Overview...

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: \( \text{X:="Hi" * "there".} \)

Prepare for Code Generation

- Insert explicit type conversions: \( \text{X:=5+6.7 \Rightarrow X:=\text{float(5)+6.7}.} \)
- 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.

\[
\begin{align*}
\text{VAR I, K : INTEGER;} \\
\text{PROCEDURE P (K : INTEGER);} \\
\text{PROCEDURE Q (L : INTEGER);} \\
\text{BEGIN I := L + K; (* I_{\text{global}} := L + K *) END Q;} \\
\text{VAR J : INTEGER;} \\
\text{BEGIN I := J + K; (* I_{\text{global}} := J + K *) END P;}
\end{align*}
\]

Semantic Analysis II

Copyright © 2002 C. Collberg

Overview

- Several independent tasks have to be performed during semantic analysis:
- Go through and check the legality of the declarations (types, variables, procedures, etc) in the program. Check for:
  1. multiple declarations: \( \text{VAR x,y,x : int.} \)
  2. undeclared types: \( \text{VAR x : integer.} \)
  3. illegal symbol kinds: \( \text{VAR X : integer; VAR Y : X.} \)
- Construct symbol tables and environments to be used during name and expression analysis.
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:

```latex
\begin{align*}
\text{All Symbols} & \quad \text{Name, Position, Level, Enclosing Block,} \ldots \\
\text{Variables} & \quad \text{Type, Size,} \ldots \\
\text{Constants} & \quad \text{Type, Size, Value,} \ldots \\
\text{Types} & \quad \text{Size,} \ldots \\
\text{Records} & \quad \text{Fields} \\
\text{Arrays} & \quad \text{Index-Type, Index-Range, Element-Type} \\
\text{Procedures} & \quad \text{Formal Parameters, Size of Local Data,} \ldots \\
\end{align*}
```

```
\begin{align*}
\text{TYPE T} & \quad \text{= RECORD} \quad a, b : \text{ CHAR END;} \\
\text{VAR X : T;} \\
\text{PROCEDURE Q (a : T); BEGIN} & \ldots \text{ END Q; }
\end{align*}
```
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 ∈ \{VarDecl, ProcDecl\} THEN
    IF n.Id ∈ n.IdsIn THEN
      PRINT n.Pos ":Multiple "declaration: " n.Id;
    ENDIF;
    n.Next.IdsIn := n.IdsIn ⋃ \{n.Id\};
    Decl(n.Next);
  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.SizeOut := n.Next.SizeOut;
PROCEDURE ProcDecl (n: Node);
  n.Next.SizeIn := n.SizeIn;
  (* Rest is same as for VarDecl *)

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

Putting it all together...

Program ::= \Id: String \DeclSeq: Decl StatSeq: Stat
Decl ::= VarDecl | ProcDecl | NoDecl
VarDecl ::= \Id: String \ TypeName: String Next: Decl
  \Ids: SyTab \Count: INTEGER \Size: INTEGER
ProcDecl ::= \Id: String Args: Decl Locals: Decl StatSeq: Stat
  Next: Decl \Ids: SyTab \Count: INTEGER
  \Size: INTEGER
NoDecl ::= \Ids: SyTab \Count: INTEGER
  \Size: INTEGER
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 $\uparrow$Count: INTEGER and $\uparrow$Size: INTEGER are available.

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

⇓

Build Symbol Table

⇓

{(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)}

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.

```pascal
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;
```

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 U {Sy};
    Decl(n.Next);
  ELSIF n.Kind = ProcDecl THEN
    Sy := (Name=n.Id, Kind=PROC,No=n.CountIn);
    n.Next.IdsIn := n.IdsIn U {Sy};
    Decl(n.Next); n.IdsOut:=...
  ELSIF n.Kind = NoDecl THEN
    n.IdsOut := n.IdsIn
ENDIF
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.

**Program**

```plaintext
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;
```

**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;
```

```
SyTabs for Nested Scope...

