Languages and Compilers (SProg og Oversættere)

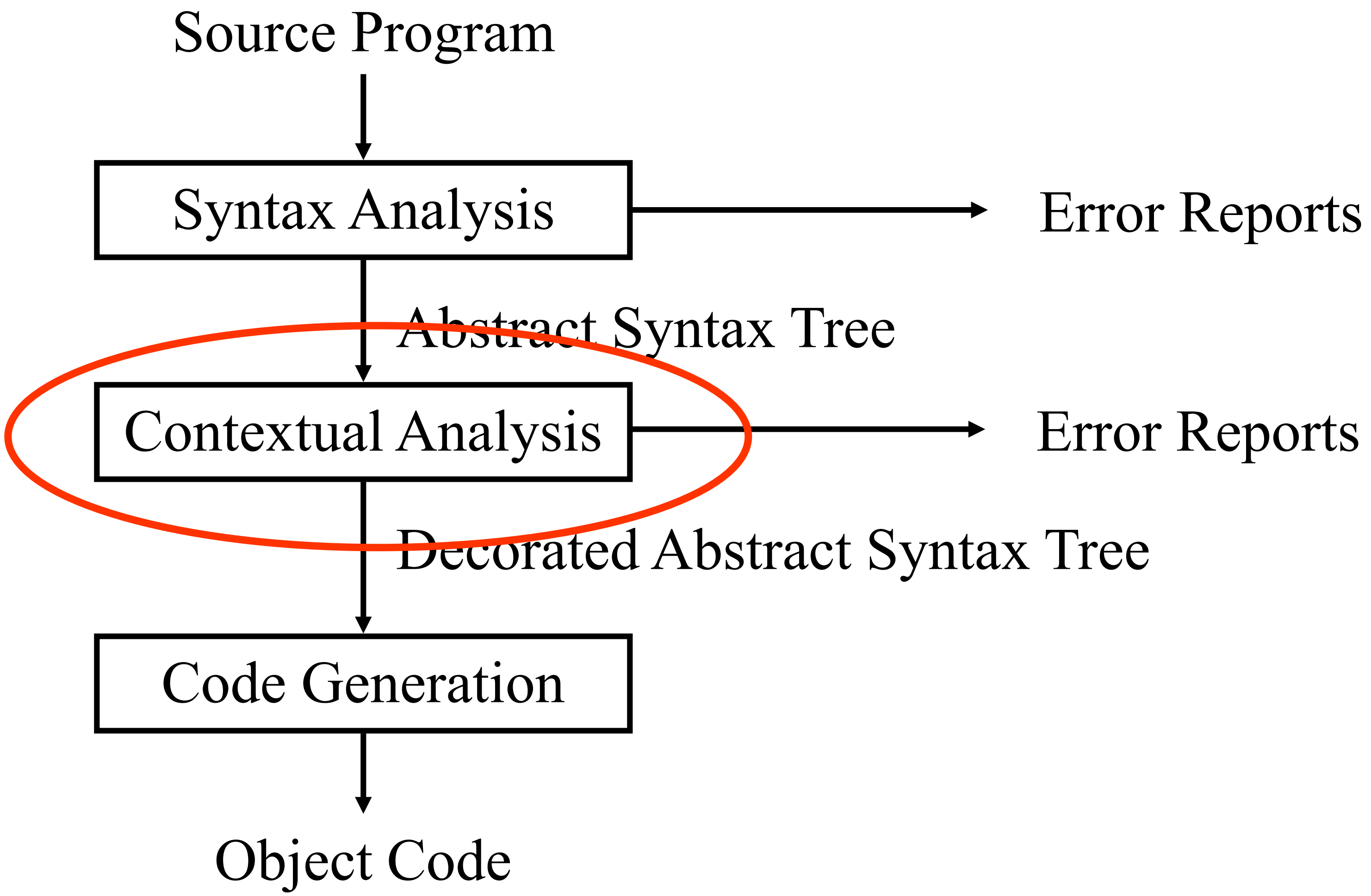
Lecture 10
Scopes and Symbol Tables

Bent Thomsen
Department of Computer Science
Aalborg University

Learning Goals

- Understand the purpose of the Contextual Analysis phase of the compiler
- Knowledge about scope and type rules
- Knowledge about Symbol Tables
- Knowledge about strategies for implementing this phase

The “Phases” of a Compiler



Programming Language Specification

- A language specification has (at least) three parts:
  - Syntax of the language: usually formal: EBNF
  - **Contextual constraints:**
    - **scope rules**
      - **often written in English, but can be formal**
      - **(see chapter 6 on p. 86-93 in Transitions and Trees)**
    - **type rules**
      - **formal or informal**
      - **See chapter 13 on p.185-210 in Transitions and Trees)**
  - Semantics:
    - defined by the implementation
    - informal descriptions in English
    - formal using operational or denotational semantics
      - See Transitions and Trees

Contextual Constraints

Syntax rules alone are not enough to specify the format of well-formed programs.

Example 1:

```
let const m~2;
in  m + x
```

Undefined! ⟹ Scope Rules

Example 2:

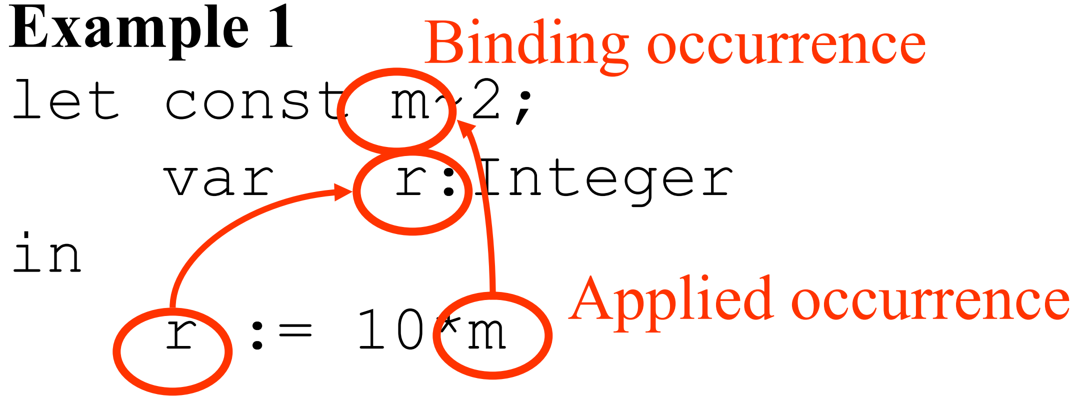
```
let const m~2 ;
    var   n:Boolean
in begin
   n := m<4;
   n := n+1
end
```

Type error! ⟹ Type Rules

Scope Rules

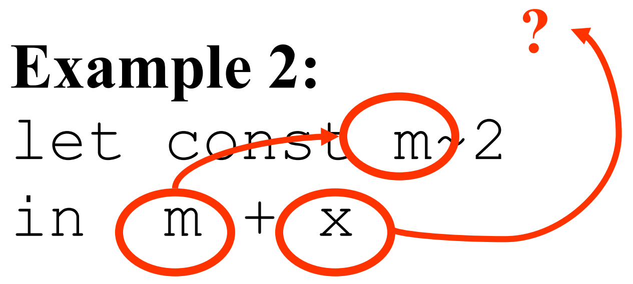
Scope rules regulate visibility of identifiers. They relate every **applied occurrence** of an identifier to a **binding occurrence**

Example 1



```
let const m~2;
    var   r:Integer
in
   r  := 10*m
```

Example 2:



```
let const m~2
in  m + x
```

Terminology:

Static binding vs. *dynamic binding*

Static scope/block structured scope vs. dynamic scope

Implicit vs. explicit binding   *(see p. 86-93 in Transitions and Trees)*

Type Rules

- In order to "tame" the behaviour of programs we can make more or less restrictive type rules
- The validity of these rules is controlled by the type cheking algorithm
- Details depend upon the type system
  - Type systems can be very complicated
    - Lets look at them later
      - Simple type system (next lecture)
      - More complex type systems (later lecture)

Type Rules

Type rules regulate the expected types of arguments and types of returned values for the operations of a language.

Examples

Type rule of `<` :

`E1 < E2` is type correct and of type `Boolean`
if `E1` and `E2` are type correct and of type `Integer`

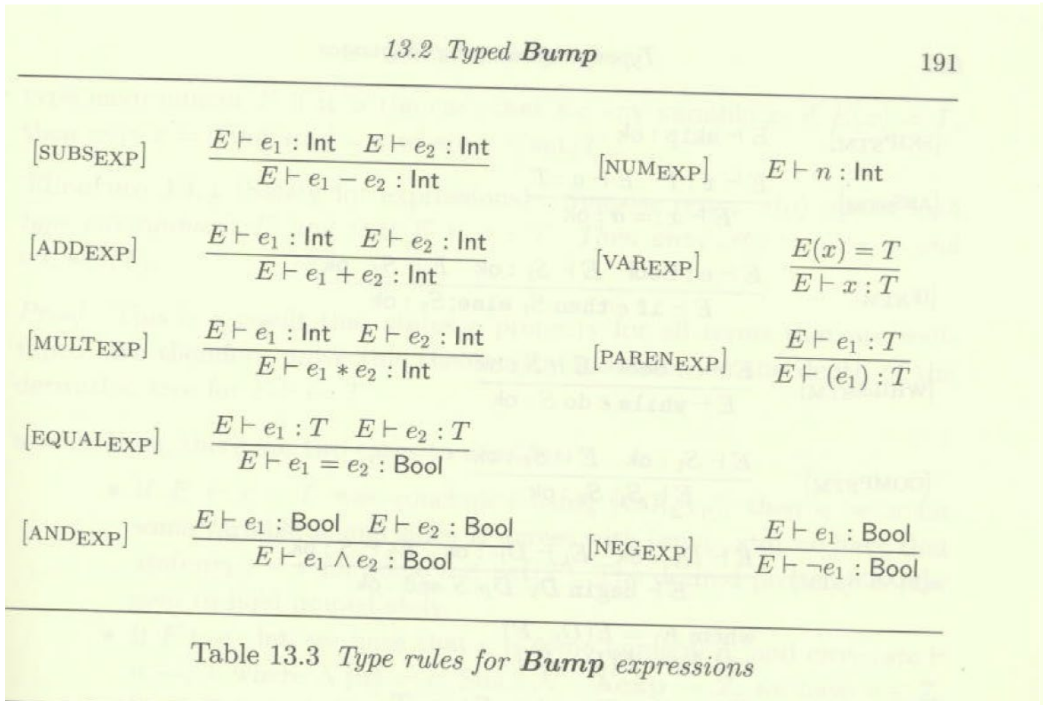
Type rule of `while`:

`while E do C` is type correct
if `E` of type `Boolean` and `C` type correct

Terminology:

Static typing vs. *dynamic typing*

See Chapter 13 in Trans. & Trees



[SUBS$_{\text{EXP}}$] $$\frac{E \vdash e_1 : \mathsf{Int} \quad E \vdash e_2 : \mathsf{Int}}{E \vdash e_1 - e_2 : \mathsf{Int}}$$

[NUM$_{\text{EXP}}$] $$E \vdash n : \mathsf{Int}$$

[ADD$_{\text{EXP}}$] $$\frac{E \vdash e_1 : \mathsf{Int} \quad E \vdash e_2 : \mathsf{Int}}{E \vdash e_1 + e_2 : \mathsf{Int}}$$

[VAR$_{\text{EXP}}$] $$\frac{E(x) = T}{E \vdash x : T}$$

[MULT$_{\text{EXP}}$] $$\frac{E \vdash e_1 : \mathsf{Int} \quad E \vdash e_2 : \mathsf{Int}}{E \vdash e_1 * e_2 : \mathsf{Int}}$$

