

Overview...

Name Analysis

- Match up each use of an identifier with the correct declaration. Report any undeclared identifiers.

Expression Analysis

- Assign types to every sub-expression.
- Report type mismatches: `X:="Hi" * "there"`.

Prepare for Code Generation

- Insert explicit type conversions: `X:=5+6.7` \Rightarrow `X:=float(5)+6.7`.
- Compute labels for conditional- and loop-statements.

Name Analysis

- Algol-like languages allow the same name to be declared in different scopes. During name analysis the use of a name is matched up with the corresponding declaration.

```
VAR I, K : INTEGER;
PROCEDURE P (K : INTEGER);
  PROCEDURE Q (L : INTEGER);
    BEGIN I := L + K; (* Iglobal := LQ + KP *) END Q;
  VAR J : INTEGER;
  BEGIN I := J + K; (* Iglobal := JP + KP *) END P;
```

Overview

- Several independent tasks have to be performed during semantic analysis:

Declaration Analysis

- Go through and check the legality of the declarations (types, variables, procedures, etc) in the program. Check for:
 1. multiple declarations: `VAR x,y,x : int`.
 2. undeclared types : `VAR x : intger`.
 3. illegal symbol kinds: `VAR X : integer; VAR Y : X`.
- Construct **symbol tables** and **environments** to be used during name and expression analysis.



University of
Arizona

CSC 453

Compilers and Systems Software

Christian Collberg
September 29, 2002

Semantic Analysis II

Copyright © 2002 C. Collberg

Building the Symbol Table

- This time we want to build a symbol table from a sequence of declarations. At the same time we want to check for multiply declared identifiers.
- In this example, the symbol table is simply a set of identifiers. Normally, we'd also want the symbol table to include other kinds of information: type, size, and offset for variables, formal parameters for procedures, etc.
- We use a threaded attribute `Ids` of type `SyTab={String}` (set of strings). `IdsIn` is inherited and holds the current set of identifiers. `IdsOut` is synthesized and returns the complete symbol table.
- Operations on `SyTabs`: Union (\cup) and member (\in).

Symbol Tables

- All compilers use a **symbol table**. In it we record all information regarding every declared item (variable, procedure, constant, etc).
- The symbol table is built during declaration analysis.
- The symbol table is used during name analysis and type checking to look up identifiers.
- Some compilers build one (huge) symbol table for the entire input program. Others build one separate symbol table for each **scope**.
- The type of information which is stored in a symbol table node depends on the declaration:

Building the Symbol Table...

- At any node `n`, `n.IdsOut-n.IdsIn` is the set of variables declared in the subtree rooted at `n`.

_____ Abstract Syntax: _____

`Program ::= \leftarrow Id:String DeclSeq:Decl StatSeq:Stat`

`Decl ::= VarDecl | ProcDecl | NoDecl`

`VarDecl ::= \leftarrow Id:String \leftarrow TypeName:String Next:Decl
 \Downarrow Ids:SyTab`

`ProcDecl ::= \leftarrow Id:String Locals:Decl StatSeq:Stat Next:Decl
 \Downarrow Ids:SyTab`

`NoDecl ::= \Downarrow Ids:SyTab`

Symbol Tables...

All Symbols Name, Position, Level, Enclosing Block, ...

Variables Type, Size, ...

Constants Type, Size, Value, ...

Types Size, ...

Records Fields

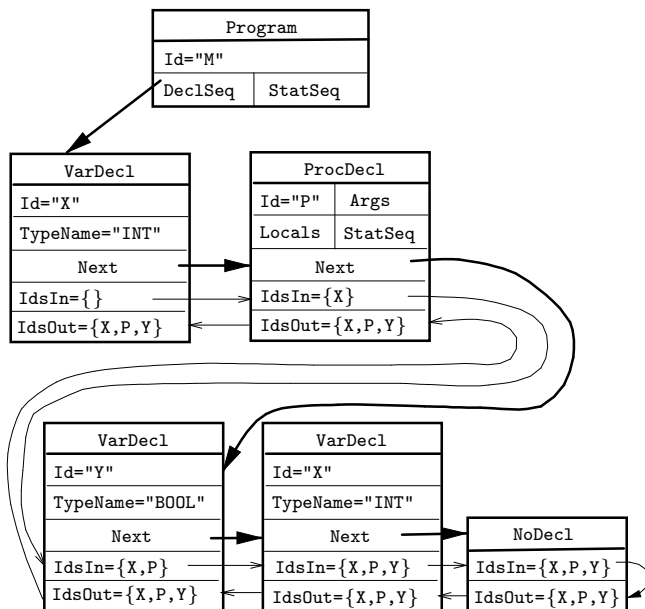
Arrays Index-Type, Index-Range, Element-Type

