A USEFUL BOUNDED RESOURCE FUNCTIONAL LANGUAGE

ME $\longrightarrow$ MIKE BURRELL (MBURREL@UWO.CA)
MARK DALEY (DALEY@CSD.UWO.CA)
JAMIE ANDREWS (ANDREWS@CSD.UWO.CA)

UNIVERSITY OF WESTERN ONTARIO (CANADA)

SOFSEM 2008

SAFETY-CRITICAL SOFTWARE

- MEDICAL EQUIPMENT
- NUCLEAR POWER PLANT CONTROLS
- AUTOMOTIVE CONTROLS
- INDUSTRIAL CONTROLLERS

FUNCTIONAL PROGRAMMING

- ☐ NO VARIABLES
- ☐ NO SIDE-EFFECTS
- ☐ NO STATEMENTS
- ☐ "ALGEBRAIC" DATA STRUCTURES

FUNCTIONAL PROGRAMMING

```
data Bool
    = True
    | False
data List a
    = Nil
    | Cons a (List a)
```

- ☐ NO VARIABLES
- ☐ NO SIDE-EFFECTS
- ☐ NO STATEMENTS
- ☐ "ALGEBRAIC" DATA STRUCTURES

FUNCTIONAL PROGRAMMING

```
data Bool
    = True
    | False
data List a
    = Nil
    | Cons a (List a)

empty :: (List a) → Bool
empty list = list {
    Nil → True;
    Cons _ _ → False;
}
```

- ☐ NO VARIABLES
- ☐ NO SIDE-EFFECTS
- ☐ NO STATEMENTS
- ☐ "ALGEBRAIC" DATA STRUCTURES

CA

- FUNCTIONAL IN STYLE
- NOT TURING-COMPLETE
  - NO GENERAL RECURSION
- CATAMORPHISM CONSTRUCT
  - GENERALIZED FOLDS

```
f x = x * x

g y x = if x < y
    then y + g y (x + 1)
    else h (y - x)

h x = g (f x) x
```

```
f x = x * x

g y x = f y + x

h x = g (f x) x
```

```
f x = x * x

g y x = if x < y
    then y + g y (x + 1)
    else h (y - x)

h x = g (f x) x
```

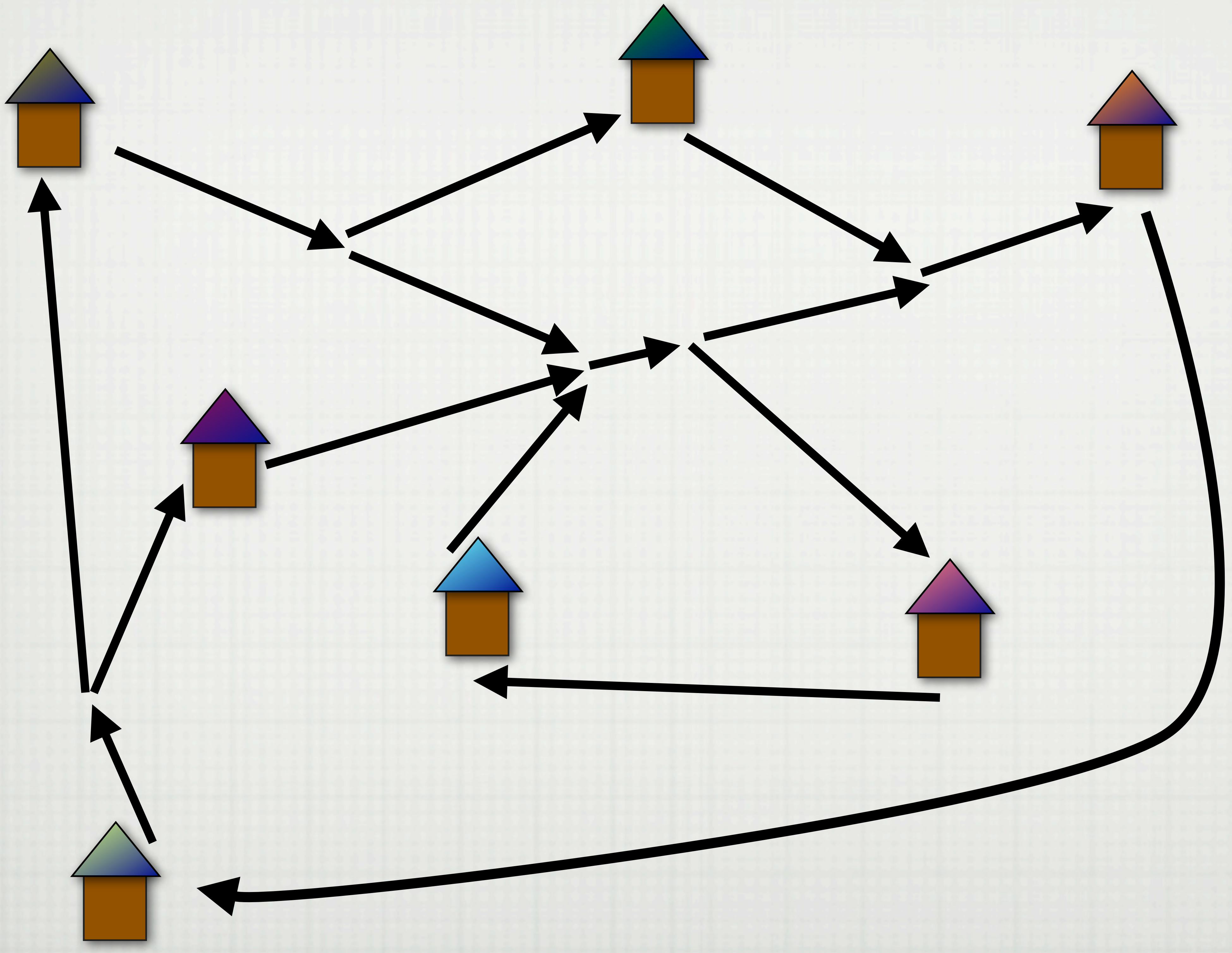
```
f x = x * x

g y x = f y + x

h x = g (f x) x
```

