Chapter 8 Symbol Table

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: April 1, 2010

current version: June 16, 2022

Chapter outline: Symbol tables

1. Constructing a symbol table
2. Block-structured languages and scopes
3. Basic implementation techniques
4. Advanced features
5. Declaration processing fundamentals
6. Variable and type declarations
7. Class and method declaration
8. An introduction to type checking
9. Summary

Prerequisites: (you may check Wikipedia for a quick review)

1. the visitor pattern
2. dynamic dispatch in object-oriented languages
3. double dispatch

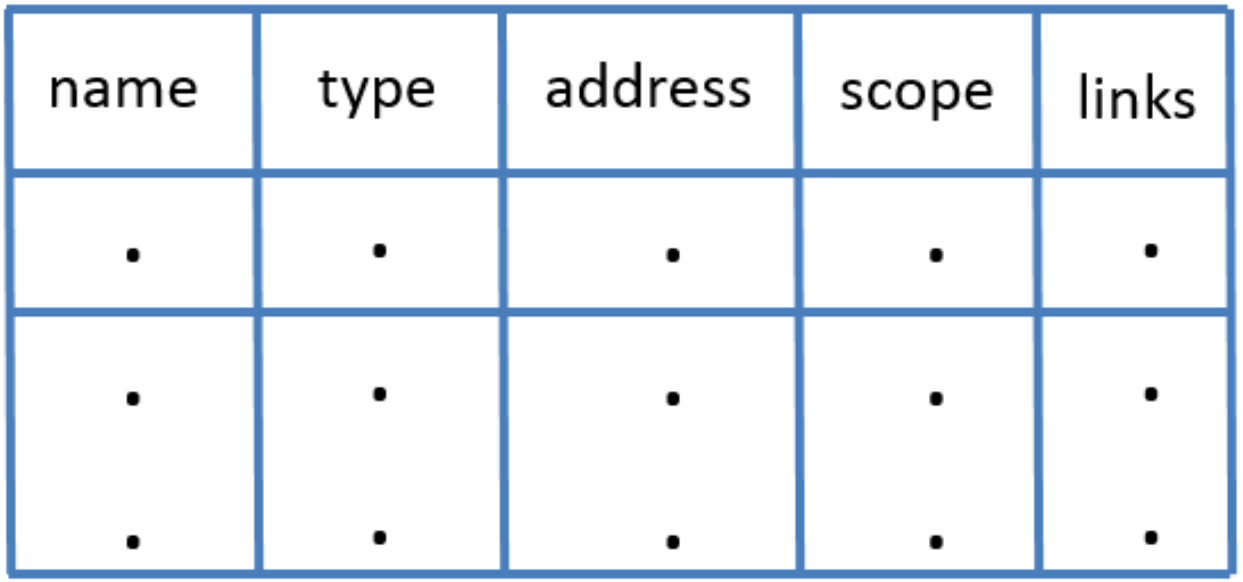
Summary of a compiler:

```
main() {
  astNodeType *astRoot;
  symTableType *symtab;

  astRoot = parse(inputfile);
  buildSymbolTable(astRoot, symtab);
  ControlFlowAnalysis(astRoot, symtab);
  DataFlowAnalysis(astRoot, symtab);
  TranslateToSSA(astRoot, symtab);
  DeadCodeElimination(astRoot, symtab);  // 1st run
  CommonSubExpression(astRoot, symtab);
  DeadCodeElimination(astRoot, symtab);  // 2nd run
  TranslateToMachineIR(astRoot, symtab, machineIR);
  RegisterAllocation(machineIR, symtab);
  generateCode(machineIR, symtab, outputfile);
}
```

A symbol table is a database of the declared *names* in a program. There are various pieces of information associated with a name, such as the class (a type name, variable name, procedure name, constant name, class name, etc.), type (`int`, `float`, `boolean`, etc.), scope, memory address, access restrictions, etc.

We collect and generate these properties when a name is declared and consult the properties when a name is used.



Symbol tables are also used in linkers and loaders, though they are simpler there. For example, the Java bytecode file includes extensive symbol table information (so that a decompiler can almost recover the original program source).

The `strip` command in Unix systems removes the symbol table from a binary file.

Debuggers and profilers also need information in the symbol table.

Two issues:

interface: how to use symbol tables

implementation: how to implement them

§8.1 Constructing a Symbol Table

Assume an AST is already built. The compiler will walk through the AST and do two things:

1. process declarations
2. connect a symbol reference with its declaration (because of overloading)

Block structures with nested scopes are common in popular programming languages. Symbol table management should take this into account.

Figure 8.1 shows an example. The compiler walks through the AST in Figure 8.1 (b). The symbols `f`, `g`, `w`, `x`, `x`, `z` are entered into the symbol table in Figure 8.1 (c). The usages of symbols in the AST are replaced by the references in Figure 8.1 (d).

A symbol table contains the names as well as various properties (such as types, addresses, sizes, scopes, etc.) of the symbols.

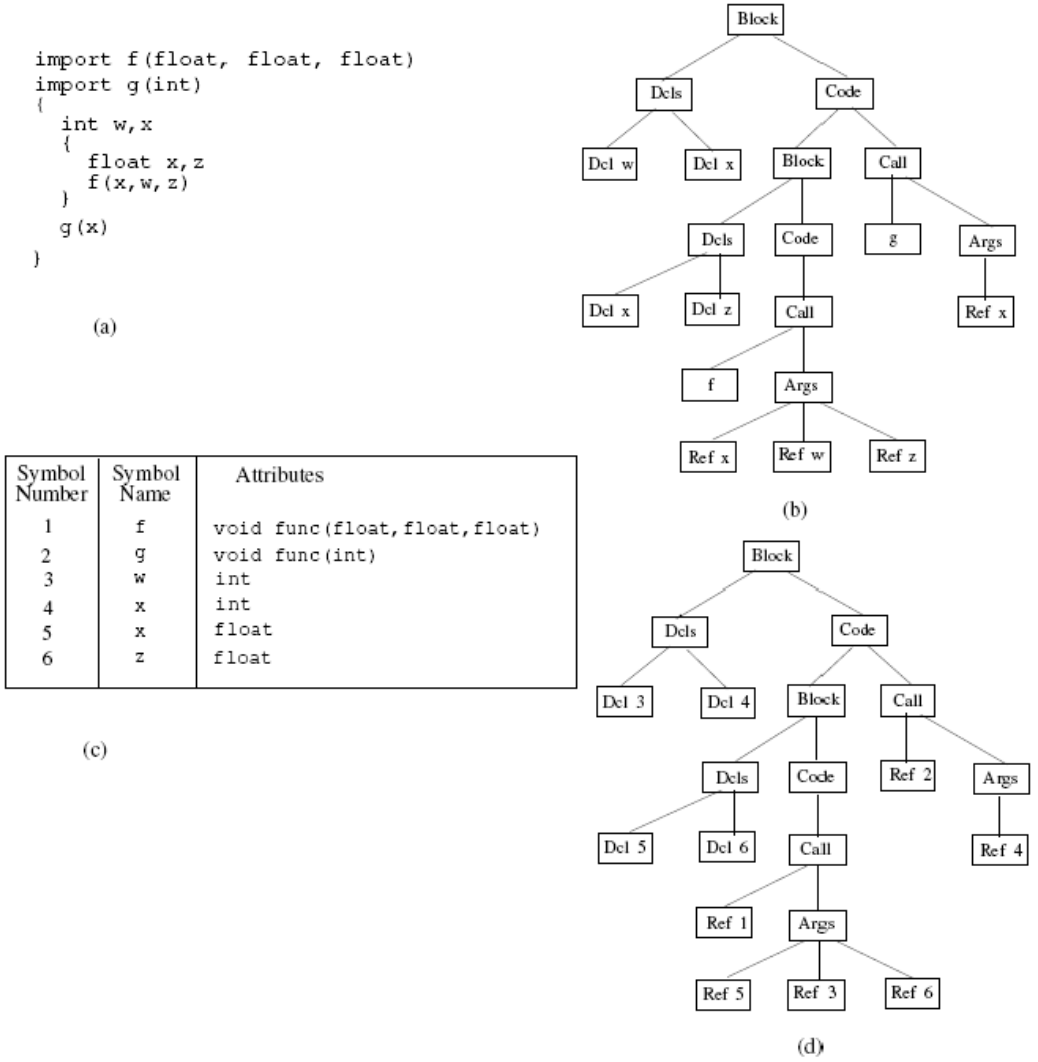


Figure 8.1: Symbol table processing for a block-structured program

§8.1.1 Static scoping

Programming languages use scopes to confine a symbol in a region. Scopes may be nested.[a] In C or Ada, we use

```
{ ......  }
begin ......  end
```

to mark a scope. Variables, types, and sometimes, functions may be declared in a scope. A name usually can be declared only once in a scope (in order to avoid confusion, but overloading may be allowed). Code in a block usually refers to names declared in that scope. However, it is possible to access names declared in other scopes, though sometimes at extra run-time cost.

C is a *flat* language in that it does not allow functions to be declared in another function. (ML allows this.) In C, all names declared in a function may be moved to the global scope with proper renaming.

[a] The simplest scope rule is to require that no variable name can be declared more than once. SSA form has this flavor.

§8.1.2 A symbol table interface

A symbol table module provides five basic functions:

- `OpenScope()`: open a new scope.
- `CloseScope()`: close the current scope and revert to a previous scope.
- `EnterSymbol(name, type)`: enter the newly declared `name` into the symbol table for the current scope with the `type`, *etc.*, information.
- `RetriveSymbol(name)`: find the entry for `name` in the symbol table.
- `DeclaredLocally(name)` determine if `name` is declared in the current scope.

Figure 8.2 is the code for building a symbol table. See also the AST in Figure 8.1. The following code performs a combination of pre-order and post-order traversal of AST.

```
procedure BuildSymboleTable()
   call ProcessNode(ASTroot)
end
procedure ProcessNode(node)
   switch( kind(node) )
   case Block: call symtab.OpenScope()
   case Dcl:   call symtab.EnterSymbol(node.name, node.type)
   case Ref:   sym := symtab.RetrieveSymbol(node.name)
               if sym == null
               then error("undeclared symbol", sym)
   end switch
   foreach c in node.GetChildren() do call ProcessNode(c)
   if ( kind(node) == Block ) then call symtab.CloseScope()
end
```

The procedure `ProcessNode(node)` performs a top-down, pre-order and post-order traversal of the tree rooted at `node`.

```
procedure      S      T    ()
   call        N    (ASTroot)
end

procedure          N    (node)
   switch (      (node))
      case Block
         call symtab.      S     ()                              (1)
      case Dcl
         call symtab.      S     (node.name, node.type)
      case Ref
         sym ← symtab.           S     (node.name)
         if sym = null
         then call       ("Undeclared symbol : ", sym)
   foreach c ∈ node.    C          () do call          N    (c)
   if      (node) = Block
   then
      call symtab.      S    ()                                 (2)
end
```

Figure 8.2: Building the symbol table

§8.2 Block-structured languages and scopes

We need to define the visibility rules of names according to the scopes.

A *scope* is a contiguous region in the program. Scopes are *properly nested* in the sense that any two scopes are either disjoint or one is contained in another. At a particular point p in the program text, the innermost scope containing p is called the *current scope*. All scopes that contain p are *open scopes*. All other scopes are *closed scopes*.

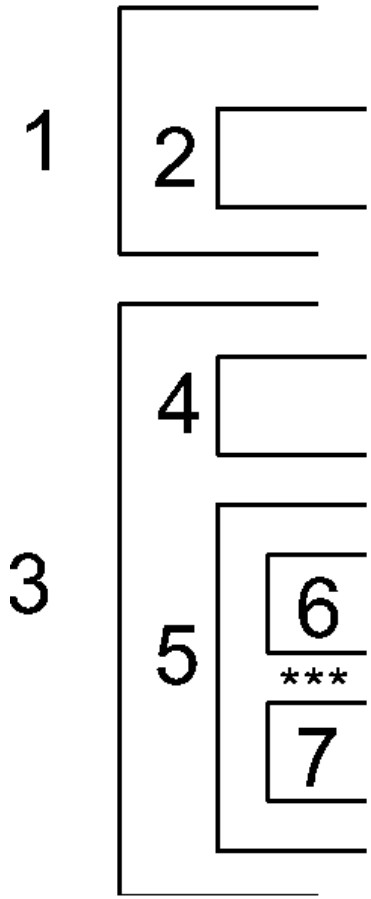


Figure 1: Nested scopes, current scope, open scopes, closed scopes.

New names can be declared only in the current scope.

There usually cannot be two distinct variables with the same name in a scope.

Only the names declared in the current or open scopes can be used at the program point p.