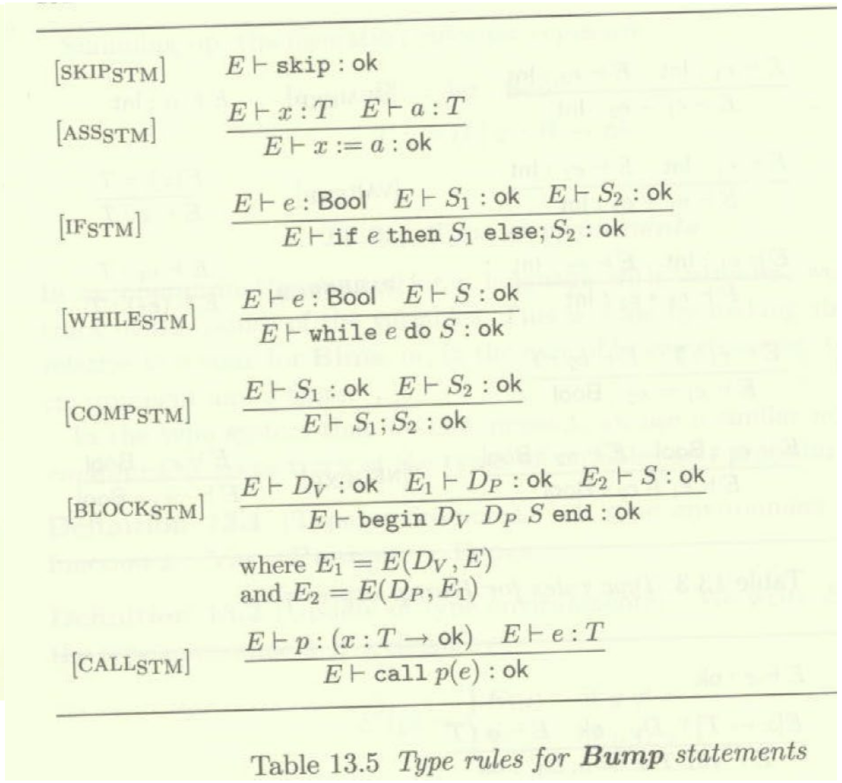
[PAREN$_{\text{EXP}}$] $$\frac{E \vdash e_1 : T}{E \vdash (e_1) : T}$$

[EQUAL$_{\text{EXP}}$] $$\frac{E \vdash e_1 : T \quad E \vdash e_2 : T}{E \vdash e_1 = e_2 : \mathsf{Bool}}$$

[AND$_{\text{EXP}}$] $$\frac{E \vdash e_1 : \mathsf{Bool} \quad E \vdash e_2 : \mathsf{Bool}}{E \vdash e_1 \wedge e_2 : \mathsf{Bool}}$$

[NEG$_{\text{EXP}}$] $$\frac{E \vdash e_1 : \mathsf{Bool}}{E \vdash \neg e_1 : \mathsf{Bool}}$$

Table 13.3 *Type rules for **Bump** expressions*



[SKIP$_{\text{STM}}$] $$E \vdash \mathtt{skip} : \mathsf{ok}$$

[ASS$_{\text{STM}}$] $$\frac{E \vdash x : T \quad E \vdash a : T}{E \vdash x := a : \mathsf{ok}}$$

[IF$_{\text{STM}}$] $$\frac{E \vdash e : \mathsf{Bool} \quad E \vdash S_1 : \mathsf{ok} \quad E \vdash S_2 : \mathsf{ok}}{E \vdash \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else};\ S_2 : \mathsf{ok}}$$

[WHILE$_{\text{STM}}$] $$\frac{E \vdash e : \mathsf{Bool} \quad E \vdash S : \mathsf{ok}}{E \vdash \mathtt{while}\ e\ \mathtt{do}\ S : \mathsf{ok}}$$

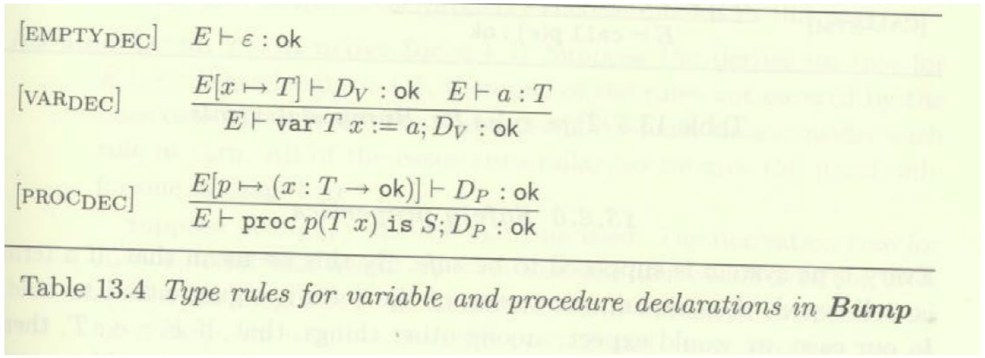
[COMP$_{\text{STM}}$] $$\frac{E \vdash S_1 : \mathsf{ok} \quad E \vdash S_2 : \mathsf{ok}}{E \vdash S_1; S_2 : \mathsf{ok}}$$

[BLOCK$_{\text{STM}}$] $$\frac{E \vdash D_V : \mathsf{ok} \quad E_1 \vdash D_P : \mathsf{ok} \quad E_2 \vdash S : \mathsf{ok}}{E \vdash \mathtt{begin}\ D_V\ D_P\ S\ \mathtt{end} : \mathsf{ok}}$$

where $E_1 = E(D_V, E)$
and $E_2 = E(D_P, E_1)$

[CALL$_{\text{STM}}$] $$\frac{E \vdash p : (x : T \to \mathsf{ok}) \quad E \vdash e : T}{E \vdash \mathtt{call}\ p(e) : \mathsf{ok}}$$

Table 13.5 *Type rules for **Bump** statements*



[EMPTY$_{\text{DEC}}$] $$E \vdash \varepsilon : \mathsf{ok}$$

[VAR$_{\text{DEC}}$] $$\frac{E[x \mapsto T] \vdash D_V : \mathsf{ok} \quad E \vdash a : T}{E \vdash \mathtt{var}\ T\ x := a; D_V : \mathsf{ok}}$$

[PROC$_{\text{DEC}}$] $$\frac{E[p \mapsto (x : T \to \mathsf{ok})] \vdash D_P : \mathsf{ok}}{E \vdash \mathtt{proc}\ p(T\ x)\ \mathtt{is}\ S; D_P : \mathsf{ok}}$$

Table 13.4 *Type rules for variable and procedure declarations in **Bump***

Typechecking

- Static typechecking
  - All type errors are detected at compile-time
  - Pascal and C are *statically typed*
  - Most modern languages have a large emphasis on static typechecking
- Dynamic typechecking
  - Scripting languages such as JavaScript, PhP, Perl and Python do run-time typechecking
- Mix of Static and Dynamic
  - object-oriented programming requires some runtime typechecking: e.g. Java has a lot of compile-time typechecking but it is still necessary for some potential runtime type errors to be detected by the runtime system

- Static typechecking involves calculating or *inferring* the types of expressions (by using information about the types of their components) and checking that these types are what they should be (e.g. the condition in an *if* statement must have type *Boolean*).

Contextual Analysis Phase

- Purposes:
  - Finish syntax analysis by deriving context-sensitive information
    - Scoping
    - (static) type checking
  - Start to interpret meaning of program based on its syntactic structure
  - Prepare for the final stage of compilation: Code generation

Contextual Analyzer

- Which contextual constraints might the compiler add?
  - Is identifier x declared before it is used?
  - Which declaration of x does an occurrence of x refer to?
  - Is x an Integer, Boolean, array or a function?
  - Is an expression type-consistent?
  - Are any names declared but not used?
  - Has x been initialized before it is being accessed?
  - Is an array reference out of bounds?
  - Does a function `bar` produce a constant value?
  - Where can x be stored? (heap, stack, …)

Why contextual analysis can be hard

- Questions and answers involve non-local information
- Answers mostly depend on values, not syntax
- Answers may involve computations

Solution alternatives:

- Abstract syntax tree
  - specify non-local computations by walking the tree
- Identification tables (sometimes called symbol tables)
  - central store for facts + checking code
- Language design
  - simplify language

To simplify the language design or not?

- Syntax vs. types
  - Bool expressions and Int expressions as syntactic categories
  - One syntactic category of Expressions with types

```
Bexp     :=     true
         |      false
         |      Bexp Bop Bexp

Bop      :=     & | or | …

IntExp   :=     Literal
         |      IntExp Iop IntExp

Iop      :=     + | - | * | / | …
```

vs

```
Exp      :=     Literal
         |      Exp op Exp

Op       :=     & | or | + | - | * | / | …
```

- Psychology of syntax errors vs. type errors
  - Most C programmers accept syntax errors as their fault, but regard typing errors as annoying constraints imposed on them

Language Issues

Example **Pascal:**

Pascal was explicitly designed to be easy to implement with a single pass compiler:

- Every **identifier** must be **declared before** its first **use.**

```
var n:integer;

procedure inc;
begin
   n:=n+1
end
```

?

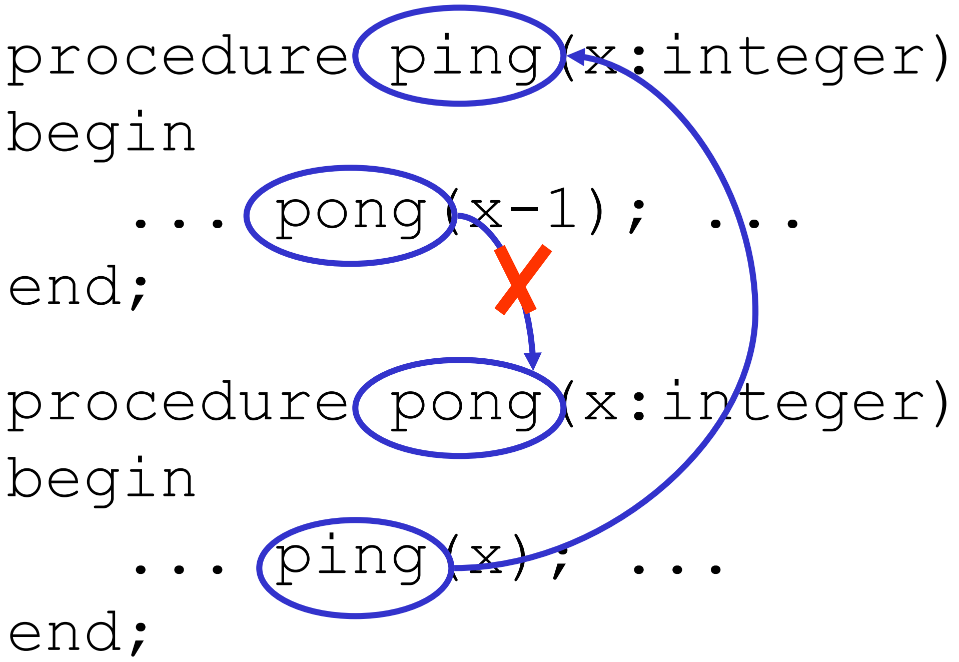
```
procedure inc;
begin
   n:=n+1
end;

var n:integer;
```

Undeclared Variable!

Language Issues

Example **Pascal:**

- Every **identifier** must be **declared before** it is **used.**
- How to handle mutual recursion then?

