Attribute Grammars

An attribute grammar consists of
1. A context free grammar describing the abstract syntax of the language.
2. A fixed set of attributes for each grammar symbol of the abstract syntax. The attributes store information associated with the symbol, such as type, value, symbol table, code, etc.
3. A set of attribute evaluation rules for each production of the abstract syntax.

Example

```
A ::= B C
{ C.d := B.c + 1;
  A.b := A.a + B.c; }
```

---

Attribute Grammars...

_____ During Compiler Construction _____
1. Describe Concrete Syntax.
2. Describe Abstract Syntax.
3. Give mapping from concrete to abstract syntax (this is the parser’s role).
5. Compute attribute evaluation order.
6. Implement the evaluator.

_____ During Parsing _____
1. Build abstract syntax tree.
2. Decorate tree with input attributes.

_____ During Semantic Analysis _____
1. Compute attributes.

---

Synthesized Attributes

- Synthesized attributes move values up the tree (from the leaves towards the root).
- The value of a synthesized attribute is determined from the values of its children.

```
A ::= B C { A.a := f(B.x, C.b) }
```
Example I

- Compute the max of an AST of ints.
- "max" is synthesized.

\[ E_0 ::= E_1 \ E_2 \]
\[ \{ E_0.\text{max} := \text{MAX}(E_1.\text{max}, E_2.\text{max}) \} \]
\[ E ::= V \{ E.\text{max} := V.\text{value} \} \]

![Diagram of Example I with max values]

Example II

- Compute the height of the AST.

\[ E_0 ::= E_1 E_2 \]
\[ \{ E_0.\text{height} := \text{MAX}(E_1.\text{height}, E_2.\text{height}) + 1 \} \]
\[ E ::= V \]
\[ \{ E.\text{height} := 0 \} \]

![Diagram of Example II with height values]

Example I...

- We write a tree-walk evaluator implementing the semantic rules:

  ```plaintext
  PROCEDURE E (n: Node) IF n.\text{Kind}=Leaf THEN E = n.\text{value};
  ELSE E(n.\text{left}) = \text{MAX}(E(n.\text{left}), E(n.\text{right}), E(n.\text{right}.\text{max});
  END
  ```

Example II...

- We write a tree-walk evaluator implementing the semantic rules:

  ```plaintext
  PROCEDURE E (n: Node) IF n.\text{Kind}=Leaf THEN n.\text{height} := 0;
  ELSE E = \text{MAX}(E(n.\text{left}.\text{height}, E(n.\text{right}.\text{height}) + 1;
  END
  ```

![Diagram for Example II showing tree-walk evaluator]
Example III

- Write a type checker for Pascal expressions.
- These are the type rules for arithmetic in Pascal:

\[
\text{int} + \text{int} \Rightarrow \text{int} \\
\text{int} + \text{real} \Rightarrow \text{real}
\]

Anything else is an error.

- `type` is a synthesized enumerated attribute that can take on the values (int, real, char, error).

Example III...

\[
E_0 := E_1 + E_2 \\
E := \begin{cases} 
E_0 \text{.type := } & \\
\text{INT} & \text{IF } E_1 \text{.type=int } \& \ E_2 \text{.type=int } \text{THEN } \\
\text{REAL} & \text{ELSE IF } E_1 \text{.type=real } \& \ E_2 \text{.type=int } \text{THEN } \\
\text{REAL} & \text{ELSE IF } E_1 \text{.type=int } \& \ E_2 \text{.type=real } \text{THEN } \\
\text{ERROR} & \text{ELSE } \\
E_1 \text{.Kind=intConst THEN } & E_1 \text{.type := int; } \\
E_1 \text{.Kind=realConst THEN } & E_1 \text{.type := real; } \\
E_1 \text{.Kind=charConst THEN } & E_1 \text{.type := char; }
\end{cases}
\]
**Slide 7-12**

- Use to collect information from the leaves of a subtree.
- There must be one attribute evaluation rule for each synthesized attribute occurring on the left hand side of the production.

\[
L ::= R \ S \ { \ L.\text{Syn} := R.\text{Syn} } 
\]

**Inherited**

- Use to inform the nodes of a subtree of the environment (context) in which they occur.
- There must be one attribute evaluation rule for each inherited attribute occurring on the right hand side of the production.

\[
L ::= R \ S \ { \ R.\text{Inh} := L.\text{Inh}; \\
S.\text{Inh} := L.\text{Inh}; } 
\]

**Threaded**

- Use to collect information from the nodes of a subtree.
- Consists of one inherited and one synthesized attribute.

