Attribute Grammars

1 Introduction

- The implementation of programming language features has been to define these techniques in terms of semantic routines that we have assumed would be called directly by the parser.
- Attribute grammars, developed by Knuth, were proposed as a means of including semantics with the context–free syntax of a language.

2 Attribute Grammars

2.1 Introduction

- An *attribute grammar* is a translation grammar with the following additional specifications:
  1. Each symbol has an associated finite set of attributes. These attributes represent information associated with the symbol, such as type, value, code sequence, etc.
  2. The associated attributes can be divided into two classes: *synthetic* and *inherited.* Synthetic attributes pass information up the syntax tree, and inherited attributes pass information down the syntax tree.
  3. Terminal symbols may have only synthetic attributes. These are supplied with the terminal by the scanner.
  4. Nonterminal symbols may have both inherited and synthetic attributes. All inherited attributes of the start symbol are supplied as initial values before evaluation begins.
  5. Each context–free production has a set of associated attribute evaluation rules.
  6. Given a production, the value of an inherited attribute on the right–hand side is computed as a function of the other attributes in the production.
  7. Given a production, the value of a synthetic attribute on the left-hand side is computed as a function of other attributes in the production.

2.2 Example 1

- Consider the grammar is Figure 1 that generates constant expressions using the operators $+$ and $*$.
- All symbols, both terminals and nonterminal, have a synthetic attribute called *val.*
- Consider the attributed syntax tree for $12 + 3 * 6$.

Productions	*Attribute Rules*
$E^1 \rightarrow E^2 + T$	$E^1.val = E^2.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T^1 \rightarrow T^2 * F$	$T^1.val = T^2.val + F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow a$	$F.val = a.val$
$F \rightarrow (E)$	$F.val = E.val$

Figure 1: Attribute grammar for arithmetic expressions

Attributes

VL :	$type\{inh\}$
TYPE :	$vtype\{syn\}$
V :	$name\{syn\}$

Productions	*Attribute Rules*
$Decl \rightarrow$ **TYPE V** VL	$givetype(V.name, TYPE.vtype)$; $VL.type = TYPE.vtype$
$VL \rightarrow$, **V** VL^1	$givetype(V.name, VL.vtype)$; $VL^1.vtype = VL.vtype$
$VL \rightarrow \lambda$	

Figure 2: Grammar G_1

2.3 Attribute rules

- Attribute rules and information flow are often straightforward.
- *Nothing* is assumed about attribute rules.
- It is possible the attribute evaluation rules may not even terminate.
- Attribute rules may have side effect (such as generating code).
- The way information flows in a syntax tree is determined by the functional dependencies of the attribute rules. Often this is simply top–down or bottom–up.

2.4 Example 2

- The following shows an example of attributes that need to flow down a parse tree. The example is of declarations.
- Consider the grammar, G_1, in Figure 2 that only has mainly inherited attributes.
- Consider the parse tree for the following example: $\mathbf{TYPE}_{real}$ $\mathbf{V}_1$, $\mathbf{V}_2$, $\mathbf{V}_3$

2.5 Example 3

- The grammar, G_2, (shown in Figure 3) shows very complex flow patterns.

Attributes

$S:$	$A\{inh\}, B\{syn\}$
$X:$	$C\{inh\}, D\{syn\}$
$Y:$	$E\{inh\}, F\{syn\}$
$Z:$	$H\{inh\}, G\{syn\}$

Productions		*Attribute Rules*
$S \rightarrow XYZ$	$Z.H = S.A$	$\{inh \leftarrow inh\}$
	$X.C = Z.G$	$\{inh \leftarrow syn\}$
	$S.B = X.D - 2$	$\{syn \leftarrow syn\}$
	$Y.E = S.B$	$\{inh \leftarrow syn\}$
$X \rightarrow x$	$X.D = 2 * X.C$	$\{syn \leftarrow inh\}$
$Y \rightarrow y$	$Y.F = Y.E * 3$	$\{syn \leftarrow inh\}$
$Z \rightarrow z$	$Z.G = Z.H + 1$	$\{syn \leftarrow inh\}$