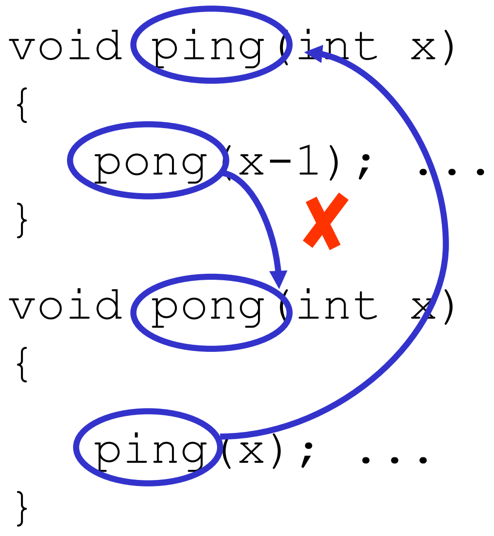


```
procedure ping(x:integer)
begin
    ... pong(x-1); ...
end;

procedure pong(x:integer)
begin
    ... ping(x); ...
end;
```

C was designed for a single pass compiler

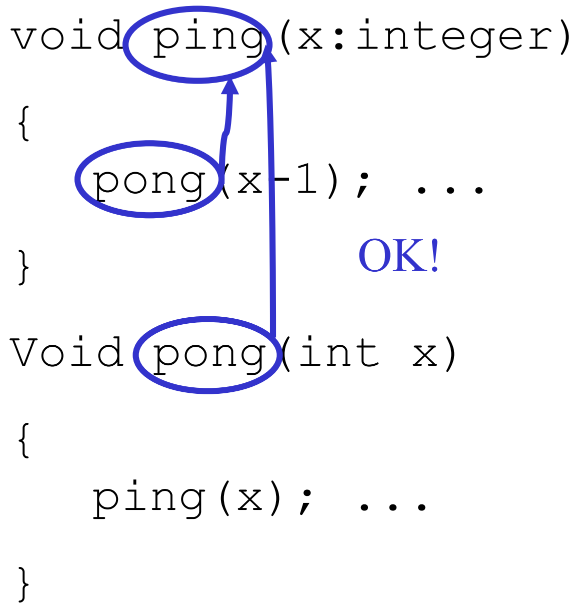
Mutual recursion problem**:**

- Every **identifier** must be **declared before** it is **used.**
- How to handle mutual recursion then?



```
void ping(int x)
{
   pong(x-1); ...
}

void pong(int x)
{
   ping(x); ...
}
```

```
void pong(int x);
```



```
void ping(x:integer)
{
   pong(x-1); ...
}
Void pong(int x)
{
   ping(x); ...
}
```

Language Issues

Example **Pascal:**

- Every **identifier** must be **declared before** it is **used.**
- How to handle mutual recursion then?

```
forward procedure pong(x:integer)

procedure ping(x:integer)
begin
    ... pong(x-1); ...
end;

procedure pong(x:integer)
begin
    ... ping(x); ...
end;
```

OK!

Language Issues

Example **SML:**

- Every **identifier** must be **declared before** it is **used.**
- How to handle mutual recursion then?

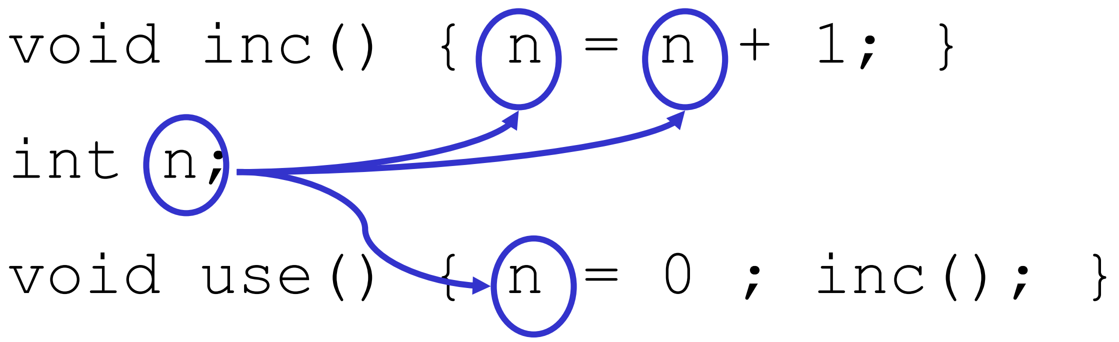
```
fun ping(x:int)=
    ... pong(x-1) ...

and pong(x:int)=
    ... ping(x) ...
;
```

OK!

Language Issues

Example **Java:**

- **identifiers** can be **declared before** they are **used.**
- thus a Java compiler needs at least two passes



```
Class Example {
        void inc() { n = n + 1; }
        int n;
        void use() { n = 0 ; inc(); }
}
```

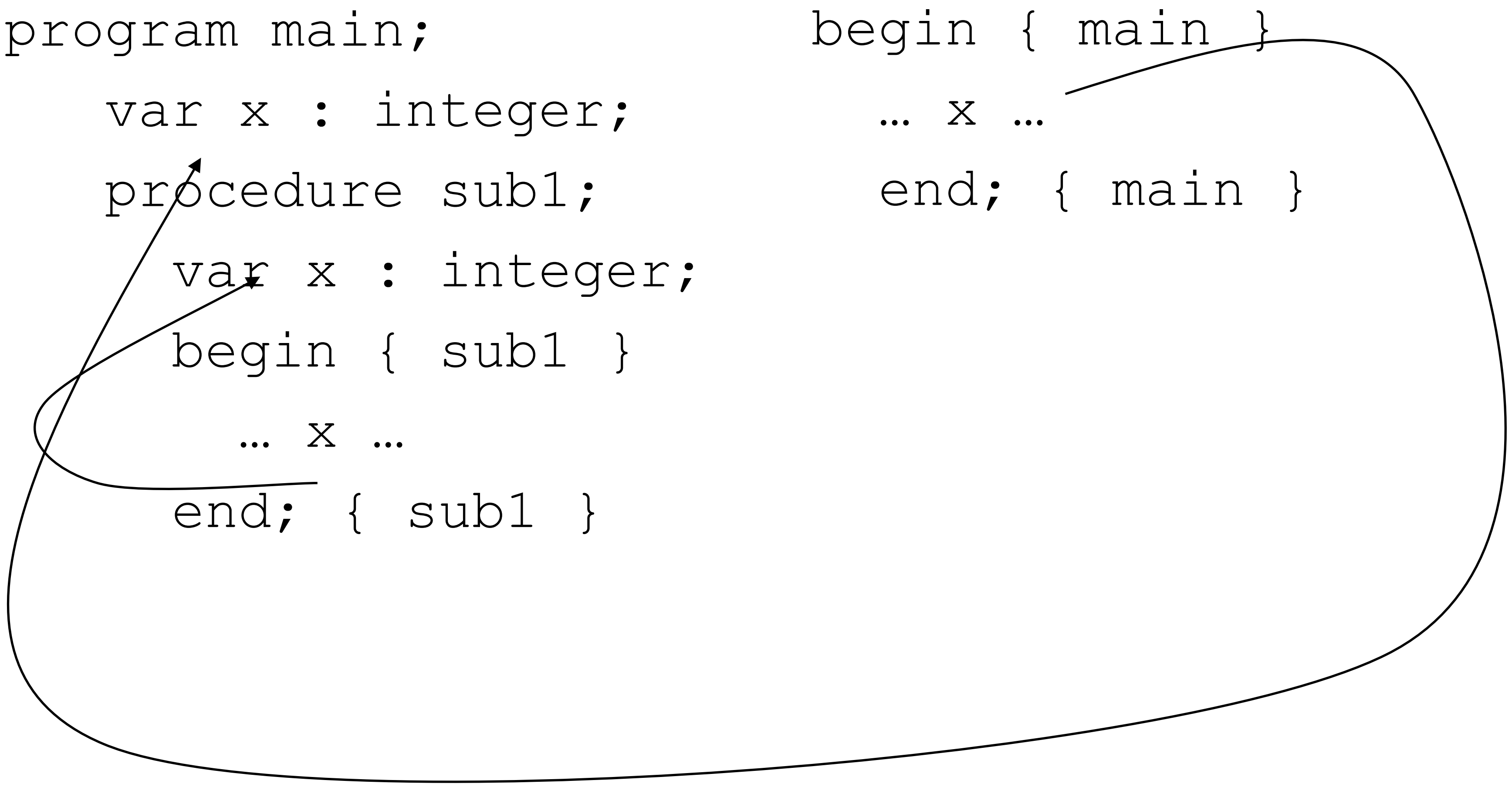
Scope of Variable

- Range of program that can reference that variable (ie access the corresponding data object by the variable’s name)
- Variable is *local* to program or block if it is declared there
- Variable is *non-local* to program unit if it is visible there but not declared there
- Static vs. Dynamic scope

Static Scoping

- Scope computed at compile time, based on program text
- To determine the name of a used variable we must find statement declaring variable
- Subprograms and blocks generate hierarchy of scopes
  - Subprogram or block that declares current subprogram or contains current block is its *static parent*
- General procedure to find declaration:
  - First see if variable is local; if yes, done
  - If non-local to current subprogram or block recursively search static parent until declaration is found
  - If no declaration is found this way, undeclared variable error detected

Example



```
program main;
   var x : integer;
   procedure sub1;
      var x : integer;
      begin { sub1 }
         … x …
      end; { sub1 }
```

```
begin { main }
   … x …
   end; { main }
```

Example (from p. 88 in Transitions and Trees)

```
begin
        var x:= 0;
        var y:= 42

        proc p is x:= x+3;
        proc q is call p;

        begin
                var x:=9;
                proc p is x := x+1;
                call q;
                y := x
        end
end
```

Assuming static scope for procedures and variables,
What is the value assigned to y ?

Value of y is 9, assuming static scope for procedures and variables

Dynamic Scope

- Now generally thought to have been a mistake
- Main example of use: original versions of LISP
  - APL, PostScript
  - (Note: Scheme uses static scope)
  - Perl allows variables to be declared to have dynamic scope
- Determined by the calling sequence of program units, not static layout
- Name bound to corresponding variable most recently declared among still active subprograms and blocks

Example

```
program main;
    var x : integer;
    procedure sub1;
      begin { sub1 }
         … x …
      end; { sub1 }
```

```
procedure sub2;
    var x :
  integer;
      begin { sub2
  }
      … call sub1 …
      end; { sub2 }
  … call  sub2…
  end; { main }
```

Example (from p. 88 in Transitions and Trees)

```
begin
        var x:= 0;
        var y:= 42

        proc p is x:= x+3;
        proc q is call p;

        begin
                var x:=9;
                proc p is x := x+1;
                call q;
                y := x
        end
end
```

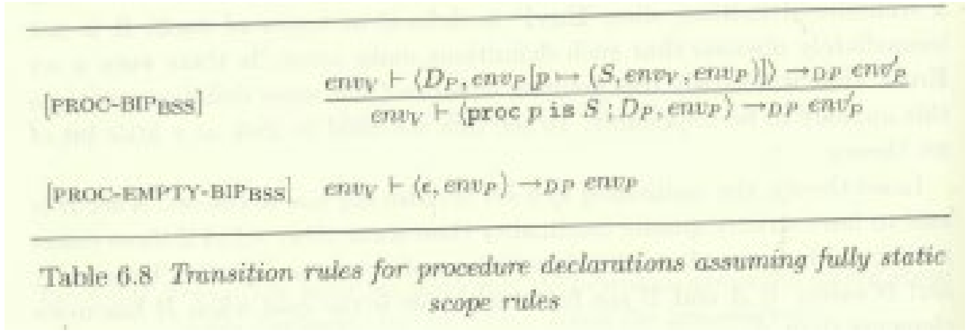
Assuming dynamic scope for procedures and variables,
What is the value assigned to y ?