If a name is declared in more than one current/open scopes, the one declared in the innermost scope is referenced.

```
begin integer x;
    begin real x;
        begin
            y := x;  // which x?
        end
    end
end
```

When a scope is closed, we need to locate (and erase) all the names

declared in that scope.

```
begin integer x, y;
    begin
        real x;
        integer y;
        boolean z;
        . . .
    end
    // inner x, y, z are not accessible here.
    y := x;    // outer x and y
end
```

Languages also offer additional provisions. In C, there are `extern` variables, which are placed in the global scope (but with restricted visibility).

In Java, every class can reference any class's `public static` fields with fully qualified names.

C and C++ offer a *compilation-unit* scope, where names declared

outside of all methods, static or not, are available within the compilation unit's methods.

In C, all function definitions are available in the global scope unless it is a `static` definition.

In C++ and Java, names declared within a class are available to all methods in the class.

In C++ and Java, a class's `protected` fields and methods are available in all subclasses of that class.

Example. C allows global, static, local and block scopes.

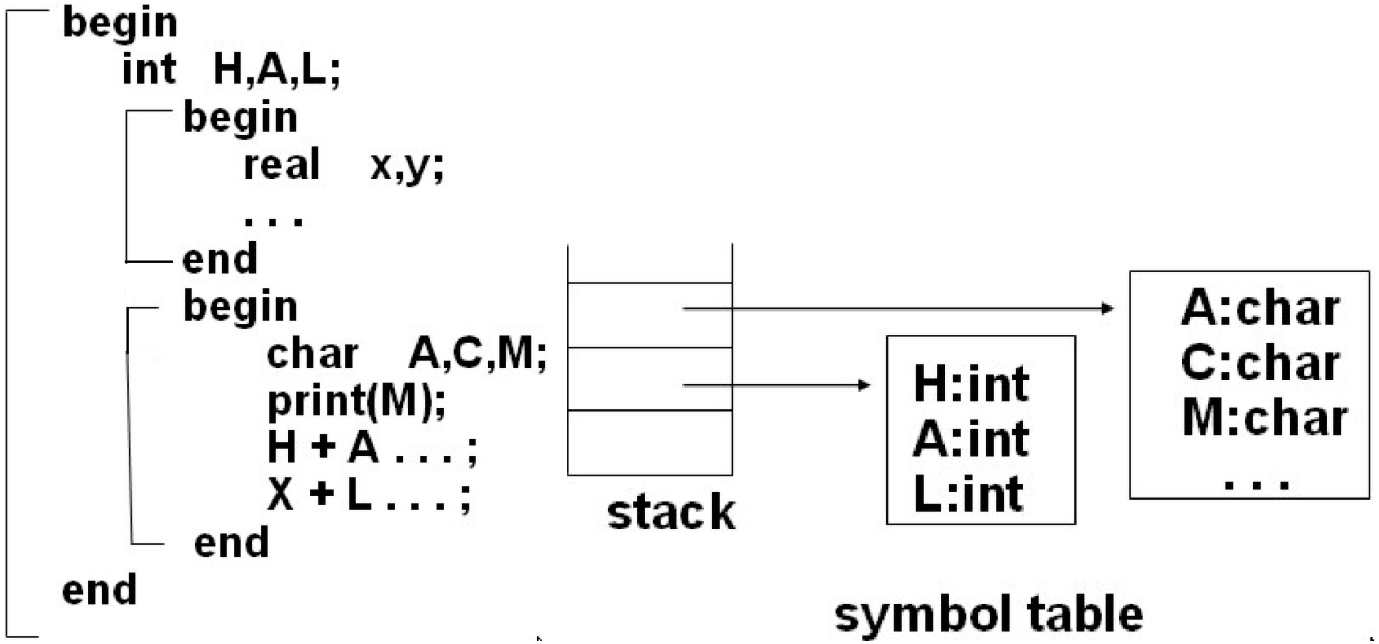
```
int i;
static int f(int y) {
    static int z;
    float w;
    if (y > 3) {
        boolean flag;
        . . .
    } else {
        char q;
        . . .
    }
}
```

§8.2.2 One symbol table or many?

For block structures, we can use either a big table for the whole program or one small table for each scope. Between these two extremes, there are intermediate choices—we may allocate a symbol table for each class, package, or module, etc.

A. Many small tables

1. Create one symbol table per scope.
2. Use a stack of tables.
3. The symbol table for the current scope is on stack top.
4. The symbol tables for other enclosing scopes are placed under the current one.
5. Push a new table when a new scope is entered.
6. Pop a symbol table when a scope is closed.



To search for a name, we check the symbol tables on the stack from top to bottom.

(-) We may need to search multiple tables. For example, a global name is defined in the bottom-most symbol table.

(-) Space may be wasted if a fixed-sized hash table is used to implement symbol tables.

- hash table too big – waste memory space

- hash table too small – collisions

In practical programs, most references are for either local variables or global variables.

Example. Figure 8.3 is the stack of symbol tables for Figure 8.1.

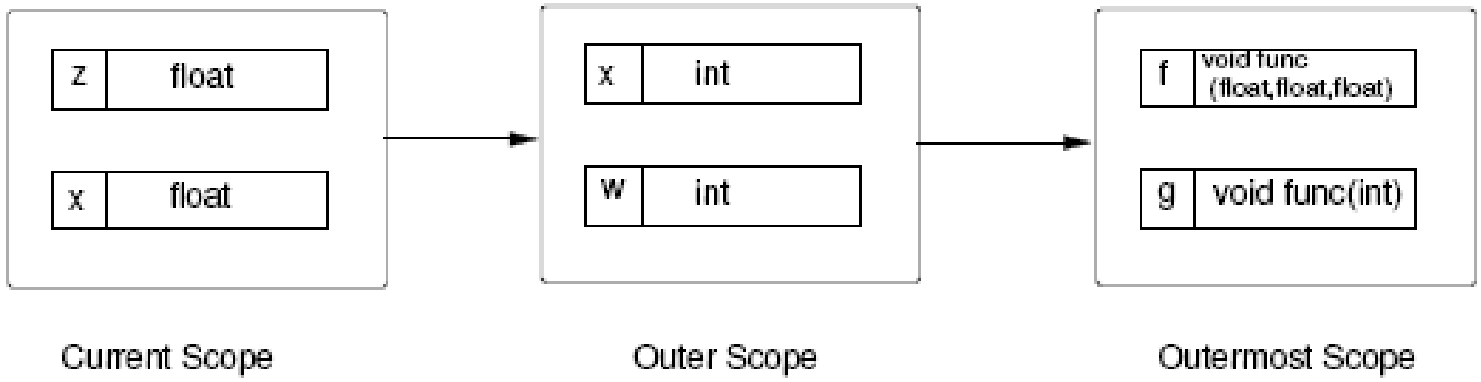
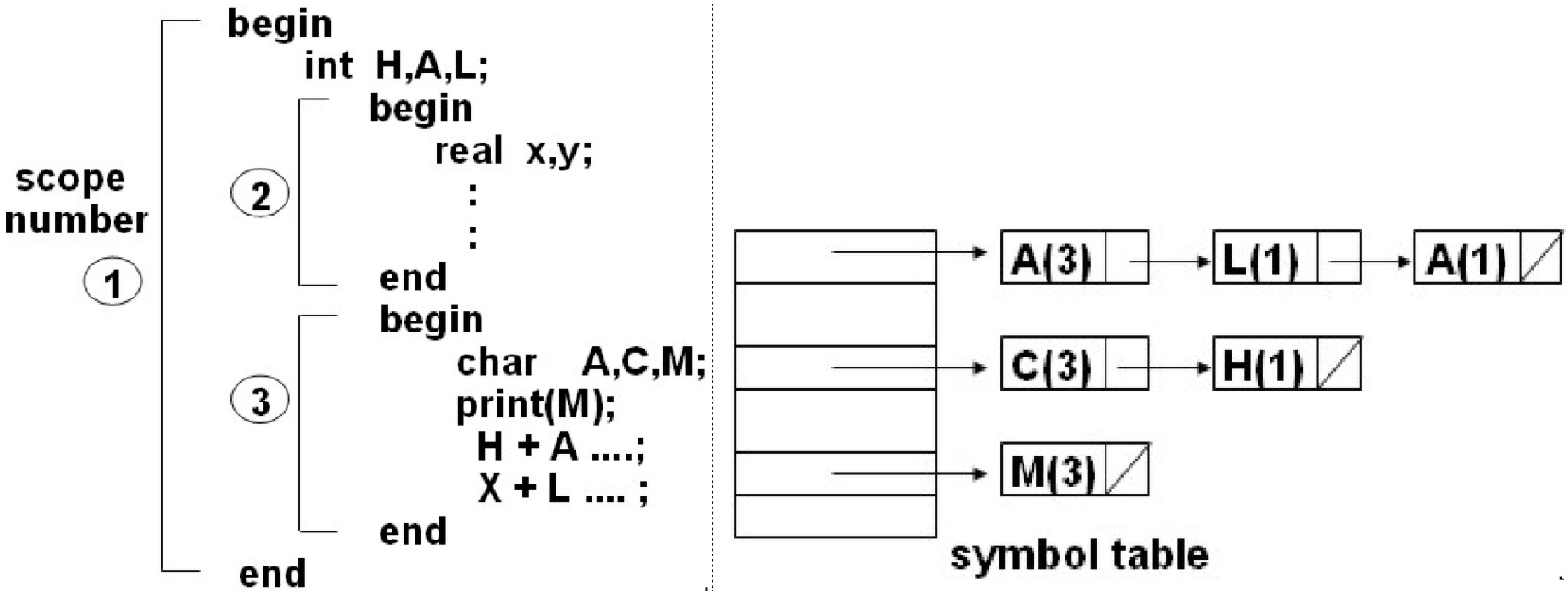


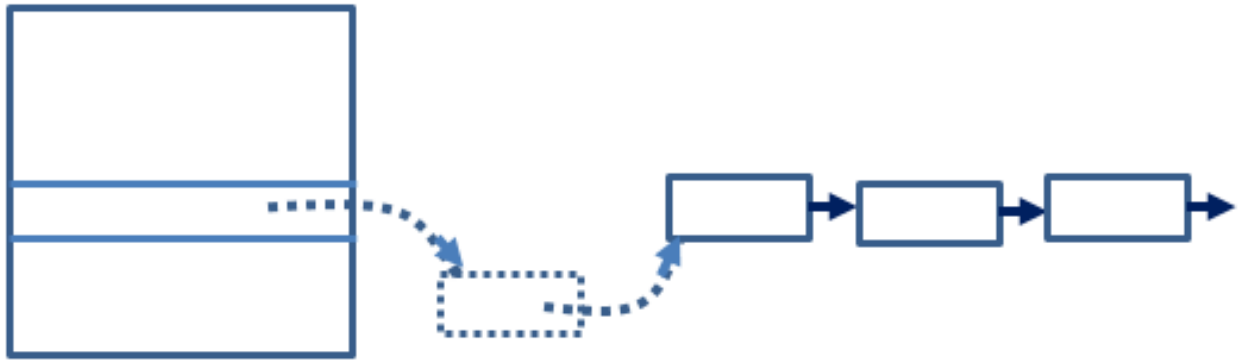
Figure 8.3: A stack of symbol tables, one per scope

B. One big table

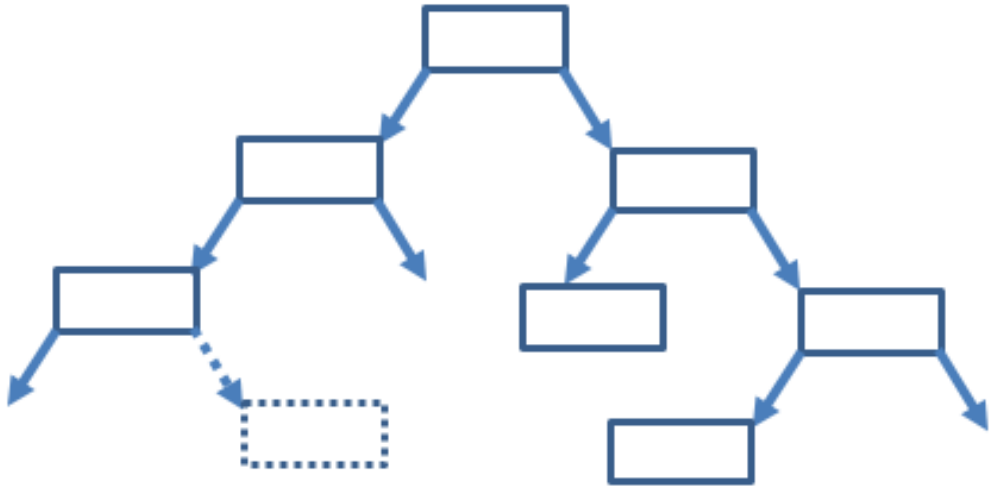
1. All names are in the same table.
2. What if the same name is declared several times, each in a different scope?
   - Each scope is given a scope number. Each name is annotated with the scope number of the scope it is declared.
   - The pair <name, scope number> should be unique in the table.
   - To search a name is easy. New names are placed at the front of collision chains (if hash table + collision chains are used).
   - To close a scope, we need to (locate and) remove all entries defined in that scope. (We need to examine *every* collision chain.)


scope number
1
2
3
begin
int H,A,L;
begin
real x,y;
:
:
end
begin
char A,C,M;
print(M);
H + A ....;
X + L .... ;
end
end
A(3)
L(1)
A(1)
C(3)
H(1)
M(3)
symbol table

Addition to a collision chain is placed in the front.