Figure 3: Grammar G_2

- The order of evaluation is thus $S.A$ (supplied as an initial value), $Z.H$, $Z.G$, $X.C$, $X.D$, $S.B$, $Y.E$, $Y.F$.
- This sort of attribute flow can drive an evaluator crazy. Many attribute evaluators limit the kinds of attribute flow they will allow (such as left to right).
- It is possible to have circularity (replace Z.H = S.A with Z.H = S.B) in the definitions. We are in a loop because S.B is indirectly defined in terms of itself. No legitimate order of attribute evaluation exists. There are tests for circularity but it is exponential in the size of the grammar tested.
- The evaluators we will consider will not allow circulatority. We will consider some of these evaluators next.

2.6 Summary of Attributes

- Contrasting the above.

Synthesized attributes	information passed up parse tree	rules associated with left hand side occurrences of nonterminal
Inherited attributes	information passed down parse tree	rules associated with right hand side occurrences of nonterminals

3 Evaluators

- This section looks at attribute evaluators. The first section deals with a restricted form

of attribute grammars such that the evaluator is simple. The next two sections deal with evaluators that traverse the tree multiple times or on the fly.

3.1 Simple Assignment Form and Action Symbols

- A common evaluation rule is to simply allow the assignment of an attribute value or a constant to another attribute. These are called *copy rules.*
- Because these copy rules can be handled automatically by an attribute evaluator, a special form of attribute grammar, called *simple assignment form*, is often used.
- In this form, *action symbols* are used to realize all nontrivial attribute rules. Input values to the action symbol are its inherited attributes, and output values are its synthetic attributes.
- Action symbols can be used to enforce context–sensitive restrictions. They can also be used to both check and compute attribute values. They are a convenient abstractions of semantic routines.
- Attribute evaluators can automate everything but action symbols which are written by hand.
- Consider the example, grammar G_3 (shown in Figure 4). This is the expression grammar with the added feature that a maximum value is passed along. If the computed value gets larger than the maximum, an error is given.
- Consider the evaluation of $30 * 30 + 125$ where Max $= 1000$. Show the syntax tree and then decorate the tree with the attributes.

3.2 Tree–Walk Attribute Evaluators

- This section considers ways in which attributes can be evaluated.
- Assume that the syntax tree has been created and labeled with the inherited attributes of the start symbol and the synthetic attributes of the terminals.
- The syntax tree is then traversed until all the attributes have been evaluated.

3.2.1 Left–to–Right Traversal Methods

- Common traversal orders are depth–first or left–to–right.
- If necessary, more than one traversal of the syntax tree is used.
- Consider the traversal method shown in Figure 5 that will work for any *noncircular* attribute grammar.
- At least one attribute will be evaluated during a pass.
- Worst case time of $O(n^2)$.

Attributes

Inh(E) = Inh(T) = Inh(P)	= {Max}
Syn(E) = Syn(T) = Syn(P)	= {Val}
Inh(<Add>) = Inh(<Mult>)	= {v1,v2}
Syn(<Add>) = Syn(<Mult>)	= {Result}
Inh(<Check>)	= {Val,Max}
Inh(<Check>)	= {Result}

Productions	*Copy Rules*
$E^1 \rightarrow E^2 +$ T <Add>	<Add>.v1 = E^2.Val
	<Add>.v2 = T.Val
	<Add>.Max = E^1.Max
	E^1.Val = <Add>.Result
	E^2.Max = E^1.Max
	T.Max = E^1.Max
E → T	E.Val = T.Val
	T.Max = E.Max
$T^1 \rightarrow T^2 *$ P <Mult>	<Mult>.v1 = T^2.Val
	<Mult>.v2 = P.Val
	<Mult>.Max = T^1.Max
	T^1.Val = <Mult>.Result
	T^2.Max = T^1.Max
	P.Max = T^1.Max
T → P	P.Max = T.Max
	T.Val = P.Val
P → C <Check>	<Check>.Max = P.Max
	<Check>.Val = C.Val
	P.Val = <Check>.Result
P → (E)	E.Max = P.Max
	P.Val = E.Val

