Typechecking Designators

- Designators reference memory locations.
  1. Simplest case: \( X \).
  2. Structured types complicate things: \( X \^[5].V[7].p \).
- Designators are typechecked using the symbol table type graph.
- An attribute \( \$Type\$.Type\_In \) stores the type of partially processed designator.
- A synthesized attribute \( Type\_Out \) returns the type of the complete designator.

Structured Types

- Declarations of structured types (arrays, records, pointers) become a type graph of type dependencies in the symbol table.

**TYPE**

\[
S = \text{ARRAY }[1..10]\ \text{OF} \ \text{CHAR}; \\
U = \text{POINTER TO} \ S; \\
T = \text{RECORD} \ A: \text{INTEGER}; B: U;
\]

**PROCEDURE**

\[
P (V AR \ X : T); \ldots \\
\text{VAR} \ X : T; C : \text{CHAR}; \\
\text{BEGIN} \\
P(X.B^[5]); \quad \text{(* L-Value *)} \\
X.B^[5] := "x"; \quad \text{(* L-Value *)} \\
C := X.B^[5]; \quad \text{(* R-Value *)} \\
\text{END}
\]
ELSIF n.Kind = PointerRef THEN
  IF TypeKind(n.TypeIn) ≠ Pointer THEN
    PRINT "Pointer Type Expected"
  ENDIF;
  n.Next.TypeIn := PtrType(n.TypeIn);
  Des(n.Next);
  n.TypeOut := n.Next.TypeOut;
ELSIF n.Kind = NoDes THEN
  n.TypeOut := n.TypeIn;
END;

PROCEDURE Des (n : Node);
  IF n.Kind = VarRef THEN
    Symbol := Lookup(n.Id,n.Env);
    n.Next.TypeIn := GetType(Symbol);
    Des(n.Next);
    n.Type := n.Next.TypeOut;
  ELSIF n.Kind = FieldRef THEN
    IF TypeKind(n.TypeIn) ≠ Record THEN
      PRINT "Record Type Expected" ENDIF;
    Symbol := FindField(n.Id,n.TypeIn);
    n.Next.TypeOut := FieldType(Symbol);
    Des(n.Next);
    n.TypeOut := n.Next.TypeOut;
  ELSIF n.Kind = ArrayRef THEN
    IF TypeKind(n.TypeIn) ≠ Array THEN
      PRINT "Array Type Expected" ENDIF;
    Expr(n.Expr);
    IdxType := ArrayIndexType(n.TypeIn);
    IF n.Expr.Type ≠ IdxType THEN
      PRINT "Wrong Index Type" ENDIF;
    n.Next.TypeIn := ArrayType(n.TypeIn);
    Des(n.Next);
    n.TypeOut := n.Next.TypeOut;
  ELSIF n.Kind = PointerRef THEN
    IF TypeKind(n.TypeIn) ≠ Pointer THEN
      PRINT "Pointer Type Expected" ENDIF;
    Symbol := Lookup(n.Id,n.Env);
    n.Next.TypeIn := GetType(Symbol);
    Des(n.Next);
    n.Type := n.Next.TypeOut;
  ELSEIF n.Kind = NoDes THEN
    n.TypeOut := n.TypeIn;
  END;
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).

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.

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.

```plaintext
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 ...
```
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.
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).

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
END.

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 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.