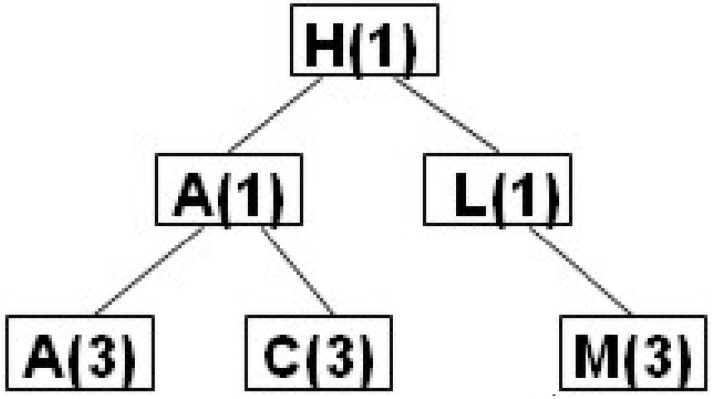
Addition to a binary tree is placed at the bottom.



One global table cannot be implemented with binary trees easily.

1. It is not easy to delete all entries declared in a scope when the scope is closed.
2. To find a name, we need to search for the last entry (rather than the first entry).

Consider the case when the identifier `A` is searched for.



Comparisons:

many small tables	one global table
simpler	more complicated
slower search	faster search
less efficient (with a hash table)	efficient storage use (for hash tables) Need space for scope numbers.
hash tables or search trees	hash tables
Good for keeping symbol tables of closed scopes (e.g. in multi-pass compilers)	Good for 1-pass compilers Entries may be discarded after the scope is closed.

One big table is more complicated.

- All names are in the same table.
- We need to attach additional flags on the entries to indicate the scopes of the names.
- Searching is fast.
- Due to space efficiency, one big table can be implemented with hashing.
- It is not easy to locate all the names declared in a scope when the scope is closed.

Many small tables are simpler.

- We need a stack of symbol tables, one for each open scope. When a scope is open, push a new empty symbol onto the stack. When a scope is closed, pop one symbol table from the stack.
- Searching a name is done by searching the symbol tables in the stack.
- Searching is slow.
- Due to space efficiency, many small tables cannot be implemented with hashing.
- It is easy to locate all the names declared in a scope when the scope is closed.

```
procedure BuildSymbolTable
  call ProcessNode(ASTroot)
end

procedure ProcessNode(node)
  switch(kind)
    case Block: call symtab.OpenScope()
    case Dcl:   call symtab.EnterSymbol(node.name, node.type)
    case Ref:   sym := symtab.RetrieveSymbol(node.name)
                if sym == null then Error("undeclared symbol")
  end case
  foreach c in node.GetChildren() do call ProcessNode(c)
  if kind(node) == Block
  then call symtab.CloseScope()
end
```

Figure 8.2 Building the symbol table

§8.3 Basic implementation techniques

The basic implementation techniques of symbol tables include ordered lists, binary search trees, AVL trees, and hash tables. We need to consider the space requirements and the time it takes to search a name in the symbol table.

1. unordered list (linked list/array): if there are only a few names.
2. ordered list: binary search on arrays
   (-) expensive insertion
   (+) good for a fixed set of names (e.g. reserved words, assembly opcodes, etc.)
3. binary search tree: On average, searching takes $O(\log n)$ time. However, names in programs are not chosen randomly.
4. hash table: most common
   (+) constant-time search

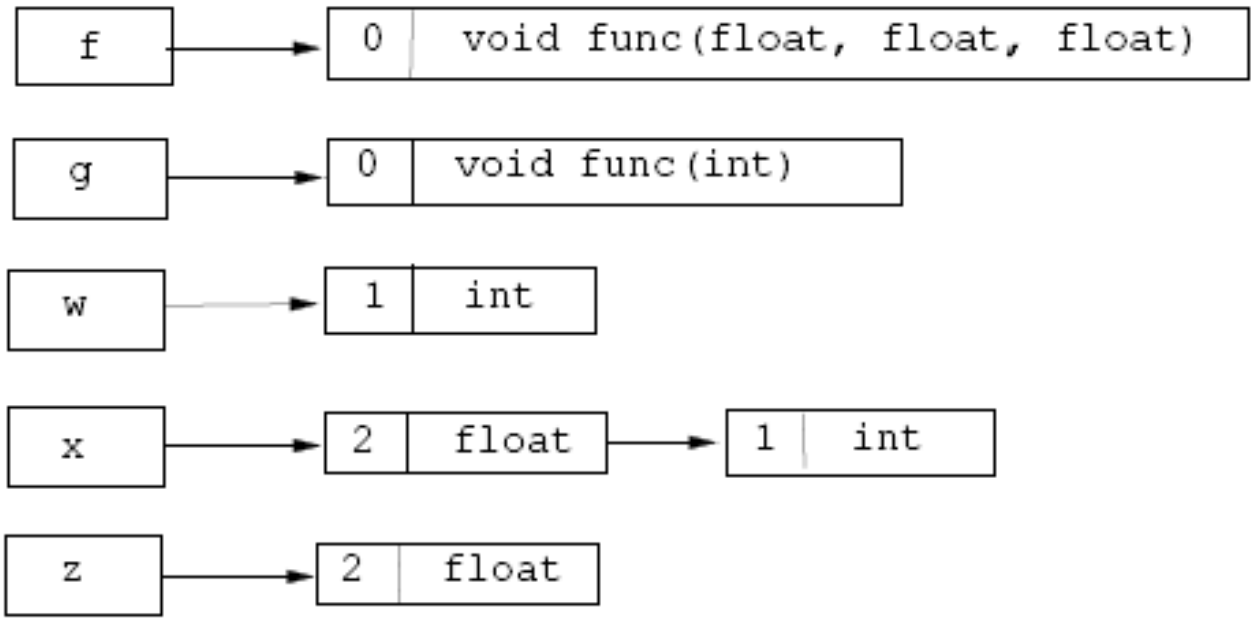


Figure 8.4: An ordered list of symbol stacks

Binary Search Trees

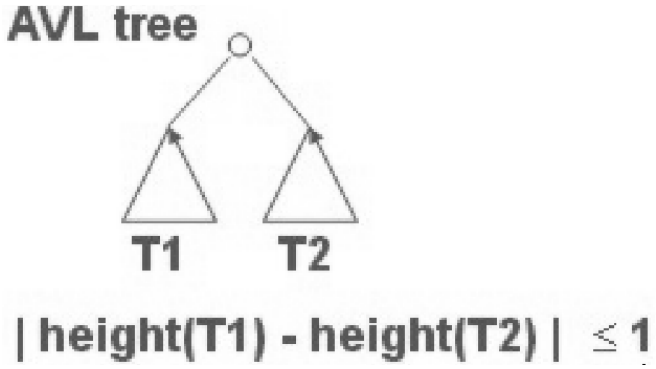
For balanced tree, search takes $O(\log n)$ time.

For random input, search takes $O(\log n)$ time. However, average search time is 38% greater than that for a balanced tree.

In the worst case, search takes $O(n)$ time.

Ex. The input `(A B C D E)` results in a linear list.
The input `(A E B D C)` also results in a linear list.

Solution. Keep the tree approximately balanced. Insertion/deletion may need to move some subtrees to keep the tree approximately balanced.



(+) space = $O(n)$

Comparison. Hash table needs a fixed size regardless of the number of entries.

Application: Sometimes, we may use multiple symbol tables. Binary search tree is better than hashing in this case.

Hash Tables

$hash : name \rightarrow address$

1. Hash is easy to compute.
2. Hash is uniform and randomizing.

Example. $(C_1 + C_2 + \ldots + C_n) \bmod m$
$(C_1 * C_2 * \ldots * C_n) \bmod m$
$(C_1 + C_n) \bmod m$
$(C_1 \oplus C_2 \oplus \ldots \oplus C_n) \bmod m$

Conflicts resolution

linear resolution Try $h(n), h(n)+1, h(n)+2, \ldots$.

Problematic if the hash table did not reserve enough empty slots.

add-the-hash rehash Try

$$h(n), (2 * h(n)) \bmod m, (3 * h(n)) \bmod m, \ldots$$

where m must be a prime number.

quadratic rehash Try $h(n), (h(n)+1) \bmod m, (h(n)+4) \bmod m, \ldots$.

chaining

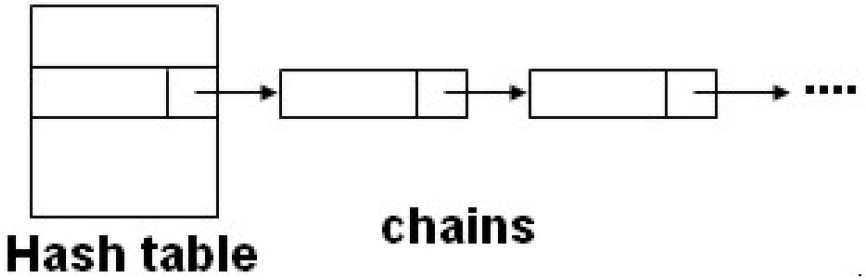


Figure 2: Chaining

Advantages of chaining:

1. There is less space overhead of the hash table.
2. It does not fail catastrophically when the hash table is almost full.

A disadvantage of chaining is that we need to traverse down the chain, comparing the incoming identifier with those in the symbol table. As the lengths of identifiers get longer in modern programming languages (consider the mangled names in C++), it takes a lot of time to search for a name.

The chains may be organized as search trees, rather than linear lists.

More importantly, we can easily remove all names defined in a scope when the scope is closed easily.

```
{ var  a;

        {  var a, b, c;
           ...
           ...
         }

}
```

compiler, hash table: case study:

"...Libraries have unresolved symbols when they use functions or global variables from other libraries. these symbols are resolved during dynamic linking. The dynamic linker locates every unresolved symbol each library needs by searching the symbol table of each loaded library in order from first loaded to last loaded. This can become quite time consuming, especially in cases such as GNOME or OpenOffice.org where 50 ˜ 150 libraries are loaded and up to 200,000 symbols must be resolved. ..."

§8.3.2 Name Space

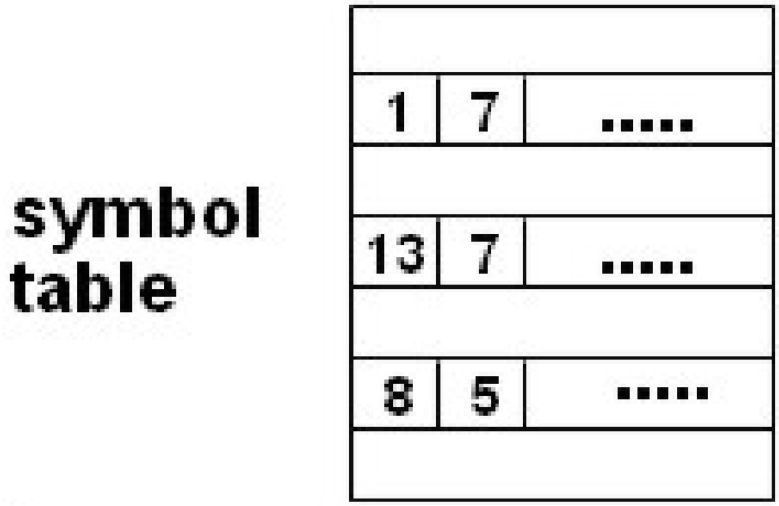
Should we store the identifiers in the symbol table?

Name lengths differ significantly, e.g., `i`, `account_receivable`, `a_very_very_long_name`, `VisibilityPartiallyObscured`.

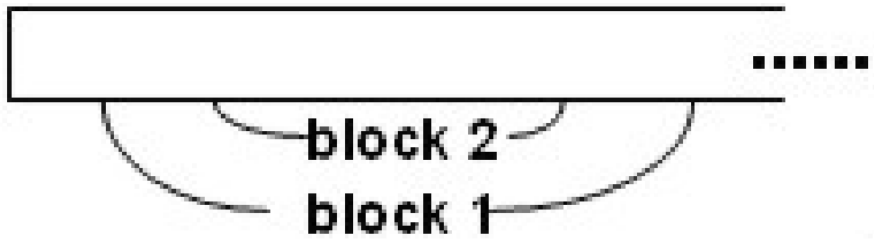
Space is wasted if space of max size is reserved.

