Attribute grammars

Formal framework based on grammar and parse tree

Idea: "attribute" the tree

- can add attributes (*fields*) to each node
- specify equations to define values
- both inherited and synthesized attributes

Attribute grammars are very general. Can be used for

- infix to postfix translation of arithmetic expressions (*ASU p.34*)
- type checking (*context-sensitive analysis*)
- construction of intermediate representation (*AST*)
- desk calculator (*interpreter*)
- code generation (*compiler*)

Attribute grammars

Aho, Sethi, & Ullman describe *syntax-directed definitions*. These are just *attribute grammars* by another name.

Attribute grammar

- generalization of context-free grammar
- each grammar symbol has an associated set of attributes
- augment grammar with rules defining attribute values
- high-level specification, independent of evaluation scheme
  - Note: **translation scheme** has evaluation order

Dependencies among attributes

- values are computed from constants & other attributes
- **synthesized attribute** - value computed from children
- **inherited attribute** - value computed from siblings & parent
- *key notion:* induced dependency graph

Attribute grammars

Note:

- terminals can be associated with values returned by the scanner. These input values are associated with a **synthesized** attribute.
- distinguished non-terminal (*start symbol*) cannot have inherited attributes.
- synthesized attributes of a grammar symbol can depend on inherited attributes of the *same* symbol.
- semantic rules are defined for each production separately
- semantic rules associated with a rule $A ::= \alpha$ have to specify the values for all
  - synthesized attributes for A (*root*)
  - inherited attributes for grammar symbols in α (*children*)

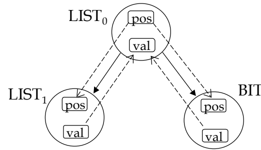
Example attribute grammar

A grammar to evaluate signed binary numbers

Production	Semantic Rules
1 NUM ::= SIGN LIST	LIST.pos ← 0
	NUM.val ← if SIGN.neg
	then -LIST.val
	else LIST.val
2 SIGN ::= +	SIGN.neg ← false
3 SIGN ::= -	SIGN.neg ← true
4 LIST ::= BIT	BIT.pos ← LIST.pos
	LIST.val ← BIT.val
5 $LIST_0$ ::= $LIST_1$ BIT	$LIST_1$.pos ← $LIST_0$.pos + 1
	BIT.pos ← $LIST_0$.pos
	$LIST_0$.val ← $LIST_1$.val + BIT.val
6 BIT ::= 0	BIT.val ← 0
7 BIT ::= 1	BIT.val ← $2^{BIT.pos}$

Example

Production	Semantic Rules
5 $LIST_0$ ::= $LIST_1$ BIT	$LIST_1$.pos ← $LIST_0$.pos + 1
	BIT.pos ← $LIST_0$.pos
	$LIST_0$.val ← $LIST_1$.val + BIT.val



Note:

- semantic rules define a partial dependency graph
- structure can be used to derive characteristics of generated total dependency graphs

Attribute grammars

The attribute dependency graph

- nodes represent attributes
- edges represent the flow of values
- graph is specific to parse tree
- size is related to parse tree's size
- can be built alongside parse tree

The dependency graph must be acyclic

Evaluation order

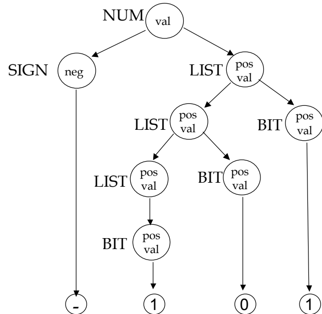
- Topological sort of the dependency graph to order attributes
- using this order, evaluate the rules

This order depends on both the grammar and the input string

Example attribute grammar

Example

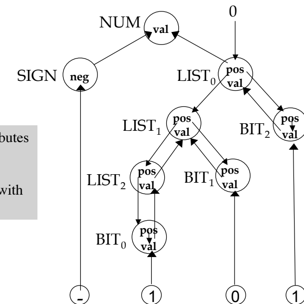
Parse tree for -101



Example grammar dependency graph

- val and neg are *synthesized* attributes
- pos is an *inherited* attribute

$LIST_0$.pos is an *inherited* attribute with an empty dependency set.