Value of y is 10, assuming dynamic scope for procedures and variables

Value of y is 12, assuming static scope for procedures and dynamic of variables

Formal rules

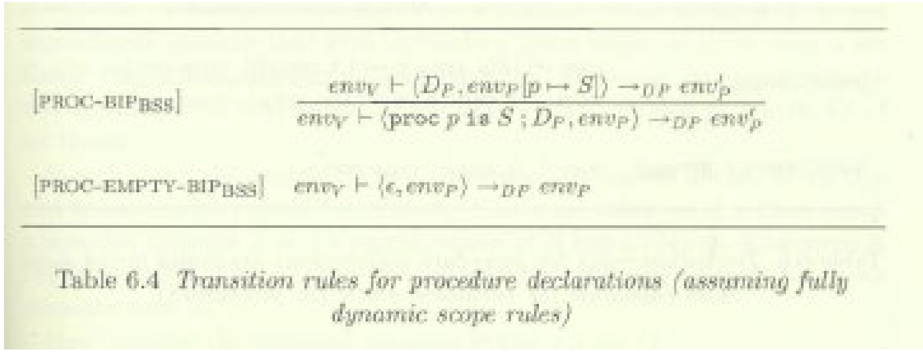
(from p. 89-93 in Transitions and Trees)



[PROC-BIP$_{\text{BSS}}$] $$\frac{env_V \vdash \langle D_P, env_P[p \mapsto (S, env_V, env_P)]\rangle \rightarrow_{DP} env'_P}{env_V \vdash \langle \texttt{proc}\ p\ \texttt{is}\ S\ ; D_P, env_P\rangle \rightarrow_{DP} env'_P}$$

[PROC-EMPTY-BIP$_{\text{BSS}}$] $env_V \vdash \langle \varepsilon, env_P\rangle \rightarrow_{DP} env_P$

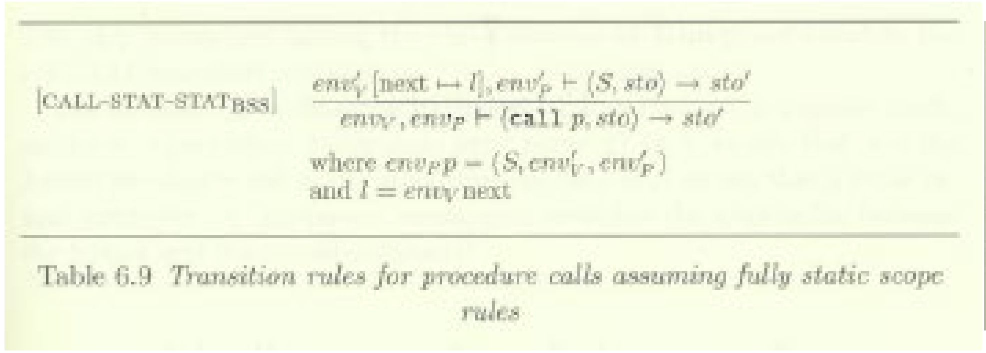
Table 6.8 *Transition rules for procedure declarations assuming fully static scope rules*



[PROC-BIP$_{\text{BSS}}$] $$\frac{env_V \vdash \langle D_P, env_P[p \mapsto S]\rangle \rightarrow_{DP} env'_P}{env_V \vdash \langle \texttt{proc}\ p\ \texttt{is}\ S\ ; D_P, env_P\rangle \rightarrow_{DP} env'_P}$$

[PROC-EMPTY-BIP$_{\text{BSS}}$] $env_V \vdash \langle \varepsilon, env_P\rangle \rightarrow_{DP} env_P$

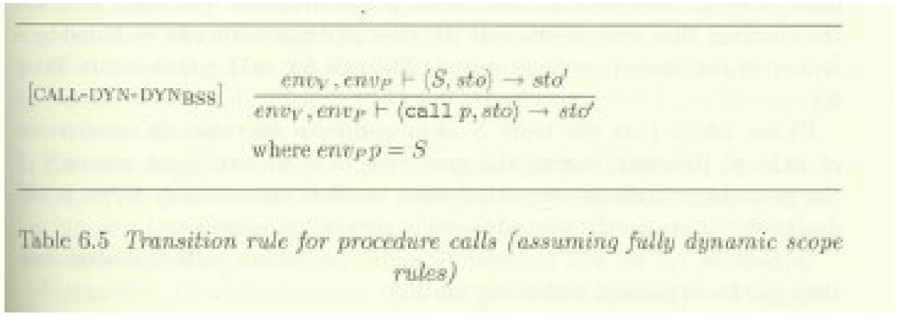
Table 6.4 *Transition rules for procedure declarations (assuming fully dynamic scope rules)*



[CALL-STAT-STAT$_{\text{BSS}}$] $$\frac{env'_V[\text{next} \mapsto l], env'_P \vdash \langle S, sto\rangle \rightarrow sto'}{env_V, env_P \vdash \langle \texttt{call}\ p, sto\rangle \rightarrow sto'}$$

where $env_P\, p = (S, env'_V, env'_P)$
and $l = env_V\, \text{next}$

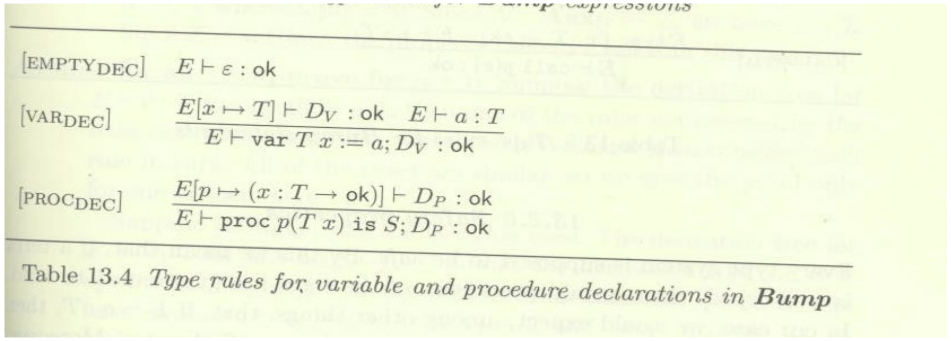
Table 6.9 *Transition rules for procedure calls assuming fully static scope rules*



[CALL-DYN-DYN$_{\text{BSS}}$] $$\frac{env_V, env_P \vdash \langle S, sto\rangle \rightarrow sto'}{env_V, env_P \vdash \langle \texttt{call}\ p, sto\rangle \rightarrow sto'}$$

where $env_P\, p = S$

Table 6.5 *Transition rule for procedure calls (assuming fully dynamic scope rules)*



[EMPTY$_{\text{DEC}}$] $E \vdash \varepsilon : \mathsf{ok}$

[VAR$_{\text{DEC}}$] $$\frac{E[x \mapsto T] \vdash D_V : \mathsf{ok} \quad E \vdash a : T}{E \vdash \texttt{var}\ T\ x := a; D_V : \mathsf{ok}}$$

[PROC$_{\text{DEC}}$] $$\frac{E[p \mapsto (x : T \rightarrow \mathsf{ok})] \vdash D_P : \mathsf{ok}}{E \vdash \texttt{proc}\ p(T\ x)\ \texttt{is}\ S; D_P : \mathsf{ok}}$$

Table 13.4 *Type rules for variable and procedure declarations in* ***Bump***

Pause

Organization of a Compiler

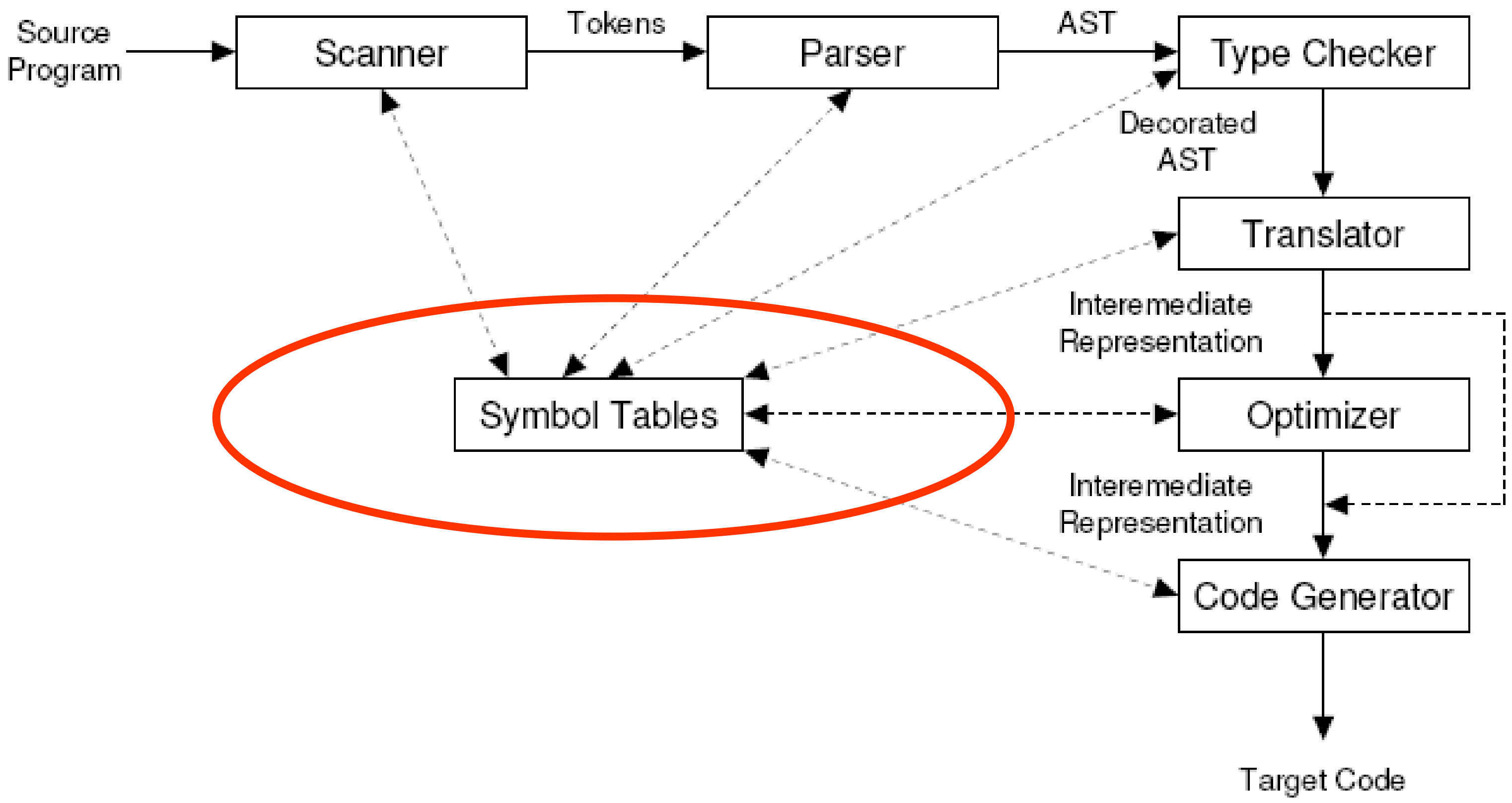


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

Identification Table

- The identification table (also often called symbol table) is a dictionary-style data structure in which we somehow store identifier names and relate each identifier to its corresponding **attributes**.
- Typical operations:
  - Empty the table
  - Add an entry (Identifier -> Attribute)
  - Find an entry for an identifier
  - (open and close scope)