Procedures Formal Parameters, Size of Local Data, ...

```
TYPE T = RECORD a, b : CHAR END;
```

```
VAR X : T;
```

```
PROCEDURE Q (a: T); BEGIN ... END Q;
```



```
Program ::=  $\leftarrow$ Id:String DeclSeq:Decl StatSeq:Stat
{ DeclSeq.IdsIn:={ } }
```

```
VarDecl ::=  $\leftarrow$ Id:String  $\leftarrow$ TypeName:String
           Next:Decl  $\Downarrow$ Ids:SyTab
{
  CHECK Id  $\in$  IdsIn  $\Rightarrow$  ERROR("Multiple Declaration")
  Next.IdsIn := IdsIn  $\cup$  Id
  IdsOut := Next.IdsOut
}
```

```
Decl ::=  $\Downarrow$ Ids:SyTab
{ IdsOut := IdsIn }
```

Putting it all together...

- Obviously, We don't write one tree-walk evaluator for each attribute. Rather, we walk over the tree once (or maybe twice or three times, depending on the language) and evaluate as many attributes as possible.
- In this example we'll just use several attribute evaluation rules in order to
 1. Check for multiple declarations and build a symbol table containing all the identifiers.
 2. Assign an offset to each variable and compute the total size of the variables declared.
 3. Assign a unique number to each declared identifier and count the number of identifiers.

```
PROCEDURE Program (n: Node);
  n.DeclSeq.IdsIn:={ }; Decl(n.DeclSeq);

PROCEDURE Decl (n: Node);
  IF n.Kind  $\in$  {VarDecl,ProcDecl} THEN
    IF n.Id  $\in$  n.IdsIn THEN
      PRINT n.Pos ":Multiple "declaration: " n.Id;
    ENDIF;
    n.Next.IdsIn := n.IdsIn  $\cup$  {n.Id};
    Decl(n.Next);
    n.IdsOut := n.Next.IdsOut
  ELSIF n.Kind = NoDecl THEN
    n.IdsOut := n.IdsIn
  ENDIF
```

```

PROCEDURE VarDecl (n: Node);
  n.Next.SizeIn := n.SizeIn + size(n.TypeName);
  n.Next.CountIn := n.CountIn + 1;
  IF n.Id ∈ n.IdsIn THEN
    PRINT n.Pos ":Multiple declaration: " n.Id;
  ENDIF;
  n.Next.IdsIn := n.IdsIn ∪ {n.Id};
  Decl(n.Next);
  n.CountOut := n.Next.CountOut;
  n.SizeOut := n.Next.SizeOut;
  n.IdsOut := n.Next.IdsOut
PROCEDURE ProcDecl (n: Node);
  n.Next.SizeIn := n.SizeIn;
  (* Rest is same as for VarDecl *)

```

Putting it all together...

Abstract Syntax:

Program ::= \leftarrow Id:String DeclSeq:Decl StatSeq:Stat

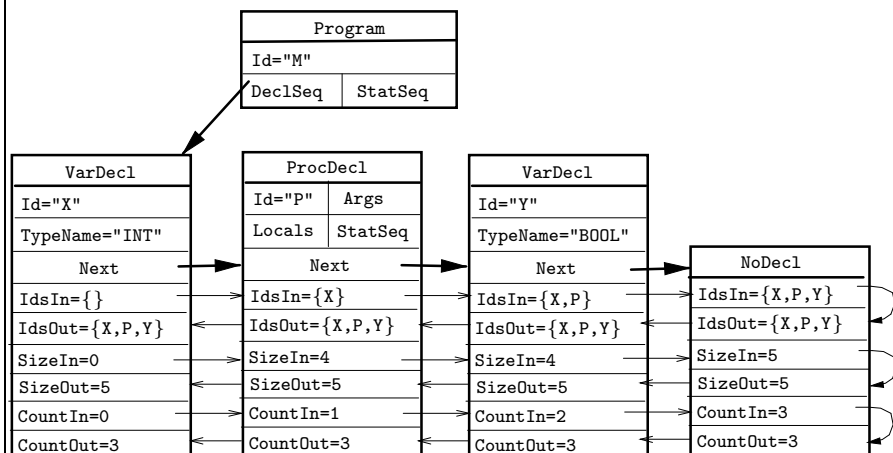
Decl ::= VarDecl | ProcDecl | NoDecl

VarDecl ::= \leftarrow Id:String \leftarrow TypeName:String Next:Decl
 \Updownarrow Ids:SyTab \Updownarrow Count:INTEGER \Updownarrow Size:INTEGER