Solution : store the identifiers in the *name space*, and store the index and length of an identifier in the symbol table.

1							8					13							
a	c	c	o	u	n	t	t	e	m	p	1	b	a	l	a	n	c	e	...



Usually, the symbol table manages the name space. Names in the string space may be re-used. Individual scopes may use different name spaces. However, for block-structured languages, a single string space is good for re-claiming memory (when the name becomes invisible). Sometimes we still need to keep the name even if it becomes invisible to the compiler.



Example.

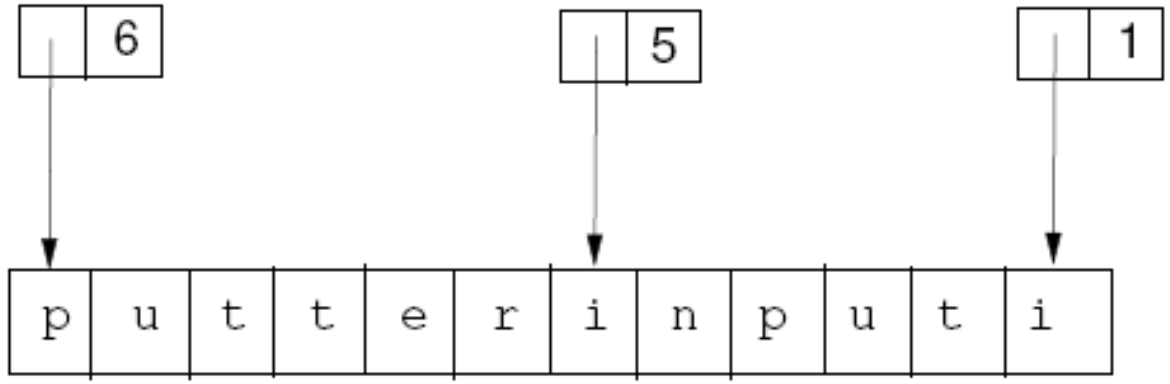
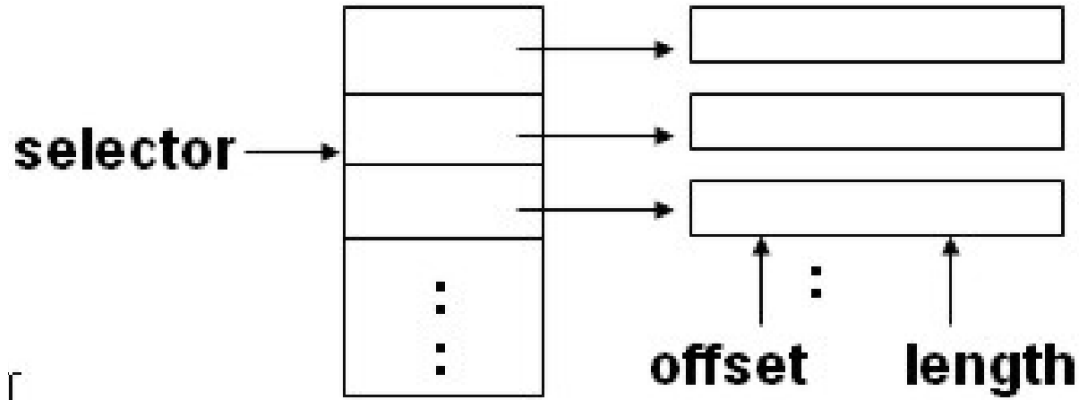


Figure 8.5: Name space for symbols *putter*, *input*, and *i*

How large is the name space?

- too big: waste space
- too small: run out of space

We may use a *segmented name space* and dynamically allocate one segment at a time.



We use a tripe *(selector, offset, length)* to identify a name.

§8.3.3 An efficient symbol table implementation

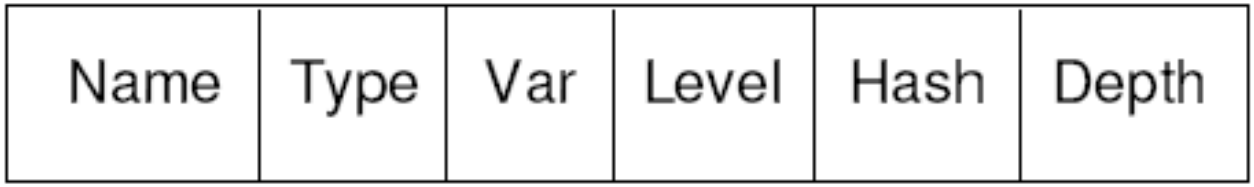


Figure 8.6: A symbol table entry

Figure 8.6 shows an entry of the symbol table.

The `Hash` field is used to thread names with the *same hash value*. This facilitates search and deletion of names.

The `Var` field is a pointer that references to the *same name* in the next outer scope. When a name's scope is closed, the name pointed to by the `Var` field becomes active again. The `Var` fields of the same name essentially build a stack of scopes from inside out.

The `Level` field threads all names declared in the *same scope*. This

facilitates deletion of names when the current scope is closed.

The `Depth` field records the nesting depth of a name's declaring scope.

Figure 8.8 shows the symbol table implementation. There are 6 names in the table. There are two index structures: the hash table on the left and the scope display on the right.

The hash table allows efficient insertion and lookup of names. The scope display is a stack of scopes. All names declared in the same scope are threaded with the `Level` field.

Figure 8.7 is the code for building the symbol table.

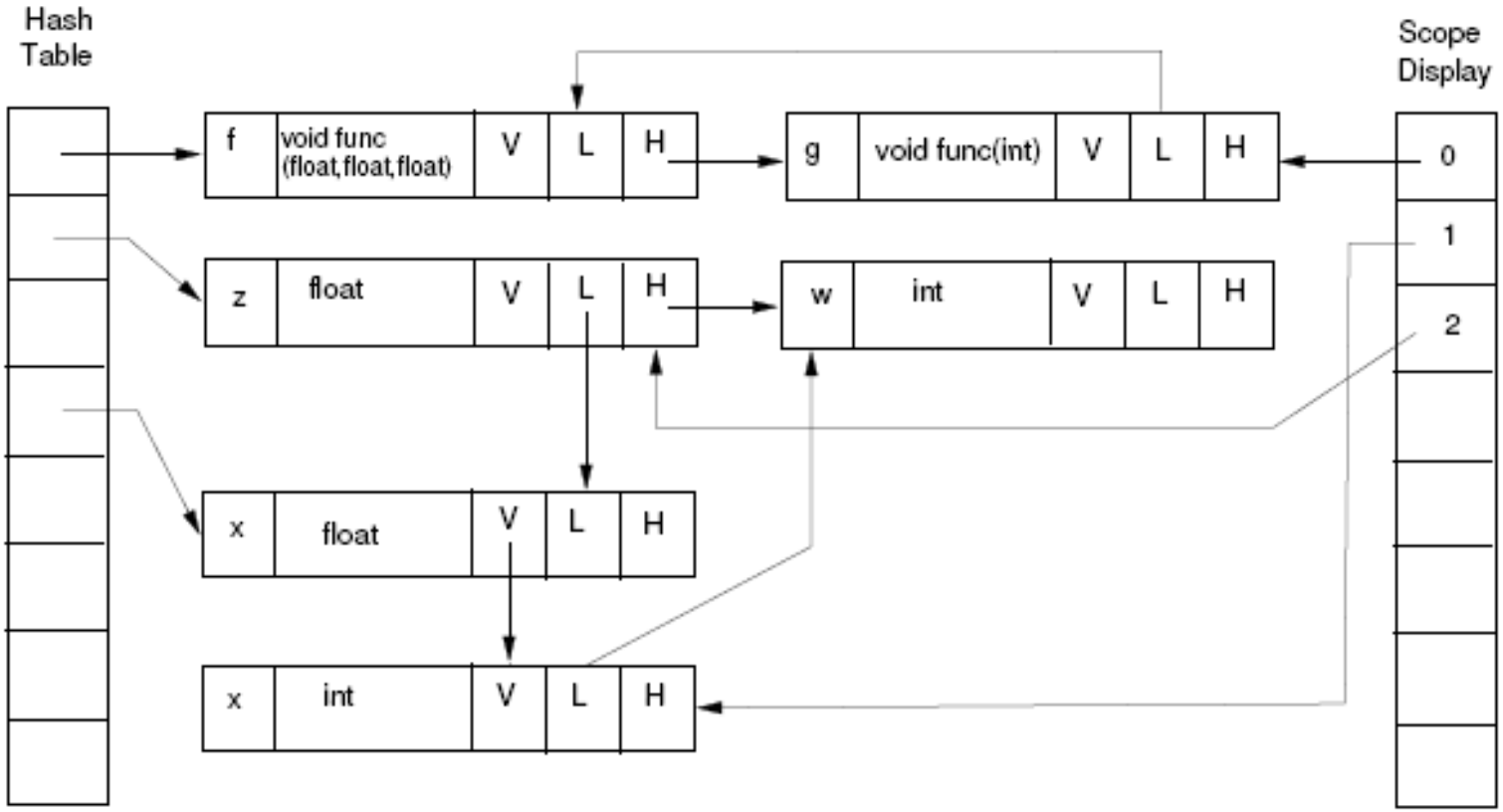


Figure 8.8: Detailed layout of the symbol table for Figure 8.1. The V, L, and H fields abbreviate the Var, Level, and Hash fields, respectively

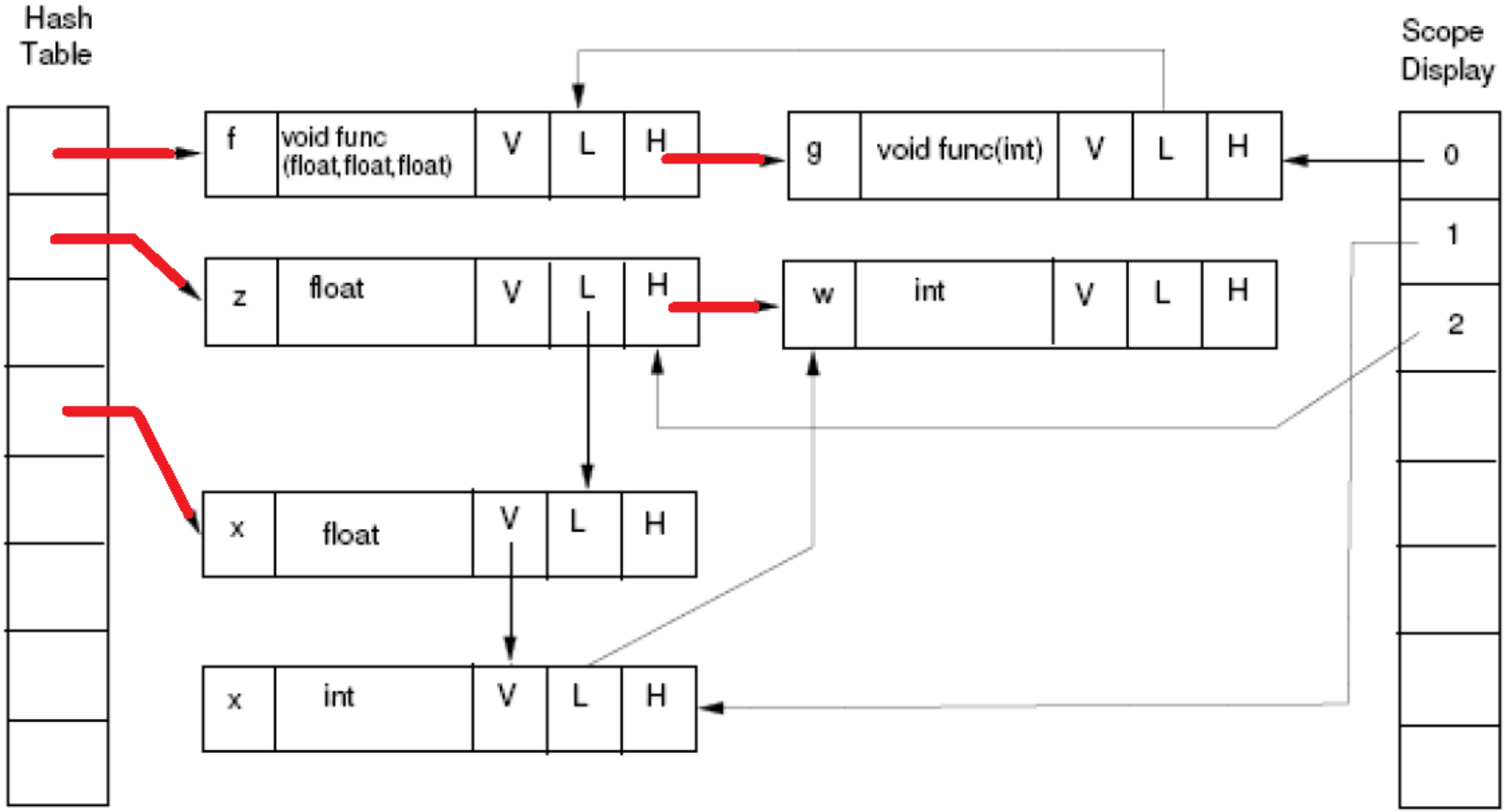


Figure 8.8: Detailed layout of the symbol table for Figure 8.1. The V, L, and H fields abbreviate the Var, Level, and Hash fields, respectively

