Decorated Attribute Grammars

Attribute Evaluation Meets Strategic Programming

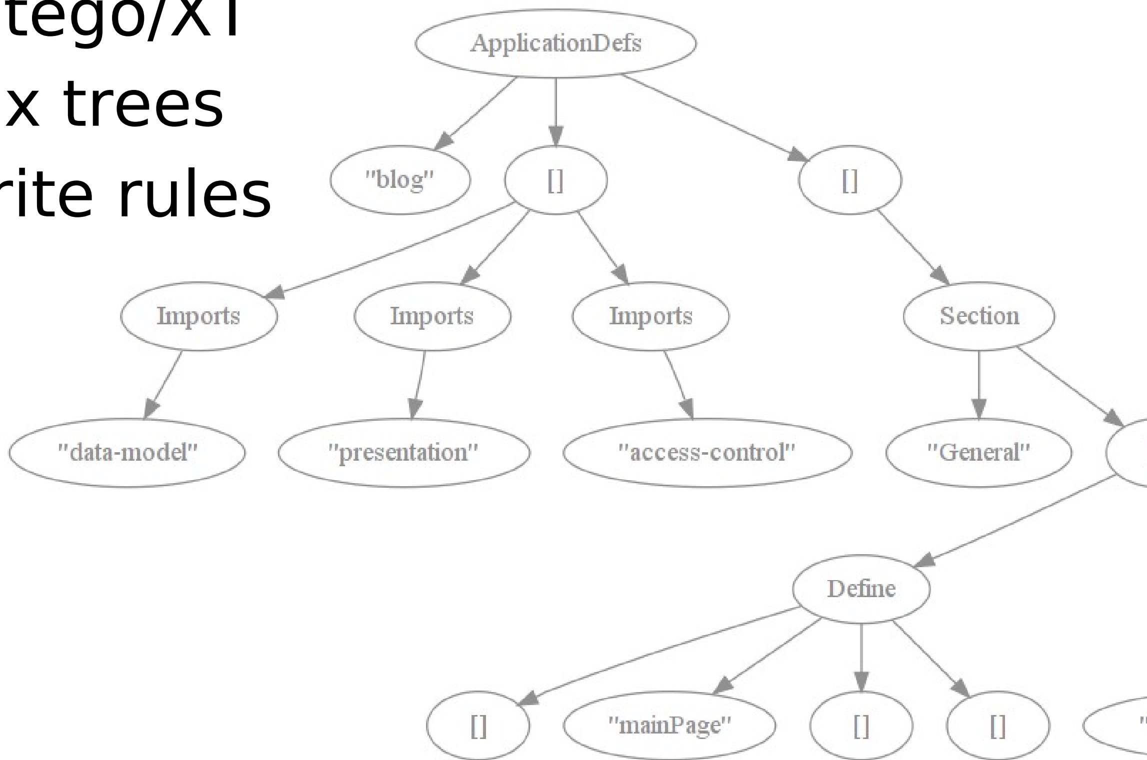
CC 2009, York, UK

Lennart Kats, TU Delft (*me*)
Tony Sloane, Macquarie
Eelco Visser, TU Delft

March 19, 2009

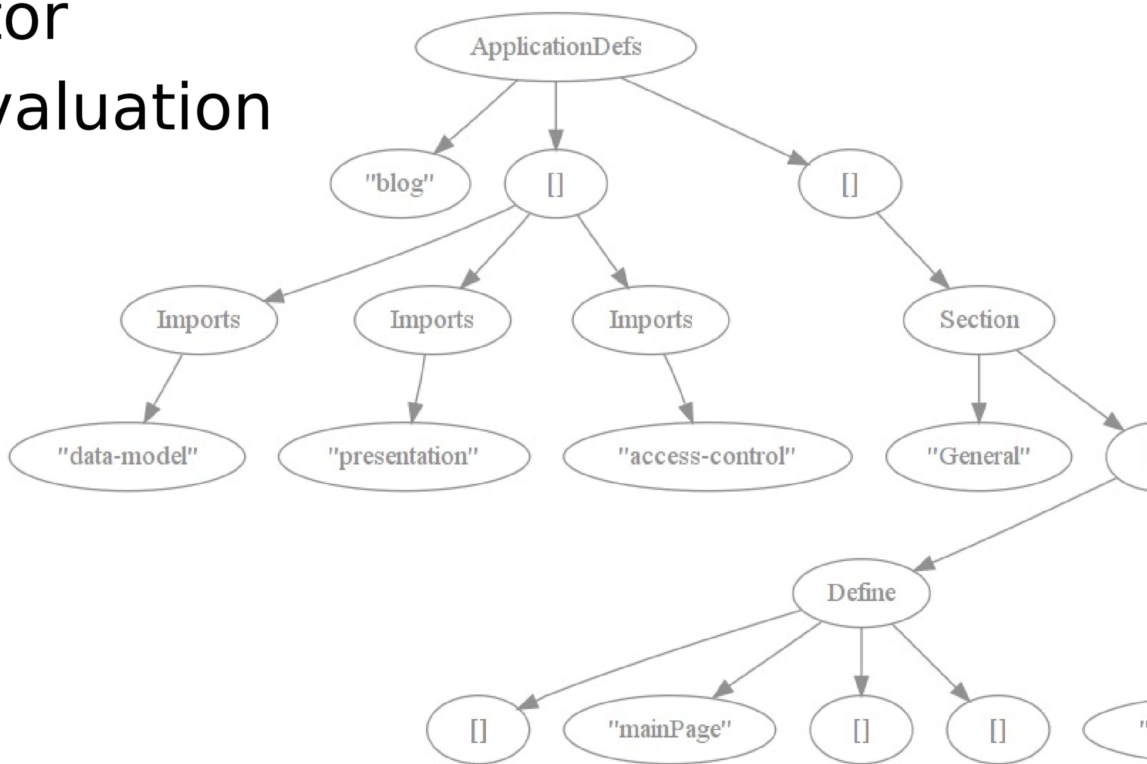
Context

- Domain-specific languages
  - example: WebDSL
