CSc 453 — Compilers and Systems Software

12 : Semantic Analysis IV

Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

September 27, 2009

1

Optimizing Tree-Walk Evaluators

2 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.

3

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

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

PROCEDURE Assign(n:  Node; Env:EnvT);
   Des(⇓Env, ⇑DesType);
   Expr⇓(Env, ⇑ExprType);
   IF DesType  ≠  ExprType THEN ···
```

4

Dynamic Tree-Walk Evaluators

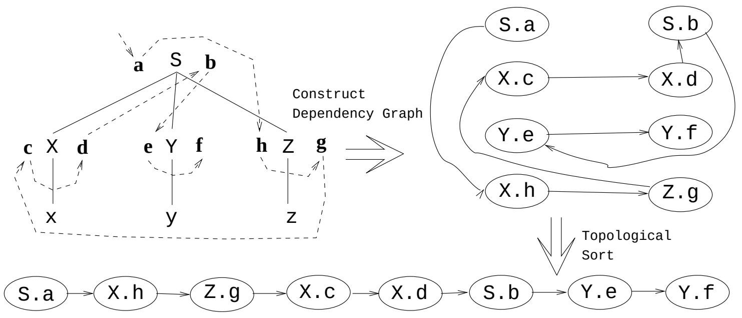
5 Dynamic Tree-Walkers

- 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).

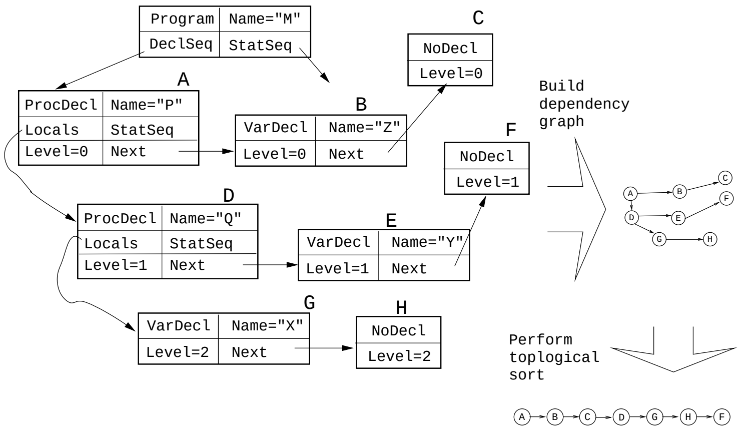
6 Dynamic Tree-Walkers...

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.

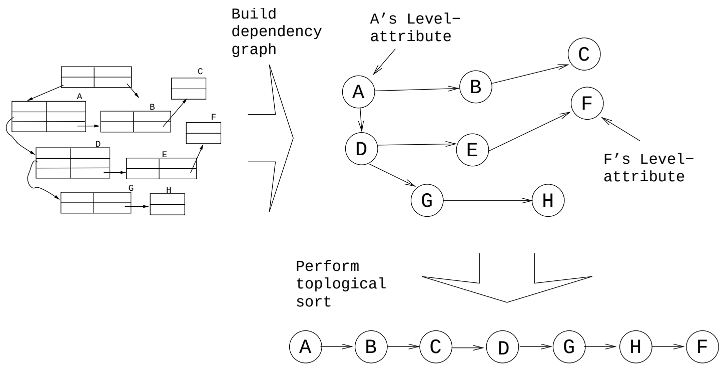
7 Dynamic Tree-Walkers...



8



9



10

Forward References

11 Forward References

- Some languages (Ada, Modula-3) allow declarations to come in an arbitrary order.
- Modula-2 allows forward references for procedures and variables, but types and constants must be declared before use.

