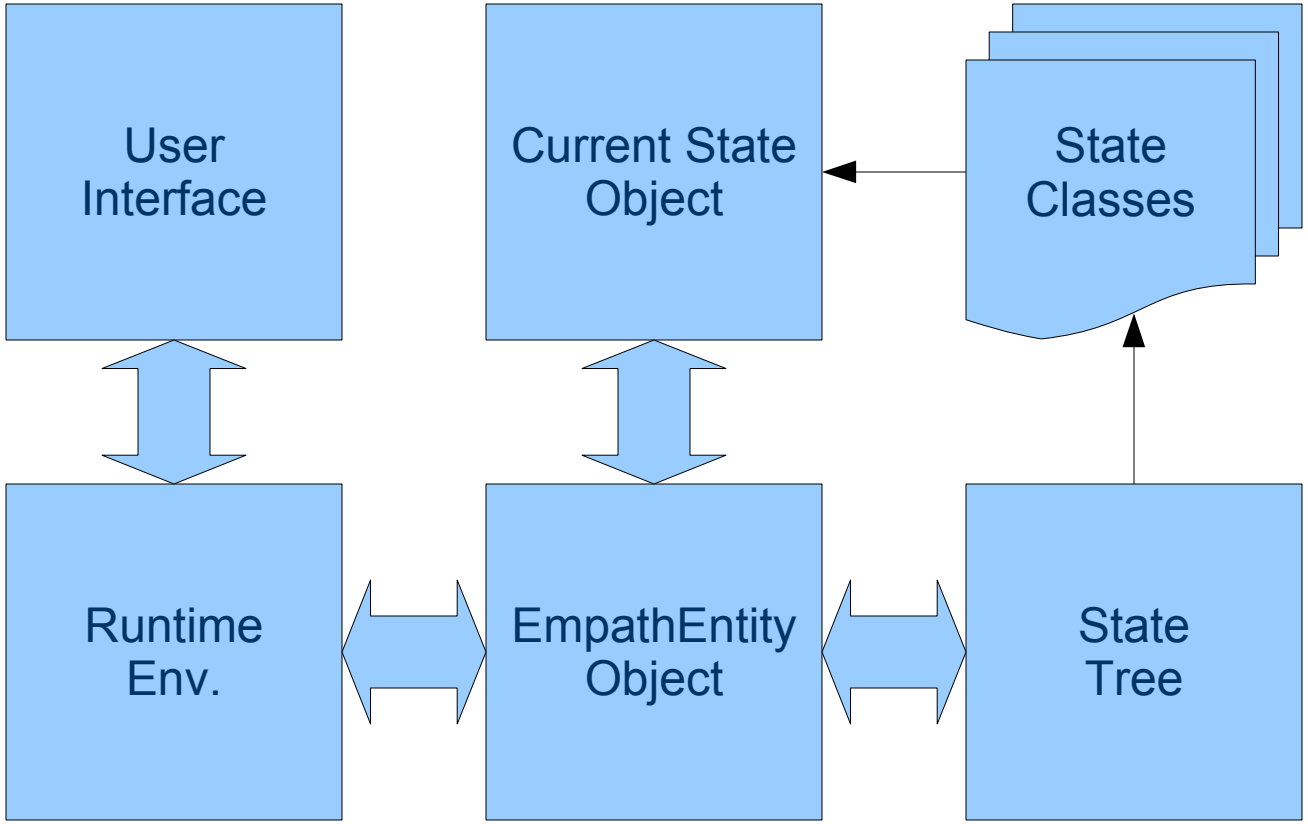
Template



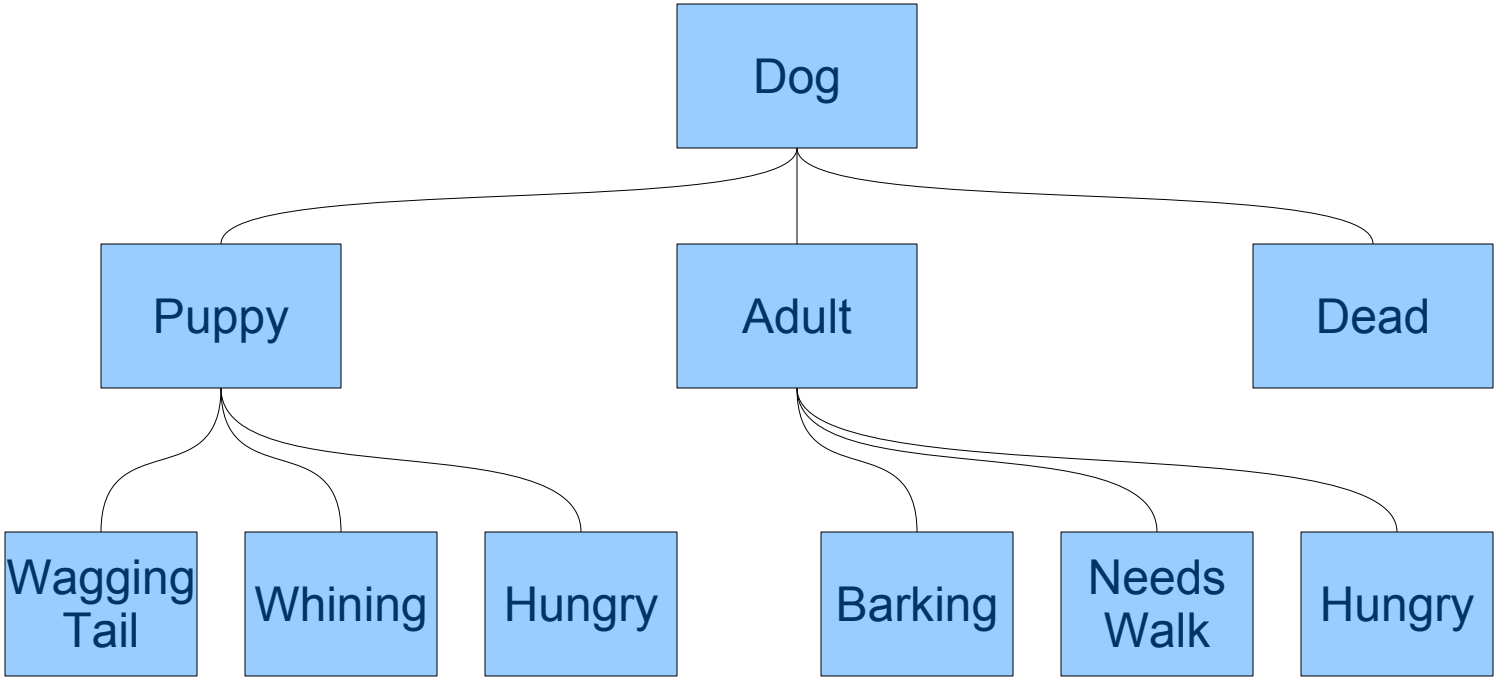

```
public class DogPuppy extends
Dog {
protected static Range
milkDrunk=new Range(0,100,0);

public DogPuppy () {
  setLabel("puppy");
}
}
```

Generated .java File

Runtime Environment



The State Tree



Trigger & Transition Handling

- Runtime signals Entity to evaluate transitions
- Entity calls current State Tree node's evaluate method
  - State Tree Node loops through all of the current state's triggers
  - Returns either the State Class for a new state or null
- Entity instantiates new State Class, replacing former State.
- Repeats instantiation for init states as needed

Automated Testing

EmpathLexerTest: lexical analysis

EmpathParserTest: parser

LineNumberTest: testing of error reporting

FuncSSATest: ssa for functions

StateTransTest: ssa for state transitions

CodeGenWalkerTest: func code generation

Lessons Learned: Technical

- Infrastructure (cvs, ide) was important
- Learned about compilers, ASTs etc
- Learned non-compiler subjects too
- Test-orientated development

- Our language was unexpectedly ambitious

Lessons Learned: Team Work

- Divided work well
- Could work independently
- Team came together
- Quality team

- Difficult to coordinate
- Differing work styles
- Differing commitment to project
- Pressure from other courses and outside

Credits

Jeremy: architecture, target code, user interface, runtime

Nalini: lang design, parser, walker, ssa, unit testing, presentation

Sampada: lang design, parser, walker, unit testing, functional code gen, documentation

Will: lexer, code templates, testing, project management, infrastructure