



Compiler construction

Guest lecture: attribute grammars

Alex Gerdes

Vårtermin 2019

Chalmers University of Technology — Gothenburg University

What are attribute grammars?

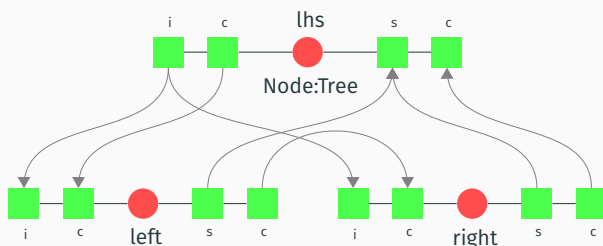


- An attribute grammar is a means to associate attributes (semantics) with the productions of a grammar (syntax)
- Describe tree traversals, for example maps, folds
- Attributes:
 - synthesized** these 'travel' upward through a syntax tree, from child to parent
 - inherited** these 'travel' downward through a syntax tree, from parent to child
 - (chained)** these are inherited as well as synthesized

Attributes



```
data Tree | Node left, right :: Tree
         | Leaf value      :: Int
```



Why are they useful?



- Allows for modular development, separation of concerns
- Reduces boilerplate code
- Example applications:
 - Scope/type checking
 - AST transformations, for example α -renaming
 - Code generation
 - Pretty printing

UUAGC

- The UU attribute grammar system
- It is implemented as a preprocessor to Haskell
- UHC is developed with UUAGC
- Available on Hackage

Binary trees: sum



```
data Tree | Node left, right :: Tree
         | Leaf value      :: Int

attr Tree
  syn sum :: Int

sem Tree
  | Node lhs.sum = @left.sum + @right.sum
  | Leaf lhs.sum = @value

tree :: Tree
tree = Node (Leaf 1) (Node (Leaf 2) (Leaf 3))
```

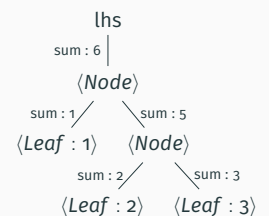
Binary trees: sum



```
data Tree
  | Node left, right :: Tree
  | Leaf value      :: Int

attr Tree
  syn sum :: Int

sem Tree
  | Node lhs.sum = @left.sum
                  + @right.sum
  | Leaf lhs.sum = @value
```



Binary trees: depth and index



```
attr Tree
  inh depth :: Int
  chn index :: Int

sem Tree
  | Node loc.depth = @lhs.depth + 1
        left.depth = @depth
        right.depth = @depth

sem Tree
  | Node left.index = @lhs.index
        right.index = @left.index
        lhs.index = @right.index
  | Leaf lhs.index = @lhs.index + 1
```

Binary trees: depth and index using copy rules



```
attr Tree
  inh depth :: Int
  chn index :: Int

sem Tree
  | Node loc.depth = @lhs.depth + 1

sem Tree
  | Leaf lhs.index = @lhs.index + 1
```

The archetypal repmin problem



```
data Root
  | Root Tree

attr Tree
  inh gmin :: Int
  syn lmin use {'min'} {0} :: Int

attr Root Tree
  syn result :: self

sem Tree
  | Leaf lhs.lmin = @value
    .result = Leaf @lhs.gmin
sem Root
  | Root tree.gmin = @tree.lmin
```

A small expression language



```
{
data Type = Int | Bool
data Value = I Int | B Bool
type Env = M.Map String Var
data Var = V { typ :: Type }
}

data Expr
  | Con val :: {Value} -- Constants
  | Var name :: String -- Variables
  | Let name :: String -- Let binding
    expr :: Expr
    body :: Expr
  | Add x, y :: Expr -- Add operator
  | And x, y :: Expr -- Logical and operator

deriving Expr : Ord, Eq, Show
```

Type checking



```
attr Expr
  inh env :: {Env}
  syn ty :: {Type}

sem Expr
  | Con lhs.ty = case @val of I _ -> Int ; _ -> Bool
  | Var lhs.ty = lookupTy @name @lhs.env
  | Let lhs.ty = @body.ty
    body.env = M.insert @name (V @expr.ty) @lhs.env
  | Add lhs.ty = tyCheck Int @x.ty @y.ty
  | And lhs.ty = tyCheck Bool @x.ty @y.ty
{
lookupVar v = fromMaybe (error "Unbound var!") . M.lookup v
lookupTy v = typ . lookupVar v
tyCheck t t1 t2 | t1 == t2 && t == t1 = t1
                | otherwise = error "Type mismatch"
}
```

Free variables



```
attr Expr
  syn free use {++} {[]} :: Strings

-- Free variables
sem Expr
  | Var lhs.free = [@name]
  | Let lhs.free = @body.free \\ [@name]
```

Live coding

Further reading



- Why AGs matter:
[https://wiki.haskell.org/The_Monad.Reader/Issue4/Why_Attribute_Gra](https://wiki.haskell.org/The_Monad.Reader/Issue4/Why_Attribute_Gr)
- UUAGC manual:
<http://www.cs.uu.nl/docs/vakken/mcco/downloads/agmanual.pdf>
- UHC: <https://wiki.haskell.org/UHC>
- Good luck!