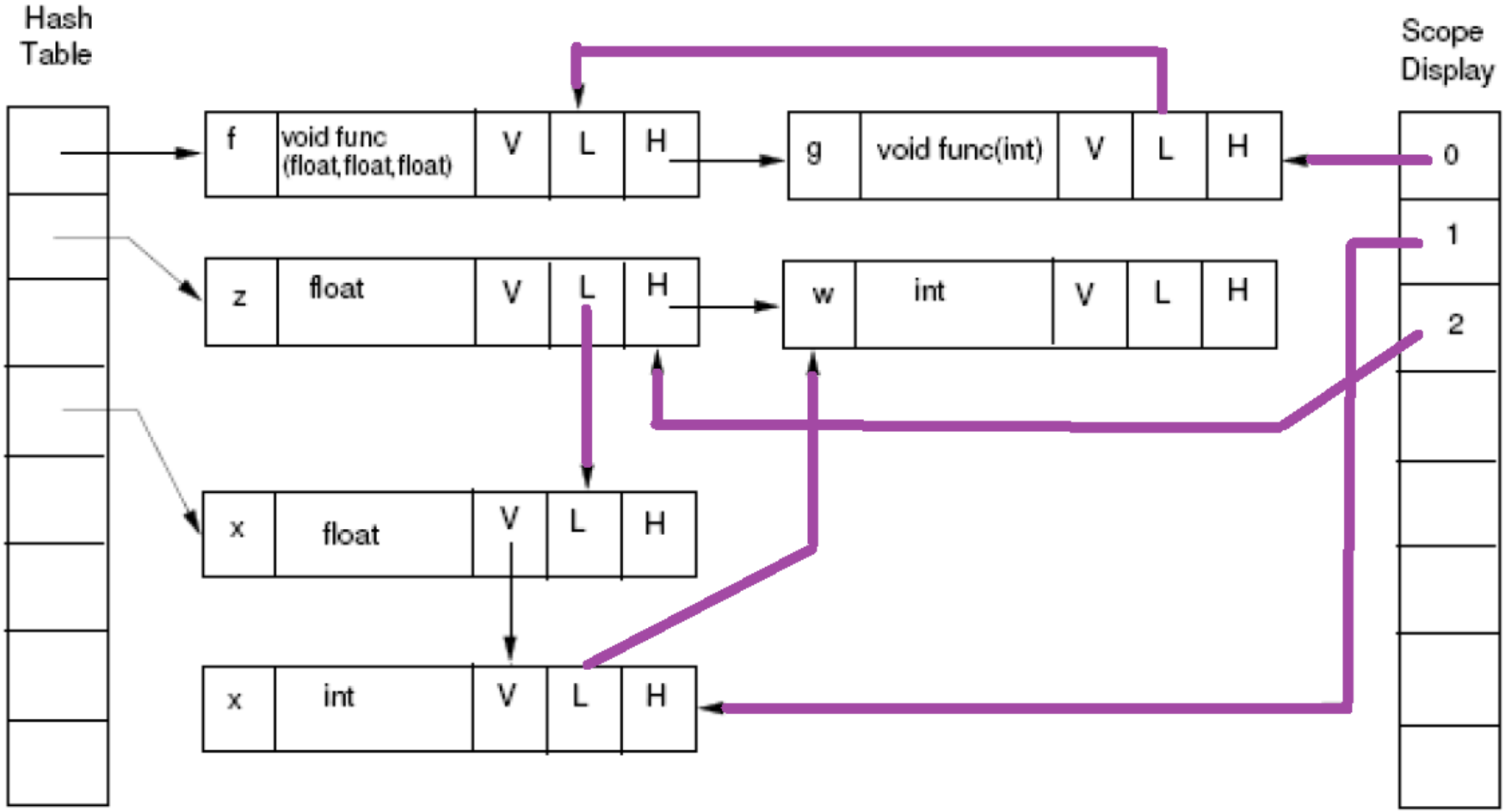


Figure 8.8: Detailed layout of the symbol table for Figure 8.1. The V, L, and H fields abbreviate the Var, Level, and Hash fields, respectively

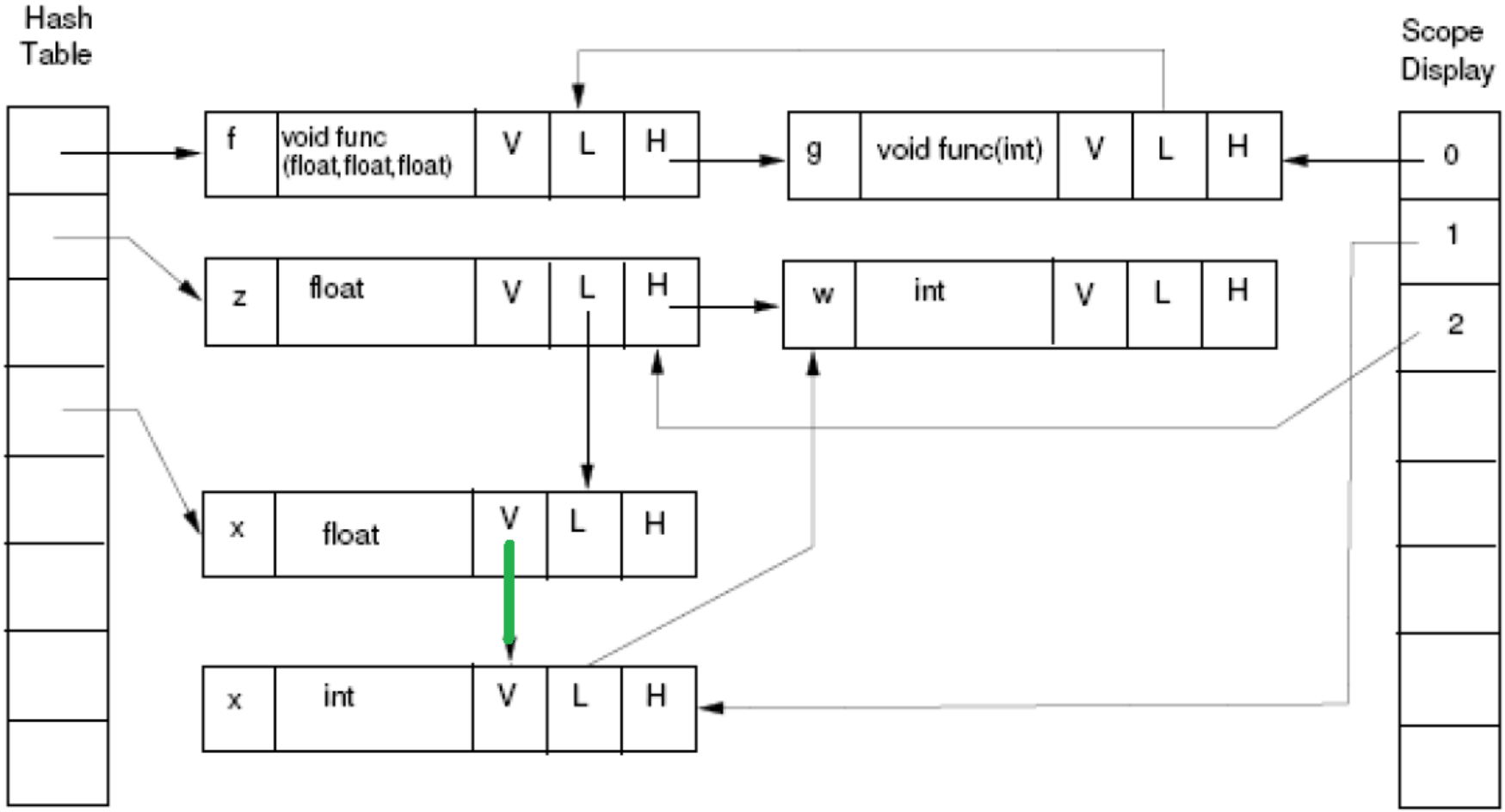


Figure 8.8: Detailed layout of the symbol table for Figure 8.1. The V, L, and H fields abbreviate the Var, Level, and Hash fields, respectively

Figure 8.7 Symbol table management

```
procedure OpenScope()
   depth := depth + 1
   scopeDisplay[depth] := null
end

procedure CloseScope()
   foreach sym in scopeDisplay[depth] do
       prevsym := sym.var
       call Delete(sym)
       if prevsym != null then call Add(prevsym)
   depth := depth - 1
end


function RetrieveSymbol(name) return Symbol
   sym := HashTable.Get(name)
```

```
    while sym != null do
        if sym.name == name then return(sym)
        sym := sym.hash
    end
    return(null)
end

procedure EnterSymbol(name, type)
    oldsym := RetrieveSymbol(name)
    if oldsym != null and oldsym.depth == depth
    then error("duplicate definitions of", name)
    newsym := CreateNewSymbol(name, type)
    // add to scopeDisplay
    newsym.level         := scopeDisplay[depth]
    scopeDisplay[depth] := newsym
    newsym.depth         := depth
    // add to hash table
    if oldsym == null the call Add(newsym)
```

```
    else call Delete(oldsym)
         call Add(newsym)
    newsym.var := oldsym
end

function DeclaredLocally(name) return boolean
    // See exercise 7
end
```

```
procedure      S      ( )
    depth ← depth + 1
    scopeDisplay[depth] ← null
end

procedure      S      ( )
    foreach sym ∈ scopeDisplay[depth] do                    (3)
        prevsym ← sym.var
        call        (sym)                                   (4)
        if prevsym ≠ null                                   (5)
        then call      (prevsym)
    depth ← depth – 1
end

function           S       (name) returns Symbol
    sym ← HashTable.     (name)
    while sym ≠ null do                                     (6)
        if sym.name = name
        then return (sym)
        sym ← sym.hash                                      (7)
    return (null)                                           (8)
end

procedure      S       (name, type)
    oldsym ←           S       (name)
```

We make use of two routines:

1. `Delete(sym)`: remove the symbol table entry `sym` from the collision chain in `HashTable.Get(sym.name)`. The symbol is not deleted. It is temporarily removed from the hash table and will be attached to the symbol table shortly via the `var` field. In particular, its `Var` and `Level` fields remain intact.
2. `Add(sym)`: add the symbol `sym` to the collision chain in `HashTable.Get(sym.name)`. We assume that there is not such an entry prior to the addition.

The `OpenScope()` procedure creates a new empty scope and increments the global `Depth` variable.

The `CloseScope()` procedure deletes all symbols declared in the current scope and then decrements the global `Depth` variable.

The `RetrieveSymbol()` procedure examines a collision chain to find the desired symbol.

The `EnterSymbol(sym, type)` procedure first locates the currently

active definition of `sym`, should one exist in the table. A new symbol-table entry is created and added to the hash table and the `scopeDisplay[depth]`.

You will need to adjust this symbol-table structure for additional features in programming languages, such as record fields, overloaded names, etc.

§8.4 Advanced Features

In addition to the basic block structure, contemporary programming languages provide many advanced features for controlling the visibility of names. These include

1. overloading
2. name hiding and promotion
3. modification of search rules

We will adjust the basic symbol table implementation for these advanced features.

§8.4.1 Record and type names

The only restriction on field names in a record (or `struct` in C) is that they be unique within the record.

```
A : record ①
    A : int;

    X : record ②
        A : real;
        B : boolean;
        end
    end
```

Usually, references to fields must be completely qualified, e.g., `A`, `A.A`, `A.X.A`.

In PL/1 and COBOL, incomplete qualifications are allowed, e.g., `A.X.B` or `A.B`, as long as there are no confusions. Incomplete qualification is deemed a bad practice.

In C, `typedef` gives a name to a type. Most compilers allow the

scanner to call a backdoor function to the symbol table in order to determine if a name is a type name or not.

How to handle field names?

1. There is one small table for each record type.
   This table is a field of the `record` type.
   Ex. For `R.A`, first find `R` then find `A` in `R`'s symbol table.

   (+) easy to implement

   (-) waste space if a hash table is used, but good for a binary tree.