  - language composition and extension
- SDF/SGLR + Stratego/XT
  - abstract syntax trees
  - traversal, rewrite rules

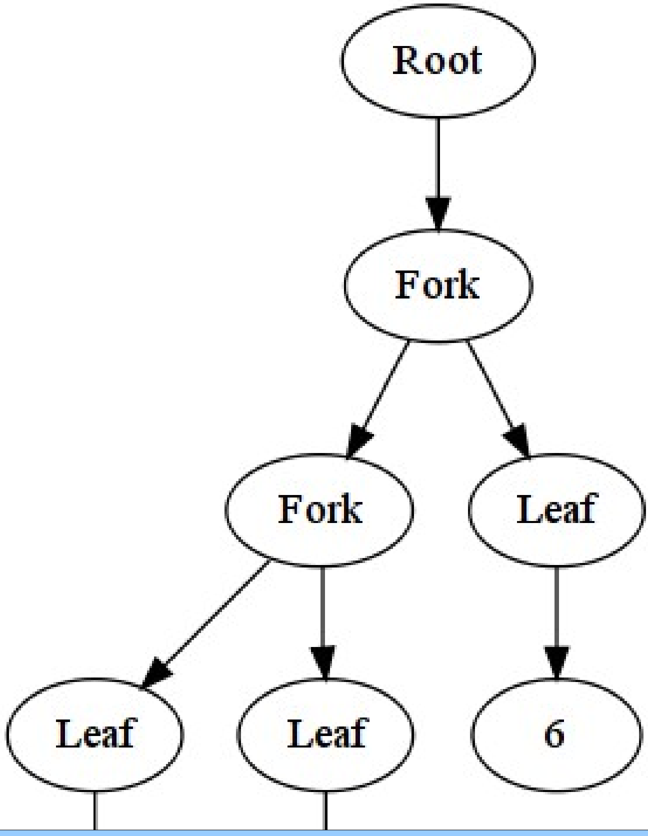


Trees and Attribute Grammars

- Attributes
  - Declarative, compositional equations
  - Express dependencies between nodes
- Attribute evaluator
  - Determines evaluation order