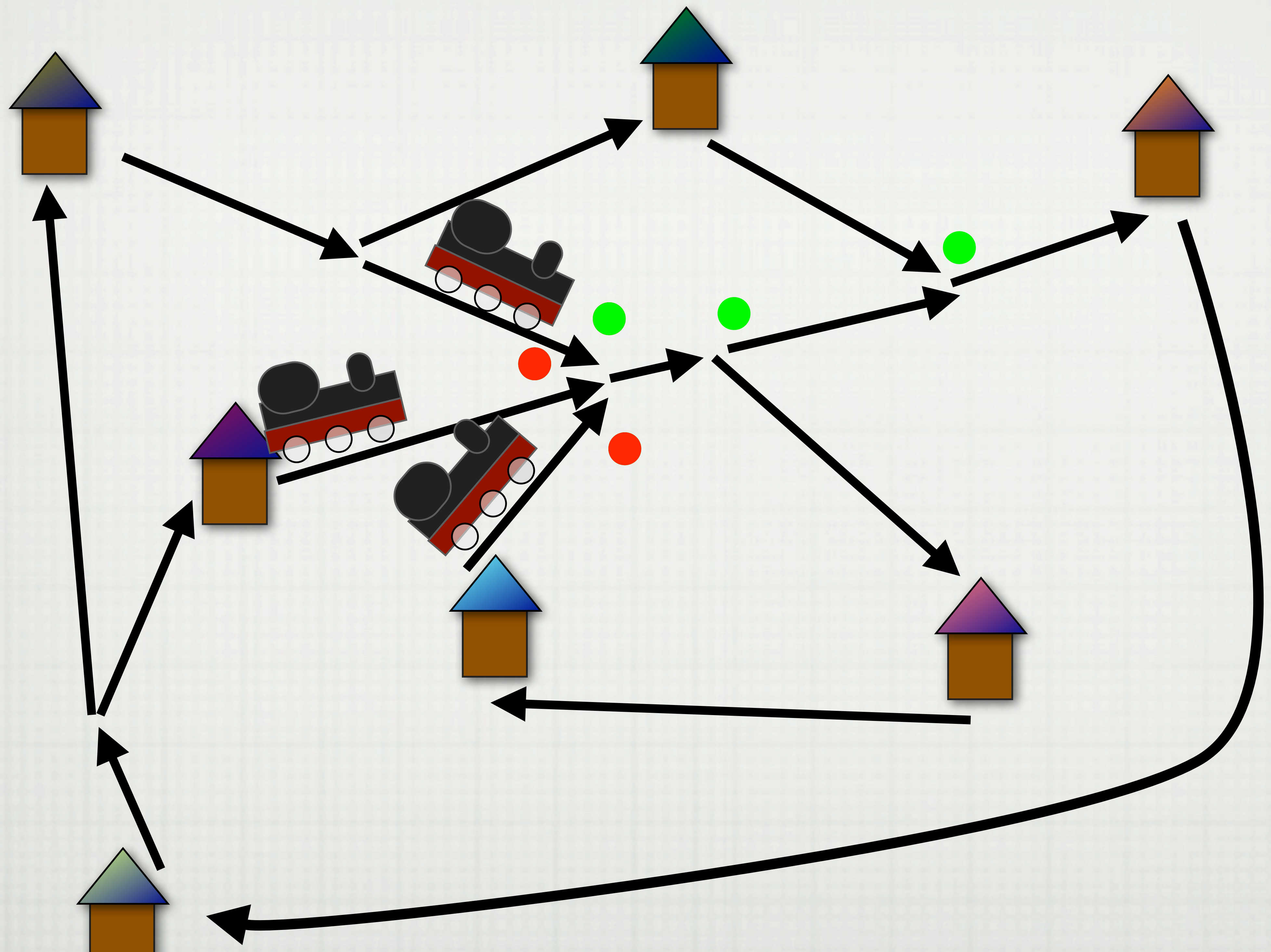
```
f x = x * x

g y x = if x < y
    then y + g y (x + 1)
    else h (y - x)

h x = g (f x) x
```

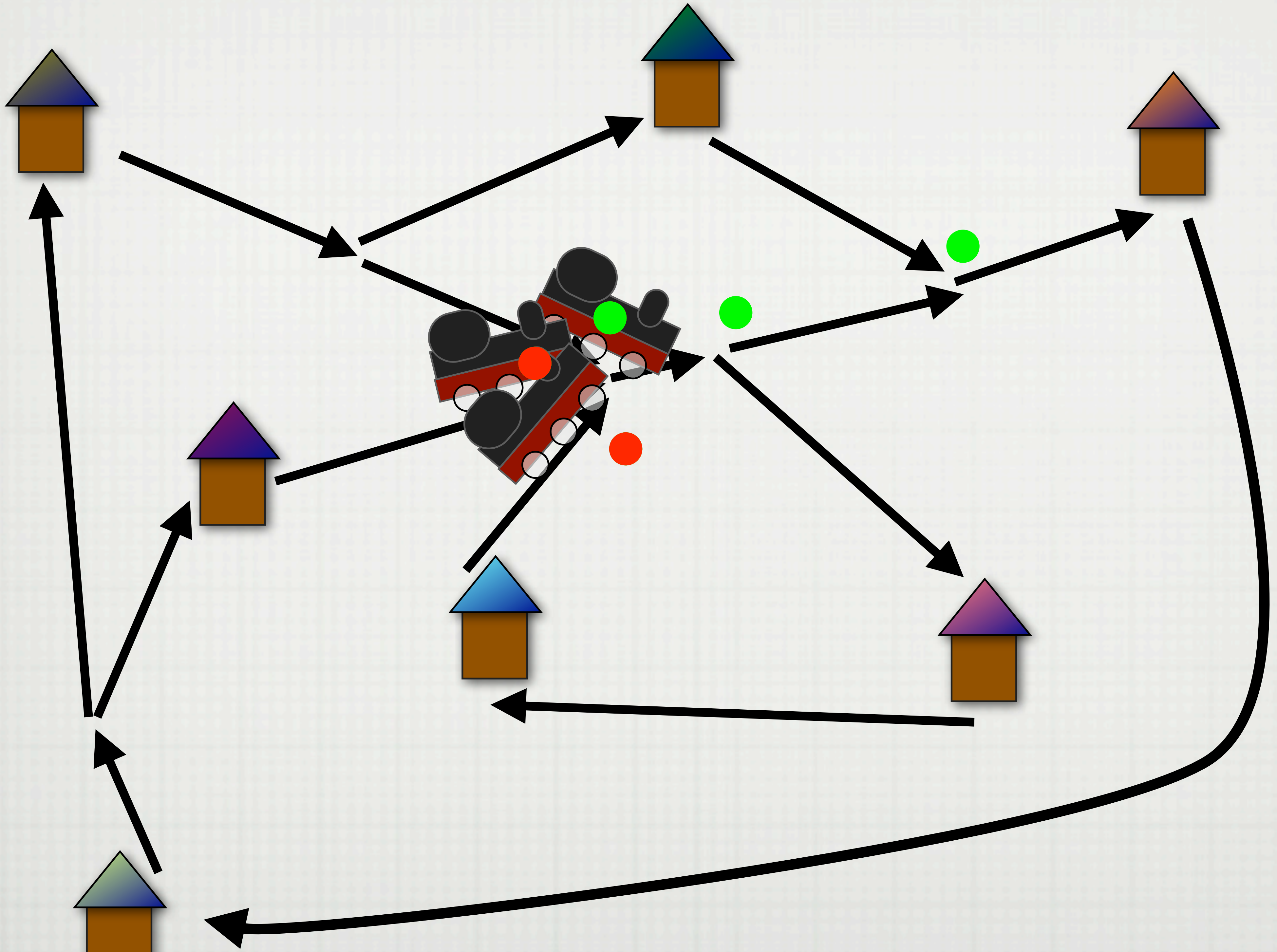
```
f x = x * x

g y x = f y + x

h x = g (f x) x
```

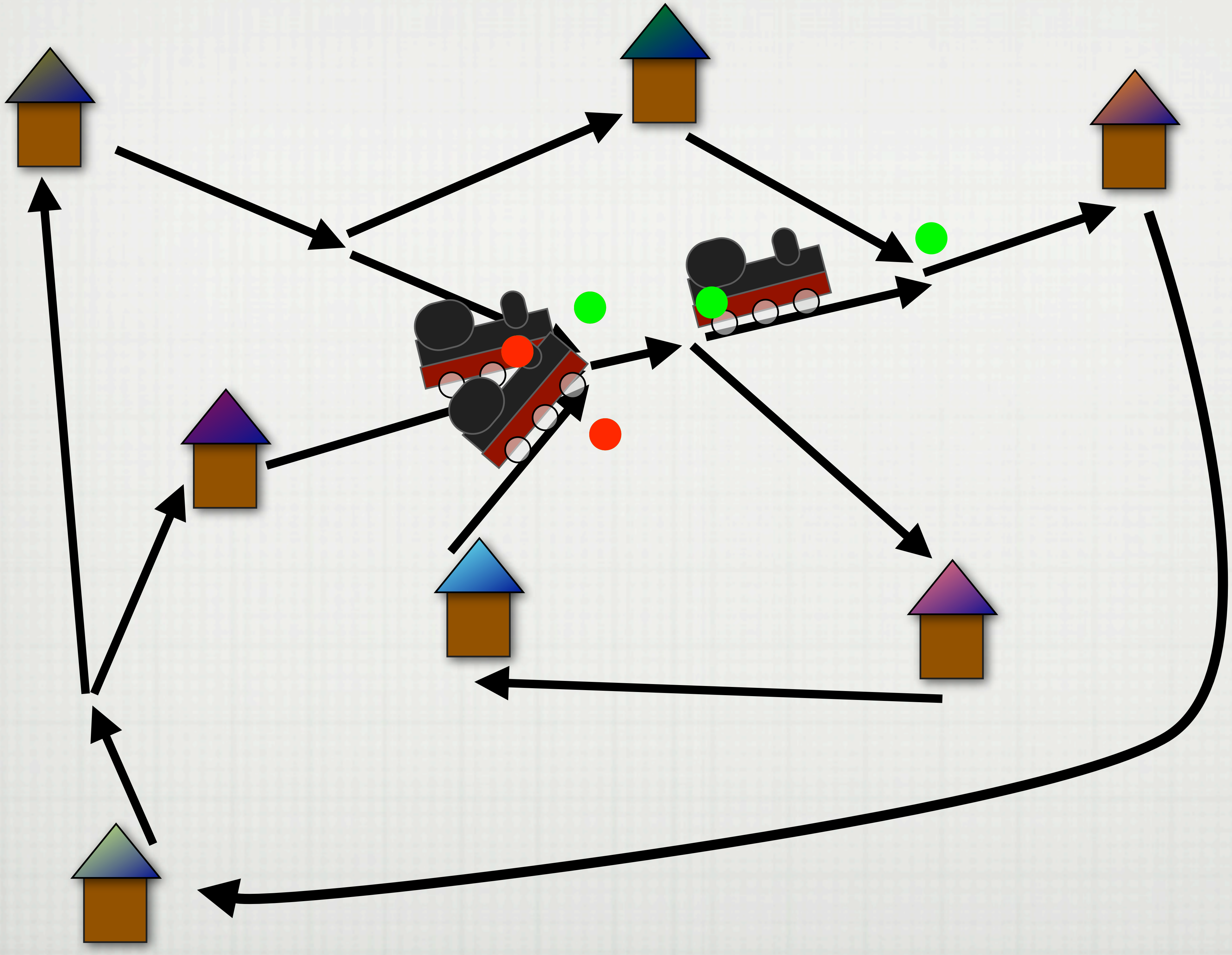
```
f x = x * x

g y x = if x < y
        then y + g y (x + 1)
        else h (y - x)

h x = g (f x) x
```