```
PROGRAM M;
   PROCEDURE P ();
      VAR X : INTEGER;
      BEGIN
         X := Y + 1;  ⇐ Forward ref to Y!
         Q();  ⇐ Forward ref to Q!
      END P;
   VAR Y : INTEGER;
   PROCEDURE Q (); BEGIN END Q;
BEGIN END M;
```

12 Forward References...

- We have to process declarations two times. The first time we build (partial) symbol tables, the second time we build environments and process statements.
- $\Updownarrow$`Ids1`:**SyTabT** is built during pass 1, $\Updownarrow$`Ids2`:**SyTabT** during pass 2.
- $\Updownarrow$`Ids1`:**SyTabT** will basically only store the names and kinds of identifiers, to be used during name lookup.
- $\Updownarrow$`Ids2`:**SyTabT** will store complete symbols.
- Statements are processed in a third pass.

```
PROCEDURE Program (n:  Node);
   Program_Pass1();
   Program_Pass2();
   Program_Pass3();
END;
```

13 Forward References — First pass

```
PROCEDURE Program_Pass1 (n:  Node);
   StdEnv := {INT,REAL,CHAR,TRUNC,FLOAT};
   n.DeclSeq.IdsIn1:= {};
   Decl_Pass1(n.DeclSeq);
END;

PROCEDURE Decl_Pass1 (n:  Node);
   IF n.Kind=ProcDecl THEN
      ProcDecl_Pass1(n);
   ELSIF n.Kind=VarDecl THEN
      VarDecl_Pass1(n);
   ELSIF ··· ENDIF
END;
```

14

```
PROCEDURE VarDecl_Pass1 (n:  Node);
-- Check for multiple declaration of the variable.
  Sy := (Name=n.Id,Kind=VAR); ⇐No Type!
  n.Next.IdsIn1 := n.IdsIn1 ∪ {Sy};
  Decl_Pass1(n.Next);
  n.IdsOut1:=n.Next.IdsOut1;
END;

PROCEDURE ProcDecl_Pass1 (n:  Node);
  n.Formals.IdsIn1 := {};
  Decl_Pass1(n.Formals);
  n.Locals.IdsIn1 := n.Formals.IdsOut1;
  Decl_Pass1(n.Locals);
  Sy:=(Name=n.Id,Kind=PROC);⇐No Formals!
  n.Next.IdsIn1 := n.IdsIn1 ∪ {Sy};
  Decl_Pass1(n.Next);
  n.IdsOut1:=n.Next.IdsOut1;
END;
```

15 Forward References — Second Pass

```
PROCEDURE Program_Pass2 (n:  Node);
   StdEnv := {INT,REAL,CHAR,TRUNC,FLOAT};
   n.DeclSeq.Env := cons(n.DeclSeq.IdsOut1,StdEnv);
   n.DeclSeq.IdsIn2:= {};
   Decl_Pass2(n.DeclSeq);
END;

PROCEDURE VarDecl_Pass2 (n:  Node);
-- Check if the type is declared.
   T := Lookup(n.TypeName,n.Env);
   Sy := (Name=n.Id,Kind=VAR, Type=T); ⇐ Type!
   n.Next.IdsIn2 := n.IdsIn2 ∪ {Sy};
   Decl_Pass1(n.Next);
   n.IdsOut2:=n.Next.IdsOut2;
END;
```

16

```
PROCEDURE ProcDecl_Pass2 (n:  Node);
-- Use symbols from pass 1 as part of
-- the env for locals and formals.
   n.Locals.Env := n.Formals.Env :=
      cons(n.Locals.IdsOut1, n.Env);

-- Build new sy tab from locals & formals.
   n.Formals.IdsIn2:={};
   Decl_Pass2(n.Formals);
   n.Locals.IdsIn2 := n.Formals.IdsOut2;
   Decl_Pass2(n.Locals);

