CS412/CS413

Introduction to Compilers

Tim Teitelbaum

Lecture 16: Attribute Grammars

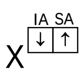
26 Feb 07

Attribute Grammars

- An extension of CFGs to define “semantics” of sentences in language
- Knuth, 1968
- Intuition:
  – Decorate each parse-tree node with attributes, i.e., variables defined by equations in terms of constants and neighboring attributes in the tree
  – Evaluate the attributes like a spreadsheet evaluates cells defined by equations, i.e., order of evaluation determined automatically

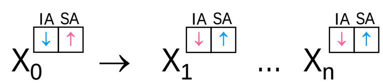
Attributes

- Let G be a context-free grammar $\langle V,\Sigma,S,\rightarrow\rangle$
- Associate with every $X \in (V \cup \Sigma)$ a set of attributes A(X)
- Notation. If $a \in A(X)$, we denote it X.a
- Let A(X) be partitioned into disjoint sets
  – synthesized attributes, SA(X)
  – inherited attributes, IA(X)

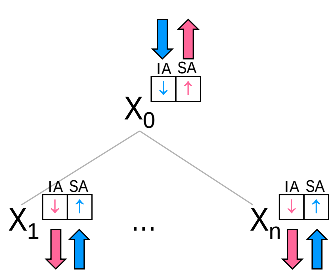


Occurrences

- Let p be a production $X_0 \rightarrow X_1 \ldots X_n$ of G
- Each X_i is a symbol occurrence of p
- Input(p) = $IA(X_0) \oplus SA(X_1) \oplus \ldots \oplus SA(X_n)$
- Output(p) = $SA(X_0) \oplus IA(X_1) \oplus \ldots \oplus IA(X_n)$
- Each attribute in Input(p) or Output(p) is an attribute occurrence of p



Input and Output Occurrences



Equations

- Let p be a production $X_0 \rightarrow X_1 \ldots X_n$ of G
- An attribute equation of p defines $a \in$ Output(p) in terms of attributes in Input(p) $\oplus$ Output(p)
- An attribute grammar is well formed if
  – IA(S) = $\emptyset$
  – SA(a) = $\emptyset$, for all $a \in \Sigma$
  – Every output attribute of every production has precisely 1 defining equation
- An attribute grammar is in normal form if only input attributes occur on RHS of equations

Example

- Productions
  - S → E
  - E → E + E
  - E → NUM
  - E → ID
  - E → **let** ID **=** E **in** E
- Sample sentence
  - **let** x **=** 1 **in let** y **=** x+1 **in** x+y
- Attributes
  - Inherited: E.env
  - Synthesized: S.value, E.value, NUM.value, ID.name

Example, cont.

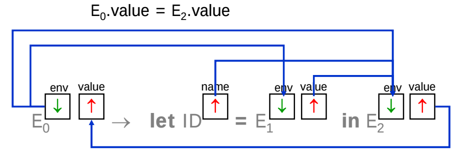
S → E
  E.env = EmptyEnvironment()
  S.value = E.value
$E_0 \to E_1 + E_2$
  E_1.env = E_0.env
  E_2.env = E_0.env
  E_0.value = E_1.value + E_2.value
E → NUM
  E.value = NUM.value
E → ID
  E.value = Lookup(ID.name, E.env)
E_0 → **let** ID **=** E_1 **in** E_2
  E_1.env = E_0.env
  E_2.env = Insert(ID.name, E_1.value, E_0.env)
  E_0.value = E_2.value

Direct Dependency Graph

- Let p be a production $X_0 \to X_1 \ldots X_n$ of G
- D_p, the direct dependency graph of p, is the directed graph ⟨A(p),E(p)⟩, where
  - Nodes: A(p) = Input(p) ⊕ Output(p)
  - Edges: E(p) = { ⟨a_1, a_2⟩ | a_2 depends on a_1 }
- An attribute grammar is locally acyclic if for every production p, D_p is acyclic

Example, cont.

E_0 → **let** ID **=** E_1 **in** E_2
  E_1.env = E_0.env
  E_2.env = Insert(ID.name, E_1.value, E_0.env)
  E_0.value = E_2.value

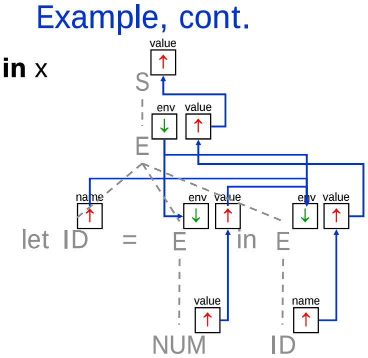


Dependency Graph

- Let T be a derivation tree for some x ∈ L(G)
  - Each subtree corresponding to production p is a production instance in T
  - Each symbol occurrence in p is a symbol instance in T
  - Each attribute occurrence in p is an attribute instance in T
  - Each edge in D_p is a dependence instance in T
- D(T), the dependency graph for T, has
  - Nodes: the attribute instances of T
  - Edges: the dependence instances of T

Example, cont.

let x **=** 1 **in** x



Noncircularity

- An attribute grammar is noncircular if for every derivation tree, T D(T) is acyclic
- We are only interested in noncircular grammars

Evaluation

- Given a derivation tree T, evaluate the attibute instances of T in topological order w.r.t. D(T)
- Dynamic evaluation: Obtain the topological order using either
  - topological sort, or
  - depth first search backwards from nodes of out-degree 0
- Static evaluation: Analyze the grammar in advance and determine tree traversal schemes with interleaved evaluations such that for any possible derivation tree T, evaluations will be in topological order

Topological Sort

```
W := ∅;
for each node n with indegree(n)=0 do
   W := W ∪ {n};
while W ≠ ∅ do
  select n from W;
  remove n from W;
  for each successor n’ of n do
      remove edge <n,n’>;
      if indegree(n’)=0 then W := W ∪ {n’}
```


S-attributed

- An attribute grammar is S-attributed iff it only has synthesized attributes.
- Evaluation: Use end-order traversal of derivation tree (e.g., during a bottom-up parse) to obtain topological evaluation order
- Yacc, Bison, and Cup only support S-attributed grammars

L-attributed

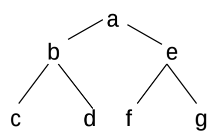
- Defined so that can be evaluated in one left-to-right pass, (e.g., during a top-down parse)
- Every RHS inherited attribute depends only on
  - LHS inherited
  - any RHS attribute to the left
- Every LHS synthesized attribute depends only on
  - LHS inherited
  - any RHS

Alternating Pass Evaluation

- Alternate between L-attributed and R-attributed passes.
- In pass i, all attributes evaluated in previous passes are known values available for during the evaluations during pass i
- An attribute grammar is alternating pass if there exists k alternating passes sufficient to evaluate any derivation tree T

Efficient Use of Sequential Storage

- Reverse of left-to-right endorder is right-to-left preorder (and vice-versa) so can make efficient use of sequential storage medium



Endorder: c d b f g e a

Right-to-left preorder: a e g f b d c