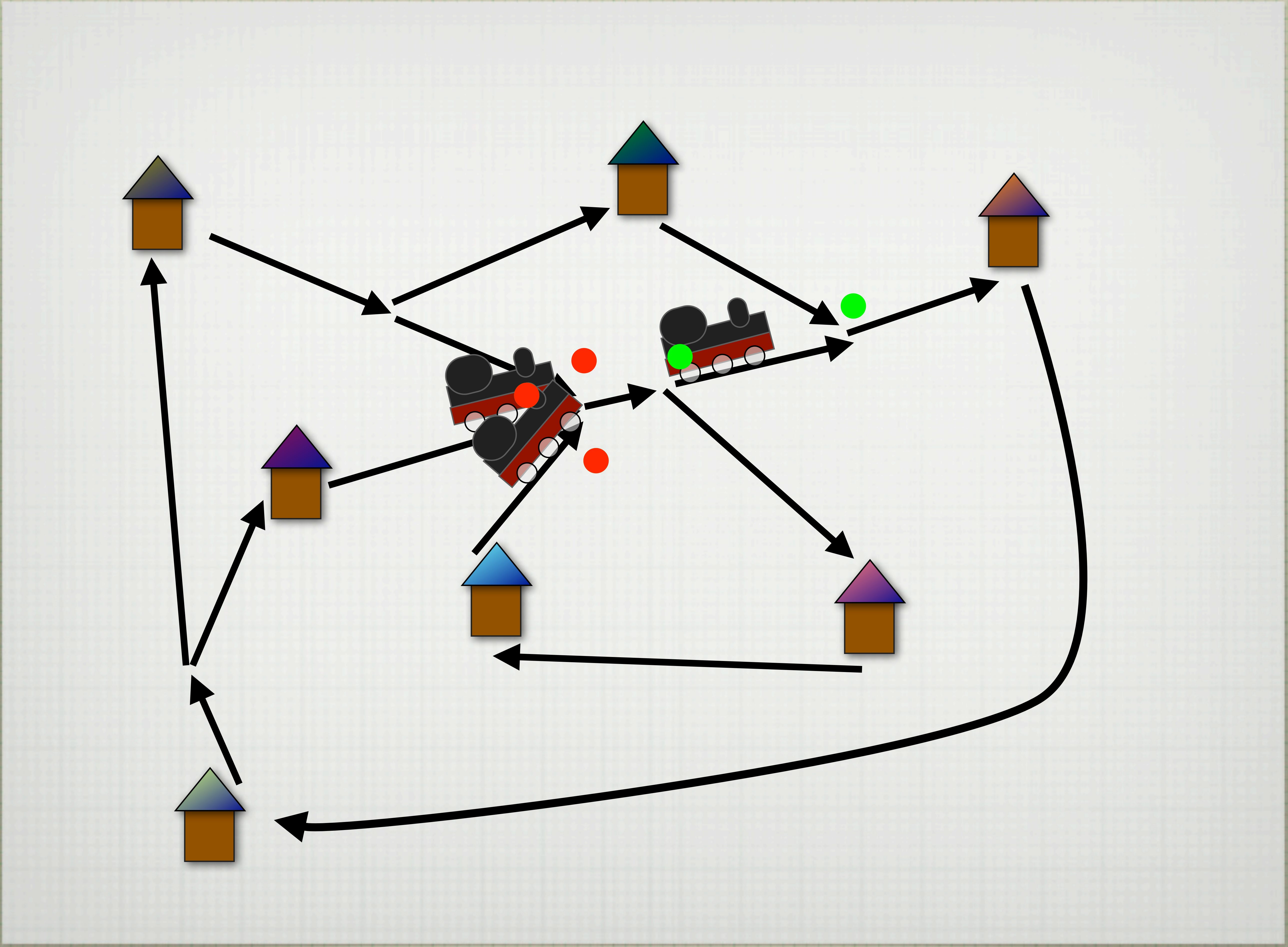
```
f x = x * x

g y x = f y + x

h x = g (f x) x
```