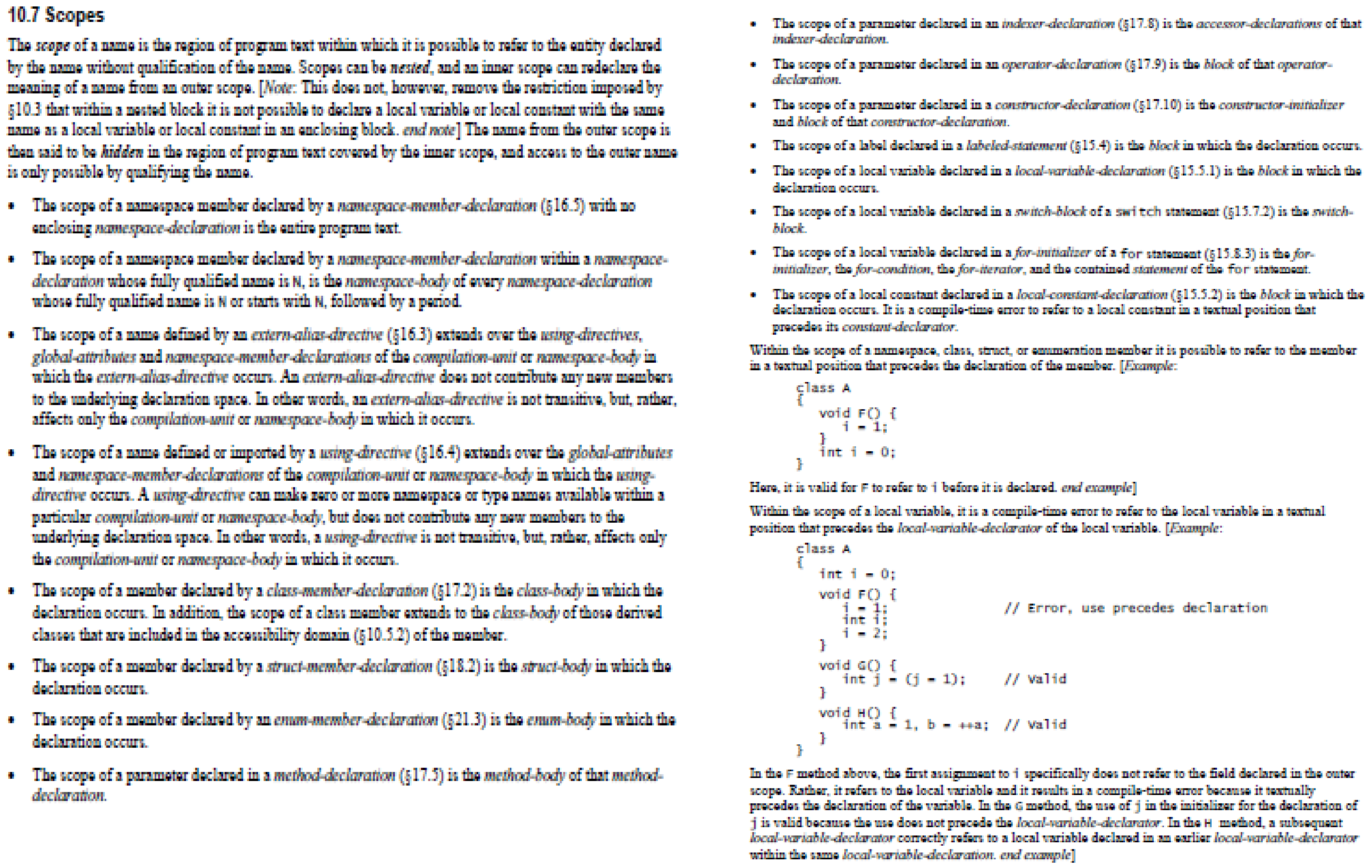
Identification Table

- The organization of the identification table depends on the programming language.
- Different kinds of "block structure" in languages:
  - Monolithic block structure: e.g. ac, BASIC, COBOL
  - Flat block structure: e.g. Fortran (and functions in C)
  - Nested block structure => Modern "block-structured" PLs (e.g. Algol, Pascal, C, C++, Scheme, Java,…)

a **block =** an area of text in the program that corresponds to some kind of boundary for the visibility of identifiers.

block structure = the textual relationship between blocks in a program.

C# scope definition



10.7 Scopes

The *scope* of a name is the region of program text within which it is possible to refer to the entity declared by the name without qualification of the name. Scopes can be *nested*, and an inner scope can redeclare the meaning of a name from an outer scope. [*Note*: This does not, however, remove the restriction imposed by §10.3 that within a nested block it is not possible to declare a local variable or local constant with the same name as a local variable or local constant in an enclosing block. *end note*] The name from the outer scope is then said to be *hidden* in the region of program text covered by the inner scope, and access to the outer name is only possible by qualifying the name.

- The scope of a namespace member declared by a *namespace-member-declaration* (§16.5) with no enclosing *namespace-declaration* is the entire program text.
- The scope of a namespace member declared by a *namespace-member-declaration* within a *namespace-declaration* whose fully qualified name is `N`, is the *namespace-body* of every *namespace-declaration* whose fully qualified name is `N` or starts with `N`, followed by a period.
- The scope of a name defined by an *extern-alias-directive* (§16.3) extends over the *using-directives*, *global-attributes* and *namespace-member-declarations* of the *compilation-unit* or *namespace-body* in which the *extern-alias-directive* occurs. An *extern-alias-directive* does not contribute any new members to the underlying declaration space. In other words, an *extern-alias-directive* is not transitive, but, rather, affects only the *compilation-unit* or *namespace-body* in which it occurs.
- The scope of a name defined or imported by a *using-directive* (§16.4) extends over the *global-attributes* and *namespace-member-declarations* of the *compilation-unit* or *namespace-body* in which the *using-directive* occurs. A *using-directive* can make zero or more namespace or type names available within a particular *compilation-unit* or *namespace-body*, but does not contribute any new members to the underlying declaration space. In other words, a *using-directive* is not transitive, but, rather, affects only the *compilation-unit* or *namespace-body* in which it occurs.
- The scope of a member declared by a *class-member-declaration* (§17.2) is the *class-body* in which the declaration occurs. In addition, the scope of a class member extends to the *class-body* of those derived classes that are included in the accessibility domain (§10.5.2) of the member.
- The scope of a member declared by a *struct-member-declaration* (§18.2) is the *struct-body* in which the declaration occurs.
- The scope of a member declared by an *enum-member-declaration* (§21.3) is the *enum-body* in which the declaration occurs.
- The scope of a parameter declared in a *method-declaration* (§17.5) is the *method-body* of that *method-declaration*.
- The scope of a parameter declared in an *indexer-declaration* (§17.8) is the *accessor-declarations* of that *indexer-declaration*.
- The scope of a parameter declared in an *operator-declaration* (§17.9) is the *block* of that *operator-declaration*.
- The scope of a parameter declared in a *constructor-declaration* (§17.10) is the *constructor-initializer* and *block* of that *constructor-declaration*.
- The scope of a label declared in a *labeled-statement* (§15.4) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *local-variable-declaration* (§15.5.1) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *switch-block* of a `switch` statement (§15.7.2) is the *switch-block*.
- The scope of a local variable declared in a *for-initializer* of a `for` statement (§15.8.3) is the *for-initializer*, the *for-condition*, the *for-iterator*, and the contained *statement* of the `for` statement.
- The scope of a local constant declared in a *local-constant-declaration* (§15.5.2) is the *block* in which the declaration occurs. It is a compile-time error to refer to a local constant in a textual position that precedes its *constant-declarator*.

Within the scope of a namespace, class, struct, or enumeration member it is possible to refer to the member in a textual position that precedes the declaration of the member. [*Example*:

```
class A
{
   void F() {
      i = 1;
   }
   int i = 0;
}
```

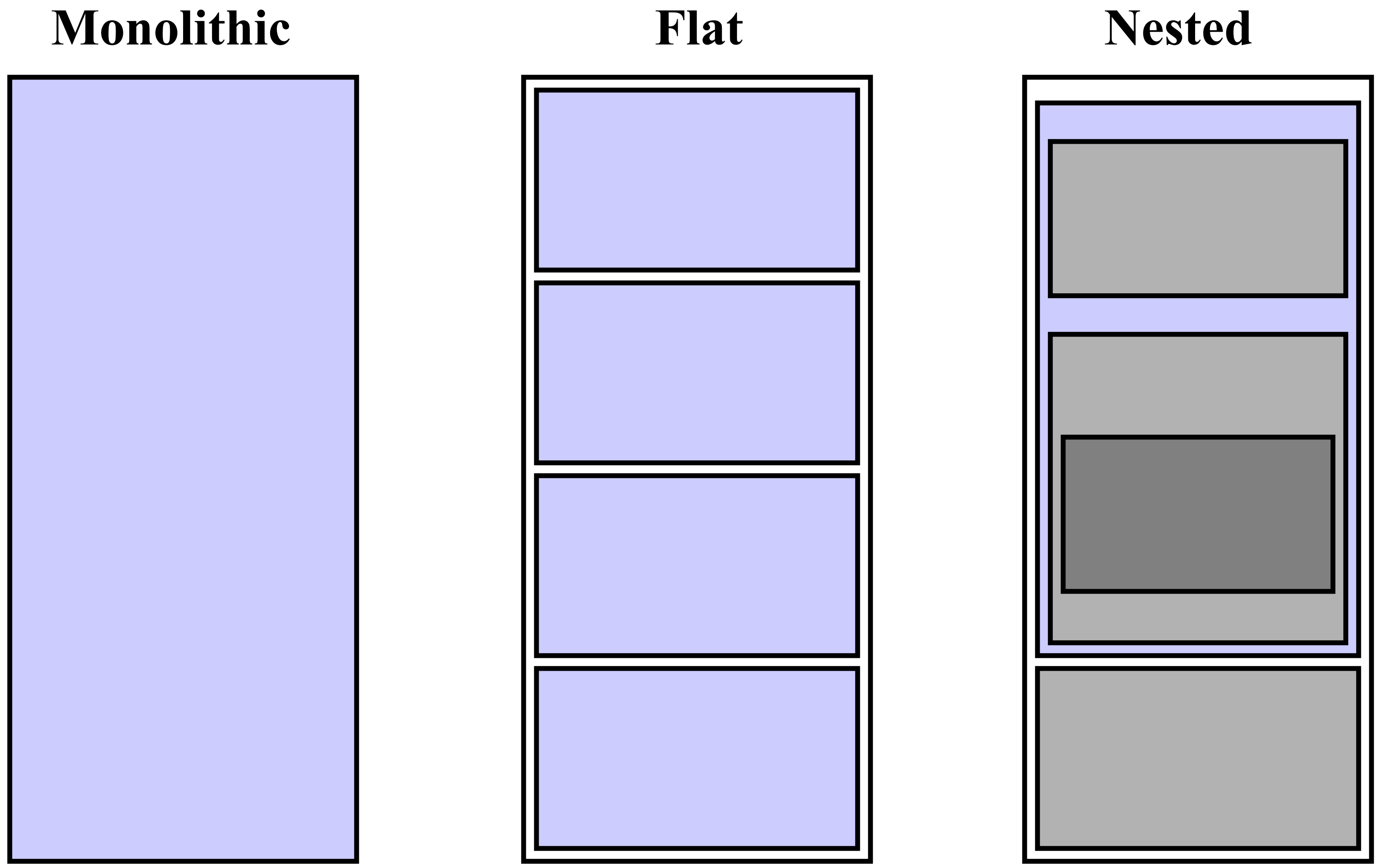
Here, it is valid for `F` to refer to `i` before it is declared. *end example*]

Within the scope of a local variable, it is a compile-time error to refer to the local variable in a textual position that precedes the *local-variable-declarator* of the local variable. [*Example*:

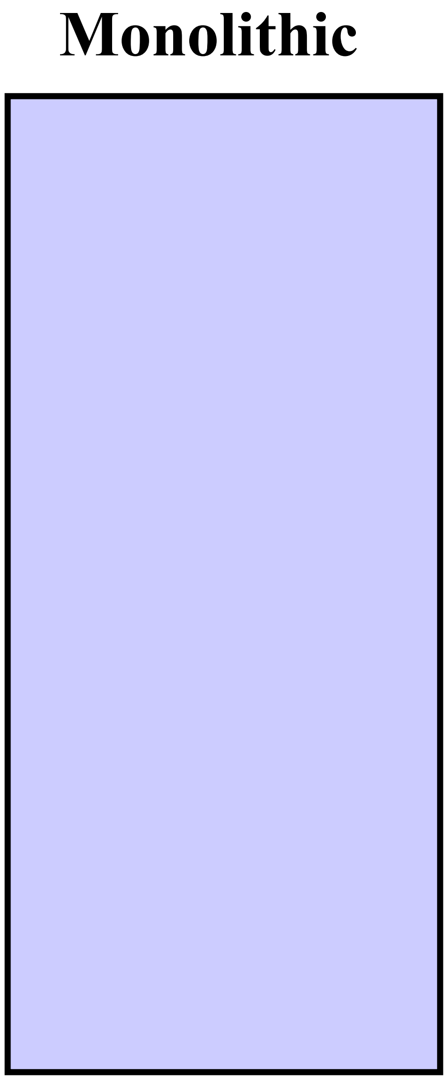
```
class A
{
   int i = 0;
   void F() {
      i = 1;                 // Error, use precedes declaration
      int i;
      i = 2;
   }
   void G() {
      int j = (j = 1);       // Valid
   }
   void H() {
      int a = 1, b = ++a;    // Valid
   }
}
```