Basic Example: Global Minimum



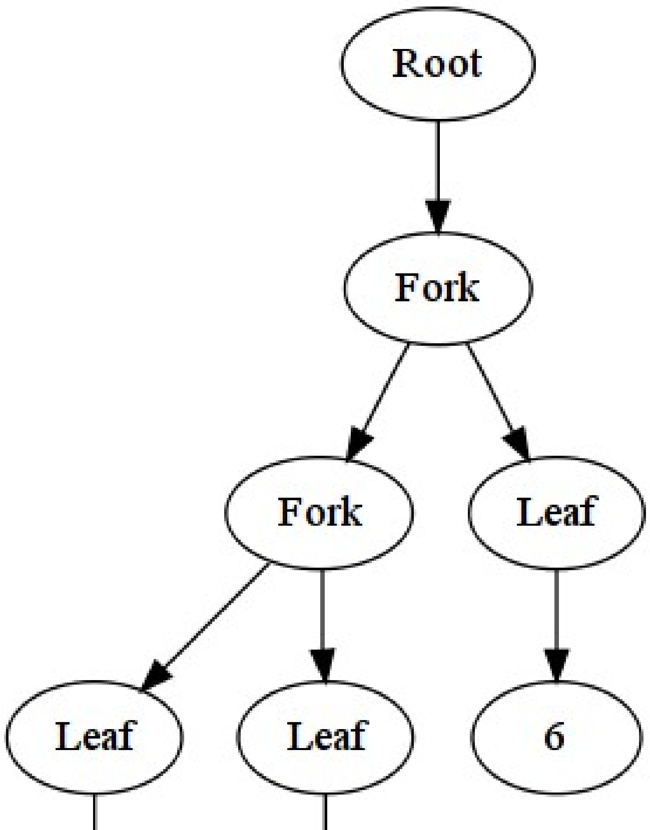
- Synthesized: flows up

def Root(*t*):
  id.min := *t*.min

def Fork(t_1, t_2):
  id.min := <min> (t_1.min,t_2.min)

def Leaf(*v*):
  id.min := *v*

Basic Example: Global Minimum



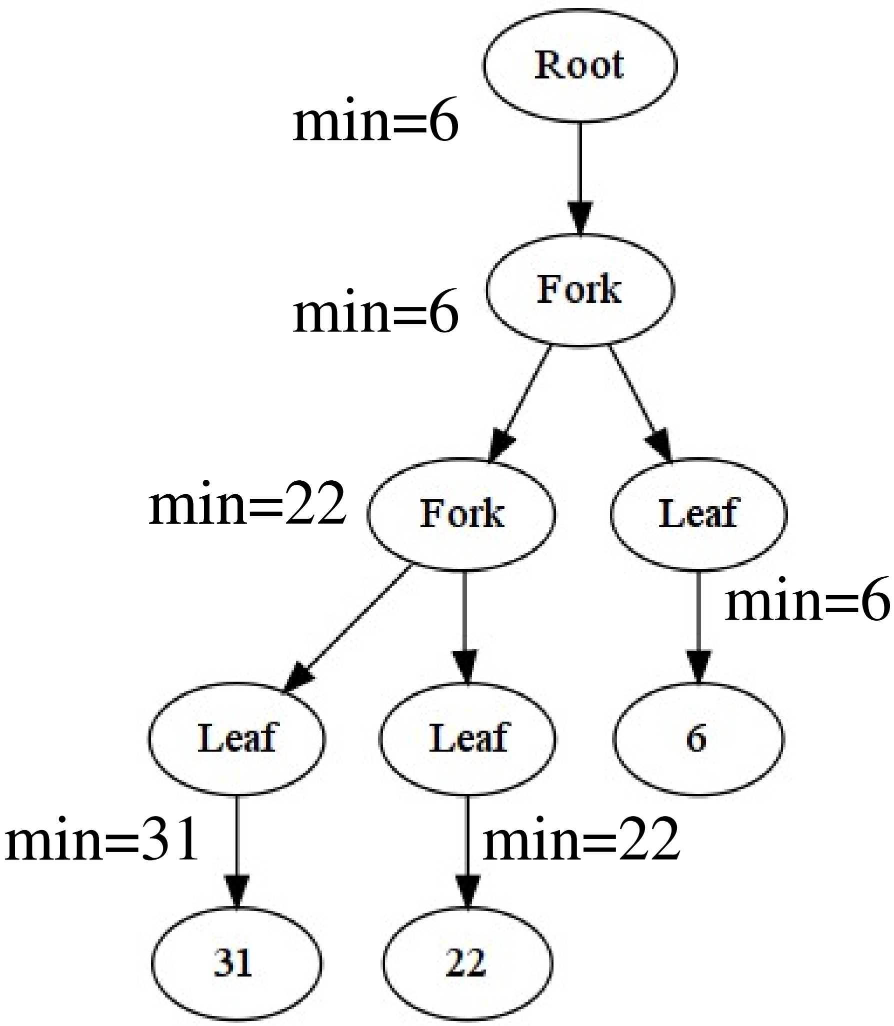
- Synthesized: flows up
- Inherited: flows down

