



University of
Arizona

CSc 453

Compilers and Systems Software

Christian Collberg
October 5, 2001

Attribute Grammars II

Copyright © 2001 C. Collberg

Optimizing Tree-Walkers

- Storing every attribute in the AST may take up a lot of space. Sometimes we can make some optimizations:
 1. Inherited attributes can be passed as input arguments to the recursive procedures.
 2. Synthesized arguments can be returned as function results (or as reference parameters).
- This won't work for **output attributes**, attributes that will be needed by later compilation phases.

```
PROCEDURE Program (n: Node);
  Std := {INT,REAL,CHAR,TRUNC,FLOAT};
  Decl(n.DeclSeq, ↓ {}, ↑ IdsOut, ↓ Std);
  xEnv := cons(IdsOut,StdEnv);
  Stat(n.StatSeq, ↓ xEnv);
END;
```

Slide 11-1

Optimizing Tree-Walkers...

```
PROCEDURE Program (n: Node);
  Std := {INT,REAL,CHAR,TRUNC,FLOAT};
  Decl(n.DeclSeq, ↓ {}, ↑ IdsOut, ↓ Std);
  xEnv := cons(IdsOut,StdEnv);
  Stat(n.StatSeq, ↓ xEnv);
END;

PROCEDURE Decl (n: Node);
  IdsIn:SyTabT; VAR IdsOut:SyTabT;
  Env:EnvT;
  ...
END;

PROCEDURE Assign (n: Node; Env:EnvT);
  Des(↓ Env, ↑ DesType);
  Expr↓(Env, ↑ ExprType);
  IF DesType ≠ ExprType THEN ...
END;
```

Slide 11-2

Dynamic Tree-Walkers

- The major problem with building a tree-walk evaluator is to find an order (a **visit sequence**) in which to traverse the AST and evaluate the attributes.
- So far, we have built **Static Evaluators**. With this type of evaluator the visit sequence is determined by the compiler designer at compiler construction time.
- If we're not concerned with efficiency, then we can build a **Dynamic Evaluator**, one for which the visit sequence is determined at **compile time** (i.e. when we're performing semantic analysis).

Slide 11-3

Dynamic Tree-Walkers...

1. Build the abstract syntax tree during parsing.
2. Build the dependency graph for the attributes of the tree.
 - The nodes of the graph are the attributes of the tree.
 - There's an edge from node a to node b if b depends on a , i.e. if a has to be computed before b .
3. Perform a topological sort of the dependency graph.
4. If a cycle is detected abort the compile: "Cyclic evaluator, compilation aborted".
5. Otherwise, evaluate the tree attributes in the order computed.

Slide 11-4

Topological Sorting

- Problem: How to get dressed? I can't put my clothes on in an arbitrary order: socks have to be put on before shoes, shirt before tie, and so on.
- Each garment becomes a node in a DAG, and there's an edge from u to v if I have to put on u before v .

_____ Topological Sorting: _____

"Order the nodes of the graph G in a sequence such that if $x \rightarrow y$ is an edge in G then x comes before y in the ordering."

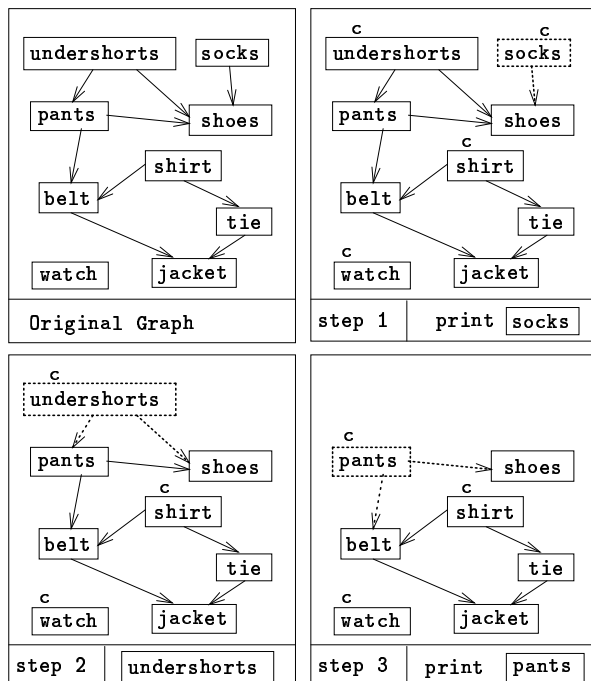
_____ Simple Algorithm: _____

Repeat until no more nodes:

- Pick a node n without predecessors.
- Print n .
- Delete n and all its outgoing edges.

Slide 11-5

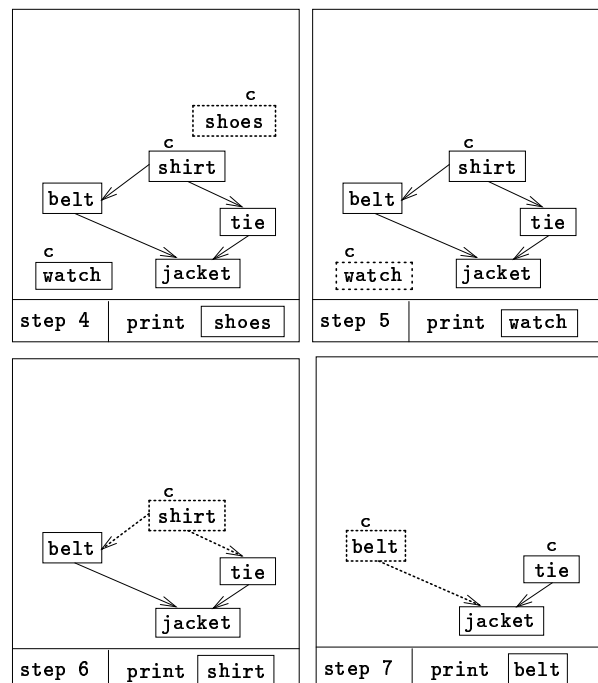
Topological Sorting...



- Candidates are marked with a c .

Slide 11-6

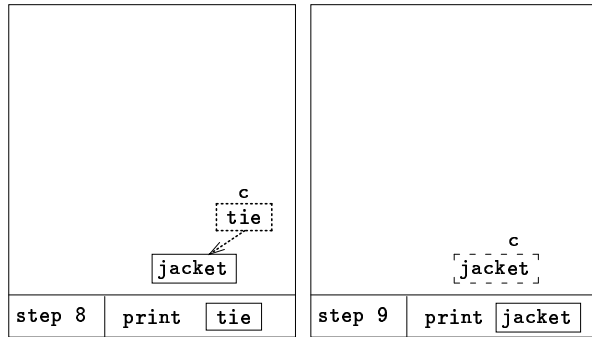
Topological Sorting...



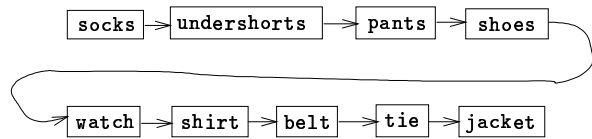
- When there is more than one candidate to choose from, we pick one at random.

Slide 11-7

Topological Sorting...

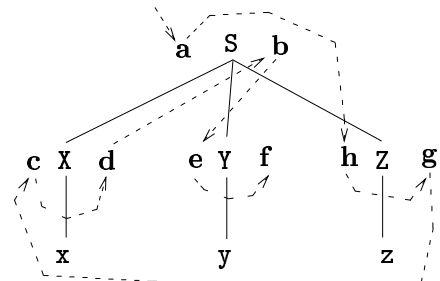


Complete Topological Order

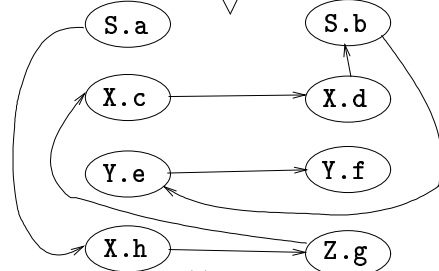


- There are often many possible topological orders to choose from. For example, since the watch node has no dependencies at all, I can put it on at any time.