\[
L ::= R \ S \{ \\
R.\text{In} := L.\text{In}; \\
S.\text{In} := R.\text{Out}; L.\text{Out} := S.\text{Out}; } 
\]

\[
S ::= \ldots \{ \ S.\text{Out} := f(S.\text{In}); \} 
\]

\[
R ::= \ldots \{ \ R.\text{Out} := g(R.\text{In}); \} 
\]

**Example**

- INH: A.a, C.d; SYN: A.b, B.c;
- A ::= B C \{ \ C.d := B.c + 1; \\
A.b := A.a + B.c; \}

**Slide 7-13**

**Synthesized**

- Given its value during parsing.
- Output Available to later phases.
- Synthesized Computed from children's attributes.
- Inherited Computed from parent's and siblings' attributes.
- Threaded Computed from children's and parent's attributes.
**Inherited Attributes I...**

- Max is the largest value allowed in the constant expression.

\[
\begin{align*}
\text{TYPE T} &= [0..1000]; \\
\text{CONST C : T} &= 30 \times 30 + 125;
\end{align*}
\]

\[
\begin{align*}
\text{E}_0 &:= \text{E}_1 + \text{E}_2 \\
\{ &\quad \text{E}_0.\text{val} := \\
&\quad \text{IF } \text{E}_1.\text{val} + \text{E}_2.\text{val} > \text{E}_0.\text{Max} \\
&\quad \text{THEN error} \\
&\quad \text{ELSE } \text{E}_1.\text{val} + \text{E}_2.\text{val} \\
&\quad \text{E}_1.\text{Max} := \text{E}_0.\text{Max}; \\
&\quad \text{E}_2.\text{Max} := \text{E}_0.\text{Max}; \\
\}
\end{align*}
\]

(* Same as above but with * instead of +. *)

\[
\begin{align*}
\text{E}_0 &:= \text{Const} \\
\{ &\quad \text{E}_0.\text{val} := \text{IF Const.val} > \text{E}_0.\text{Max} \\
&\quad \text{THEN error ELSE Const.val} \\
\}
\end{align*}
\]

---

**Inherited Attributes I...**

**Slide 7–16**

**Slide 7–17**

**Slide 7–18**

**Slide 7–19**
**Inherited Attributes II**

- Often we must pass a value first up through one sub-tree, and then down another subtree. We need one synthesized and one inherited attribute.
- Example: assign types to declared variables. TypeSy in synthesized, Type is inherited.

```
VAR X, Y, Z : INTEGER;
```

```
VarDecl ::=
  IdList.Type
  
Var X, Y, Z : INTEGER;

IdList0
  IDENT: X Type:
  IdList1

IdList0
  IDENT: Y Type:
  IdList1
```

```
IdList0
  IDENT: Z Type:
  IdList1

IdList := NULL { }
```

**Slide 7-20**

**Inherited Attributes II...**

```
VarDecl ::= IdList TypeName
  
  
  
TypeCheck := IDENT
  
TypeSy := IDENT
  
THEN Int ELSE error ENDF

IdList0 ::= IDENT IdList1
  
  
  
IdList1.Type := IdList0.Type
  
SyTab.insert(IDENT);
  
SyTab.setType(IDENT,IdList0.Type);
```

```
IdList := NULL { }
```

**Slide 7-21**

**Inherited Attributes III**

- Check for multiply declared identifiers.
- Ids is an inherited attribute of type Set of Name that is passed down the tree. Declared identifiers are collected in Ids on the way down.

```
VAR X, Y, Z : INTEGER;
```

```
VarDecl ::= IdList.Type
  
Var X, Y, Z : INTEGER;

IdList0
  IDENT: X Ids: { }
  
IdList1

IdList0
  IDENT: Y Ids: {X} |
  
IdList1

IdList0
  IDENT: Z Ids: {X,Y}
  
IdList1

IdList := NULL { }
```

**Slide 7-22**

**Inherited Attributes III...**

```
VarDecl ::= IdList TypeName
  
  
  
TypeCheck ::= IDENT
  
TypeSy ::= IDENT
  
  
THEN Int ELSE error ENDF

IdList0 ::= IDENT IdList1
  
  
  
IdList1.Type := IdList0.Type
  
  
  
SyTab.insert(IDENT);
  
  
  
SyTab.setType(IDENT,IdList0.Type);
```

```
IdList := NULL { }
```

**Slide 7-23**
Threaded Attributes

- A threaded attribute is really the combination of two attributes, an inherited In-attribute, and a synthesized Out-attribute.
- The inherited attribute is used to gather information in a subtree, and the synthesized attribute is used to bring that information back up to the root.
- Threaded attributes are similar to Prolog’s Accumulator Pairs.