In the `F` method above, the first assignment to `i` specifically does not refer to the field declared in the outer scope. Rather, it refers to the local variable and it results in a compile-time error because it textually precedes the declaration of the variable. In the `G` method, the use of `j` in the initializer for the declaration of `j` is valid because the use does not precede the *local-variable-declarator*. In the `H` method, a subsequent *local-variable-declarator* correctly refers to a local variable declared in an earlier *local-variable-declarator* within the same *local-variable-declaration*. *end example*]

Different kinds of Block Structure... a picture



Monolithic Block Structure



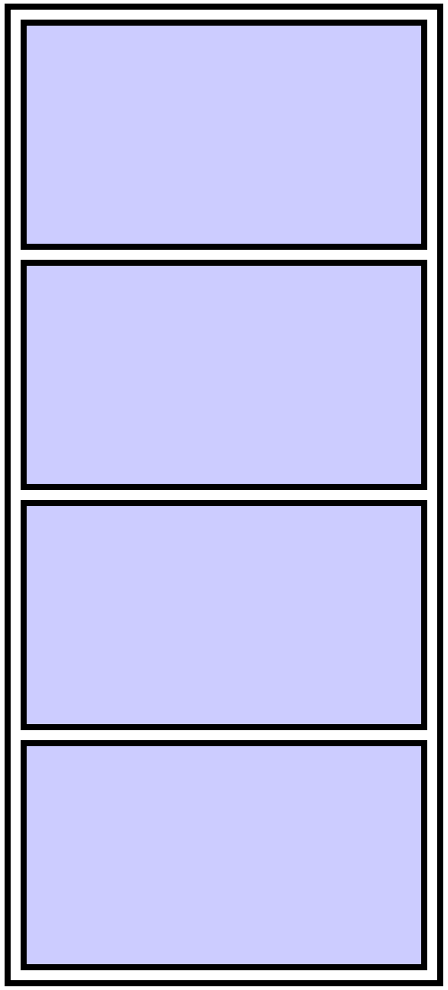
A language exhibits **monolithic block structure** if the only block is the entire program.

=> Every identifier is visible throughout the entire program

Very simple scope rules:

- No identifier may be declared more than once
- For every applied occurrence of an identifier *I* there must be a corresponding declaration.

Flat Block Structure



A language exhibits **flat block structure** if the program can be subdivided into several disjoint blocks

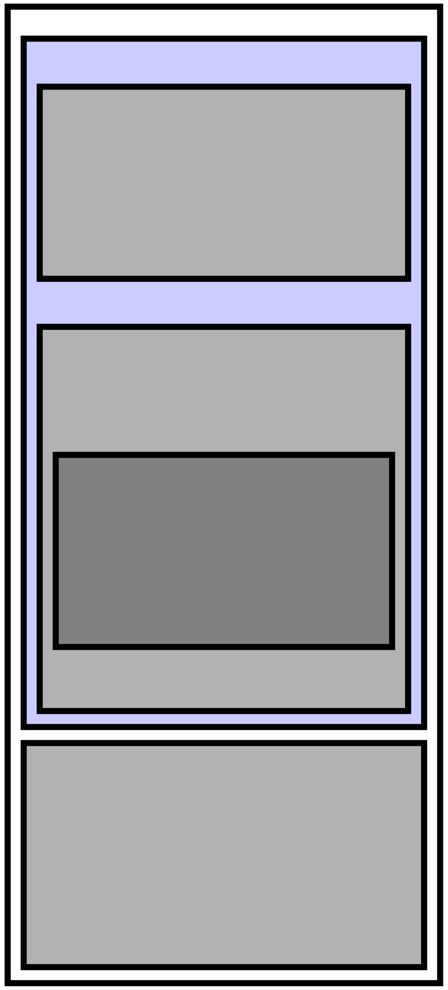
There are two scope levels: global or local.

Typical scope rules:

- a globally defined identifier may be redefined locally
- several local definitions of a single identifier may occur in different blocks (but not in the same block)
- For every applied occurrence of an identifier there must be either a local declaration within the same block or a global declaration.

Nested Block Structure

Nested



A language exhibits **nested block structure** if blocks may be nested one within another (typically with no upper bound on the level of nesting that is allowed).

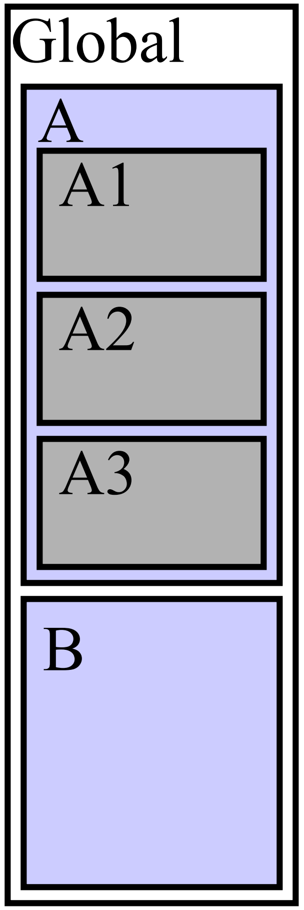
There can be any number of scope levels (depending on the level of nesting of blocks):

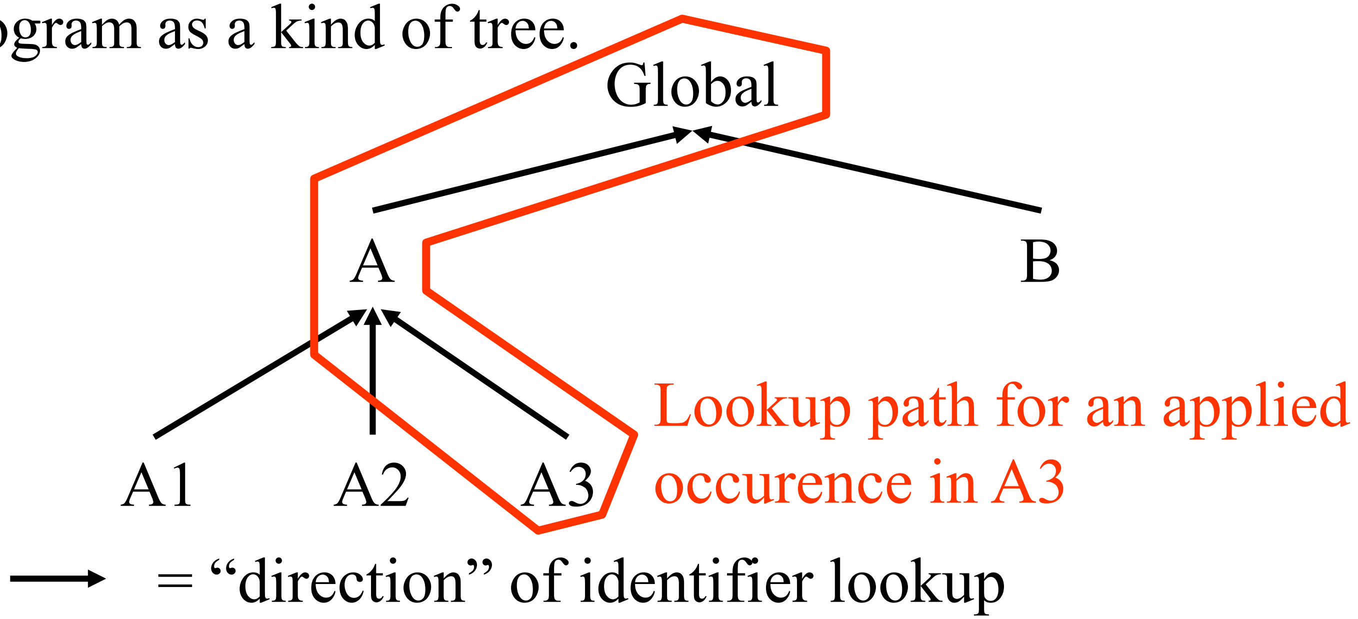
Typical scope rules:

- no identifier may be declared more than once within the same block (at the same level).
- for any applied occurrence there must be a corresponding declaration, either within the same block or in a block in which it is nested.

Identification Table

For a typical programming language, i.e. statically scoped language and with nested block structure we can visualize the structure of all scopes within a program as a kind of tree.




Global
A
B
A1
A2
A3
Lookup path for an applied occurence in A3
= "direction" of identifier lookup

At any one time (in analyzing the program) only a single path on the tree is accessible.
=> We don't necessarily need to keep the whole "scope" tree in memory all the time.