- Definition of the action symbols:

  <Add>: if v1 + v2 > Max **then** Error **else** Result = v1 + v2 **end if**

  <Mult>: if v1 * v2 > Max **then** Error **else** Result = v1 * v2 **end if**

  <Check>: if Val > Max **then** Error **else** Result = Val **end if**

Figure 4: Grammar G_3

```
do
  VisitNode(S) /* S is start symbol */
while (attributes remain to be evaluated)

void VisitNode (N : Node)
{
  if (N is a nonterminal)
    for (i = 1; i <= M; i++)
      if (Xi is not a terminal) /* i.e., nonterminal or action */
      {
        Evaluate all possible inherited attributes of Xi
        VisitNode(Xi)
      } /* if */
  Evaluated all possible synthetic attributes of N
} /* VisitNode */
```

Figure 5: Tree evaluator

- Consider grammar G_2 in Figure 3 with complicated flow.
  - After one pass the attributes Z.H and Z.G are evaluated. Of course the the initial value of S.A is available.
  - After the second pass the attributes S.B, X.C, and X.D are evaluated.
  - Finally after a third pass the attributes Y.E and Y.F are evaluated.
- This is a very general and crude algorithm. It would be nice to limit the number of passes required to evaluated the attributes.
- We want to consider the case in which one pass suffices. Attribute grammars for which one left–to–right pass always allows all attributes to be evaluated are termed *L–attributed.* An attribute grammar is L–attributed if and only if:
  - Each inherited attribute of a right–hand–side symbol depends only on inherited attributes of the left–hand–side and arbitrary attributes of symbols to the *left* of the given right–hand–side symbol.
  - Each synthetic attribute of the left–hand–side depends only on inherited attributes of that symbol and arbitrary attributes of right–hand–side symbols.
  - Each synthetic attribute of an action symbol depends only on its inherited attributes.
- These limitations on attribute flow, in effect, assume the attribute evaluation in the order shown in Figure 6 for a production $X \rightarrow Y_1 \ldots Y_n$.
- Note that attribute grammars G_1 and G_3 (but not G_2) are L–attributed.

Evaluate X's inherited attributes.
Evaluate Y_1's inherited attributes.
Call VisitNode(Y_1) to get Y_1's synthetic attributes.
.
.
.
Evaluate Y_n's inherited attributes.
Call VisitNode(Y_n) to get Y_n's synthetic attributes.
Evaluate X's synthetic attributes.

Figure 6: Evaluation for L–Attributed grammars

- We can contrive other traversal methods, but there are grammars that still take $O(n^2)$ evaluation time.
- In summary, depth–first traversals (left–to–right or right–to–left) are very general, but except in the L–attributed case, they are quite inefficient in that nodes are repeatedly visited that either can't be evaluated or that already have been entirely evaluated.