Slide 7–24

Threaded Attributes...

- We often need to count the number of elements in a subtree, or give each element a unique number.
- For example, when we process the list of actual parameters in a procedure call statement, we need to count the number of expressions in order to see if the count matches the number of formal parameters.
- They way to do this is to start out with the count 0 at the top of the subtree, pass that value down the tree, adding 1 to it whenever a new “countable” node is found.

Slide 7–25

Threaded Attributes...

- When a leaf is reached, we pass the value back up the tree until we reach an internal node which has children who have yet to be visited. The value is again incremented as it is passed into the unvisited subtree.
- This amounts to performing an inorder traversal.
- Passing a value down the tree is done using the inherited attribute In. Passing it up the tree is done using the synthesized attribute Out.

Slide 7–26

Threaded Attributes I

- Give each E-node a unique number. Also count the number of E-nodes.
- In is inherited.
- Out is synthesized.

\[
T ::= E \\
\{ E.In := 1; \}
\]

\[
E_0 ::= E_1\ E_2 \\
\{ \\
E_1.In := E_0.In + 1; \\
E_2.In := E_1.Out + 1; \\
E_0.Out := E_2.Out; \\
\}
\]

\[
E ::= \text{Const} \\
\{ E.Out := E.In \}
\]

Slide 7–27
Threaded Attributes I...

Threaded Attributes II

- Compute the size of a sequence of variable declarations.
- \( \text{size(INTEGER)} = 4 \), \( \text{size(BOOLEAN)} = 1 \).
- \( \text{In} \) is inherited and holds the current size. \( \text{Out} \) is synthesized.
- At any node \( n \), \( \text{Out} - \text{In} \) is the size of the variables declared in the subtree rooted at \( n \).
- After semantic analysis, \( \text{Out} \) of the root node will hold the total size of the variables in the declaration sequence

\[
\begin{align*}
\text{VAR } & \ A : \ \text{INTEGER}; \\
& \ \text{VAR } \ B, \ C : \ \text{BOOLEAN};
\end{align*}
\]

Slide 7–28

\[
\begin{align*}
\text{DeclSeq} & : = \text{Decl} \{ \ \text{Decl.In} \ := \ 0; \ \} \\
\text{Decl}_0 & : = \text{VarDecl} \ \text{Decl}_1 \\
& \{ \ \text{VarDecl.In} \ := \ \text{Decl}_0 . \text{In}; \\
& \quad \text{Decl}_1 . \text{In} \ := \ \text{VarDecl.Out}; \\
& \quad \text{Decl}_0 . \text{Out} \ := \ \text{Decl}_1 . \text{Out}; \ \}
\end{align*}
\]

\[
\begin{align*}
\text{Decl} & : = \text{NULL} \{ \text{Decl.Out} \ := \ \text{Decl.In}; \}
\end{align*}
\]

\[
\begin{align*}
\text{VarDecl} & : = \text{IdList.TypeName} \\
& \{ \ \text{IdList.Type} \ := \ \text{TypeName.TypeSy}; \\
& \quad \text{IdList.In} \ := \ \text{VarDecl.In}; \\
& \quad \text{VarDecl.Out} \ := \ \text{IdList.Out}; \ \}
\end{align*}
\]

\[
\begin{align*}
\text{IdList}_0 & : = \text{Ident.IdList}_1 \\
& \{ \ \text{IdList}_1 . \text{Type} \ := \ \text{IdList}_0 . \text{Type} \\
& \quad \text{IdList}_1 . \text{In} \ := \ \text{IdList}_0 . \text{In} + \\
& \quad \text{SIZE} (\text{IdList}_0 . \text{Type}) \}
\end{align*}
\]

\[
\begin{align*}
\text{IdList} & : = \text{NULL}\{\text{IdList.Out}:=\text{IdList.In};\}
\end{align*}
\]

Slide 7–29

Slide 7–30

Slide 7–31
Threaded Attributes III...

- This example is almost identical to the previous one.
- Instead of computing the size of the variable sequence, this time we’ll return the set of identifiers that were declared.
- This is also similar to the inheritance example where we sent a list of identifiers down the tree in order to check for multiply declared identifiers. This time we’re using a synthesized attribute as well, to bring the list of identifiers back up to the top of the sub-tree.

Threaded Attributes III...

- Compute the list of declared variables.
- In (inherited) holds the set of currently known variables.
- Out (synthesized) brings the set back up the tree.
- At any node $n$, Out-In is the set of variables declared in the subtree rooted at $n$.