2. Treat field names like ordinary names.
   Put them in the same symbol table.
   Need a record number field (like a scope number).
   Each record variable has a unique record number.
   E.g., for `R.A`, first find `R` (and `R`'s record number) then find `A` with `R`'s record number.
   For ordinary names, the record number is 0.

§8.4.2 Overloading and type hierarchy

In most programming languages, a name may denote different objects at the same time. Object-oriented languages, such as Java and C++, allow method name overloading provided each definition has a unique type signature, which is used later for overload resolution. For instance, we may have the following three declarations:

```
void print(int)
void print(real)
void print(String)
```

Some compilers make a function's *type signature* as a part of the function's name, such as "`print(int):  void`". Other compilers collect all function definitions with the same name in a list. Subsequently, an *overload resolution* procedure will pick an appropriate definition for each invocation of the function name from the list.

Overloading is different from hiding, which is used in conventional scope rules for block structures.

Can we use the same name as a variable and a function simultaneously?

Ex. In Pascal, a name may denote a function and its return value.

```
function abc() : integer
begin
      abc := abc + 1;
end
```

Ada allows much more general overloading—procedure names, function names, operators, and enumeration literals all can be overloaded. For example,

```
function +(X, Y: complex): complex;
function +(U, V: polar): polar;
type month is (Jan, Feb, ..., Oct, Nov, Dec);
type base is (Bin, Oct, Dec);
```

The new `Oct` overloads, rather than hides, the old `Oct`.

There are algorithms to determine the meaning of an overloaded name (chapter 11) in a given context.

C allows a name to be used as a local variable, a struct name, and a label.

C++ allows operators (`+`, `-`, `[ ]`, ...) to be overloaded.

In object-oriented programming languages, overloading and type/class inheritance together makes programming more flexible but compiling more complicated.

Java overloading and inheritance

```
class A           { . . . }
class B extends A { . . . }
class C extends B { . . . }
class D           { int h(A x) { return 1; } }
class E extends D { int h(C x) { return 2; } }
class F extends E { int h(A x) { return 3; } }
class G extends F { int h(B x) { return 4; }
                    int h(C x) { return 5; } }
class H {
      main(. . .) {
      B b = new C();
      E e = new G();
      . . . e.h(b) . . . --- 3
      }       }
```

§8.4.3 Implicit Declaration

Ex. FORTRAN implicitly declares variables and their types.

Ex. Algol 60 implicitly declares labels.

Ex. Scripting languages such as Javascript contains no explicit declarations.

Ex. Ada implicitly declares `for`-loop indices and opens a new scope. The following program contains two different `i`'s.

```
i : integer;
i := 5;
for i in 1 .. 9 loop
      . . .
end loop;
print i;
```

Two solutions for Ada:

1. Actually create a new scope.
2. Put a loop index in the same scope but possibly hide an existing name temporarily.

Ex. **(labels in Pascal)** In Pascal, label declarations and label usages do not mesh well. Specifically, we may not be able to jump to a label (i.e., use that label) within the scope of its declaration.

```
label 99;
begin
     for i := 1 to n do begin
       99:  x := x + 1;
     end
     goto 99;    // illegal
end
```

Solution: Mark the label (`99`) as inaccessible outside the `for`-loop.

§8.4.4 Export and import directives

The export rules allow selected names declared in a local scope to be visible outside that scope. Export rules are typically associated with modularization, such as Ada packages, C++/Java classes, C compilation units, and MODULA-2 modules.

C++ and Java provides three access modifiers:

1. `private`: for friends
2. `protected`: for subclasses and friends
3. `public`: for general public

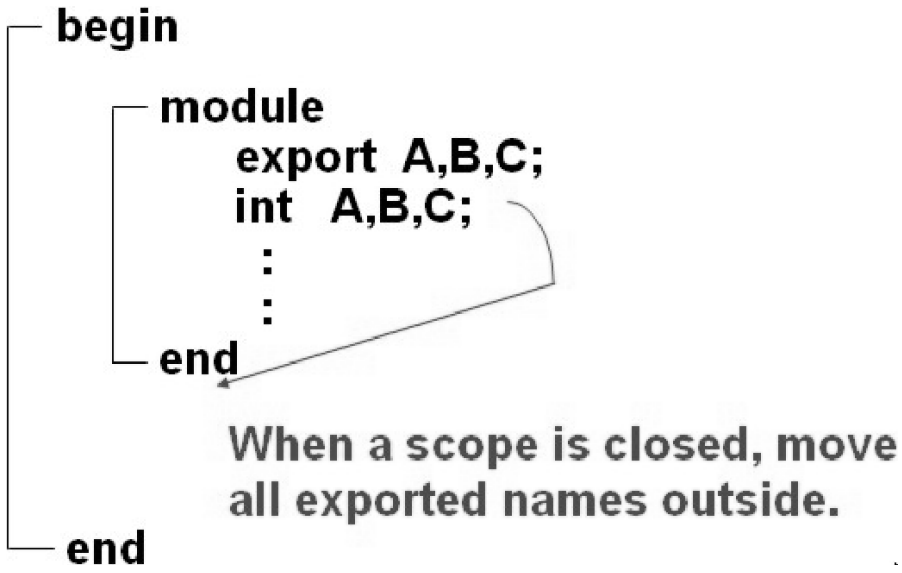
Furthermore, visibility may change in private inheritance.

In contrast to `public`, all methods in C are visible outside their compilation units unless they are prefixed with the `static` modifier.

Example. A module in Modula-2.

```
module IntStack
export push, pop;   // export rule
const  max = 25;
var    stack : array [ 1 .. max ] of int;
       top   : 1 .. max;
procedure push(int);
procedure pop() : int;
begin top := 1; end IntStack;
```

How to handle export rules?



How to find all exported names when a scope is closed?

1. When a hash table + collision chains are used, we need to scan each collision chain.
2. When a binary tree is used for each scope, the exported names cluster around the root (if the `export` statements come before any declarations).

Example. For Ada packages, there is a specification part, which contains all exported names. Each name has a boolean `exported` field in the symbol table. Use a `close_scope()` procedure to handle exported names.

specification (exported)

```
package IntStack is
    procedure  push(int);
    function   pop(): int;
end  package
```

implementation

```
package body IntStack is
    max: const int := 25;
    stack: array ......
    top: int;
    procedure push() { .... }
    function pop ....
begin  top := 1;  end IntStack
```

The specification part of a package is exported. The exported names are used with qualifications, e.g. `IntStack.push`. They can also be

imported with `use` clauses.

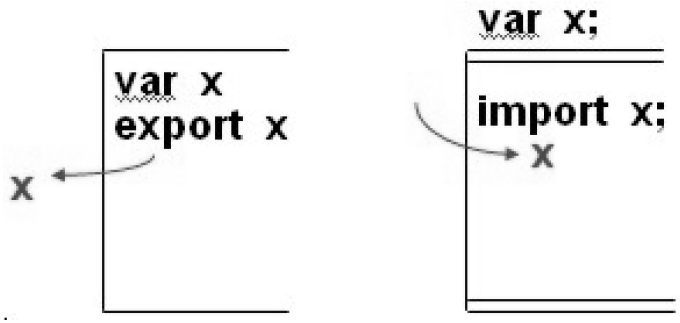
Import rules

For a large program, we hope to restrict a name's visibility to the smallest possible area. The export rules may serve this purpose. Each module may selectively export the names declared in that module.

On the other hand, there are also corresponding *import rules*. The import rules allow a module *fooModule* to select, among the global names exported from other modules, only those *fooModule* needs. C uses `include` to import names in a header file. Java uses `import` to import packages and classes. Ada uses `use` for importing names. C++ has the `using` statements.

export: to the outside

import: to the inside



The purpose of import rules is more precise control, for increasing the reliability of programs. An example of *import*:

```
module thingStack
     import Thing;  // Thing is a type
     var    stack : array [1..3] of Thing;
     . . .
end
```

In Java, we use

```
import A.*;
import B.C.D;
```

In Python, we use

```
from collections import deque
import logging
```

§8.4.5 Altered search rules

Some programming languages offer special rules to alter the search process:

1. `with` statements in Pascal
2. qualified names
3. `use` clauses in Ada
4. `namespace` and `using` in C++

1. `with` statements in Pascal

```
with R do <stmt>
```

An example of the `with` statement:

```
type R = record   a : int;   b : boolean   end;
var  q : R;
     a : float;
     . . .
with q do
     c := a + b;   // Which a is this a?
```

Within `<stmt>`, a name is searched in the following order:

1. declarations within `<stmt>`
2. fields of record type `R`
3. other scopes in the normal order

Consider the following nested `with` statement:

```
with R1 do
     with R2 do
          with R3 do
               c := a + b;
```

To process `with` statements,

1. easy if each record or each scope has its own symbol table. Use a stack of symbol tables.
2. for a single big table,
   (a) Open a new scope.

(b) Keep a stack of all open `with` statements.

2. **qualified names**: similar to record's fields

`IntStack.A`

1. Find `IntStack`'s symbol table or scope number.
2. Find `A` in that symbol table or in that scope.

If packages do not nest, things are simpler.

In Ada, packages may nest.

3. `use` clauses in Ada

```
package foo;
use pkg1, pkg2;
...  x ...
end package;
```

Names in `pkg1` and `pkg2` will become visible.

Two rules:

1. Local definition has precedence if a name is defined both locally and in imported packages.
2. If a name is defined in both `pkg1` and `pkg2`, the name is not directly visible.

Three Solutions:

1. Enter *all* names of a package into the local symbol table when it is imported.
   This is expensive if the package is big.

2. Search a name in the local symbol table when it is used.
   If the name is not found in the local table, search *all* packages.
   Do not duplicate the names in the local symbol table.

3. Similar to (2). After a name is found, put it in the local symbol table to avoid repeated search.

For practical programs, there is *locality*: After a name is referenced, the probability that the name will be referenced again soon is very high.

How to reduce the probability of finding a bomb in an airplane?

4. namespace and using in C++

```
#include <iostream>
using namespace std;

// first name space
namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }

   // second name space
   namespace second_space {
      void func() {
         cout << "Inside second_space" << endl;
      }
   }  // end of second_space
}  // end of first_space
```

```
using namespace first_space::second_space;
int main () {
   // This calls function from second name space.
   func();
   return 0;
}
```

Note the second namespace is inside the first namespace. If we compile and run above code, this would produce the following result:

```
Inside second_space
```

Namespaces can be nested where you can define one namespace inside another name space.

A namespace can be defined in several parts and so a namespace is made up of the sum of its separately defined parts. The separate parts of a namespace can be spread over multiple files.

You do not need to `using` the *whole* namespace. You can choose only the items you want, such as `using std::cout`.

We may declar a namespace in the `alpha.h` file, define the operations in the namespace in the `alpha.cpp` file, and use the namespace in other files.

C++ does not recommend `using`. Instead, `std::cout`, `std::string` should be used.

```
// In alpha.h
namespace betaServer {
    void Foo();
    int Bar();
}


// In alpha.cpp
#include "alpha.h"
using namespace betaServer;
```

```
void betaServer::Foo() { // use fully-qualified name here
   // no qualification needed for Bar()
   Bar();
}

int betaServer::Bar(){ return 0; }
```

Forward reference is also an issue for symbol table management. Some languages require forward references be declared. But this is not always possible. For example, in C, ...

Forward Reference

In Pascal, pointer types may introduce forward references. In the following example, `P` is a pointer-to-`real` type.

```
type T = integer;
procedure abc();
     type    P = ^ T; (which T does this T refer to?)
     . . .
     type    T = real;
```

Forward references cause difficulty for compilers. It would be ideal if we can totally ban forward references. However, sometimes we do need forward references.

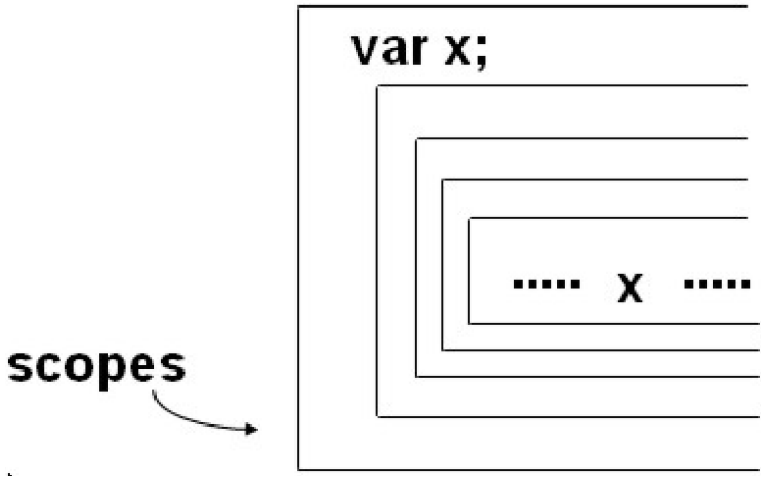
In C, we also have forward reference.

```
struct a { . . . struct b *nextb; . . . }
struct b { . . . struct a *nexta; . . . }
```

Some problems for non-local goto in Algol 60.

```
L: . . .
   proc abc()
   begin
        goto L;  -- which L does this L refer to?
        . . .
    L:  . . .
   end;
   . . .
```

In the most general case, suppose `x` may be a forward reference. We need to check all the scopes from inside out.