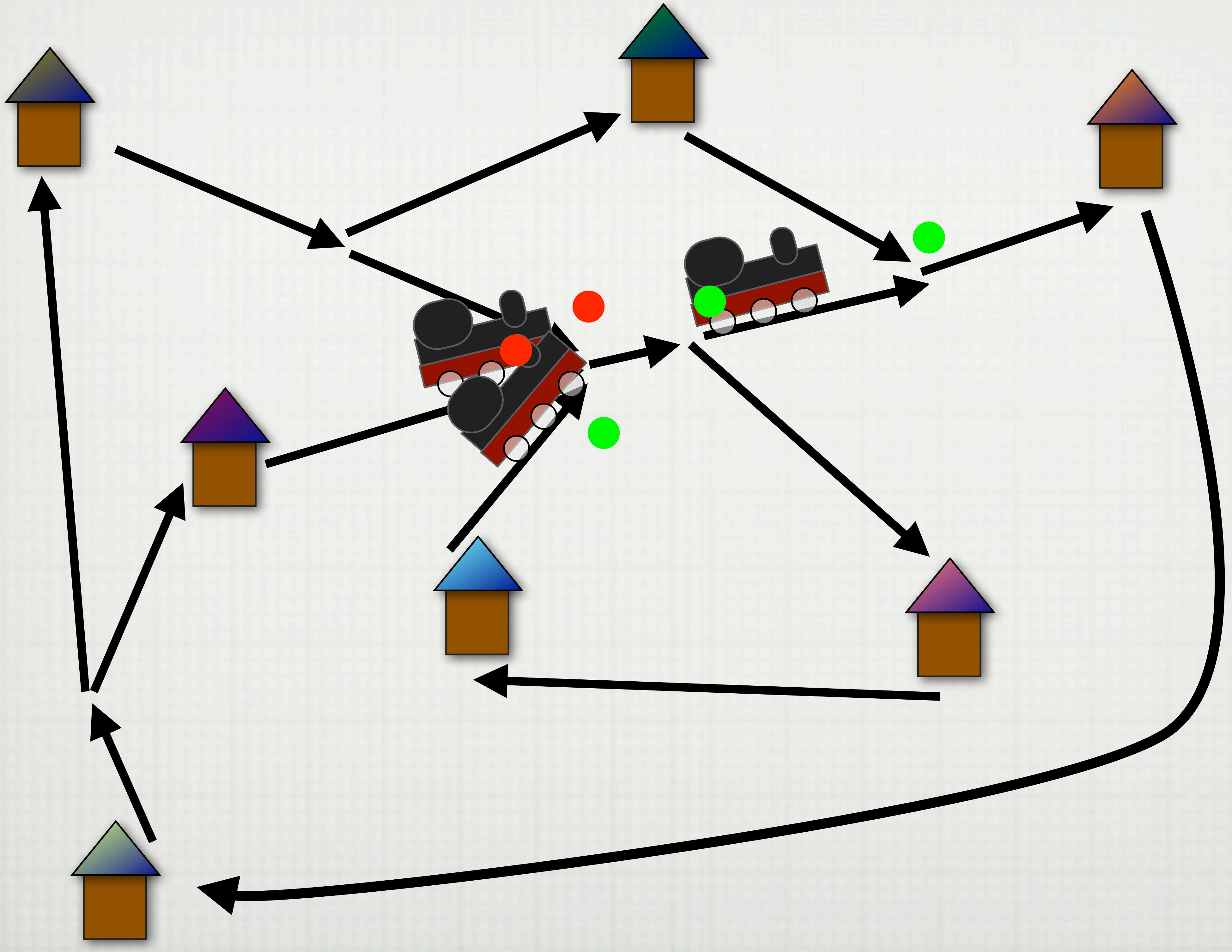


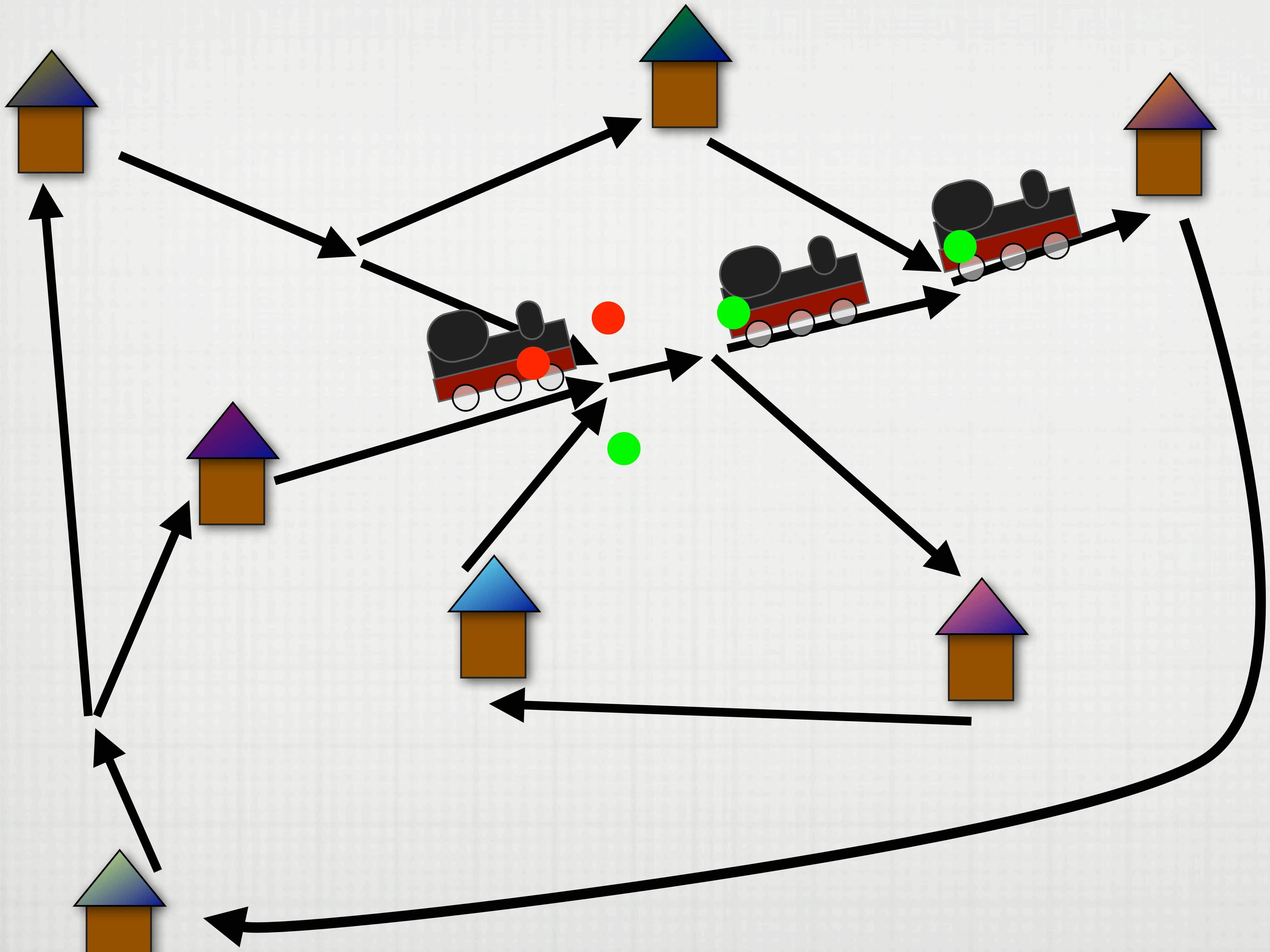


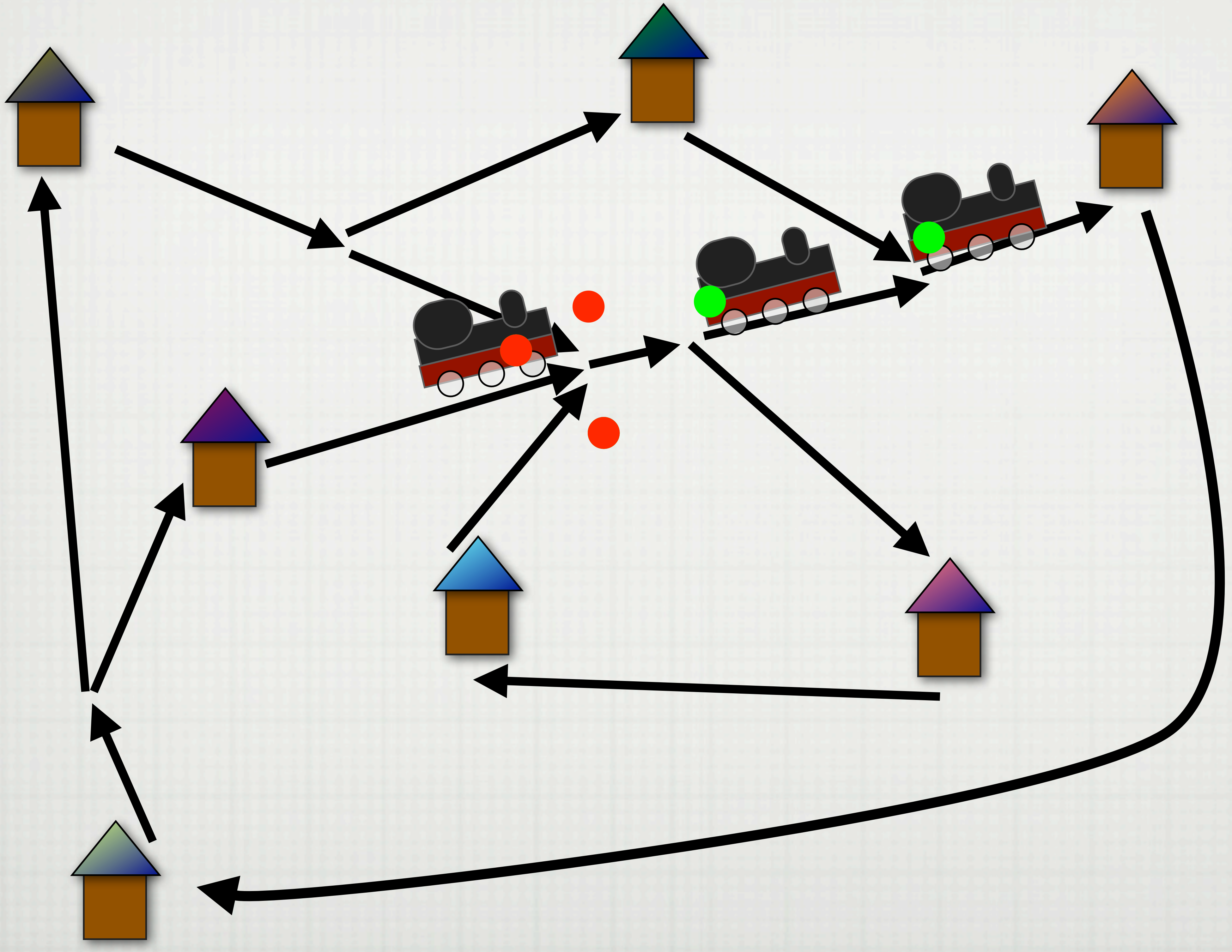


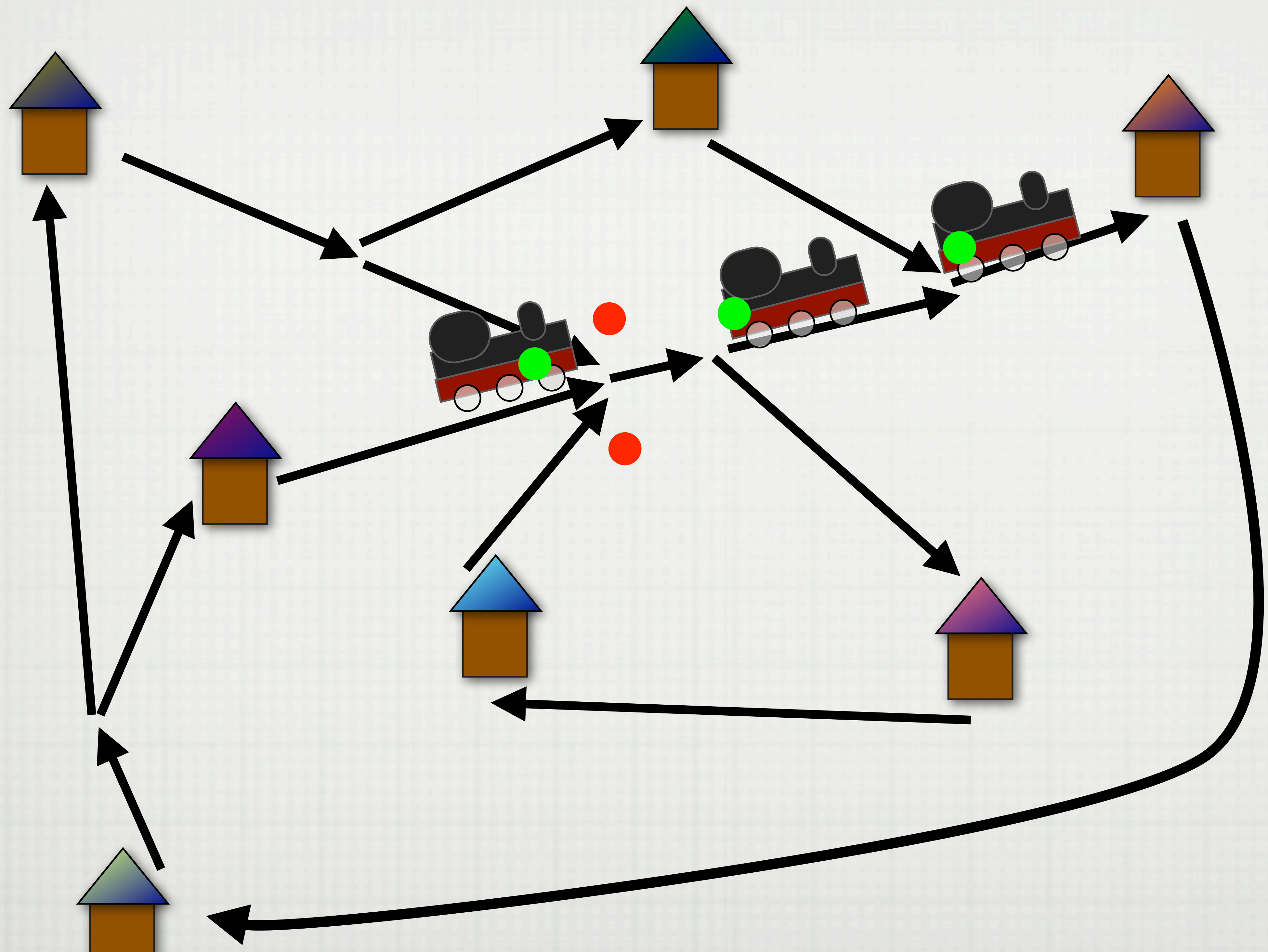


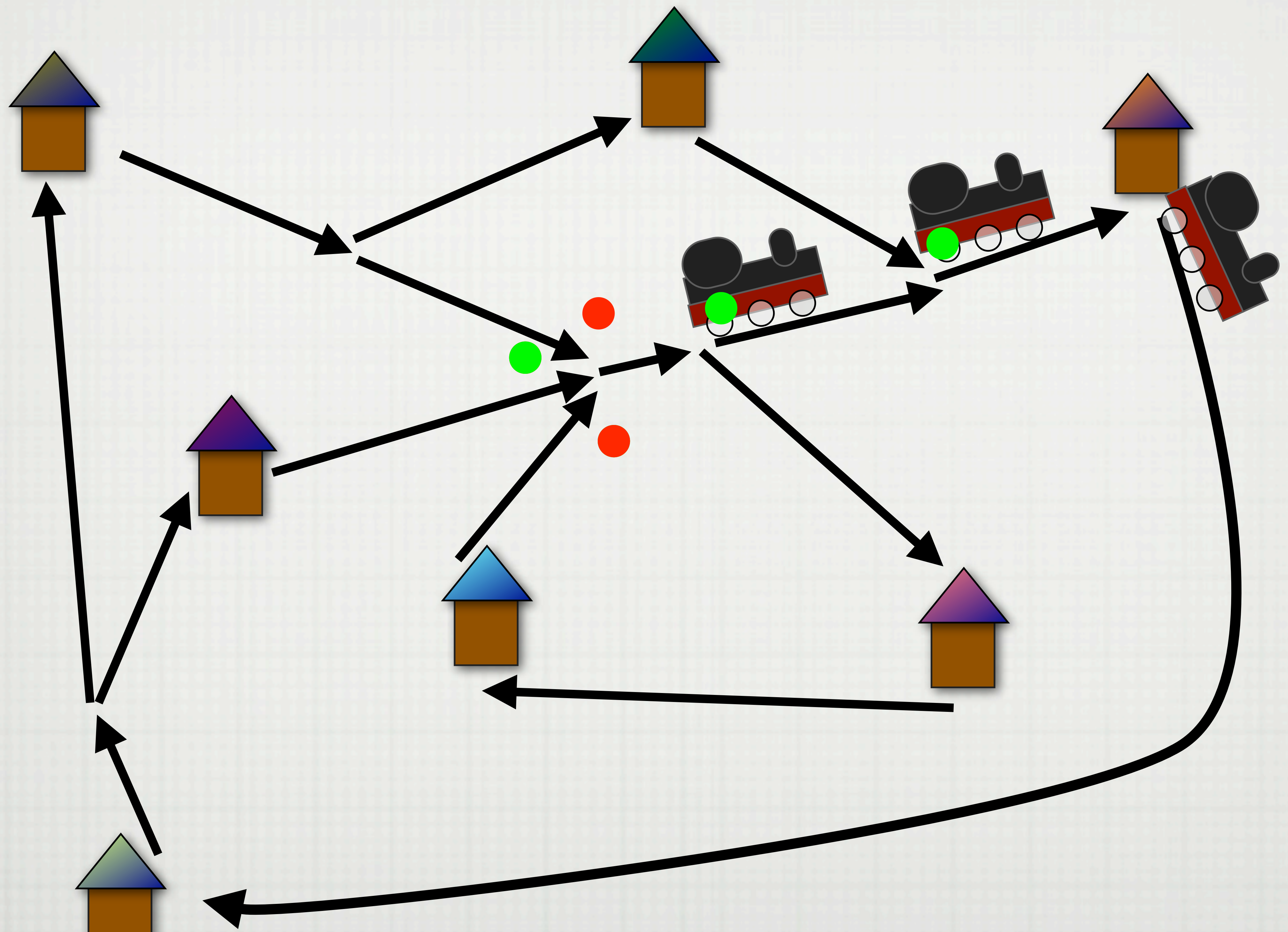