-- Build new sytab entry for the procedure.  Include formals.
   Sy := (Name=n.Id,Kind=PROC, Formals=n.Formals.IdsOut2);
   n.Next.IdsIn2 := n.IdsIn2 ∪ {Sy};
   Decl_Pass2(n.Next);
   n.IdsOut2:=n.Next.IdsOut2;
```

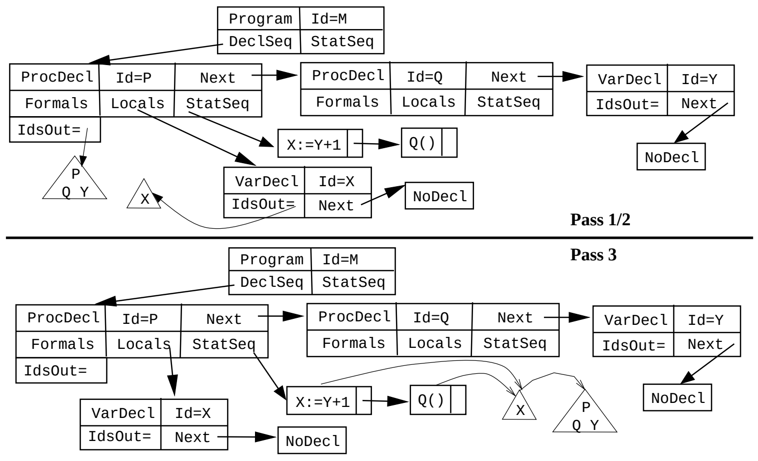
17 Forward References — Third pass

```
PROCEDURE Program_Pass3 (n:  Node);
   n.DeclSeq.Env := n.StatSeq.Env :=
      cons(n.DeclSeq.IdsOut2, StdEnv);
   Decl_Pass3(n.DeclSeq);
   Stat(n.StatSeq);
END;
PROCEDURE Decl_Pass3 (n:  Node);
   IF n.Kind=ProcDecl THEN
      ProcDecl_Pass3(n);
   ENDIF
END;
```

18

```
PROCEDURE ProcDecl_Pass3 (n:  Node);
   n.Locals.Env := n.StatSeq.Env :=
      cons(n.Locals.IdsOut2, n.Env);
   Decl_Pass3(n.Locals);
   Stat(n.StatSeq);
   n.Next.Env:=n.Env; Decl_Pass3(n.Next);
END;
```

19 Forward References — Example 1



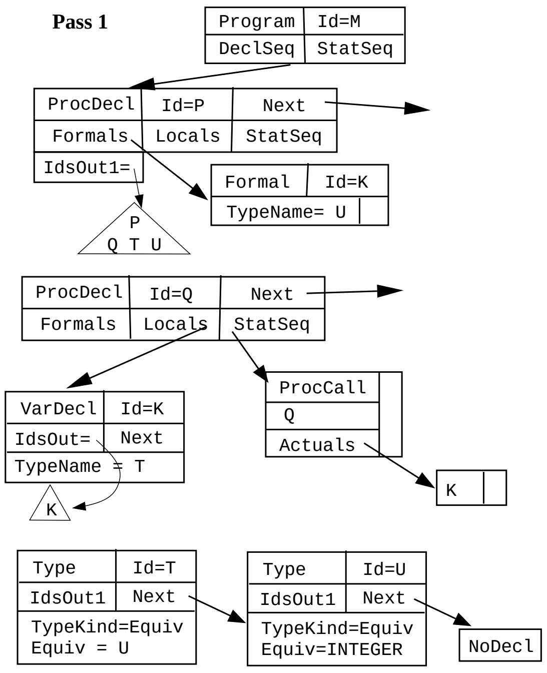
20 Forward References — Example 2

- Why can't we process the statements during the first pass? Types complicate things. Before we can process the call to `P` at $\boxed{2}$, we need to have processed `P`'s formals at $\boxed{1}$. We can't process `P's` formals until we've seen `U` at $\boxed{4}$.
- We might be able to do the first pass **as the AST is being built**, or do pass 2 as we return from the pass 1 recursion.

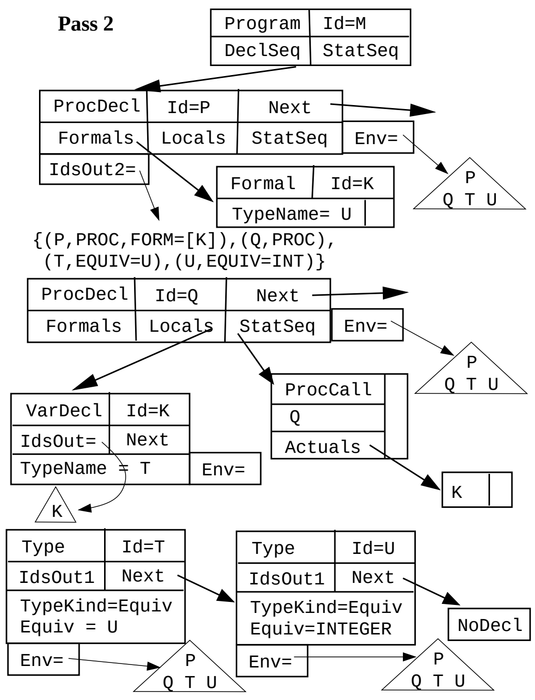
```
PROCEDURE P (S : U); ⇐ 1
   BEGIN ··· END P;