The `label` problem in Pascal is simpler since labels have to be declared.

This is why most languages require declarations precede statements.

How to forbid forward reference?

In Pascal, constants cannot be forward referenced. Then consider

```
const C = 10
procedure abc();
   const D = C;
   E: array[D] of integer;
   F: integer;
   . . . (3 million lines of code)
   const C = 20;  (what if C+20?)
   begin . . .  end
```

So what is the value of `D`? — error!

It is not easy to detect this error.

Most Pascal compilers cannot detect this error, including SUN `pc`.

C uses `define` to define constants.

Due to this difficulty, Ada changes the visibility rule. . . .

How to handle forward reference?

We need more than one pass:

- one for collecting all definitions
- others for processing declarations and statements.

For Pascal pointer types,

1. Link together all references to type `T`.
2. Determine the real `T` after the type section is completely examined.

For `goto` labels, use backpatch or symbolic labels in one-pass compilers.

More general forward references, e.g., in PL/1, must be processed in multiple passes.

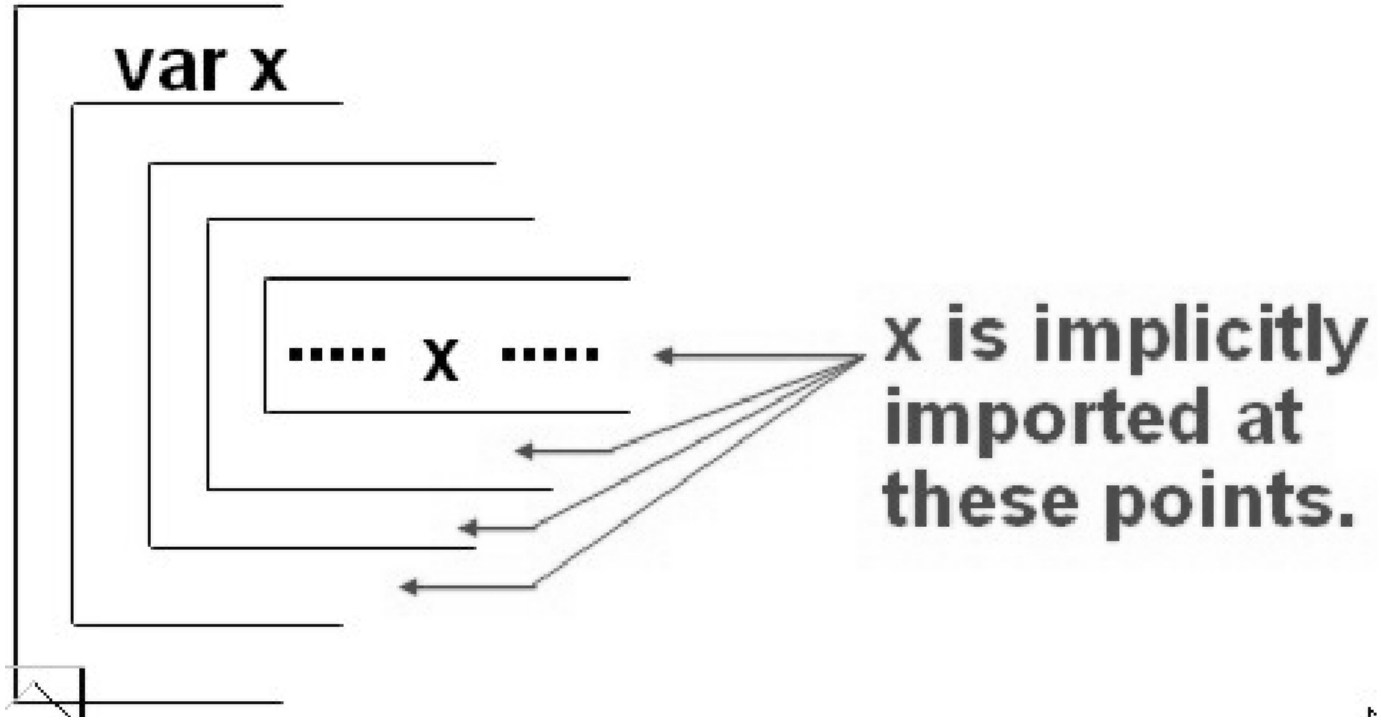
```
A = B + C
int  A, B, C;
```

To detect illegal forward reference,

```
const C = 10;

procedure abc;
     const D = C;
     -- here assume C is imported
     . . .
     const C = 20;
     -- here we find C is in conflict with the imported C
```

In general, assume x may be forward-referenced.



Should we tolerate this indecent program?

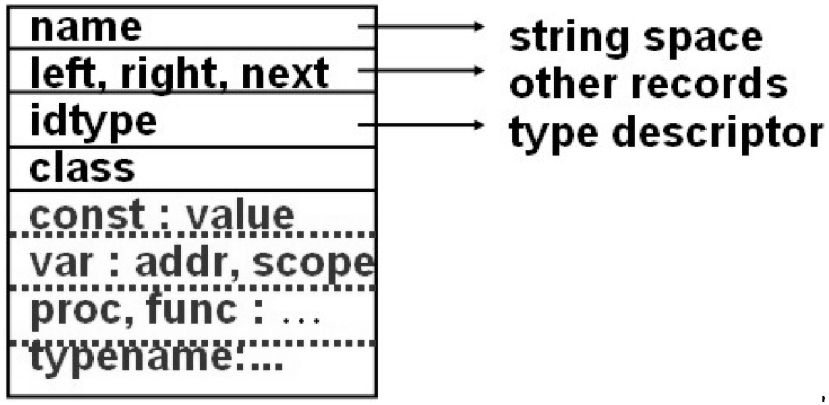
```
const C = 10;
. . .
procedure abc;
   const D = C;
         . . .
   const C = 10; -- different C's with the same values
   begin . . . end
. . .
```

Can we use a type name that is currently under construction?

```
struct foo {
    int x;
    float y;
    struct foo z;        // ok?
    struct foo *w;       // ok?
}
```

§8.5 Declaration processing fundamentals

A *symbol table* consists of a set of entries, one for each entity. An entry has the form:



A name could denote either a constant, a variable, a type, a function, a procedure, an enumeration literal, a class, a package, etc. This is noted in the `class` field.

For different classes of names, the entries contain different pieces of information (properties). For a *constant*, the entry contains the actual value of the constant. For a *variable*, the entry contain its address and scope.

All the properties of a name are stored in an `Attributes` structure. Every name/symbol has an Attributes structure.

Since different classes of identifiers carry different properties, if the compiler is written in C, we can use a `union` type for the `Attributes` structure. On the other hand, if the compiler is written in an object-oriented language (e.g., Java), we can define a base class `Attributes`, which contain the fields that are common to all classes of identifiers. Then a subclass is defined for each class of identifiers. See Figure 8.9. Figure 8.9 (a) is for an identifier that represents a variable. while Figure 8.9 (b) is for an identifier that represents a type.

Every type has a TypeDescriptor.

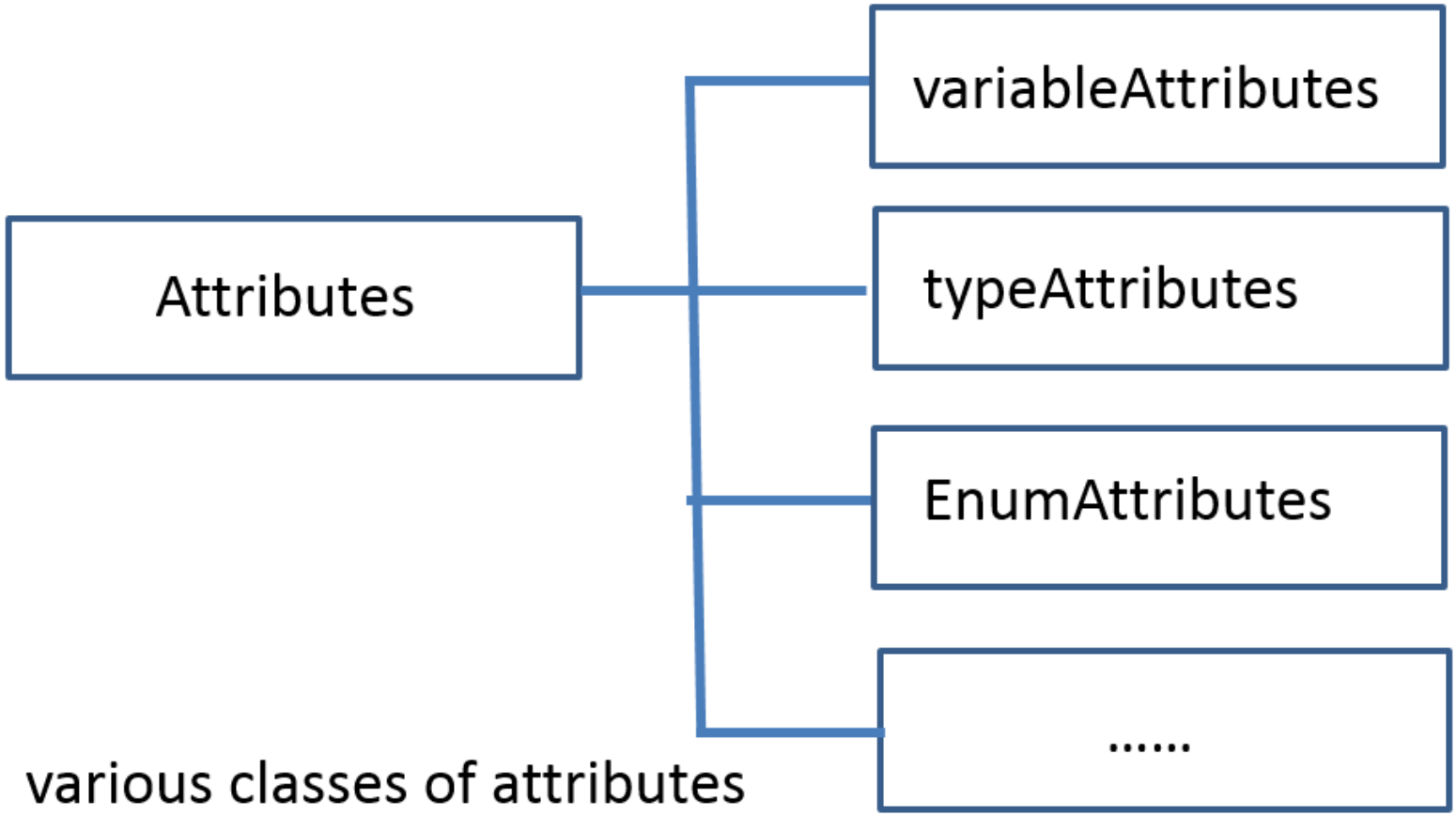
The `TypeDescriptor` class similarly has several subclasses, such as `EnumTypeDescriptor`, `StructTypeDescriptpor`, `ArrayTypeDescriptor`, etc. in later sections.

Union types in C and variant records in Pascal.

```
  struct Attributes {
   char name[20];
   int idtype;
   int class; // one of constant, variable, function, etc.
   union {
       int value;          // if this id is a constant.
       float address;      // if this id is a variable.
       char *parameters;   // if this id is a function.
   } u;
 } symtab[NSYM];
```

subclasses in Java

```
class Attributes {
  char name[20];
  int idtype;
  int class;
}
class ConstAttrubutes extends Attributes {
      int value;
}
class VariableAttributes extends Attributes {
      float address;
}
class FunctionAttributes extends Attributes {
      char *parameters;
}
class typeAttributes extends Attributes { }
class EnumAttributes extends Attributes { }
```



various classes of attributes

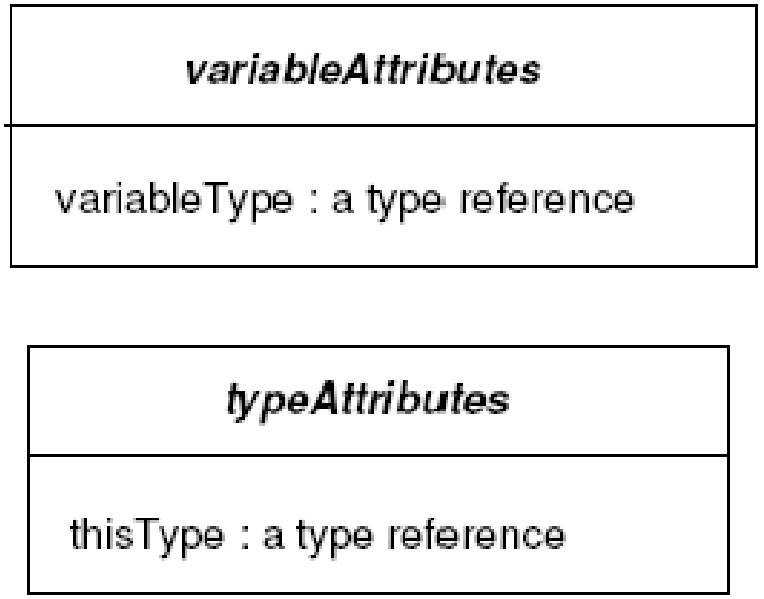


Figure 8.9: Attribute Descriptor Structures

routines for symbol tables:

1. `SymbolTable.RetrieveSymbol(name)` returns a pointer to an attribute record.
2. To enter a symbol into the symbol table, use

```
SymbolTable.EnterSymbol(name, attribute)
```

Declarations

When a name is declared, an entry for it is created in the symbol table. Each name has certain properties. The properties come from two sources:

1. declared in the program, such as type, number of elements in an array, dimension, range, etc.
2. generated by the compiler, such as addresses of variables, etc.