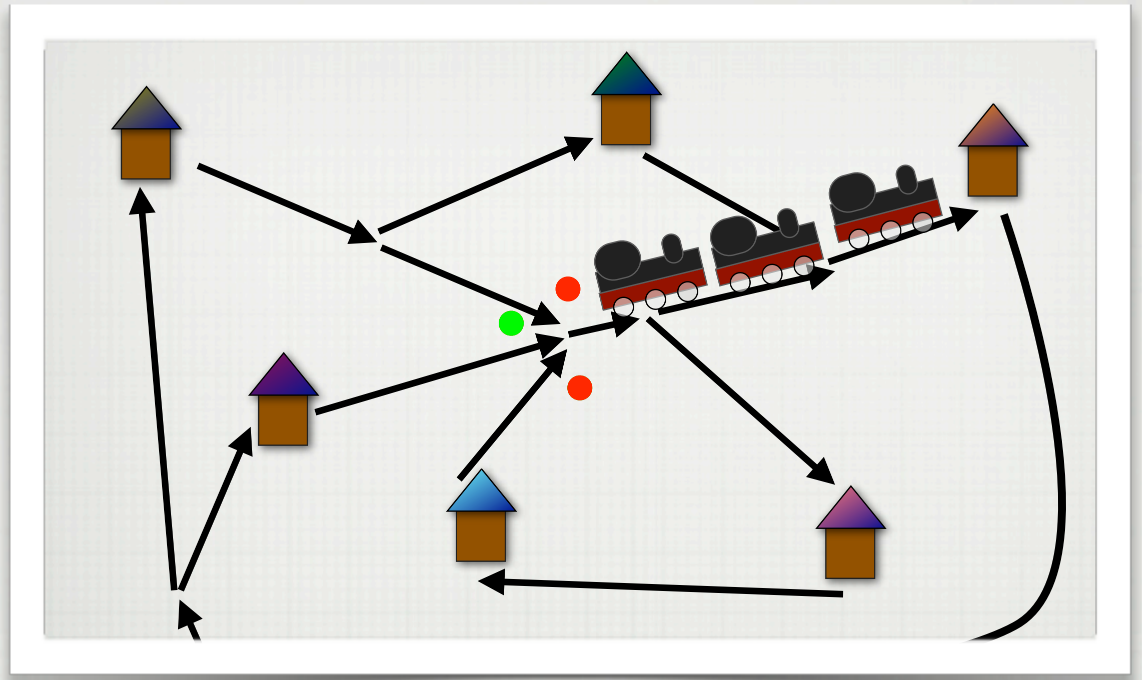




```
data Signal
    = Stop
    | Go


data Vehicle
    = Train TrainStop TrainStop Nat
```

TrainStop is anything that can be compared for equality.

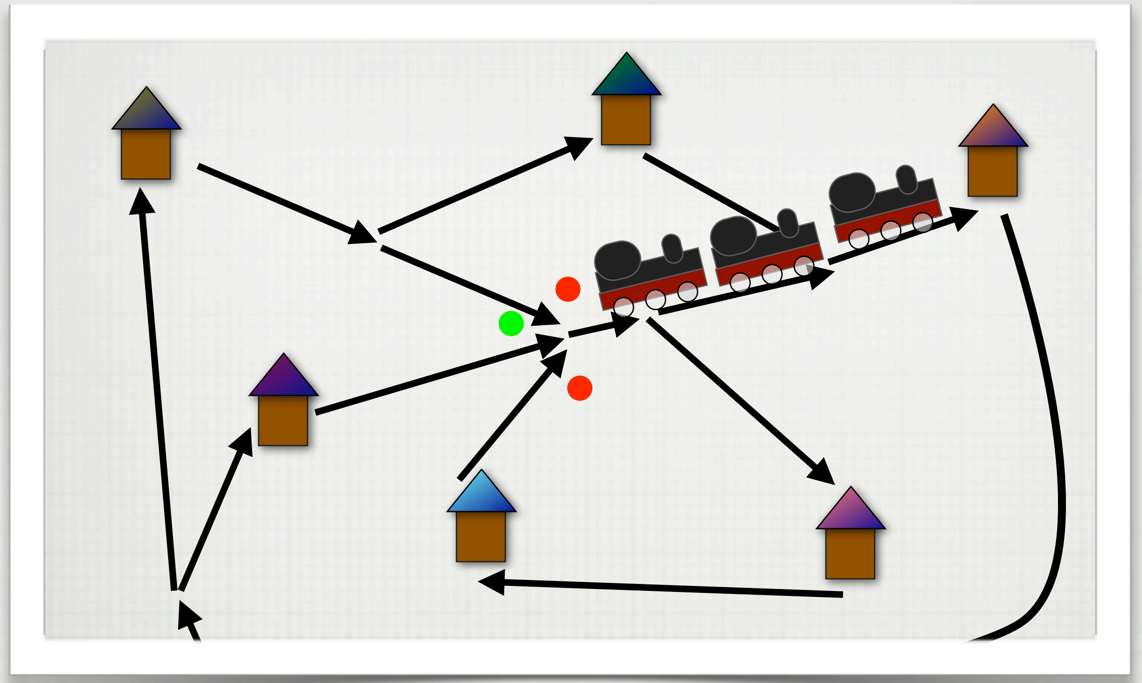
```
data Signal
    = Stop
    | Go
```



```
data Vehicle
    = Train TrainStop TrainStop Nat
```

TrainStop is anything that can be compared for equality.

```
data Vehicle
    = Train TrainStop
            TrainStop Nat
```