ProcDecl ::= \leftarrow Id:String Args:Decl Locals:Decl StatSeq:Stat
 Next:Decl \Updownarrow Ids:SyTab \Updownarrow Count:INTEGER
 \Updownarrow Size:INTEGER

NoDecl ::= \Updownarrow Ids:SyTab \Updownarrow Count:INTEGER
 \Updownarrow Size:INTEGER

Putting it all together...



```

PROCEDURE Program (n: Node);
  n.DeclSeq.IdsIn := {};
  n.DeclSeq.CountIn:=0;
  n.DeclSeq.SizeIn:=0;
  Decl(n.DeclSeq);

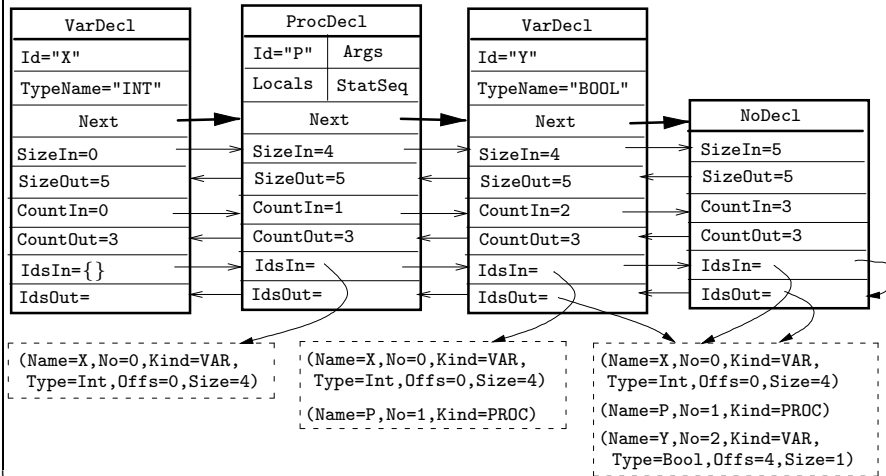
```

```

PROCEDURE Decl (n: Node);
  IF n.Kind = VarDecl THEN VarDecl(n);
  ELSIF n.Kind = ProcDecl THEN ProcDecl(n);
  ELSIF n.Kind = NoDecl THEN
    n.CountOut := n.CountIn;
    n.SizeOut := n.SizeIn;
    n.IdsOut := n.IdsIn
  ENDIF

```

Symbol Tables...



Slide 10-18

Symbol Tables

- For all symbols we'll store **Kind** (VAR or PROC), **Name**, and **Number** (every identifier has a unique number). For variables we'll also store **Size**, **Type**, and **Offset** (address).
- We'll assume that \Downarrow Count:INTEGER and \Downarrow Size:INTEGER are available.

```
VAR X : INTEGER;
PROCEDURE P (); BEGIN ...END P;
VAR Y : BOOLEAN;
```

\Downarrow Build Symbol Table \Downarrow

```
{ (Name="X", No=0, Kind=VAR, Size=4, Type=Int, Offset=0),
  (Name="P", No=1, Kind=PROC),
  (Name="Y", No=2, Kind=VAR, Size=1, Type=Bool, Offset=4)}
```

Slide 10-16

Implementing Symbol Tables

Symbol tables are sets of tuples. Any set data structure will do fine. Hash tables, binary search trees, or linked lists will be OK, depending on the size of the table.

```
TYPE KindT = (Var, Proc, Type, Const);
DataT = RECORD
    Name: String; Number: INTEGER; Pos: Position;
CASE Kind : KindT OF
    Var : Type:String; ... |
    Const : Value:INTEGER;...|
END;
Next : SyTab;
END;
SyTab = POINTER TO DataT;
```