def Root(*t*):
  id.min := *t*.min
  t.gmin := *t*.min

def Fork(t_1, t_2):
  id.min := <min> (t_1.min,t_2.min)
  t_1.gmin := id.gmin
  t_2.gmin := id.gmin

def Leaf(*v*):
  id.min := *v*

Basic Example: Global Minimum

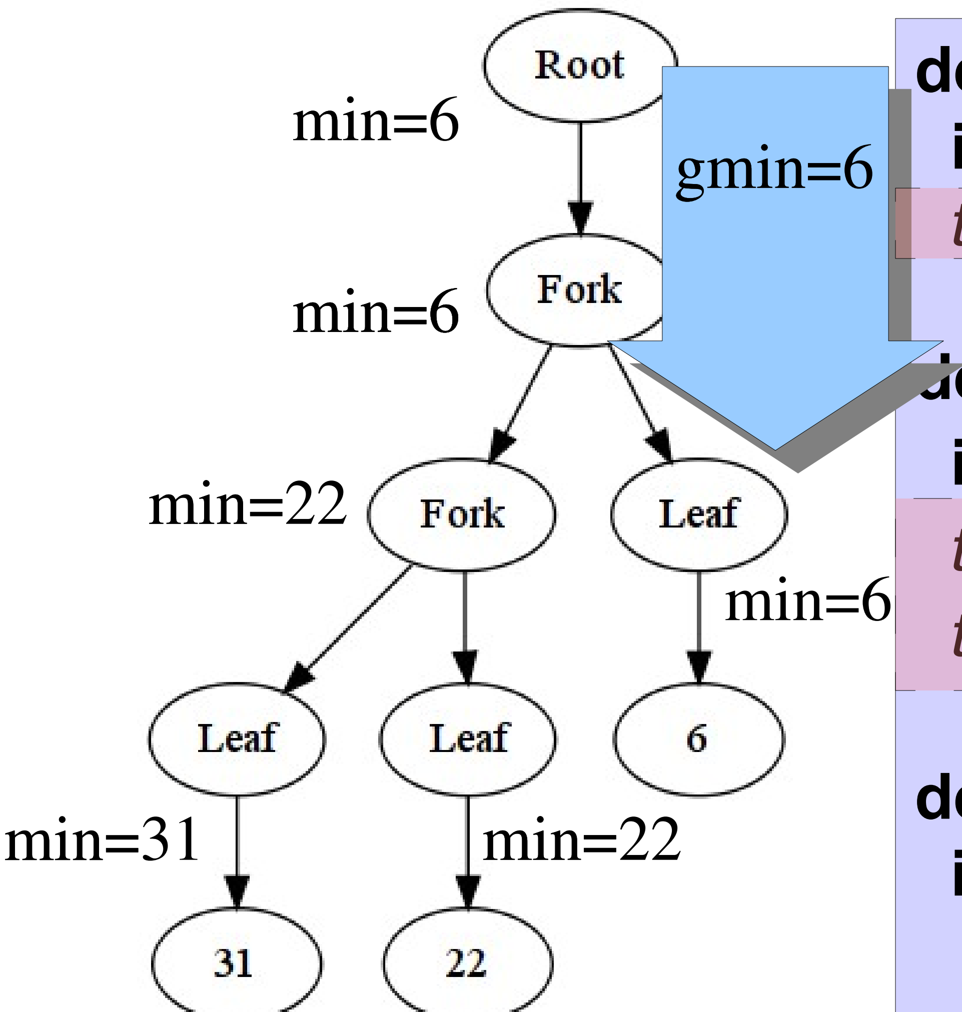



```
def Root(t):
  id.min := t.min
  t.gmin := t.min

def Fork(t1, t2):
  id.min  := <min> (t1.min,t2.min)
  t1.gmin := id.gmin
  t2.gmin := id.gmin

def Leaf(v):
  id.min := v
```