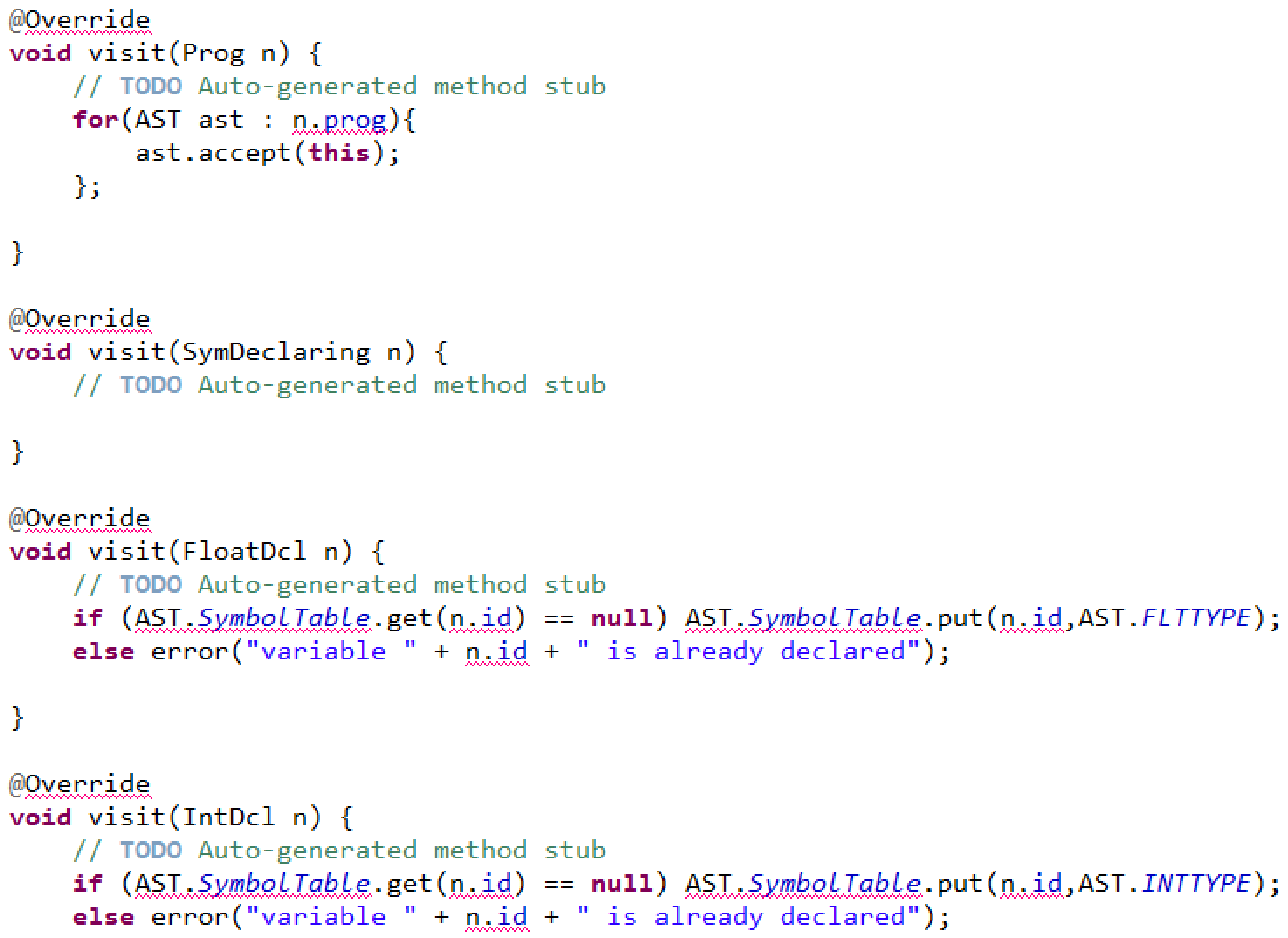
A Symbol Table Interface

- Methods
  - OpenScope()
  - CloseScope()
  - EnterSymbol(name, type)
  - RetreiveSymbol(name)
  - DeclaredLocally(name)
- Ex.
  - (Fig. 8.2) Code to build the symbol table for the AST in Fig. 8.1

```
procedure buildSymbolTable( )
    call processNode( $ASTroot$ )
end

procedure processNode( $node$ )
    switch (kind( $node$ ))
        case $Block$
            call $symtab$.openScope( )                                    (1)
        case $Dcl$
            call $symtab$.enterSymbol( $node.name, node.type$ )
        case $Ref$
            $sym \leftarrow symtab$.retrieveSymbol( $node.name$ )
            if $sym$ = null
            then  call error( "Undeclared symbol : ", $sym$ )
    foreach $c \in node$.getChildren( ) do  call processNode( $c$ )
    if kind( $node$ ) = $Block$
    then
        call $symtab$.closeScope( )                                       (2)
end
```

Figure 8.2: Building the symbol table

Ac SymbolTableFilling



```
@Override
void visit(Prog n) {
    // TODO Auto-generated method stub
    for(AST ast : n.prog){
        ast.accept(this);
    };

}

@Override
void visit(SymDeclaring n) {
    // TODO Auto-generated method stub

}

@Override
void visit(FloatDcl n) {
    // TODO Auto-generated method stub
    if (AST.SymbolTable.get(n.id) == null) AST.SymbolTable.put(n.id,AST.FLTTYPE);
    else error("variable " + n.id + " is already declared");

}

@Override
void visit(IntDcl n) {
    // TODO Auto-generated method stub
    if (AST.SymbolTable.get(n.id) == null) AST.SymbolTable.put(n.id,AST.INTTYPE);
    else error("variable " + n.id + " is already declared");
```

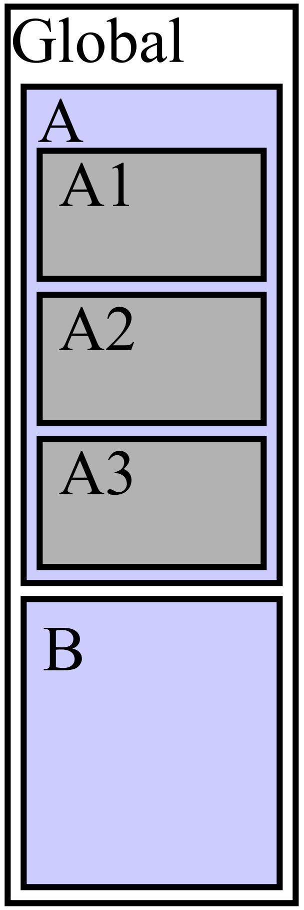
One Symbol Table or Many?

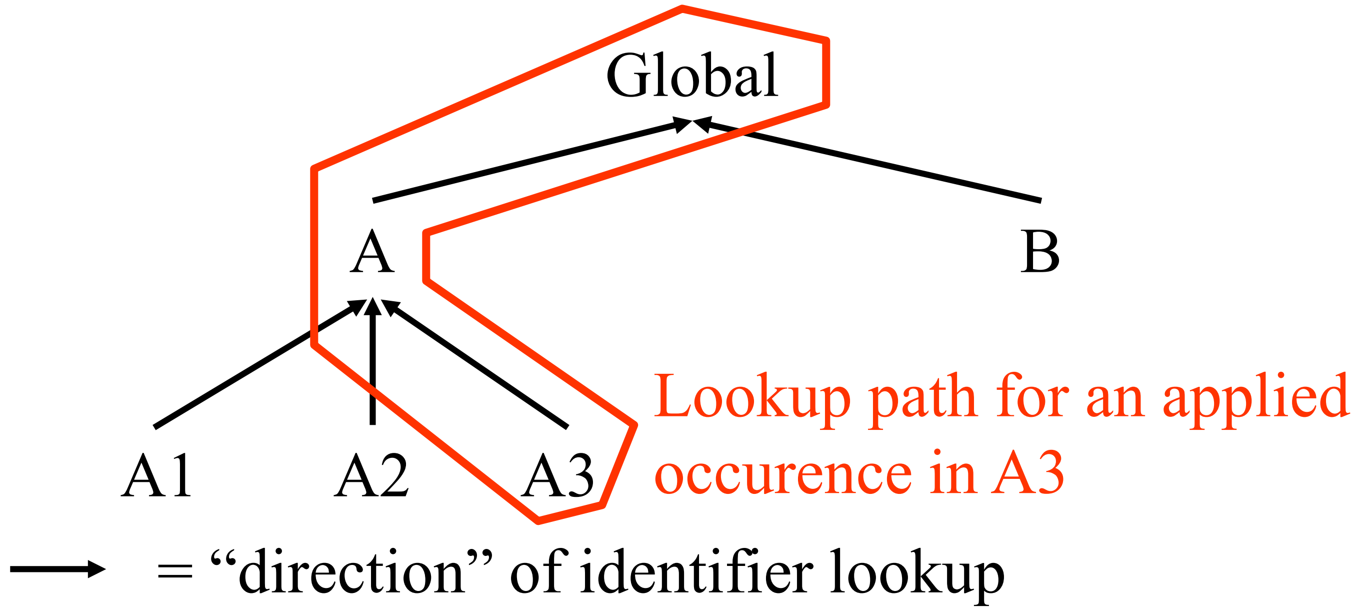
- Two common approaches to implementing block-structured symbol tables
  - A symbol table associated with each scope
  - Or a single, global table

An Individual Table for Each Scope

- Because name scope are opened and closed in a last-in first-out (LIFO) manner, a stack is an appropriate data structure for a search
  - The innermost scope appears at the top of stack
  - OpenScope(): pushes a new symbol table
  - CloseScope(): pop
- Disadvantage
  - Need to search a name in a number of symbol tables
  - Cost depending on the number of nonlocal references and the depth of nesting

Individual Table for each scope




Global
A
B
A1
A2
A3
Lookup path for an applied occurence in A3
= "direction" of identifier lookup

At any one time (in analyzing the program) only a single path on the tree is accessible.
=> We can keep a stack of identification tables, one for each "active" scope.

One Symbol Table

- All names in the same table
  - Uniquely identified by the scope name or depth
- RetrieveSymbol() need not chain through scope tables to locate a name

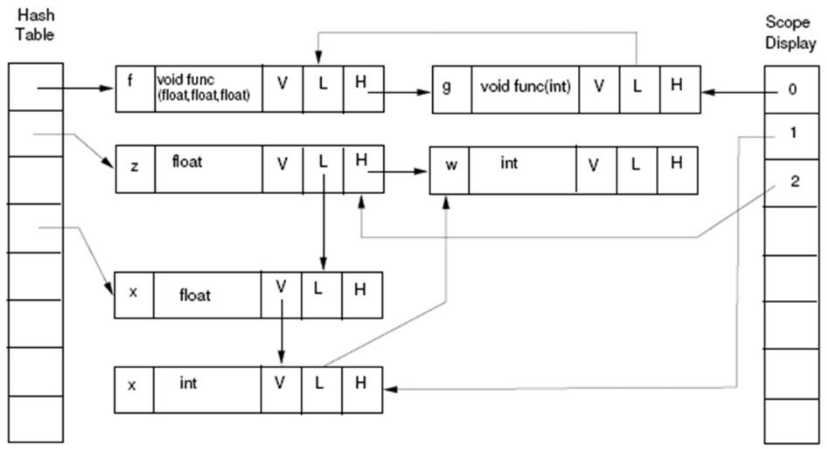


Figure 8.8: Detailed layout of the symbol table for Figure 8.1. The V, L, and H fields abbreviate the Var, Level, and Hash fields, respectively

Entering and Finding Names

- Examine the time needed to insert symbols, retrieve symbols, and maintain scopes
  - In particular, we pay attention to the cost of retrieving symbols
  - Names can be declared no more than once in each scope, but typically referenced multiple times

- Various approaches
  - Unordered list
    - Insertion: fast, Retrieval: linear scan, Impractically slow
  - Ordered list
    - Fast retrieval , but expensive insertion
  - Binary search trees
    - Insert, search: O(log n),
  - Balanced trees
    - Insert, search: O(log n) – avoids worst case for binary trees
  - Hash tables
    - Insert, search: O(1), given sufficiently large table, a good hash function and appropriate collision-handling techniques

Advanced Features

- Extensions of the simple symbol table framework to accommodate advanced features of modern programming languages
  - Name augmentation (overloading)
  - Name hiding and promotion
  - Modification of search rules

Implicit Declarations

- In some languages, the appearance of a name in a certain context serves to declare the name as well
  - E.g.: labels in C
  - In Fortran: inferred from the identifier’s first letter
  - In Ada: an index is implicitly declared to be of the same type as the range specifier
  - A new scope is opened for the loop so that the loop index cannot clash with an existing variable
    - E.g. for (int i=1; i<10; i++) { … }
  - Variables in dynamic languages like Python

Symbol Table Summary

- The symbol table organization in this chapter efficiently represents scope-declared symbols in a block-structured language
- Most languages include rules for symbol promotion to a global scope
- Issues such as inheritance, overloading, and aggregate data types must be considered
  - Records, objects and classes

Declaration Processing Fundamentals

- Attributes in the symbol table
  - Internal representations of declarations
  - Identifiers are used in many different ways in a modern programming language
    - Variables, constants, types, procedures, classes, and fields
    - Every identifier will not have the same set of attributes
  - We need a data structure to store the variety of information
    - Using a struct that contains a tag, and a union for each possible value of the tag
    - Using object-based approach, Attributes and appropriate subclasses