3.2.2 Alternate Traversal Methods

- Consider the idea: Every nonterminal and action symbol needs to be visited at least once but, once visited, a symbol *need not be visited again* until at least one more attribute for that symbol is made available (that is, is evaluated).
- This simply says that all information flow *into* a subtree is through its root, and thus a subtree evaluation is keyed by evaluating the root's attributes.
- Consider the improved VisitNode routine shown in Figure 7.
- `State(A)` is the *set of attributes* of `A` evaluated when `A` was last visited. NV is used if `A` has never been visited.
- All terminal symbols have an initial state equal to the set of all the terminal's synthetic attributes. The start symbol has an initial state equal to the set of inherited attributes.
- `Atr(A)` gives all the attributes currently evaluated for `A`.
- Nodes are only visited if new attributes have become available since the last visit. This gives evaluation in linear time.
- Consider grammar G_2 again. We start with Atr(S) = {A} and call VisitNode2(S):
  1. Immediately evaluate Z.H so Atr(S) = {A}, Atr(Z) = {H}.
  2. Now Atr(X) = $\emptyset \neq$ State(X) = {NV}, so X is visited.
  3. VisitNode2(X). No attributes can be evaluated, so we set State(X) = $\emptyset$ and return.
  4. Now State(X) = Atr(X) = $\emptyset$, so we visit Y.

```
void VisitNode2 (N : Node)
{
  if (N is an Action Symbol)
    evaluated all possible attributes in N
  else  /* N is a nonterminal */
    while (TRUE)
    {
      Evaluate all possible attributes in the production rooted by N
      if (there exists an offspring, Xi, of N such that
           State(Xi) ~= Atr(Xi))
        VisitNode2(Xi)
      else
        break
    } /* while */
  State(N) = Atr(N)
} /* VisitNode2 */
```

Figure 7: New evaluation routine

5. VisitNode2(Y). No attributes can be evaluated, so we set State(Y) = $\emptyset$ and return.
6. Now we have State(Y) = Atr(Y) = $\emptyset$, so we visit Z.
7. VisitNode2(Z). We evaluate Z.G, and State(Z) becomes {G,H}.
8. Now X.C can be evaluated. Since Atr(X) = C $\neq$ State(X) = $\emptyset$, we visit X.
9. VisitNode2(X). We evaluate X.D, and State(X) becomes ={C,D}.
10. Now S.B and Y.E can be evaluated. Since Atr(Y) = E $\neq$ State(Y) = $\emptyset$, we visit Y.
11. VisitNode2(Y). We evaluate Y.F, and State(Y) becomes {E,F}.
12. No more attributes can be evaluated and no more nodes need to be visited, so we are done.

3.3 On–the–fly Attribute Evaluators

- Unlike tree–walk evaluators, on–the–fly evaluators evaluate attributes in conjunction with a parse rather than after it.
- As such, they are good models of one–pass syntax–direct compilers.
- These kinds of evaluators discard attribute values when they are no longer needed in order to evaluate other attributes. Thus translations are usually realized via side effect (code or IR generation) or by building a translation into a synthetic attribute of the goal symbol.

3.3.1 LL(1) L–Attributed Evaluators

- Assume the following:
  - Attribute grammar is L–attributed.
  - Underlying CFG is LL(1).
  - Grammar is in simple assignment form.
- Evaluator uses an attribute stack (really just a semantic stack).
- When a nonterminal is prediced, its inherited attributes are pushed onto the stack.
- As the right–hand–side of a production is recognized, inherited and then synthetic attributes of each symbol are pushed onto the stack.
- When the entire right–hand–side is recognized, all attributes of the right–hand–side are popped and the synthetic attributes of the left–hand–side are pushed.
- Consider what would happen for $X \rightarrow YZ$:
  - Push inherited attributes of X.
  - Push inherited attributes of Y.
  - Push synthetic attributes of Y (after recognizing Y).
  - Push inherited attributes of Z.
  - Push synthetic attributes of Z (after recognizing Z).
  - Pop the attributes of the right–hand–side and push synthetic attributes of X.
- There needs to be a means of manipulating the attribute stack. To do this, *copy symbols* are introduced. These are essentially action symbols that move attributes rather than computing them. These copy symbols are automatically generated from copy rules and appear when inherited or synthetic attributes are manipulated.
- A simple example will help. Consider Figure 8 that is a simplified expression grammar. It is LL(1), L–attributed, and in simple assignment form. Copy symbols are denoted #n. Replacing copy rules with copy symbols generates the grammar in Figure 9.
- Consider the parse of $10 + 11 + 12$. Show the attribute stack, the parse stack, and the remaining input.
- There are obvious optimizations that would be preformed in practice.

3.3.2 LR S–Attributed Evaluators

- The advantage of LR parsers over LL parsers is their ability to delay production recognition until after the entire right–hand–side has been recognized.
- For attribute evaluation, this advantage limits the attribute flow that can be accommodated.