Slide 10-19

```
PROCEDURE Decl (n: Node);
```

```
IF n.Kind = VarDecl THEN
```

```
    Sy := (Name=n.Id, No=n.CountIn, Offset=n.SizeIn,
           Kind=VAR, Size=size(n.TypeName), Type=n.TypeName);
```

```
    n.Next.IdsIn := n.IdsIn  $\cup$  {Sy};
```

```
    Decl(n.Next);
```

```
    n.IdsOut:=n.Next.IdsOut
```

```
ELSIF n.Kind = ProcDecl THEN
```

```
    Sy := (Name=n.Id, Kind=PROC, No=n.CountIn);
```

```
    n.Next.IdsIn := n.IdsIn  $\cup$  {Sy};
```

```
    Decl(n.Next); n.IdsOut:=...
```

```
ELSIF n.Kind = NoDecl THEN
```

```
    n.IdsOut := n.IdsIn
```

```
ENDIF
```

Slide 10-17

SyTabs for Nested Scope...

```
PROGRAM M;  
  PROCEDURE P ();  
    VAR X : INTEGER;  
    PROCEDURE Q (  
      X : CHAR;  
      Z : INTEGER);  
      VAR Y : INTEGER;  
      VAR Z : CHAR;  
    BEGIN END Q;  
  BEGIN END P;  
BEGIN END M;
```

Slide 10-22

SyTabs for Nested Scope

- This time we are going to build one symbol table for each nested scope.
- Note that the formal parameters and local variables of a procedure belong to the same scope.
- The next slide shows the abstract syntax for a language with variable and procedure declarations. The following slides show an example program, the attribute grammar, and the tree-walker.

Slide 10-20

```
Formal, VarDecl ::=  $\leftarrow$ Id:String  $\leftarrow$ TypeName:String  
                Next:Decl  $\Downarrow$ Ids:SyTab  
{ Next.IdsIn := IdsIn  $\cup$  {(Name=Id,Kind=VAR,...)};  
  IdsOut:=Next.IdsOut }
```

```
ProcDecl ::=  $\leftarrow$ Id:String Locals:Decl  
           Formals:Decl StatSeq:Stat Next:Decl  $\Downarrow$ Ids:SyTab  
{ Formals.IdsIn := {};  
  Locals.IdsIn := Formals.IdsOut;  
  Next.IdsIn := IdsIn  $\cup$   
    {(Name=Id,Kind=PROC,Vars=Locals.IdsOut,...)};  
  IdsOut:=Next.IdsOut }
```

```
NoDecl ::=  $\Downarrow$ Ids:SyTab { IdsOut := IdsIn }
```

Slide 10-23

SyTabs for Nested Scope...

```
Program ::=  $\leftarrow$ Id:String DeclSeq:Decl StatSeq:Stat
```

```
Decl ::= VarDecl | ProcDecl | NoDecl
```

```
VarDecl ::=  $\leftarrow$ Id:String  $\leftarrow$ TypeName:String Next:Decl  
            $\Downarrow$ Ids:SyTab
```

```
Formal ::=  $\leftarrow$ Id:String  $\leftarrow$ TypeName:String Next:Decl  
           $\Downarrow$ Ids:SyTab
```

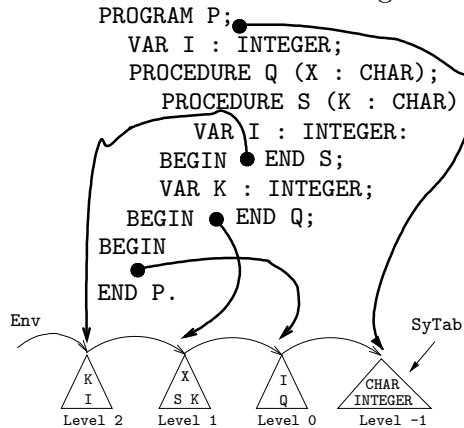
```
ProcDecl ::=  $\leftarrow$ Id:String Args:Decl Locals:Decl Formals:Decl  
            StatSeq:Stat Next:Decl  $\Downarrow$ Ids:SyTab
```

```
NoDecl ::=  $\Downarrow$ Ids:SyTab
```

Slide 10-21

Building Environments

- Environments are used to represent scope information. They are linked lists of symbol tables where each symbol table represents the identifiers declared in a given scope.