Attribute grammars

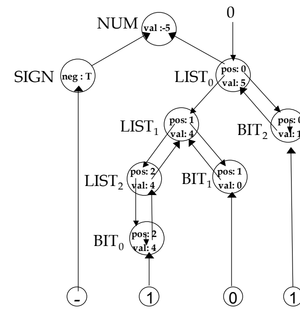
A topological order for the example

1. SIGN.neg
2. $LIST_0$.pos
3. $LIST_1$.pos
4. $LIST_2$.pos
5. BIT_0.pos
6. BIT_1.pos
7. BIT_2.pos
8. BIT_0.val
9. $LIST_2$.val
10. BIT_1.val
11. $LIST_1$.val
12. BIT_2.val
13. $LIST_0$.val
14. NUM.val

Evaluate in this order

Yields NUM.val: -5

Example grammar final result



The evaluation process is also called *decorating the parse tree*

Classification of evaluation methods

(*ASU*'s taxonomy)

Parse-tree methods (dynamic)

1. build the parse tree
2. build the dependency graph
3. topological sort the graph
4. evaluate it (cyclic graph fails)

Rule-based methods (treewalk)

1. analyze rules at compiler-generation time
2. determine a static ordering at that time
3. evaluate nodes in that order at compile time

Ad-hoc methods (passes, dataflow)

1. ignore the parse tree and grammar
2. choose a convenient order and use it
   (forward-backward passes, alternating passes)

Problems

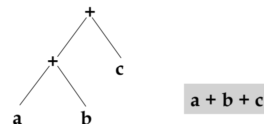
- circularity
- best evaluation strategy is grammar dependent

Example: AST construction

***Abstract Syntax Tree* (AST):** Intermediate program representation.

***Syntax directed translation*:** Uses parser as a driver to compute context-sensitive information, to build intermediate representations, and/or to generate code.

Production	Semantic Rules
1 E' ::= E	E'.ptr ← E.ptr
2 E ::= E_1 + T	E.ptr ← create-node("+", E_1.ptr, T.ptr)
3 E ::= T	E.ptr ← T.ptr
4 T ::= id	T.ptr ← create-leaf("id", id.symEntry)



Example: AST construction

stack	input	action
$	a+b+c	shift
$a	+b+c	reduce 4 {T.ptr:=create-leaf}
$(T,T.ptr)	+b+c	reduce 3 {E.ptr:=T.ptr}
$(E,E.ptr)	+b+c	shift
$(E,E.ptr)+	b+c	shift
$(E,E.ptr)+b	+c	reduce 4 {T.ptr:=create-leaf}
$(E,E.ptr)+(T,T.ptr)	+c	reduce 2 {E.ptr:=create-node}
$(E,E.ptr)	+c	shift
$(E,E.ptr)+	c	shift
$(E,E.ptr)+c		reduce 4 {T.ptr:=create-leaf}
$(E,E.ptr)+(T,T.ptr)		reduce 2 {E.ptr:=create-node}
$(E,E.ptr)		reduce 1 {E'.ptr:=E.ptr}
$(E',E'.ptr)		accept

- for each grammar symbol, entry in stack stores attribute values
- synthesized attributes can be evaluated during reduction step
- at time of evaluation, all dependent attribute values are somewhere in the stack; in fact, position is exactly known since handle is unique.

Example: AST construction

Semantic rules can be implemented as operations on the stack.

Production	Semantic Rules
1 E' ::= E	E'.ptr[top]:=E.ptr[top]
2 E ::= E_1 + T	E.ptr[top-2]:= create-node("+",E_1.ptr[top-2],T.ptr[top]) top := top - 2
3 E ::= T	E.ptr[top]:=T.ptr[top]
4 T ::= **id**	T.ptr[top+1]:= create-leaf("**id**", **id**.symEntry) top := top + 1

top *is the stack pointer*

Attribute types

Synthesized attributes

derive value from constants and children

- **S-attributed grammar:** only synthesized attributes
- S-attributed grammars can be **evaluated in one bottom-up pass**
- useful in many contexts (*calculator, AS&U 281*)
- S-attributed grammar is good match for LR parsing

Inherited attributes

derive value from constants, siblings, and parent

- **L-attributed grammar**: Each inherited attribute of X_j, $1 \leq j \leq n$, on the right side of $A ::= X_1X_2, \ldots X_n$ depends only on
  - attributes of $X_1, \ldots X_{j-1}$, and
  - inherited attributes of A
- L-attributed grammars can be **evaluated in a single DFS top-down pass**

S-attributed ⊂ L-attributed

Evaluation Methods

- The most general method is to **construct an ordering from the parse tree itself**
- Define a graph as follows:
  - For each attribute E.a to be computed add a node in the graph.
  - If E.a depends on $E_1.a_1,\ldots,E_n.a_n$ then add directed edges from $E_i.a_i$ to E.a for $1 \leq i \leq n$
- A **topological sort** of the graph is any ordering $n_1,\ldots,n_k$ of the nodes such that edges of the graph are all from left-to-right in the ordering; i.e., a node appears in the ordering after all of the nodes it depends on.
  - **A topological sort is a legal evaluation order of the attributes**
  - *Note:* for the topological sort to make sense, there can be no cycles in the graph. Cyclically defined attributes are not legal.