Attributes

Syn(E) = Syn(T) = Syn(T–List) = Syn(C)	= {Val}
Inh(T–List)	= {LeftVal}
Inh(<add>) = {v1,v2}	
Syn(<add>) = {Result}	

Productions	*Attribute Rules*
E → T T–List	E.Val = T–List.Val
	T–List.LeftVal = T.Val
T → C	T.Val = C.Val
$T\text{–List}^1$ → + T <add> $T\text{–List}^2$	<add>.v1 = $T\text{–List}^1$.LeftVal
	<add>.v2 = T.Val
	$T\text{–List}^2$.LeftVal = <add>.Result
	$T\text{–List}^1$.Val = $T\text{–List}^2$.Val
T–List → λ	T–List.Val = T–List.LeftVal

Figure 8: LL(1) grammar

E	→ T #1 T–List #2
T	→ C #3
T–List	→ + T #4 <add> #1 T–List #5
T–List	→ #1

#1:	Push copy of Top Element
#2:	Temp = Top Element; Pop 3; Push Temp
#3:	None {Equivalent to Temp = Top; Pop 1; Push Temp}
#4:	Push copy of Top – 1; Push copy of Top – 1
#5:	Temp = Top; Pop 6; Push Temp

Figure 9: Transformed grammar

Attributes

Syn(E) = Syn(C)	= {Val}
Syn(<add>)	= {Result}
Inh(<add>)	= {v1,v2}

Productions	*Attribute Rules*
$E^1 \rightarrow E^2$ + C <add>	<add>.v1 = E^2.Val
	<add>.v2 = C.Val
	E^1.Val = <add>.Result
E → C	E.Val = C.Val

Figure 10: S–attributed grammar

E → E + C #1 <add> #2
E → C #3

#1: Push copy of Top – 1; Push copy of Top – 1
#2: Temp = Top; Pop 5; Push Temp
#3: No action {In effect Temp = Top; Pop 1; Push Temp}

Figure 11: Transformed grammar

- LR techniques are limited to S–attributed grammars. An attribute grammar is S–attributed if and only if:
  - It is L–attributed.
  - Nonterminals have only synthetic attributes.
  - All action symbols (and copy symbols) occur to the right of all terminals and nonterminals in a right–hand–side.
- Consider the S–attributed grammar shown in Figure 10 and the results grammar when copy symbols are transformed in Figure 11.

3.3.3 LR LC–Attributed Evaluators

- Not allow inherited attributes is an unattractive alternative.
- In general, productions need not be recognized at their extreme left (as in LL parsing) or their extreme right (as in LR parsing) but rather they can be recognized somewhere in the middle of the right–hand–side.
- The right–hand–side of a production, $A \rightarrow \alpha\beta$, can be divided into two pieces, the left corner (α) and the trailing part (β).
- By definition, any production can be correctly recognized after the left corner is process.
- In LL(1) grammars, the left corner is always empty.

Attributes

Inh(E) = {Max}	Syn(E) = {Val}
Syn(C) = {Val}	
Inh(<Add> = {v1,v2,Max}	Syn(<Add> = {Result}
Inh(<Check>) = {Val,Max}	Syn(<Check>) = {Result}

Productions	*Attribute Rules*
$E^1 \rightarrow E^2$ & + C <Add>	E^2.Max = E^1.Max
	<Add>.v1 = E^2Val
	<Add>.v2 = C.Val
	<Add>.Max = E^1.Max
	E^1.Val = <Add>.Result
E → & C <Check>	<Check>.Val = C.Val
	<Check>.Max = E.Max
	E.Val = <Check>.Result

Figure 12: Grammar G_4

- In LR(1) grammars, the left corner can sometimes comprise the entire right–hand–side.
- In LR(1) grammars, left corners are generally quite small.
- LC attributed grammar is defined to have the following characteristics:
  - L–attributed.
  - No nonterminals that occur in a left corner that has any inherited attributes.
  - No action symbols occur in the left corner.
- Note that left corners allow S–attributed flow while trailing parts allow L–attribute flow.
- Conceptually, this is how this works:
  - Create a special recognition symbols delimiting left corners.
  - Action symbols and copy symbols only appear in the trailing part.
  - Thus a production might appear as $A \rightarrow XY\& < A1 > Z < A2 >$.
  - These productions can be parsed using ordinary LR–type techniques.