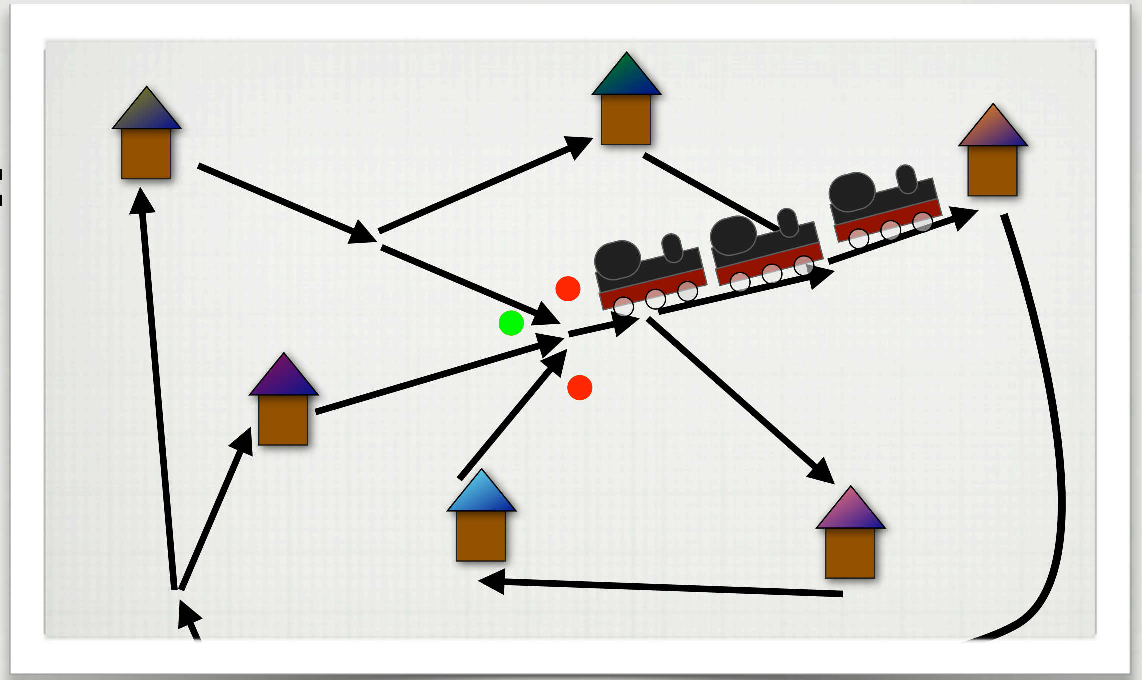
```
isApproaching :: Vehicle → TrainStop → Bool
isApproaching train stop = train {
    Train from to wait → to == stop;
}
```

```
pickTrain :: List Vehicle → Maybe Vehicle
pickTrain trains = second (trains {
    Nil → (0, Nothing);
    Cons t ts → t {
        Train _ _ tWait →
            let (wait, _) = @ts; in
            if tWait >= wait
                then (tWait, Just t)
                else @ts;
    };
})
```

```
Maybe a
    = Nothing
    | Just a
```

```
nextToGoAtStop trains stop =
    let trainsAtStop = trains {
        Nil → Nil;
        Cons t ts → if isApproaching t stop
            then Cons t @ts
            else @ts;
    }; in
    if empty trainsAtStop
        then Nothing
        else pickTrain trainsAtStop
    }
```

```
setSignal trains from to =
    (nextToGoAtStop
    trains to) {
        Nothing → Stop;
        Just t → t {
            Train s _ _ → if s == from
                then Go
                else Stop;
        };
    }
```



CATAMORPHISMS

- ☐ PATTERN MATCHING
- ☐ ITERATING OVER LISTS
- ☐ ITERATING OVER NATURAL NUMBERS

```
fib' 0 = (1, 1)
fib' 1 = (1, 1)
fib' n = let (x1, x2) = fib' (n - 1); in
   (x2, x1 + x2)
```

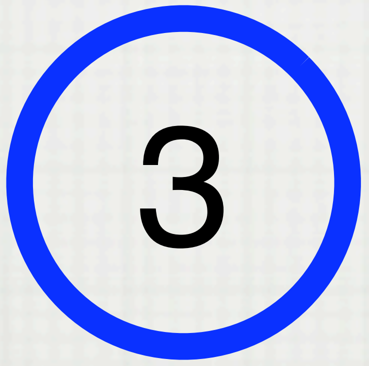
```
fib' n = n {
   Zero → (1, 1);
   Succ p → if n == 1 then (1, 1)
      else let (x1, x2) = @p; in
         (x2, x1 + x2);
}
```

```
fib n = second (fib' n)
```

- ☐ ITERATING OVER ANY RECURSIVE DATA TYPE

```
data Tree = Leaf Nat | Branch Tree Tree
{
    Leaf x → x * x;
    Branch left right → @left + @right;
}
```

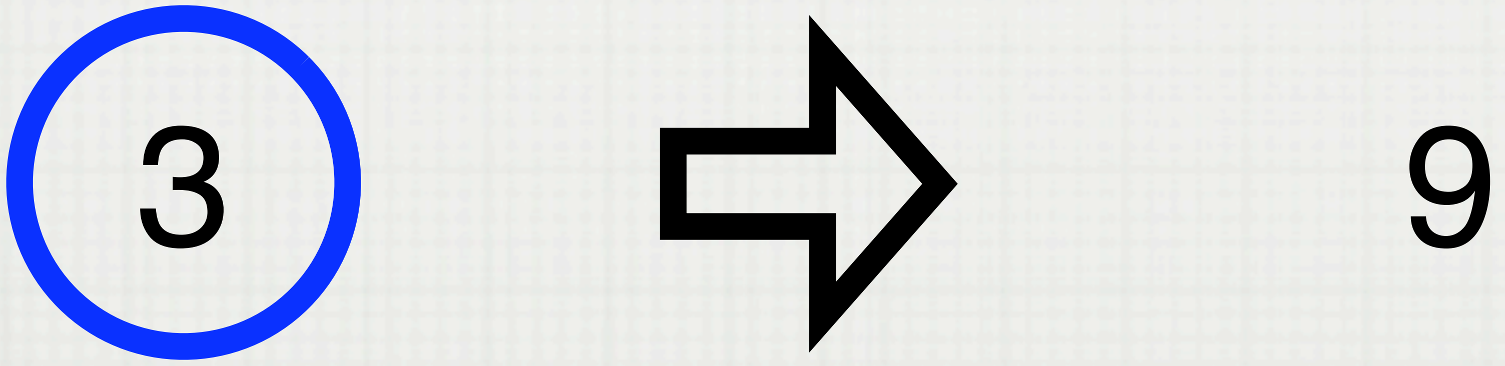
```
data Tree = Leaf Nat | Branch Tree Tree
{
    Leaf x → x * x;
    Branch left right → @left + @right;
}
```



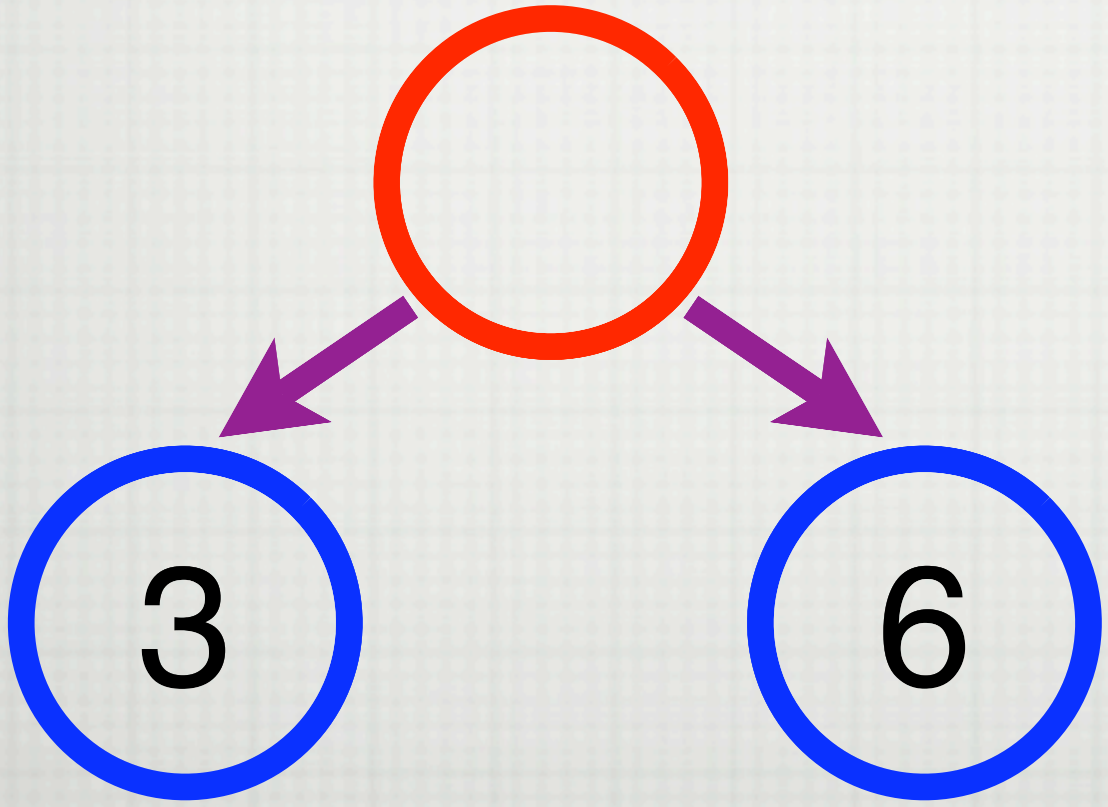
```
data Tree = Leaf Nat | Branch Tree Tree
{
    Leaf x → x * x;
    Branch left right → @left + @right;
}
```