Type Descriptor Structures

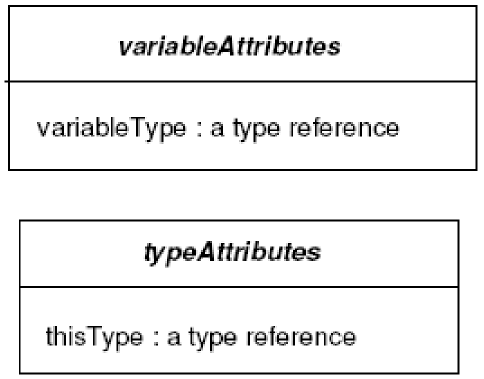


Figure 8.9: Attribute Descriptor Structures

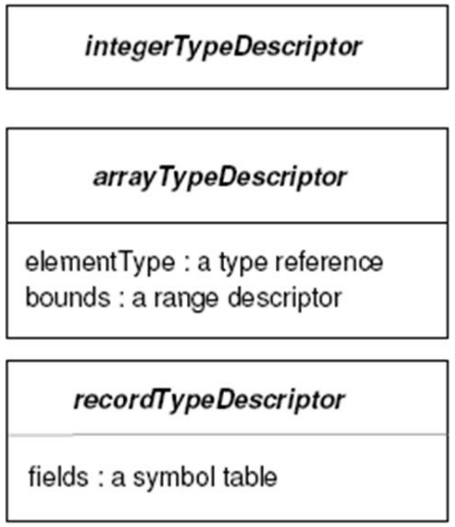
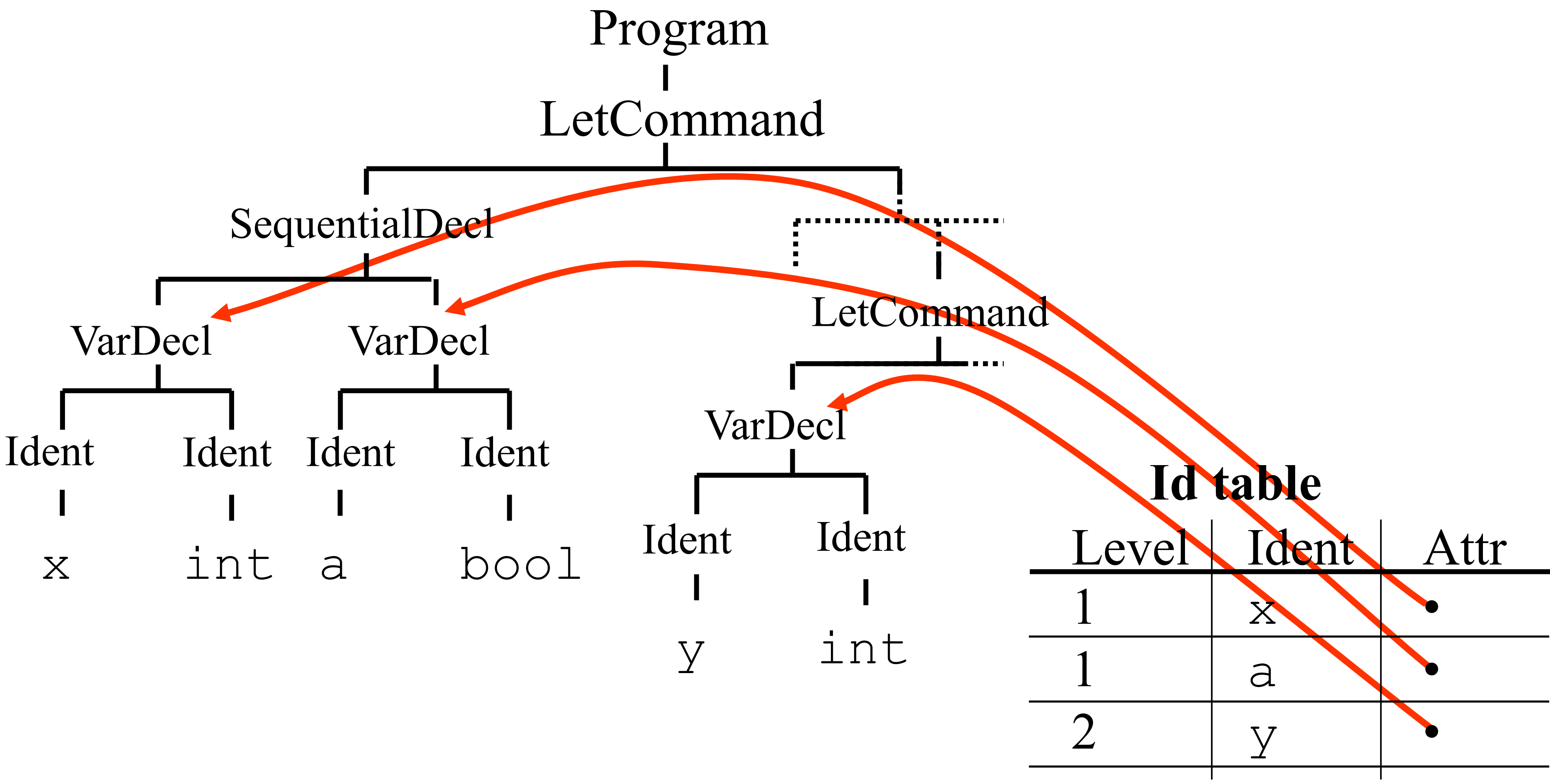


Figure 8.10: Type Descriptor Structures

Attributes as pointers to Declaration AST's



Id table

Level	Ident	Attr
1	x	•
1	a	•
2	y	•

The Standard Environment

- Most programming languages have a set of predefined functions, operators etc.
- We call this the **standard environment**

At the start of identification the ID table is not empty but... needs to be initialized with entries representing the standard environment.

Scope for Standard Environment

Should the scope level for the standard environment be the same as the globals (level 1) or outside the globals (level 0)?

– C: level 1

– Mini Triangle: level 0

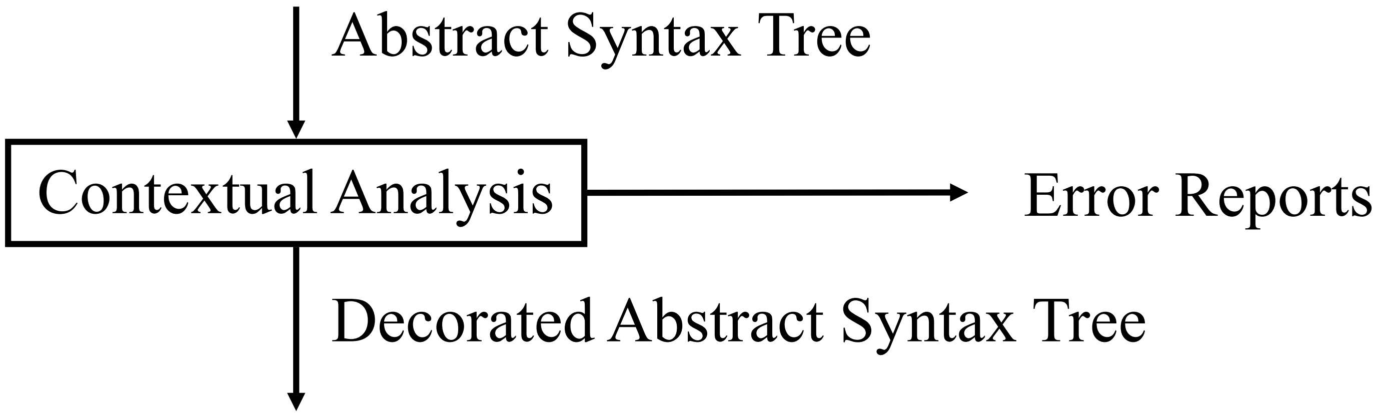
- Consequence:

```
let
  var false : Integer
in
  begin
    false := 3;
    putint ( false )
  end
```

  is a perfectly correct Mini Triangle program

- Similar with Integer or putint. . .

Contextual Analysis -> Decorated AST



Contextual analysis:

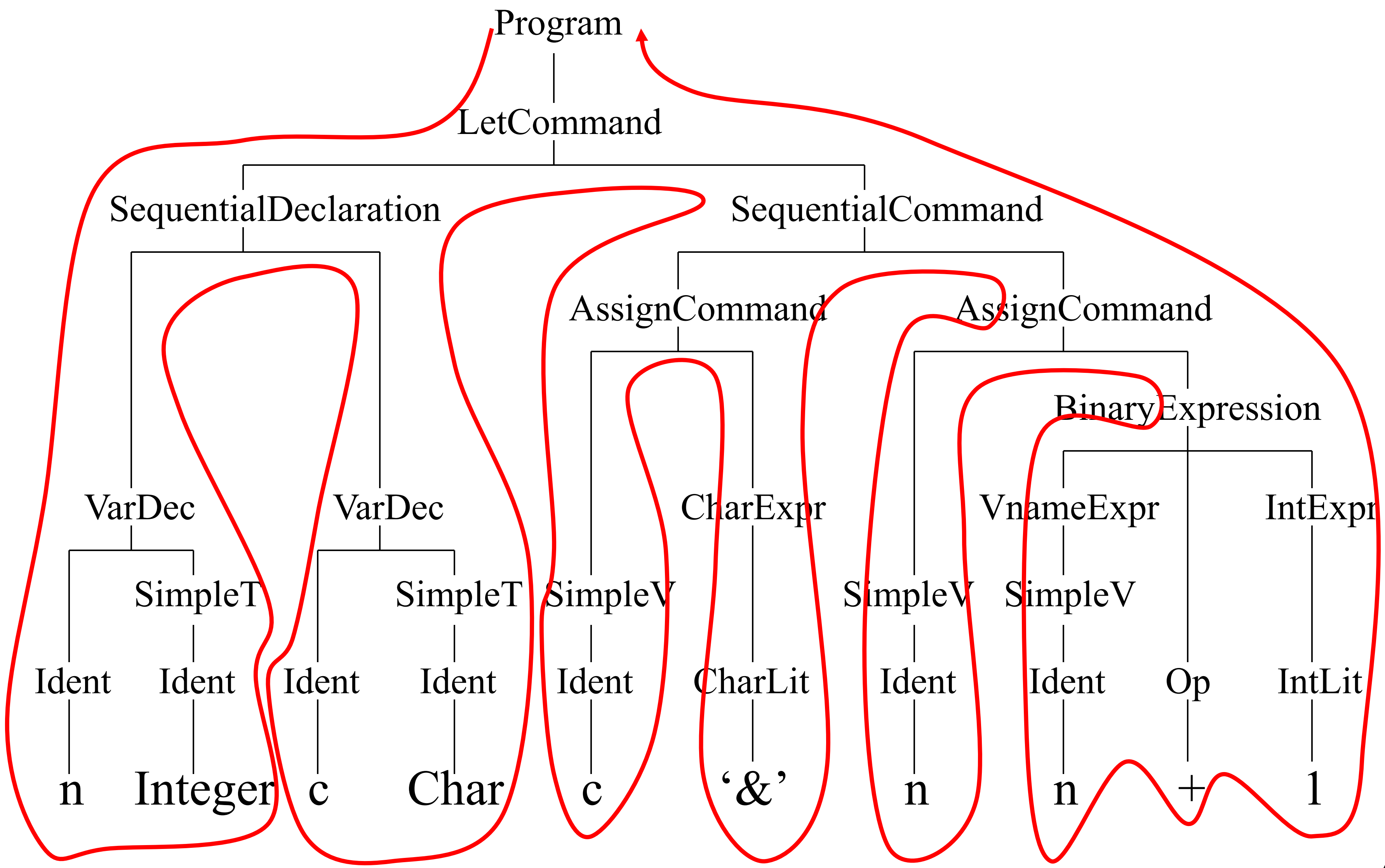
- Scope checking: verify that all applied occurrences of identifiers are declared
- Type checking: verify that all operations in the program are used according to their type rules.

Annotate AST:

- Applied identifier occurrences => declaration
- Expressions => Type

Contextual Analysis

Identification and type checking are combined into a depth-first traversal of the AST.



Implementing Tree Traversal

- “Traditional” OO approach
- Visitor approach
  - GOF
  - Using static overloading
  - Reflective
  - (dynamic)
  - (SableCC style)
- “Functional” approach
- Active patterns in Scala (or F#)
- (Aspect oriented approach)

What can you do in your project now?

- Start designing and defining:
  - Scope rules for your language
    - Informal (in structured English)
    - Formally (when you have read chapter 6 in Trans. & Trees)
- Start thinking about designing and defining
  - the type system for your language
    - Informal (in structured English)
    - Formally (when you have read chapter 13 in Trans. & Trees)
- Start thinking about implementing
  - Symbol table(s)
  - Scope cheking
  - (simple) type cheking