3.3.4 The Limited Use of Inherited Attributes in Left Corners

- Sometimes inherited attributes are still needed in left corners.
- Consider grammar G_4 in Figure 12.
- Grammar G_4 is not LC–attributed since E, a left corner symbol, has an inherited attribute.

E $\rightarrow$ E & + C #1 <Add> #2
E $\rightarrow$ & C #3 <Check> #4

#1: Push Top – 1; Push Top – 1; Push Top – 4
#2: Temp = Top; Pop 6; Push Temp
#3: Push Top; Push Top – 2
#4: Temp = Top; Pop 4; Push Temp

Figure 13: Grammar with copy rules optimized away

- Many times copy symbols needed for inherited attributes in left corners can be optimized away.
- In this example, a copy of E^1.Max to E^2.Max can be obviated by letting them share the same stack location.
- Copies of the top i $(i \leq 1)$ stack locations can be optimized away by sharing.
- Removing this copy in the left corner results in the grammar shown in Figure 13.

4 An Example

- Consider the following attribute grammar for if–statements and while–statements.

S → **if** E **then** L **end if**	E.case = false E.label = S.next L.next = S.next S.code = E.code & L.code
S → **if** E **then** L^1 **else** L^2 **end if**	E.case = false E.label = NewLabel L^1.next = S.next L^2.next = S.next S.code = E.code & L^1.code & Generate(JUMP, S.next) & Generate(LABEL, E.label) & L^2.code & Generate(LABEL, S.next)
S → **while** E **do** L **end while**	E.case = false E.label = S.next S.begin = NewLabel L.next = S.begin S.code = Generate(LABEL, S.begin) & E.code & L.code & Generate(LABEL, S.begin)
S → OtherS	S.code = OtherS.code
L → S	S.next = L.next L.code = S.code
L → L^1 S	L^1.next = NewLabel S.next = L.next L.code = L^1.code & Generate(LABEL, L^1.next) & S.code

$E \rightarrow E^1$ BoolOp E^2	E^2.label = E.label
	E^2.case = E.case
	if BoolOp.operator = OrElseOp **then**
	  E^1.case = true
	  if E.case **then**
	  E^1.label = E.label
	  E.code = E^1.code & E^2.code
	  else
	  E^1.label = NewLabel
	  E.code = E^1.code & E^2.code &
	  Generate(LABEL, E^1.label)
	  end if
	else – BoolOp.operator = AndThenOp
	  E^1.case = false
	  if E.case **then**
	  E^1.label = NewLabel
	  E.code = E^1.code & E^2.code &
	  Generate(LABEL, E^1.label)
	  else
	  E^1.label = E.label
	  E.code = E^1.code & E^2.code
	  end if
	end if

$E \rightarrow$ **not** E^1	E^1.label = E.label E^1.case = **not** E.case E.code = E^1.code
$E \rightarrow$ (E^1)	E^1.label = E.label E^1.case = E.case E.code = E^1.code

$E \rightarrow id^1$ RelOp id^2	**if** E.case **then**
	  E.code = Generate(BR, RelOp.operator, id^1.loc,
	  id^2.loc, E.label)
	else
	  E.code = Generate(BR, complement(RelOp.operator),
	  id^1.loc, id^2.loc, E.label)
	end if

$E \rightarrow$ true	**if** E.case **then**
	  E.code = Generate(JUMP, E.label)
	end if

$E \rightarrow$ false	**if not** E.case **then**
	  E.code = Generate(JUMP, E.label)
	end if