3

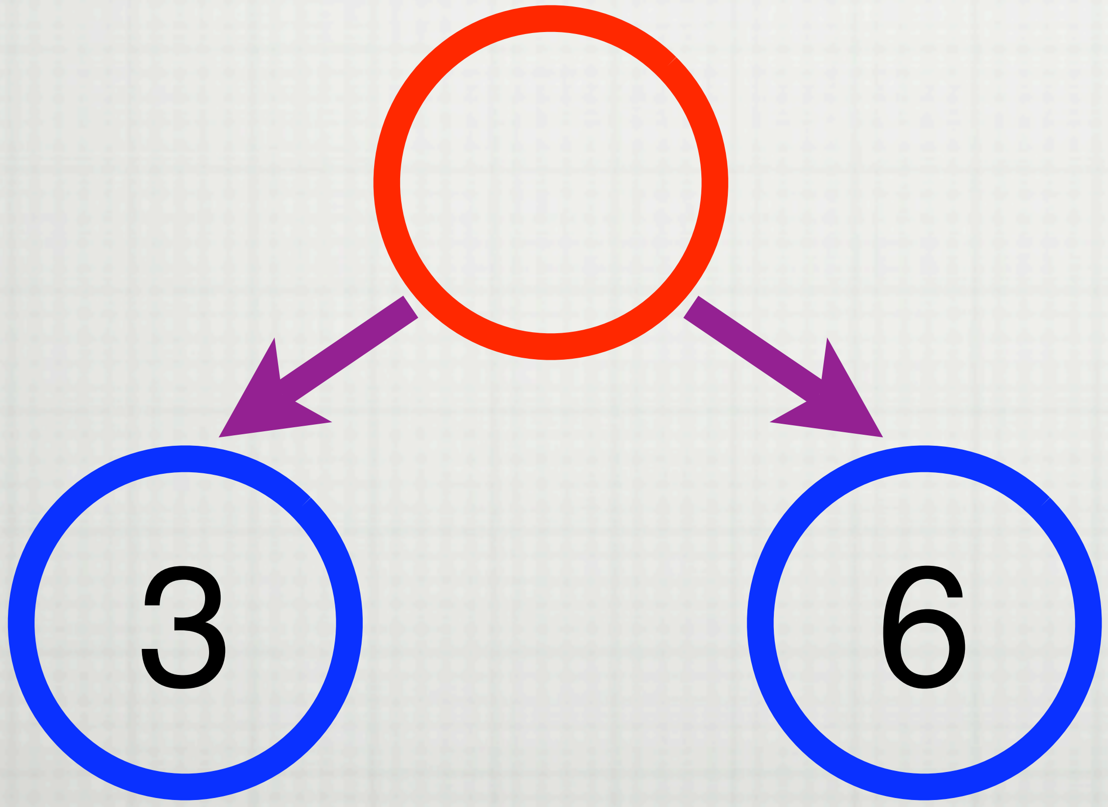
```
data Tree = Leaf Nat | Branch Tree Tree
{
    Leaf x → x * x;
    Branch left right → @left + @right;
}
```



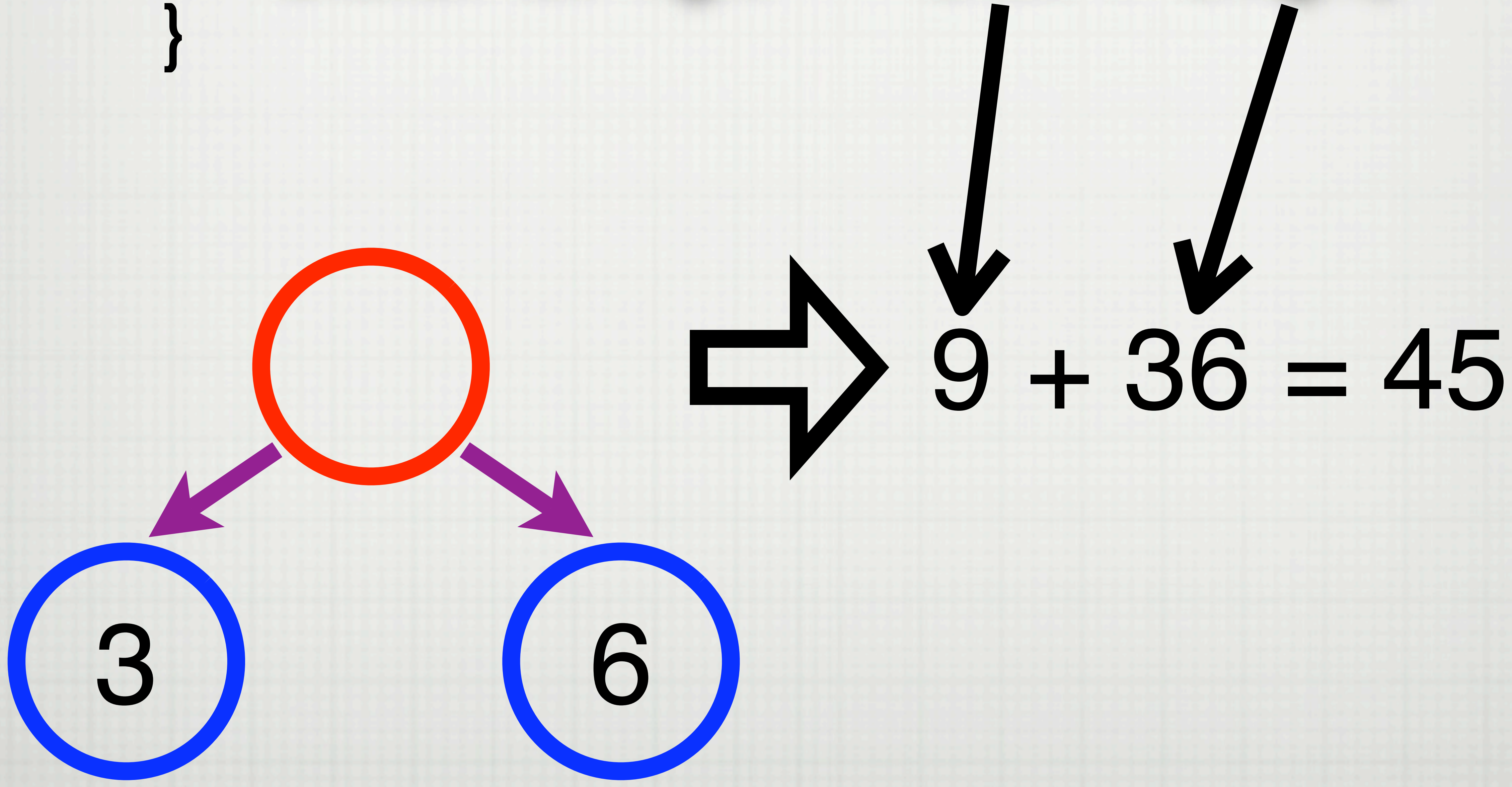
```
data Tree = Leaf Nat | Branch Tree Tree
{
    Leaf x → x * x;
    Branch left right → @left + @right;
}
```



```
data Tree = Leaf Nat | Branch Tree Tree
{
    Leaf x → x * x;
    Branch left right → @left + @right;
}
```



```
data Tree = Leaf Nat | Branch Tree Tree
{
    Leaf x → x * x;
    Branch left right → @left + @right;
}
```



WHAT'S DONE

- ☐ CONCRETE SYNTAX
- ☐ OPERATIONAL SEMANTICS
- ☐ (COOL) PROOF OF TERMINATION
  - ☐ ACTUALLY PRIMITIVE RECURSIVE

WHAT'S COMING

- ☐ (STABLE) INTERPRETER/COMPILER
- ☐ ALGORITHMS FOR BOUNDS ON TIME AND SPACE (AND MORE)
- ☐ EXTENSIONS TO CA OR OTHER LANGUAGES
- ☐ MORE THEORETICAL WORK

THANK YOU