Slide 10-26

```
PROCEDURE Decl (n: Node);
```

```
  IF n.Kind = VarDecl THEN
```

```
    Sy := (Name=n.Id,Kind=VAR,...);
```

```
    n.Next.IdsIn := n.IdsIn ∪ {Sy};
```

```
    Decl(n.Next); n.IdsOut:=n.Next.IdsOut
```

```
  ELSIF n.Kind = ProcDecl THEN
```

```
    n.Formals.IdsIn := {}; ← NEW!
```

```
    Decl(n.Formals); ← NEW!
```

```
    n.Locals.IdsIn := n.Formals.IdsOut; ← NEW!
```

```
    Decl(n.Locals); ← NEW!
```

```
    Sy := (Name=n.Id,Kind=PROC,...);
```

```
    n.Next.IdsIn := n.IdsIn ∪ {Sy};
```

```
    Decl(n.Next); n.IdsOut:=n.Next.IdsOut
```

```
  ELSIF n.Kind = NoDecl THEN n.IdsOut := n.IdsIn ENDIF
```

Slide 10-24

Building Environments...

- First a symbol table is built for each scope. Then environments are constructed and sent down to statements to be used during name analysis.
- EnvT = LIST OF SyTab = LIST OF SET OF Symbol.**

Abstract Syntax:

VarDecl ::= ←Id:String ←TypeName:String Next:Decl

⇕Ids:SyTab ⇓Env:EnvT

Formal ::= ←Id:String ←TypeName:String Next:Decl

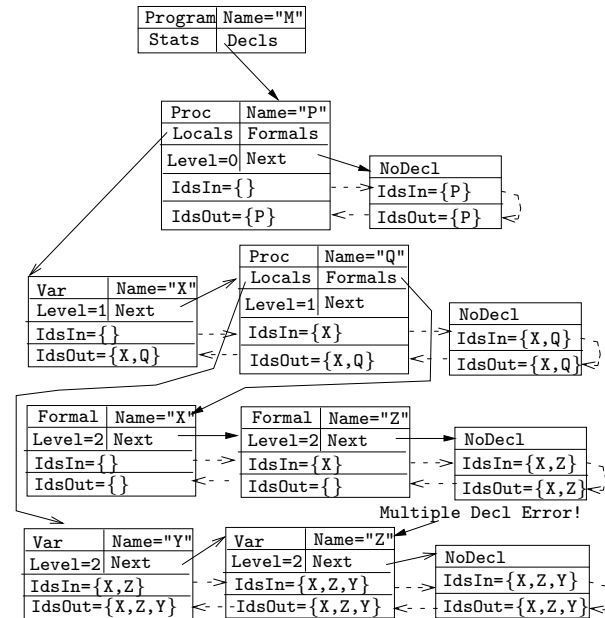
⇕Ids:SyTab ⇓Env:EnvT

ProcDecl ::= ←Id:String Locals:Decl Formals:Decl StatSeq:Stat

Next:Decl ⇓Env:EnvT ⇕Ids:SyTab

Assign ::= Des:Des Expr:Expr ⇓Env:EnvT

Slide 10-27



Slide 10-25

Building Environments...

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

```
PROCEDURE Decl (n: Node);
  IF n.Kind=VarDecl THEN VarDecl(n);
  ELSIF n.Kind=ProcDecl THEN ProcDecl(n);
  ELSIF n.Kind=NoDecl THEN (* Same *)
  ENDIF
```

Building Environments...

```
PROCEDURE Assign (n: Node);
  Des.Env := Expr.Env := n.Env; ...
```

```
PROCEDURE VarDecl (n: Node);
  IF NOT member(n.Env,n.TypeName) THEN
    PRINT n.Pos ":Identifier not declared " n.TypeName;
  ENDIF;
  n.Next.Env := Env;
  (* More here...*)
```

```
Program ::=  $\leftarrow$ Id:String DeclSeq:Decl StatSeq:Stat
{
  DeclSeq.Env := {INT,REAL,CHAR,TRUNC,FLOAT};
  DeclSeq.IdsIn:={};
  StatSeq.Env := cons(DeclSeq.IdsOut,DeclSeq.Env);
}
```

```
Formal, VarDecl ::=  $\leftarrow$ Id:String  $\leftarrow$ TypeName:String
Next:Decl  $\Downarrow$ Ids:SyTab  $\Downarrow$ Env:EnvT
{
  CHECK NOT member(Env,TypeName)
     $\Rightarrow$  ERROR("Ident not declared")
  Next.Env := Env;
}
```

```
ProcDecl ::=  $\leftarrow$ Id:String Locals:Decl Formals:Decl
StatSeq:Stat Next:Decl  $\Downarrow$ Env:EnvT
 $\Downarrow$ Ids:SyTab
{
  Formals.IdsIn := {};
  Formals.Env := Locals.Env:= Env;
  Locals.IdsIn := Formals.IdsOut;
  StatSeq.Env := cons(Locals.IdsOut, Env);
  Next.Env := Env;
}
```

```
Assign ::=  $\Downarrow$ Env:EnvT
{ Des.Env := Expr.Env := Env; }
```

Slide 10-28

Slide 10-29

Slide 10-30

Slide 10-31

Implementing Environments

_____ Symbol Table ADT _____

```

Create() : SyTabT;
Insert(S:SyTabT; N:Name) : SymbolRef;
Lookup(S:SyTabT; N:Name) : SymbolRef;
    
```

- Each symbol R has a set of attributes (Type, Size,...) that can be set/retrieved using operations Set<Attr>/Get<Attr>(S,R,A).