PROCEDURE Q ();
   VAR K : T;
   BEGIN P(K); ⇐ 2  END Q;

TYPE T = U; ⇐ 3
TYPE U = INTEGER; ⇐ 4
```

21

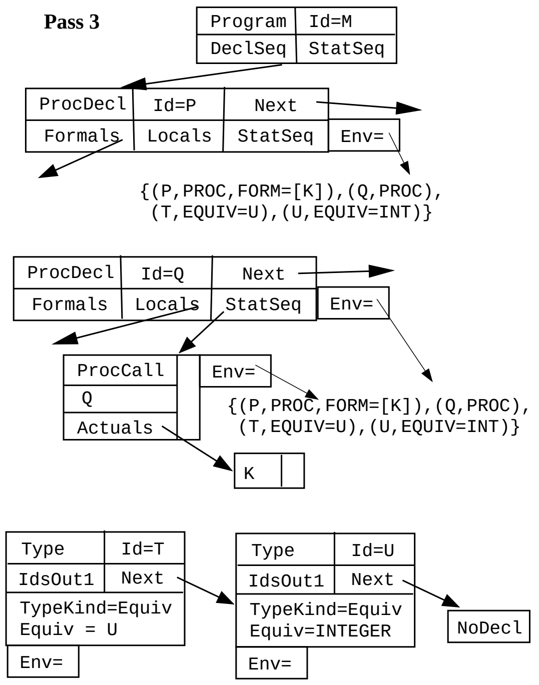
Pass 1



22



23



24

Summary

25 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.

26 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).

27 Summary...

- Some attributes (such as **type**s of expressions and **size**s 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 it's 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`.

28

Homework

29 Homework...

- Build an AST for the program below.
- Show – in detail – how the symbol tables and environments are built and how the statements are checked for type correctness. Assume that the language allows arbitrary declaration order.

```
PROGRAM M;

   PROCEDURE P (S : T);
      BEGIN S := 5; END P;

   PROCEDURE Q ();
      VAR K : T;
      BEGIN P(K); END Q;

   TYPE T = INTEGER;

BEGIN Q(); END.
```

30 Homework

- Build an AST for the program below. Show – in detail – how the assignment statements are checked for type correctness.

```
PROGRAM M;
   TYPE A = RECORD X : ARRAY [1..10] OF INTEGER; END;
   TYPE B = POINTER TO A;
   TYPE C = ARRAY [1..2] OF B;
   VAR V : C;
BEGIN
   V[1]^.X[4] := "C";
   V[2].X[4] := 5;
END.
```