Formal, VarDecl ::= \( \leftarrow \text{Id} : \text{String} \leftarrow \text{TypeName} : \text{String} \)

Next:Decl \( \uparrow \text{Ids} : \text{SyTab} \)

Next.IdsIn := IdsIn \( \cup \) \{ (Name=Id,Kind=VAR,\ldots) \};

IdsOut := Next.IdsOut

ProcDecl ::= \( \leftarrow \text{Id} : \text{String} \leftarrow \text{Locals} : \text{Decl} \)

Formals:Decl StatSeq:Stat Next:Decl \( \uparrow \text{Ids} : \text{SyTab} \)

Next.IdsIn := \{\};

Locals.IdsIn := Formals.IdsOut;

Next.IdsIn := IdsIn \( \cup \)

\{ (Name=Id,Kind=PROC,Vars=Locals.IdsOut,\ldots) \};

IdsOut := Next.IdsOut

NoDecl ::= \( \uparrow \text{Ids} : \text{SyTab} \) \{ IdsOut := IdsIn \}

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.
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

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.

Program P;
VAR I : INTEGER;
PROCEDURE Q (X : CHAR);
PROCEDURE S (K : CHAR);
VAR I : INTEGER:
BEGIN ● END S;
VAR K : INTEGER:
BEGIN ● END Q;
BEGIN
END P.

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
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 ::= \langle Id: String \rangle Decls:Decl StatSeq:Stat
{ Decls.Env := {INT,REAL,CHAR,TRUNC,FLOAT};
   Decls.IdsIn:={};
   StatSeq.Env := cons(Decls.IdsOut,Decls.Env);
}

Formal, VarDecl ::= \langle Id: String \rangle \langle TypeName: String \rangle
   Next:Decl \uparrow Ids:SyTab \downarrow Env:EnvT
{ CHECK NOT member(Env,TypeName)
   \implies ERROR("Ident not declared")
   Next.Env := Env;
}

ProcDecl ::= \langle Id: String \rangle Locals:Decl Formals:Decl
   StatSeq:Stat Next:Decl \uparrow Ids:SyTab \downarrow Env:EnvT
{ 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; }

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...*)
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.

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...*)
### 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

### 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
    · · ·
  END;
Name : String; Next : SyTabT;
END;
EnvT = POINTER TO RECORD
  SyTab : SyTabT; Next : EnvT;
END;
```

### 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).
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? Pass 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.

Summary...

- A threaded attribute $\downarrow\text{A:T}$ actually consists of two attributes: an inherited attribute $\downarrow\text{AIn:T}$ and a synthesized attribute $\uparrow\text{AOut:T}$. 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 $\downarrow\text{Ids:SyTabT}$ to construct the symbol table.
- This symbol table can then be passed down the tree (using an inherited attribute $\downarrow\text{Env:SyTabT}$) during name analysis.

- An inherited attribute is given a value **before** a recursive call is made:
  
  $\text{n.LOP.Env := n.Env; \text{Expr(n.LOP)};}$

- An synthesized attribute is given a value **before** a recursive call returns:
  
  \[
  \text{PROCEDURE \text{Expr (n: \text{Node});}}
  \]
  
  \[
  \text{IF n.Kind = IntConst THEN n.Type := "INT";}\]

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

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 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 (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 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 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 VII

• Assume that enumerated types are declared in this fashion:

```plaintext
TYPE T = ENUM[Margaret=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:

```plaintext
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"
```

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 VIII

1. Assume that enumerated types are declared in the “normal” Pascal fashion:
   \[
   \text{TYPE T = ENUM\{Marge, Homer, Bart, Maggie, Lisa\};}
   \]
   Assume furthermore that the individual identifiers are given numbers \(0, 1, 2, \ldots\).

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.

   Examples:
   \[
   \begin{align*}
   \text{TYPE T = ENUM\{a\} \Rightarrow 1 \text{ bit}} \\
   \text{TYPE T = ENUM\{a,b\} \Rightarrow 1 \text{ bit}} \\
   \text{TYPE T = ENUM\{a,b,c\} \Rightarrow 2 \text{ bits}} \\
   \text{TYPE T = ENUM\{a,b,c,d\} \Rightarrow 2 \text{ bits}} \\
   \text{TYPE T = ENUM\{a,b,c,d,e\} \Rightarrow 3 \text{ bits}} \\
   \text{TYPE T = ENUM\{a,b,c,d,e,f\} \Rightarrow 3 \text{ bits}}
   \end{align*}
   \]

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):
   \[
   \text{VAR x : CHAR; y,z,a,b : INTEGER; n,s : BOOLEAN;}
   \]

   Example 2 (Wrong):
   \[
   \text{VAR x, y, z, x, a : CHAR;}
   \]

   Example 3 (Wrong):
   \[
   \text{VAR x : CHAR; y,z,a,x : INTEGER; n,x : BOOLEAN;}
   \]