Implementing Environments...

_____ Environment ADT _____

```

Create() : EnvT;
Cons(S:SyTabT; E:EnvT) : Env;
Identify(E:EnvT; N:Name) : SymbolRef;
Member(E:EnvT; N:Name) : BOOLEAN;
    
```

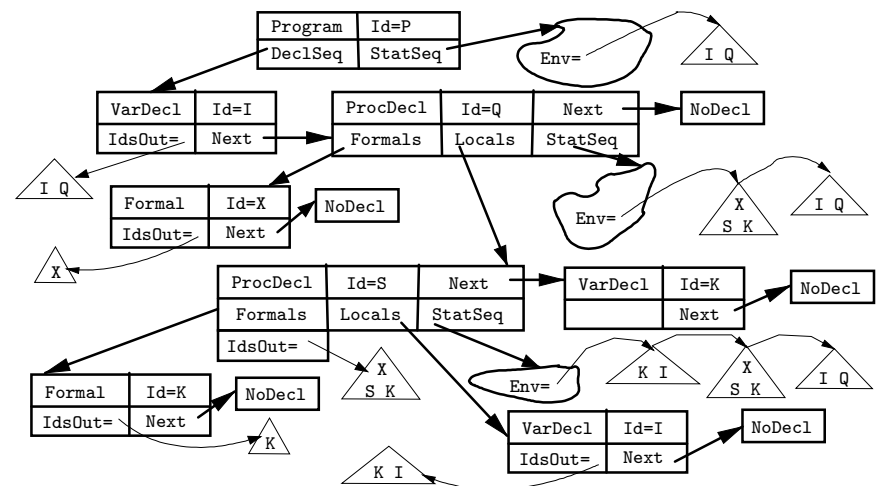
- Cons(S,E) creates a new environment consisting of the symbol table S followed by the symbol tables of E.
- Identify(E, N) searches the symbol tables of E sequentially until a definition of the name N is found.

Building Environments...

```

PROCEDURE ProcDecl (n: Node);
  n.Formals.IdsIn := {};
  n.Formals.Env := n.Locals.Env:=n.Env;
  Decl(n.Formals);
  n.Locals.IdsIn := n.Formals.IdsOut;
  Decl(n.Locals);
  n.StatSeq.Env := cons(n.Locals.IdsOut, n.Env);
  Stat(n.StatSeq);
  n.Next.Env := Env;
  (* More here...*)
    
```

Building Environments...



Readings and References

- Read Louden:

Symbol Tables 295–313

Note that Louden uses different algorithms for symbol-tables and environments than I do in this lecture.

- or read the Dragon book:

Symbol Tables 429–438

Nested Procedures 415–416

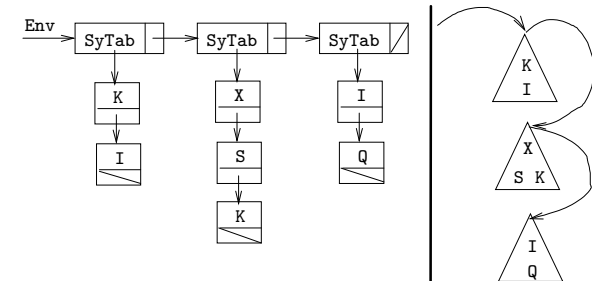
Environments 438–440

Summary

- During declaration analysis we build symbol tables that will be used during name analysis.
- A symbol table is a collection of information about the identifiers declared in a program. The kind of information that is stored for a particular identifier depends on its **kind** (variable, procedure, etc).
- For every identifier we store its name, kind, and position (line and column number in the source code where the identifier is declared).

Implementing Environments...

- Symbol tables are sets of tuples (collections of data), environments are lists of symbol tables.



Implementing Environments...

```
TYPE KindT = (Var,Proc,Type,Const);  
SyTabT =POINTER TO RECORD  
    CASE Kind : KindT OF ... END;  
    Name : String; Next : SyTabT;  
END;  
EnvT = POINTER TO RECORD  
    SyTab : SyTabT; Next : EnvT;  
END;
```

Summary...