Basic Example: Global Minimum

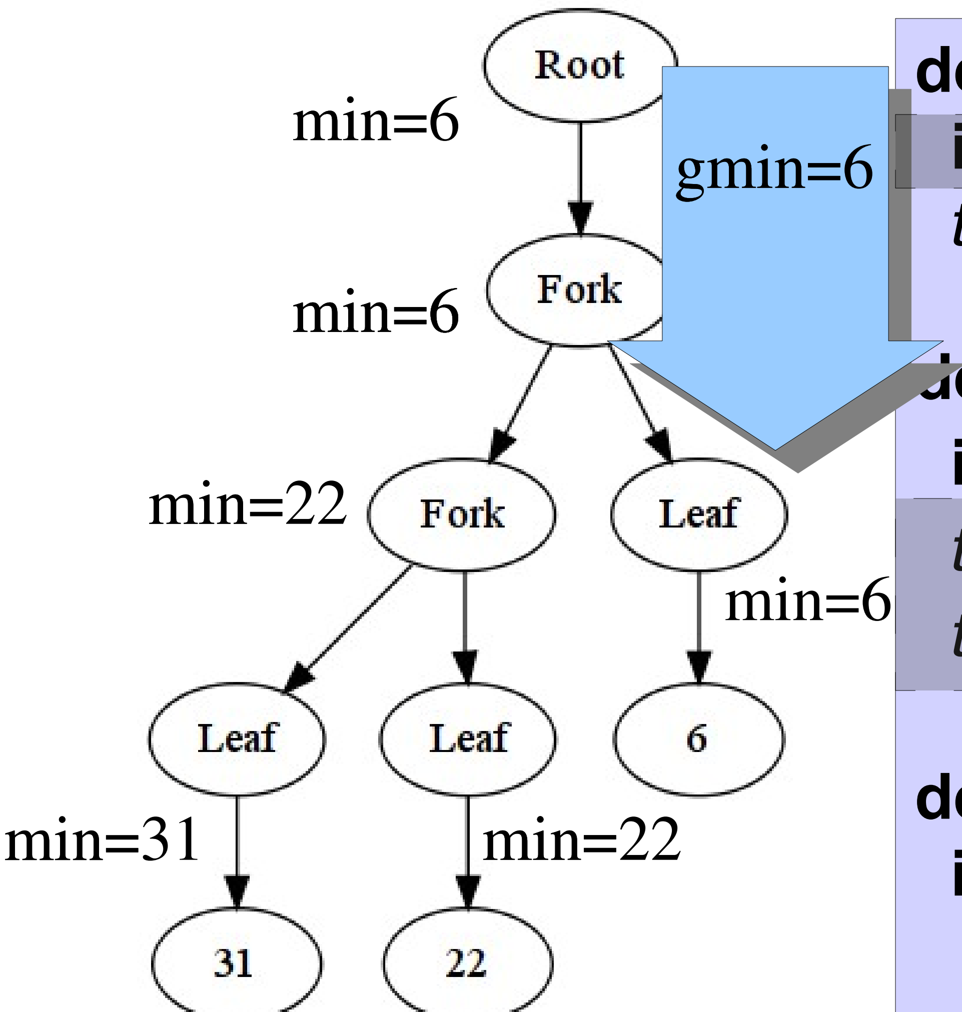


def Root(t):
  id.min := t.min
  t.gmin := t.min

def Fork(t_1, t_2):
  id.min := <min> (t_1.min, t_2.min)
  t_1.gmin := id.gmin
  t_2.gmin := id.gmin