VAR A : INTEGER;
VAR B, C : BOOLEAN;

Slide 7–32

\[
\text{DeclSeq} := \text{Decl} \{ \text{Decl.In} := \{ \}; \} \\
\text{Decl}_0 := \text{VarDecl} \; \text{Decl}_1 \\
\{ \text{VarDecl.In} := \text{Decl}_0.In; \\
\text{Decl}_1.In := \text{VarDecl.Out}; \\
\text{Decl}_0.Out := \text{Decl}_1.Out; \} \\
\text{Decl} := \text{NULL} \{ \\
\text{Decl.Out} := \text{Decl.In}; \} \\
\text{VarDecl} := \text{IdList} \; \text{TypeName} \\
\{ \text{IdList.Type} := \text{TypeName.TypeSy}; \\
\text{IdList.In} := \text{VarDecl.In}; \\
\text{VarDecl.Out} := \text{IdList.Out}; \} \\
\text{IdList}_0 := \text{Ident} \; \text{IdList}_1 \\
\{ \text{IdList}_1.Type := \text{IdList}_0.Type; \\
\text{IdList}_1.In := \text{IdList}_0.In \cup \{\text{Ident}\} \} \\
\text{IdList} := \text{NULL} \{ \text{IdList.Out} := \text{IdList.In} \}
\]

Slide 7–34

Slide 7–33

Slide 7–35
• This code implements the attribute evaluation rules of the previous slide.

PROCEDURE declSeq(n:Node)
n.Decl.In := { }; 
decl(n.Decl);
END;

PROCEDURE decl(n:Node)
IF n.Kind == NULL THEN
  n.Out := n.In;
ELSE
  n.VarDecl.In := n.In;
  varDecl(n.VarDecl);
  n.next.In := n.VarDecl.Out;
  decl(n.next);
  n.Out := n.next.Out;
END;
END;

Slide 7–36

PROCEDURE varDecl(n:Node)
typeName(n.TypeName);
n.IdList.Type := n.TypeName.TypeSy;
n.IdList.In := n.In;
idList(n.IdList);
n.Out := n.IdList.Out;
END;

PROCEDURE idList(n:Node)
IF n.Kind = NULL THEN
  n.Out := n.In;
ELSE
  n.IdList.Type := n.Type;
  n.IdList.In := n.In U {Ident};
idList(n.IdList);
n.Out := n.IdList.Out;
END;
END;

Slide 7–37

Attribute Grammar Tools

• There exist many systems for doing semantic analysis by attribute computation “automatically.” For example, in the COCKTAIL system you would describe the abstract syntax of expressions like this:

<table>
<thead>
<tr>
<th>Abstract Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
</tr>
<tr>
<td>Binary</td>
</tr>
<tr>
<td>IntConst</td>
</tr>
<tr>
<td>CharConst</td>
</tr>
<tr>
<td>Designator</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>&gt;.</td>
</tr>
<tr>
<td>&gt;.</td>
</tr>
</tbody>
</table>

• From this specification COCKTAIL would generate an module for constructing and traversing the AST.

Attribute Grammar Tools...

• From the semantic rules on the next slide, COCKTAIL produces a tree-walker that evaluates the attributes in the correct order.

• The DECLARE section lists which attributes each node in the AST will have. Each Expr node, for example, will have a synthesized output attribute named Type, of type tSymbol.

• The RULE section describes attribute computation rules, and gives context conditions (in CHECK-clauses).
Summary

- Input attributes are assigned values during parsing. These values may be used (but not changed) during semantic analysis.
- The values of output attributes are available to later compiler phases.
- The value of a synthesized attribute is determined from the values of its children. They are used to gather information from the leaves of the abstract syntax tree and to move this information up the tree (towards the root).
- An inherited attribute at a node gets its value by n's parent and siblings. Inherited attributes provide a mechanism for passing context information down the tree; i.e. to inform a node of the environment in which it exists.

Summary...

- A threaded attribute collects information contained in a subtree. It is a combination of an inherited attribute in which information is collected, and a synthesized attribute which passes the collected information back up the tree.
- Attribute grammars are declarative. Hence, attribute computations shouldn't have side-effects. Sometimes, they still do, e.g. when issuing error messages.
- There are tools for building semantic analyzers from attribute grammar specifications.

Readings and References

- You can get COCKTAIL here: http://www.first.gmd.de/cocktail. A more current, commercial, version can be bought here: http://www.cocolab.de/.
- ELI (http://www.cs.colorado.edu/~eliuser) and ANTLR (http://www.antlr.org) are other compiler tools.