- Tree-walkers use *environments* to perform *name analysis*. An environment is a list of symbol tables, where each table consists of the symbols collected in a particular scope.
- Environments are organized so that if they are searched sequentially from the start, we'll always find the correct (most closely nested) identifier first.
- Environments are passed **down** the tree (using inherited attributes) in order to inform lower level nodes about the **context** in which they occur.

Summary...

- A threaded attribute $\uparrow\mathbf{A:T}$ actually consists of two attributes: an inherited attribute $\downarrow\mathbf{AIn:T}$ and a synthesized attribute $\uparrow\mathbf{AOut:T}$. As we perform an inorder traversal of a subtree, **AIn** collects information from the tree, and **AOut** brings it back up the tree.
- Threaded attributes are used to gather information from a subtree. Since gathering information is exactly what we do when we build a symbol table, we use a threaded attribute $\uparrow\mathbf{Ids:SyTabT}$ to construct the symbol table.
- This symbol table can then be passed down the tree (using an inherited attribute $\downarrow\mathbf{Env:SyTabT}$) during name analysis.

Confused Student Email

Why do we have to store all information in the AST? Why can't we just use one global symbol-table to keep all data about all symbols? Passing these environments around seems really inefficient and confusing.

It is true that some compilers build one huge symbol table for the entire program and keep that outside the tree. This method works well for simple languages like C, which does not support nested procedures, classes, etc.

For other languages, it's better to build one symbol table for each scope, and pass them around the tree using attributes. Then we'll have complete control of the information that is available at each point in the program; we'll know exactly what information is passed into each node, and what attributes are computed at each node.

- An inherited attribute is given a value **before** a recursive call is made:
- An synthesized attribute is given a value **before** a recursive call returns:

```
PROCEDURE Expr (n: Node);  
    IF n.Kind = IntConst THEN n.Type := "INT";
```

- For a threaded attribute pair, the inherited part is given a value **before** the recursive call is made and the synthesized parts is given a value **after** the call returns:

```
n.Next.IdsIn := n.IdsIn  $\cup$  {Sy};  
Decl(n.Next);  
n.IdsOut := n.Next.IdsOut
```

Homework II

- Show the symbol tables resulting from the declarations below. Include as much information about each symbol as possible. Give each identifier a unique number (set INTEGER=1 and CHAR=2), and use these numbers to represent types.

_____ Problem (A): _____

```
PROCEDURE P (X:INTEGER; Y:CHAR);  
VAR Z:INTEGER;  
BEGIN END P;
```

Homework II...

_____ Problem (B): _____

```
TYPE T = RECORD A, B : CHAR END;  
VAR X : T;
```

_____ Problem (C): _____

```
TYPE T2 = POINTER TO CHAR;  
TYPE T2 = ARRAY 100 OF T1;  
TYPE T3 = ARRAY 20 OF T2;
```

Exam Question

Show the environment in effect at each point **i** in this program. Identifiers must be declared before use. Recursion is allowed.

```
PROGRAM M;  
2  
TYPE T = ARRAY 5 OF CHAR; VAR X : INTEGER;  
PROCEDURE P ();  
  VAR Z : T;  
  PROCEDURE Q ();  
    VAR R : CHAR; PROCEDURE Z (X:CHAR); BEGIN 3 END Z;  
    VAR Y : CHAR; PROCEDURE V (); BEGIN 4 END V;  
  BEGIN 5 END Q;  
  VAR Y : INTEGER;  
BEGIN 6 END P;  
VAR Y : INTEGER;  
BEGIN 1 END M.
```

Homework I

- Show the environment in effect at each point **i** in the program below.

```
PROGRAM M;  
VAR X : INTEGER;  
PROCEDURE P (X : CHAR);  
  VAR Z : INTEGER;  
  PROCEDURE Q (X : INTEGER);  
    VAR R : CHAR;  
    PROCEDURE Z (); BEGIN 1 END Z;  
    VAR Y : CHAR;  
  BEGIN 2 END Q;  
  VAR Y : INTEGER;  
BEGIN 3 END P;  
VAR Y : INTEGER;  
BEGIN 4 END M.
```

Homework V

- Assume a small Modula-2 like language:

_____ Concrete Syntax: _____

Block ::= BEGIN StatSeq END

AssignStat ::= ident ':=' Expr

ForStat ::= FOR ident ':=' expr TO expr [ByPart] DO StatSeq
END

ByPart ::= BY ConstExpr

Stat ::= AssignStat | IfStat | ForStat

StatSeq ::= Stat ';' StatSeq | ϵ

Expr ::= Expr + Expr | ident | IntConst

Homework V...