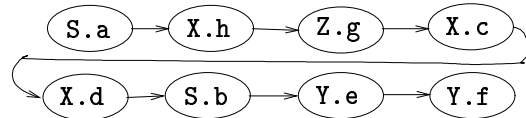
Slide 11-8



Construct
Dependency Graph

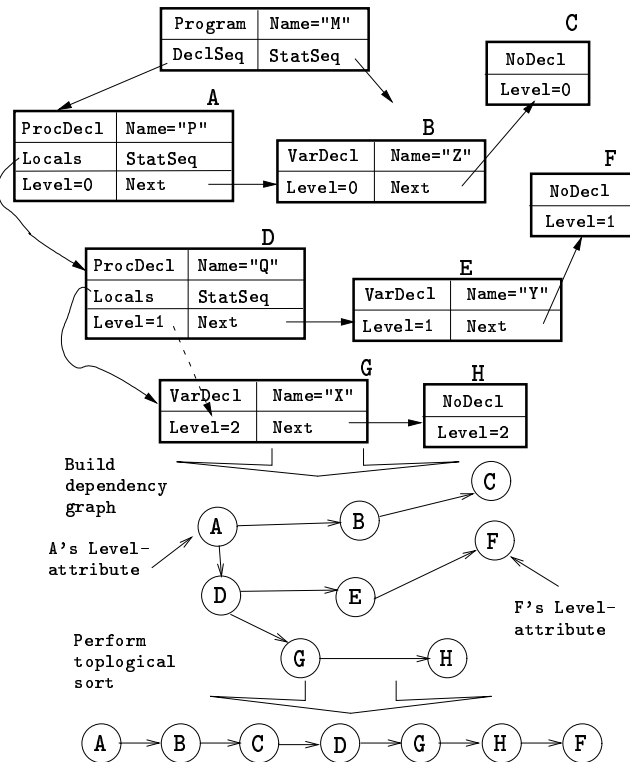


Topological
Sort



Slide 11-9

Dynamic Tree-Walkers...



Slide 11-10

Readings and References

- Read Louden: 270-277, 285-286.

Slide 11-11

Summary

- In programming languages that allow forward references (the use of an identifier before it is declared) we need to process the tree twice.
- Sometimes we may perform multiple traversals even for languages that are definition-before-use. Each traversal will compute a different subset of the attributes. Even if this is less efficient than performing a single traversal, it may lead to an evaluator that's easier to read and modify.
- The kinds of evaluators we have been building are called **static evaluators**, because the order in which the attributes are evaluated is determined at compiler construction time.

Slide 11–12

Summary...

- In a **dynamic evaluator**, the attribute evaluation order is determined at **compile time**. The idea is to build an **attribute dependency graph** from the AST (this graph encodes how one attribute may depend on [use the value of] another attribute), and using topological sorting to compute a valid evaluation order.
- It is not necessary to always store every attribute explicitly in the tree. Instead, we can pass them as arguments to the evaluator procedures. Inherited attributes will be passed by value, synthesized attributes by reference (since they return data back to the calling routine).

Slide 11–13

Summary...

- Some attributes (such as **types** of expressions and **sizes** of variables) will be needed after semantic analysis by the code generator. These attributes are called **output** attributes and must be stored explicitly in the tree.
- Some languages allow **anonymous types**, types for which the programmer need not give an explicit name. The compiler has to invent its own names for such types. Example: `TYPE T=RECORD A:POINTER TO CHAR; END;`. The compiler may give the name `T$1` (a name that no user-defined type can have) to `POINTER TO CHAR`.

Slide 11–14