def Leaf(v):
  id.min := v

Global Minimum: Identifying Copy Rules



def Root(t):
  id.min := t.min
  t.gmin := t.min

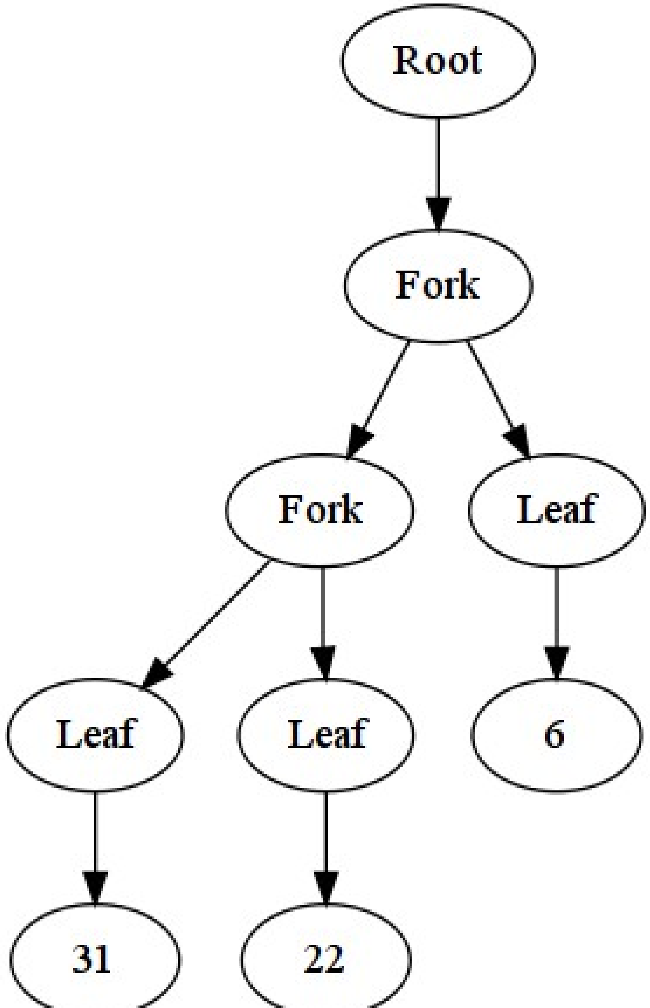
def Fork(t_1, t_2):
  id.min := <min> (t_1.min, t_2.min)
  t_1.gmin := id.gmin
  t_2.gmin := id.gmin

def Leaf(v):
  id.min := v

Introducing Decorators

- Abstract over traversal or evaluation pattern
  - Express intent
    - min: “upward flow”
    - gmin: “downward flow”
  - May introduce default behavior
  - May modify existing behavior

Example: up/down copying decorators



def Root(*t*):

