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:
     
     \[
     \text{VAR } x, y, x : \text{int}
     \]
  2. undeclared types:
     
     \[
     \text{VAR } x : \text{integer}
     \]
  3. illegal symbol kinds:
     
     \[
     \text{VAR } X : \text{integer}; \text{VAR } Y : X
     \]

- Construct symbol tables and environments to be used during name and expression analysis.

Name Analysis

- Most 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.

\[
\begin{align*}
\text{VAR } I, K & : \text{INTEGER}; \\
\text{PROCEDURE } P \ (K : \text{INTEGER}); \\
& \quad \text{PROCEDURE } Q \ (L : \text{INTEGER}); \\
& \quad \quad \text{BEGIN} \\
& \quad \quad \quad I := L + K; \quad (\ast \text{Iglobal} := Lq + Kp \ast) \\
& \quad \quad \quad \text{END } Q; \\
& \quad \quad \text{VAR } J : \text{INTEGER}; \\
& \quad \quad \text{BEGIN} \\
& \quad \quad \quad I := J + K; \quad (\ast \text{Iglobal} := Jp + Kp \ast) \\
& \quad \quad \quad \text{END } P; 
\end{align*}
\]

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:
  \[
  X := \text{"Hi" } \ast \text{"there"}.
  \]
- Prepare for Code Generation

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

Slide 9-4

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;

Slide 9-5

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. For example, for variables we’d like to store their type, size, and offset; for procedures their formal parameters.
- We use a threaded attribute \( \text{Ids} \) of type \( \text{SyTab} = \{ \text{String} \} \) (set of strings). \( \text{IdsIn} \) is inherited and holds the current set of identifiers. \( \text{IdsOut} \) is synthesized and returns the complete symbol table.
- There are two operations on \( \text{SyTabs} \) available: Union (\( \cup \)) and member (\( \in \)).

Slide 9-6

Building the Symbol Table...

- At any node \( n \), \( n.\text{IdsOut} - n.\text{IdsIn} \) is the set of variables declared in the subtree rooted at \( n \).
- Notice how similar this example is to the previous one.

------------ Abstract Syntax:  ------------

| Program | ::= | \(<\text{Id}:\text{String}\) \text{DeclSeq:Decl} |
| StatSeq | ::= | \text{Stat} |
| Decl | ::= | \text{VarDecl} | \text{ProcDecl} | \text{NoDecl} |
| VarDecl | ::= | \(<\text{Id}:\text{String}\) |
| ProcDecl | ::= | \(<\text{Id}:\text{String}\) \text{Args:Decl} |
| NoDecl | ::= | \#\text{Ids:SyTab} |

Slide 9-7
Building the Symbol Table...

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

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

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.
Putting it all together...

Slide 9–12

Putting it all together...

Slide 9–13

Putting it all together...

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"
      PRINT "declaration: " n.Id;
  END;
  n.Next.IdsIn := n.IdsIn ∪ {n.Id};
  Decl(n.Next);
  n.SizeOut := n.Next.SizeOut;
END;

PROCEDURE ProcDecl (n: Node);
  n.Next.SizeIn := n.SizeIn;
  (* Rest is same as for VarDecl *)
END;

Slide 9–14

Slide 9–15
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 $\#$Count:INTEGER and $\#$Size:INTEGER are available.

Symbol Table Example:

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) \}

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.

Linked List Implementation:

```
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;
```

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.

---

Abstract Syntax (cont):

```
Program ::= <Id: String> DecSeq:Decl  
           StatSeq:Stat
Dec ::= VarDecl | ProcDecl | NoDecl
VarDecl ::= <Id: String>  
           <TypeName: String> Next:Decl  
           [Ids:SyTab]
Formal ::= <Id: String>  
           <TypeName: String>  
           Next:Decl [Ids:SyTab]
ProcDecl ::= <Id: String> <ArgDecl:Dec>  
           Locals:Decl Formals:Decl StatSeq:Stat  
           Next:Decl [Ids:SyTab]
NoDecl ::= [Ids:SyTab]
```

---

Example Program:

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

---

PROCEDURE Decl (n: Node);

```
IF n.Kind = VarDecl THEN 
  Sy := (Name=n.Id, Kind=VAR, ...);
  n.Next.IdsIn := n.IdsIn U {Sy};
  Decl(n.Next);
ELSIF n.Kind = ProcDecl THEN 
  n.Formals.IdsIn := {};
  Decl(n.Formals);
  n.Locals.IdsIn := n.Formals.IdsOut;
  Decl(n.Locals);
  Sy := (Name=n.Id, Kind=PROC, ...);
  n.Next.IdsIn := n.IdsIn U {Sy};
  Decl(n.Next);
ELSIF n.Kind = NoDecl THEN
  n.IdsOut := n.IdsIn
ENDIF
END;
```
Building Environments

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

**Slide 9–24**

**SyTabs for Nested Scope...**

**Slide 9–25**

**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.

  - $\text{EnvT} = \text{LIST OF SyTab} = \text{LIST OF SET OF Symbol}$

  **Abstract Syntax:**

  $$
  \text{VarDecl} ::= \leftarrow \text{Id} : \text{String} \\
  \quad \leftarrow \text{Name} : \text{String} \quad \text{Next} : \text{Decl} \\
  \uparrow \text{Ids} : \text{SyTab} \downarrow \text{Env} : \text{EnvT} \\
  \\
  \text{Formal} ::= \leftarrow \text{Id} : \text{String} \leftarrow \text{Name} : \text{String} \\
  \quad \text{Next} : \text{Decl} \uparrow \text{Ids} : \text{SyTab} \downarrow \text{Env} : \text{EnvT} \\
  \\
  \text{ProcDecl} ::= \leftarrow \text{Id} : \text{String} \quad \text{Locals} : \text{Decl} \\
  \quad \text{Formals} : \text{Decl} \quad \text{StatSeq} \quad \text{Stat} \quad \text{Next} : \text{Decl} \\
  \downarrow \text{Env} : \text{EnvT} \uparrow \text{Ids} : \text{SyTab} \\
  \\
  \text{Assign} ::= \text{Des} : \text{Des} \quad \text{Expr} : \text{Expr} \downarrow \text{Env} : \text{EnvT}
  $$