Real World

- In practice, computing all of the attribute dependencies from the tree is rarely, if ever, done
- Instead, syntax-directed definitions are used where the attribute evaluation order can be determined from the actions
- The most important special case is **S-attributed grammars** (grammars with only synthesized attributes)
  - These attributes can be evaluated bottom-up during parsing

Evaluating S-attributed Definitions

- **Bottom-up evaluation**
- Keep a stack S parallel to parsing stack
  - consider production $A \rightarrow XY$ with rule A.val = X.val + Y.val
  - When reducing by $A \rightarrow XY$ the top of the S stack has X.val and Y.val
  - compute A.val
  - pop X.val and Y.val from S, push A.val
- Symmetric with reduce action on the parse stack
- Tools like Bison/Flex support S-attributed definitions

DFS (depth first search)

```
procedure DFS(n: node)
begin
   for each child m of n, from left to right do begin
        evaluate inherited attributes of m
        DFS(m)
   end
   evaluate synthesized attributes of n
End
```

Translation schemes

- Semantic actions are inserted within the RHS of rules
- Actions are " dummy" terminals on RHS
- **Evaluation order of semantic actions based on DFS walk**

This is the method used in your compiler

Example translation schemes

AST construction example:

1 E' ::= E {E'.ptr ← E.ptr}
2 E ::= E1 + T {E.ptr ← create-node("+",E1.ptr,T.ptr)}
3 E ::= T {E.ptr ← T.ptr}
4 T ::= id {T.ptr ← create-leaf("id",id.symEntry)}

Evaluation of signed binary numbers:

1 NUM ::= SIGN {LIST.pos ← 0} LIST
{NUM.val ← if SIGN.neg then -LIST.val
else LIST.val}
2 SIGN ::= + {SIGN.neg ← false}
3 SIGN ::= - {SIGN.neg ← true}
4 LIST ::= BIT {BIT.pos ← LIST.pos;
LIST.val ← BIT.val}
5 $LIST_0$ ::= $LIST_1$ {$LIST_1$.pos ← $LIST_0$.pos + 1} BIT
{BIT.pos ← $LIST_0$.pos;
$LIST_0$.val ← $LIST_1$.val + BIT.val}
6 BIT ::= 0 {BIT.val ← 0}
7 BIT ::= 1 {BIT.val ← $2^{BIT.pos}$}

Are these translation schemes correct?

L-attributed grammars

Restrictions for translation scheme to make sure that attribute grammar is L-attributed:

1. An inherited attribute for a symbol on the RHS of a rule must be computed in an action **before** that symbol.
2. An action cannot refer to an attribute of a symbol to the right of the action.
3. A synthesized attribute of the LHS can only be computed after all attributes it refers to are computed; place the action **at the end** of RHS of rule.

Example translation schemes

Let's try again

Evaluation of signed binary numbers:

1 NUM ::= SIGN {LIST.pos ← 0} LIST
{NUM.val ← if SIGN.neg then -LIST.val else LIST.val}
2 SIGN ::= + {SIGN.neg ← false}
3 SIGN ::= - {SIGN.neg ← true}
4 LIST ::= **{BIT.pos ← LIST.pos} BIT**
{LIST.val ← BIT.val}
5 $LIST_0$ ::= {$LIST_1$.pos ← $LIST_0$.pos + 1} $LIST_1$
{BIT.pos ← $LIST_0$.pos} BIT
{$LIST_0$.val ← $LIST_1$.val + BIT.val}
6 BIT ::= 0 {BIT.val ← 0}
7 BIT ::= 1 {BIT.val ← $2^{BIT.pos}$}

Example: Simple compiler

Assume a simple load/store architecture (RISC). The generated code uses a new register for each computed value.

Production	Semantic Rules
1 E' ::= id := E	E.reg# := 0 E'.code := E.code || "STORE addr(id) $R_{E.reg\#}$"
2 E ::= E_1 + T	E_1.reg# := E.reg# + 1 T.reg# := E.reg# + 2 E.code := E_1.code || T.code || "ADD $R_{E.reg\#}$ $R_{E1.reg\#}$ $R_{T.reg\#}$"
3 E ::= T	T.reg# := E.reg# E.code := T.code
4 T ::= id	T.code := "LOAD $R_{T.reg\#}$ addr(id)"

– | | is a concatenation operator
– addr(*id*) returns memory address of identifier *id*

What does the translation scheme look like?

BU evaluation of inherited attributes

Method to implement L-attributed definitions for bottom-up parsing:

- Can handle all L-attributed definitions corresponding to LL(1) grammars and some LR(1) (not all!).
- Basic idea:
  1. Transform grammar so all embedded actions of translation scheme occur at the end of RHS of some production (i.e. at a reduction). This will make sure that attribute values will be in the stack.
  2. Introduce copy rules if necessary to locate attribute values in the stack.
  3. Note: modifications to input attribute grammar must not change its LR-ness.

Attribute grammars

Advantages

- clean formalism
- automatic generation of evaluator
- high-level specification

Disadvantages

- evaluation strategy determines efficiency
- increased space requirements
- results distributed over tree