t.gmin := *t*.min

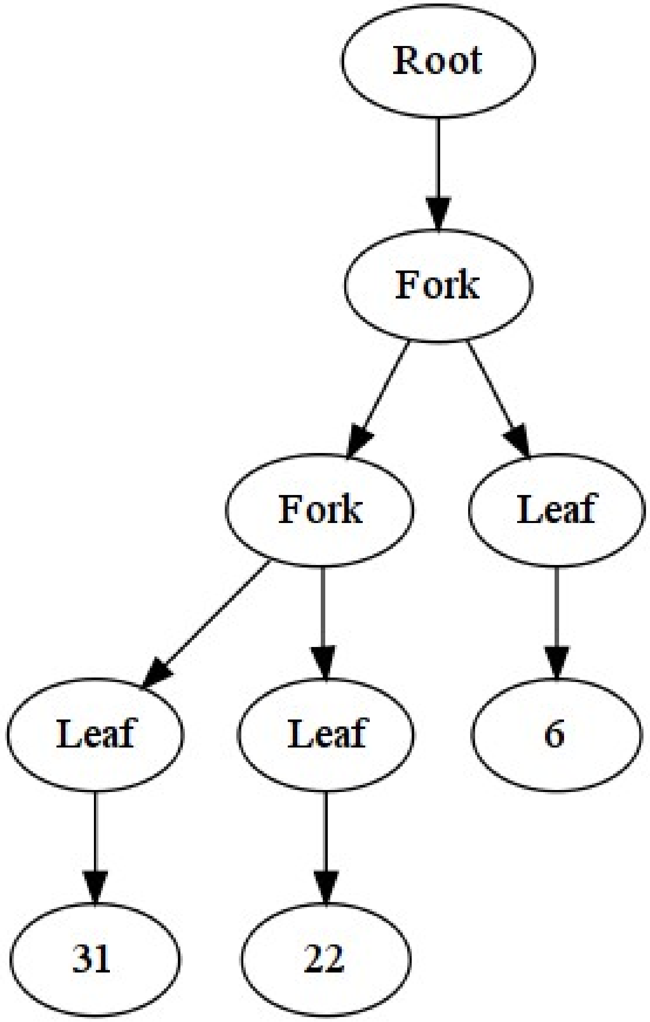
def Fork(t_1, t_2):

id.min := <min> (t_1.min,t_2.min)

def Leaf(*v*):

id.min := *v*

Example: up/down copying decorators



def Root(*t*):
  t.gmin := *t*.min

def Fork(t_1, t_2):
  id.min := <min> (t_1.min,t_2.min)

def Leaf(*v*):
  id.min := *v*

def down gmin
def up min

Introducing Decorators (ct'd)

Based on ***strategic programming***

- **Programmable**: decorators available as a library
- **Generic**: independent of a particular tree
- **Reflective**: may use properties of attributes

```
decorator down(a) =
  if a.defined then
    a
  else
    id.parent.down(a)
  end
```

```
decorator up(a) =
  if a.defined then
    a
  else
    id.child(id.up(a))
  end
```

Basic Building Blocks of Decorators

Arguments

- attribute *a*
- functions, terms

Reflective attributes

- *a*.defined
- *a*.attribute-name
- *a*.signature

```
decorator down(a) =
  if a.defined then
    a
  else
    id.parent.down(a)
  end
```

Tree access attributes

- id.parent
- id.child(*c*), id.prev-sibling, ...

Recursion

Abstraction Using Decorators (1)

Boilerplate code elimination:

- avoid repetitive code (e.g., “copy rules”)
- reduce accidental complexity
- implement some of the boilerplate-coping mechanisms of other AG systems

Abstraction Using Decorators (2)

Control over evaluation:

- **tracing**
- memoization
- assertions

```
def trace gmin

decorator trace(a) =
  t := id;
  a;
  log(|[a.attribute-name, " at ", t.path, ": ", id])
```

Abstraction Using Decorators (2)

Control over evaluation:

- tracing
- **memoization**
- assertions

```
decorator default-caching(a) =
  if id.get-cached(a) then
    id.get-cached(a)
  elseif a then
    ...
    a;
   ...set-cached...
  end
```

Abstraction Using Decorators (2)

Control over evaluation:

- tracing
- memoization
- **assertions**

```
def assert-after(<leq> (id.gmin, id.min)) gmin

decorator assert-after(a, c) =
  t := id;
  a;
  id.set-cached-for(a|t);
  if not(<c> t) then
    fatal-err(|["Assertion failed for ",
                  a.attribute-name, " at ", t.path])
  end
```

Abstraction Using Decorators (3)

Help in typical compiler front-end tasks:

- name analysis
- type analysis
- control- and data-flow analysis

$\Rightarrow$ *encapsulation of recurring attribution patterns*

Type Analysis with Aster

def type:
  Int(*i*)   → IntType
  |[var *x* : *t*;]| → *t*
  Var(*x*)   → **id**.lookup-local(|*x*).type
  |[*f*(*arg**)]| → **id**.lookup-function(|*f*, *arg**).type
  ...