1. Give an abstract syntax corresponding to the concrete syntax above.
2. Write a attribute grammar/tree-walk evaluator which checks that the **ByPart**, if present, is a constant expression.

Homework III

Build an abstract syntax tree for the program below. Show — in detail — how the symbol tables and environments are built.

```
PROGRAM M;  
  VAR X : INTEGER;  
  VAR Y : INTEGER;  
  PROCEDURE P (X : CHAR);  
    VAR Z : INTEGER;  
    PROCEDURE Q (X : INTEGER);  
      VAR R : CHAR;  
      VAR V : CHAR;  
      BEGIN END Q;  
      VAR Y : INTEGER;  
      BEGIN END P;  
    BEGIN END M.
```

Homework IV

1. Build an abstract syntax tree for the program below. Show — in detail — how the statements are type checked. Which error messages should be generated?

```
PROGRAM M;  
  VAR X : INTEGER;  
  VAR Y : INTEGER;  
  PROCEDURE P (Z : INTEGER; VAR X : CHAR);  
    VAR Z : INTEGER;  
  BEGIN  
    X := "D";  
    Y := Z + X;  
  END P;  
BEGIN  
  P(X, "C");  
END M.
```

Homework VII

- Assume that enumerated types are declared in this fashion:

```
TYPE T = ENUM[Marge=1, Bart=2,  
             Maggie=5, Lisa=10];
```

I.e., unlike Pascal, we're allowed to number the identifiers however we like.

- Give a suitable abstract syntax and a tree-walk evaluator that checks that all identifiers **and** values are unique (within the declaration).

Homework VII...

- In other words, the static semantics should flag these declarations as erroneous:

```
TYPE T1 = ENUM[Ren=3, Stimp=4, Ren=2];  
"ERROR: Multiple enumeration id: Ren"
```

```
TYPE T2 = ENUM[CB=10, Linus=4, Lucy=10];  
"ERROR: Repeated enumeration value: 10"
```

Homework VI

- Assume a small Modula-2 like language like in the previous exercise, but with **IF**-statements:

_____ Concrete Syntax Extension: _____

```
IfStat ::= IF Expr THEN StatSeq ELSE StatSeq END
```

```
IfStat ::= IF Expr THEN StatSeq END
```

```
ForStat ::= FOR ident ':=' expr TO expr [ByPart] DO StatSeq  
          END
```

```
ByPart ::= BY ConstExpr
```

Homework VI...

1. Give an abstract syntax corresponding to the concrete syntax.
2. Write a attribute-grammar/tree-walk evaluator which checks that the iteration variable of a **FOR**-loop is not changed within the body of the loop. Remember that loops can be nested!

Homework IX

1. Write a concrete and an abstract grammar for Pascal-like variable declarations.
2. Write a tree-walk evaluator that checks for multiple declarations of the same identifier.

_____ Example 1 (Correct): _____

```
VAR x : CHAR; y,z,a,b : INTEGER; n,s : BOOLEAN;
```

_____ Example 2 (Wrong): _____

```
VAR x, y, z, x, a : CHAR;
```

_____ Example 3 (Wrong): _____

```
VAR x : CHAR; y,z,a,x : INTEGER; n,x : BOOLEAN;
```

Homework VIII

- Assume that enumerated types are declared in the “normal” Pascal fashion:

```
TYPE T = ENUM[Marge, Homer, Bart,  
             Maggie, Lisa];
```

- Assume furthermore that the individual identifiers are given numbers 0, 1, 2, ...
- Give a suitable abstract syntax and a tree-walk evaluator that computes the minimum number of bits required to store variables of the type.

Homework VIII...

_____ Examples: _____

```
TYPE T = ENUM[a]           ⇒ 1 bit
```

```
TYPE T = ENUM[a,b]        ⇒ 1 bit
```

```
TYPE T = ENUM[a,b,c]      ⇒ 2 bits
```

```
TYPE T = ENUM[a,b,c,d]    ⇒ 2 bits
```

```
TYPE T = ENUM[a,b,c,d,e]  ⇒ 3 bits
```

```
TYPE T = ENUM[a,b,c,d,e,f] ⇒ 3 bits
```