**Slide 9–26**

**Slide 9–27**

**Program:**

- $\text{Dec1Seq} \quad \text{Env} = \{\text{INT, REAL, CHAR, TRUNC, FLOAT}\}$
- $\text{DecSeq} \quad \text{Ids} : \text{Inner} = \{}$
- $\text{StatSeq} \quad \text{Env} = \text{cons}(\text{DecSeq}, \text{Ids} : \text{Outer}, \text{DecSeq}, \text{Env})$
- $\text{Proc} \quad \text{Formal} = \text{Proc} \quad \text{Name} : \text{String}$
- $\text{Next} : \text{Decl} \quad \text{Ids} : \text{SyTab} \quad \text{Env} : \text{EnvT}$
- $\text{Assign} ::= \text{Des} : \text{Des} \quad \text{Expr} : \text{Expr} \downarrow \text{Env} : \text{EnvT}$
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);
END;

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
END;

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

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

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...*)
END;
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).

--------- 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.

Slide 9–32

Implementing Environments

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

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

Slide 9–33

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

Slide 9–34

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).

Slide 9–35
Summary...

- A threaded attribute $A:T$ actually consists of two attributes: an inherited attribute $\mathit{AIn}:T$ and a synthesized attribute $\mathit{AOut}:T$. As we perform an inorder traversal of a subtree, $\mathit{AIn}$ collects information from the tree, and $\mathit{AOut}$ 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 $\mathit{Ids}:\mathit{SyTabT}$ to construct the symbol table.

- This symbol table can then be passed down the tree (using an inherited attribute $\mathit{Env}:\mathit{SyTabT}$) during name analysis.

Slide 9–36

Summary...

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

  $$\mathit{nLOPEnv} := \mathit{nEnv};$$
  $$\mathit{Expr}(\mathit{nLOP});$$

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

  $$\text{PROCEDURE}\ \mathit{Expr}\ (\mathit{n}:\ \mathit{Node});$$
  $$\text{IF}\ \mathit{nKind} = \mathit{IntConst}\ \text{THEN}$$
  $$\mathit{nType} := \text{"INT"};$$

- 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:

  $$\mathit{nNextIdsIn} := \mathit{nIdsIn} \cup \{\mathit{Sy}\};$$
  $$\mathit{Decl(nNext)};$$
  $$\mathit{nIdsOut} := \mathit{nNextIdsOut}$$

Slide 9–37

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.

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.

Slide 9–38

Slide 9–39
Exam Question

- Show the environment in effect at each point \[ \text{1} \] in the program below. Identifiers must be declared before they are used. Procedures can be recursive.

PROGRAM M;
\[ \text{2} \]
TYPE T = ARRAY 5 OF CHAR;
VAR X : INTEGER;
PROCEDURE P ();
VAR Z : T;
PROCEDURE Q ();
VAR R : CHAR;
PROCEDURE Z (X:CHAR); BEGIN \[ \text{3} \] END Z;
VAR Y : CHAR;
PROCEDURE V (); BEGIN \[ \text{4} \] END V;
BEGIN \[ \text{1} \] END M.

Homework I

- Show the environment in effect at each point \[ \text{1} \] 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 \[ \text{1} \] END Z;
VAR Y : CHAR;
BEGIN \[ \text{2} \] END Q;
VAR Y : INTEGER;
BEGIN \[ \text{3} \] END P;
VAR Y : INTEGER;
BEGIN \[ \text{4} \] END M.

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 \text{INTEGER}=1 and \text{CHAR}=2), and use these numbers to represent types.

Problem (A):

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

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;

Homework III

1. Build an abstract syntax tree for the program below.
2. 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.

Slide 9–44

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

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

Slide 9–45

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
1. Give an abstract syntax corresponding to the concrete syntax.
2. Write a 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!

Slide 9–46

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).

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

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

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

Slide 9–47
**Homework VIII**

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

<table>
<thead>
<tr>
<th>Type Expression</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE T = ENUM[a]</td>
<td>1</td>
</tr>
<tr>
<td>TYPE T = ENUM[a,b]</td>
<td>1</td>
</tr>
<tr>
<td>TYPE T = ENUM[a,b,c]</td>
<td>2</td>
</tr>
<tr>
<td>TYPE T = ENUM[a,b,c,d]</td>
<td>2</td>
</tr>
<tr>
<td>TYPE T = ENUM[a,b,c,d,e]</td>
<td>3</td>
</tr>
<tr>
<td>TYPE T = ENUM[a,b,c,d,e,f]</td>
<td>3</td>
</tr>
</tbody>
</table>

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

   **Example 2 (Wrong):**
   ```pascal
   VAR x, y, z, x, a : CHAR;
   ```

   **Example 3 (Wrong):**
   ```pascal
   VAR x : CHAR;
   y,z,a,x : INTEGER;
   n,x : BOOLEAN;
   ```