Concrete syntax [Visser 2002]

VarDecl(*x*, *t*)

Reference attributes [Hedin 2000]

look up declaration nodes

var x : Int;

Type Analysis with Aster:
Using Decorators (1)

Lookup decorators require:

- Lookup type (ordered, unordered, global, ...)
- Tree nodes to fetch
- Scoping definition

def lookup-ordered(**id**.is-scope) lookup-local(|*x*):
  |[var *x* : *t*;]| → **id**
  |[*x* : *t*]| → **id**

Type Analysis with Aster: Using Decorators (2)

Lookup decorators require:

- Lookup type (ordered, unordered, global, ...)
- Tree nodes to fetch
- Scoping definition

def is-scope:
  |[{ *s** }]| → **id**
  |[function *x*(*param**) : *t* { *stm** }]| → **id**
  ...

Type Analysis with Aster: Using Decorators (3)

```
def lookup-unordered(id.is-scope) lookup-function(|x, arg*):
  |[ function x (param*) : t { stm* } ]| → id
    where
      argtype*      := arg*.map(id.type);
      paramtype* := param*.map(id.type);
      paramtype*.eq(|argtype*)
```

Type Analysis with Aster: Decorator Definitions

```
decorator down lookup-ordered(a, is-s) =
  if a then
    a
  else
    id.prev-sibling(id.lookup-inside(a, is-s))
  end

decorator lookup-inside(a, is-scope) =
  if a then
    a
  else if not(is-scope) then
    // enter preceding subtrees
    id.child(id.lookup-inside(a, is-scope))
  end
```

Decorator stacking

```
function x() : Int {
  var j : Int;
  ...
}

function y() : Int {
  if (true) {
    var i : Int;
  }
  return j;
}
```

Error Reporting Using Decorators

module Constraints

def errors:
  |[while (*e*) *s*]| → "Condition must be of type Boolean"
    where not(*e*.type ⇒ BoolType)

  ...

module Reporting

def collect-all add-error-context errors

Decorator stacking

decorator add-error-context(*a*) =
  <conc-strings > (*a*," at ", **id**.pp, " in ", **id**.file, ":",
                  id.linenumber)

Control-flow Analysis (1)

```
def default(id.default-succ) succ:
  |[ if (e) s1 else s2 ]| → [s1, s2]
  |[ return e; ]|         → []
  |[ { s1; s* } ]|        → [s1]
  ...

decorator default(a, default) =
  if a.defined then
    a
  else
    default
  end
```

Control-flow Analysis (2)

```
def down default-succ:
  Program(_)  → []
  [s1, s2 | _].s1 → [s2]
  ...
```

"But Wait, There's More!"

- Data-flow analysis
  - reaching definitions, liveness, ...
  - data-flow equations as attribute equations
  - circular (fixed point) attribute evaluation
- Deriving the reverse control flow

```
// Statement copies itself to successors
def contributes-to(id.succ) pred:
  stm → id
```

Go Figures!

• **Syntax**:	3 modules	341 lines
• **Compiler**:	25 modules	2831 lines
• **Library**:	25 modules	1845 lines
• **Tests**:	33 modules	1531 lines
Total:	86 modules	6548 lines

Concluding Remarks

- Language growth in the hands of the users
  - decorators implement automatic copy rules, self rules, collection attributes, circular attributes, ...
- Combine generic behavior with (simple) reflection
- Future:
  - Library, language extension
  - IDE integration

Decorated Attribute Grammars. Attribute Evaluation Meets Strategic Programming. Lennart C. L. Kats, Anthony M. Sloane, and Eelco Visser. *18th International Conference on Compiler Construction (CC 2009)*.

http://www.strategoxt.org/Stratego/Aster