A variable has two categories of properties:

1. those shared by all variables of the same type, e.g., size, subscript range, element type, number of fields, field names, types of fields, field ordering, etc.
2. those specific to the variable itself, e.g., starting address, code address, scope, etc.

Ex. Consider

```
a, b, c :  array[1 ..  10] of integer
```

The three variables `a`, `b`, and `c` share some information. The important thing is

No duplicate information about `a`, `b`, and `c` in the symbol table.

Duplicate information:

1. Waste space.
2. Create inconsistencies.

Figure 3: Duplication leads to inconsistency.

Some languages allow *implicit declarations*. For example, Fortran may assign types to identifiers based on their names.

In common programming languages, an identifier could denote a variable, a constant, a type, a procedure, a function, a class, a field, a label, or a package.

§8.5.2 Type descriptor structures

A *type descriptor* is a structure representing a type.

Most programming languages allow user-defined types in addition to primitive types (such as `int`, `real`, `boolean`, `char`, etc.)[a] The number of user-defined types in a programming language is usually unbounded. These user-defined types are built with type constructors, e.g.,

```
array[ . . . ] of . . .
record . . . end
set of . . .
file of . . .
```

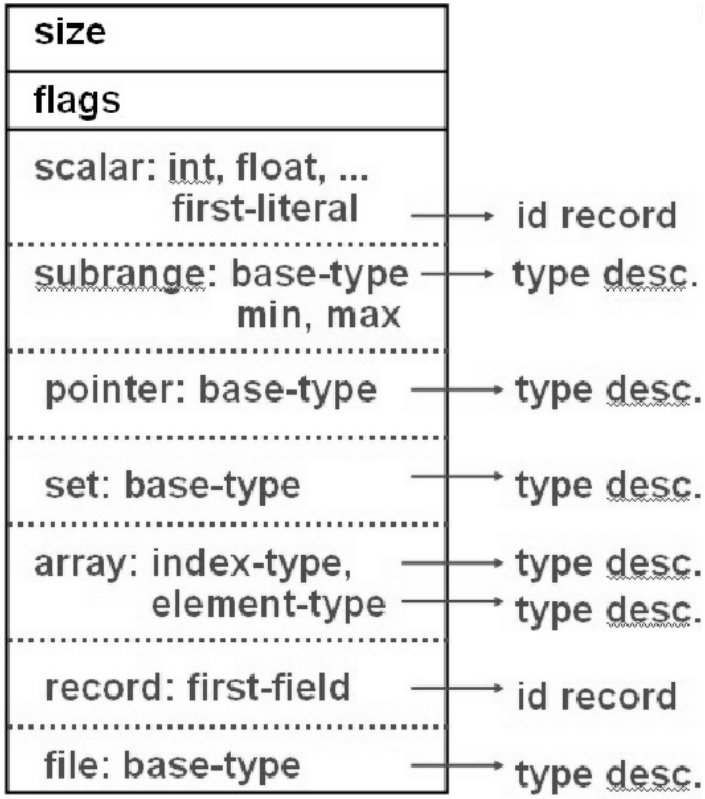
A compiler builds structures—*type descriptors*—to represent types.

Different categories of types incorporate different pieces of information. For example, a *subrange* type requires information of the base type and the minimum and maximum of the subrange. An array type requires

[a] Java includes 8 primitive types.

information about the index type (which is usually a subrange) and the element type. Note that even a primitive type, such as `integer` and `boolean`, has a type descriptor for uniform processing.

A type descriptor could be implemented as a union type in C or could be implemented as a subclass of a basic class, similar to `Attributes` in Figure 8.9. Here are the fields of a type descriptor in a union type.



The following figure shows the hierarchy of various type descriptor subclasses.

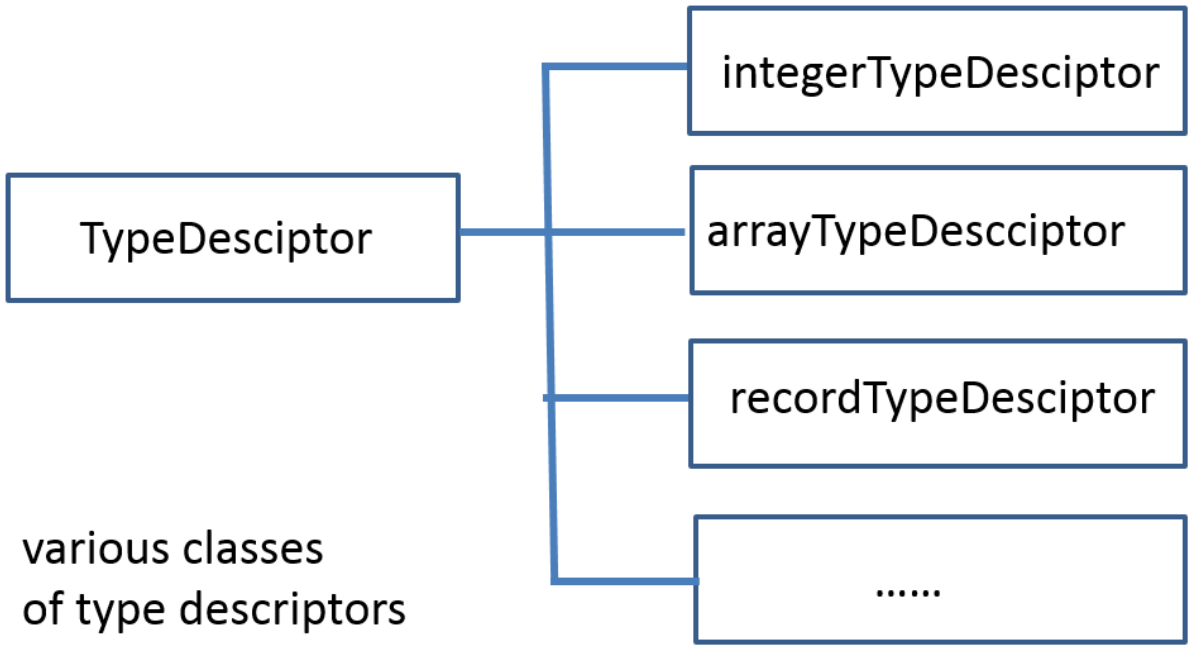


Figure 8.10 shows three type descriptor subclasses.

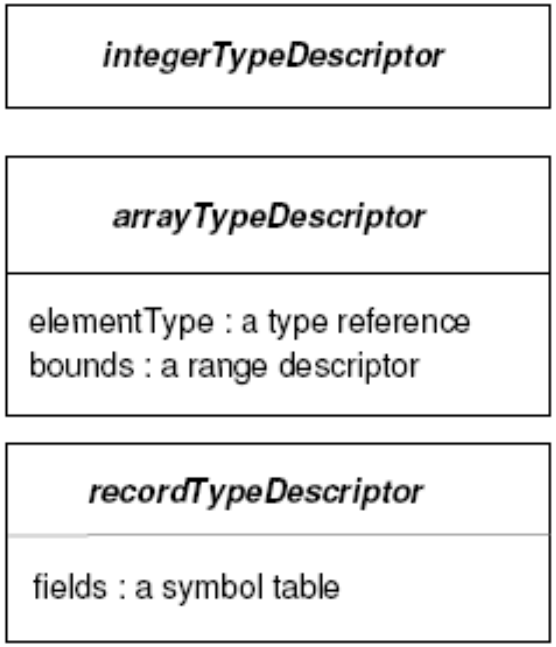


Figure 8.10: Type Descriptor Structures

Since a complex type is usually built from simpler types, it is natural to have pointers in a type descriptor that reference other type descriptors. Sometimes, a type descriptor may also contain a small symbol table.

We want a uniform representation for all types, from simple primitive types to complex programmer-defined types. A uniform representation simplifies a compiler's internal structure.

§8.5.3 Type checking using an abstract syntax tree

We will use a semantic processing pass that visits the AST. This pass will construct the symbol table from the declarations in the AST and perform necessary type checking throughout the tree.

We will use the *visitor pattern* to implement this pass.

The `SemanticsVisitor` is the top-level visitor for processing declarations and doing semantic checking on an AST's node. There is one `Visit` method for each kind of AST node in the `SemanticsVisitor` class.

The `TopDeclVisitor` processes declarations. It builds the symbol table structures for *variable*, *type*, *class*, and *method* declarations.

The `TypeVisitor` finds type infromation from type names and constructs type structures. It examines *identifier*, *array*, *structure*, and *enumeration*. It is *not* responsible for type checking.

```
class NodeVisitor
   procedure VisitChildren(node n)
       foreach c in n.getChildren() do call c.Accept(this)
   end
end

class SemanticsVisitor extends NodeVisitor
   /* Visit methods for other node types are defined in \S 8.8 */
end

class TopDeclVisitor extends SemanticsVisitor
   procedure Visit(VariableListDeclaring vld)
       /* Sec 8.6.1, p. 303 */
   end
   procedure Visit(TypeDeclaring td)
       /* Sec 8.6.3, p. 305 */
   end
   procedure Visit(ClassDeclaring cd)
       /* Sec 8.7.1, p. 317 */
   end
```

```
    procedure Visit(MethodDeclaring md)
        /* Sec 8.7.2, p. 321 */
    end
end

class TypeVisitor extends TopDeclVisitor
    procedure Visit(Identifier id)
        /* Sec 8.6.2, p. 304 */
    end
    procedure Visit(ArrayDefining arraydef)
        /* Sec 8.6.5, p. 311 */
    end
    procedure Visit(StructDefining structdef)
        /* Sec 8.6.6, p. 312 */
    end
    procedure Visit(EnumDefining enumdef)
        /* Sec 8.6.7, p. 313 */
    end
end
```

Figure 8.11 Structure of the Declarations Visitors

The type hierarchy is as follows:
`NodeVisitor` → `SemanticsVisitor` → `TopDeclVisitor` →
`TypeVisitor`

The `Visit` method will take an instance of the node type as parameter.

The code implementing the semantics pass will be scattered across many class definitions.

An alternative is a recursive traversal of the AST by a single routine with a big `switch` statement, with one branch for each kind of node in the abstract syntax tree.

The `SemanticsVisitor` marks the end of the analysis phase of the compiler.

§8.6 Variable and type declaration

Consider a simple declaration:

```
var a, b, c:  integer;
```

The relevant part of the AST looks as follows:

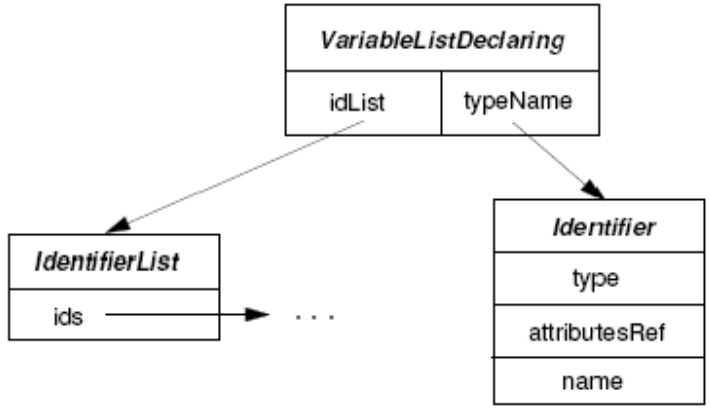


Figure 8.12: Abstract Syntax Tree for Variable Declarations

We will need a `Visit` method (in Figure 8.13) to process this simple declaration and enter information into the symbol table.

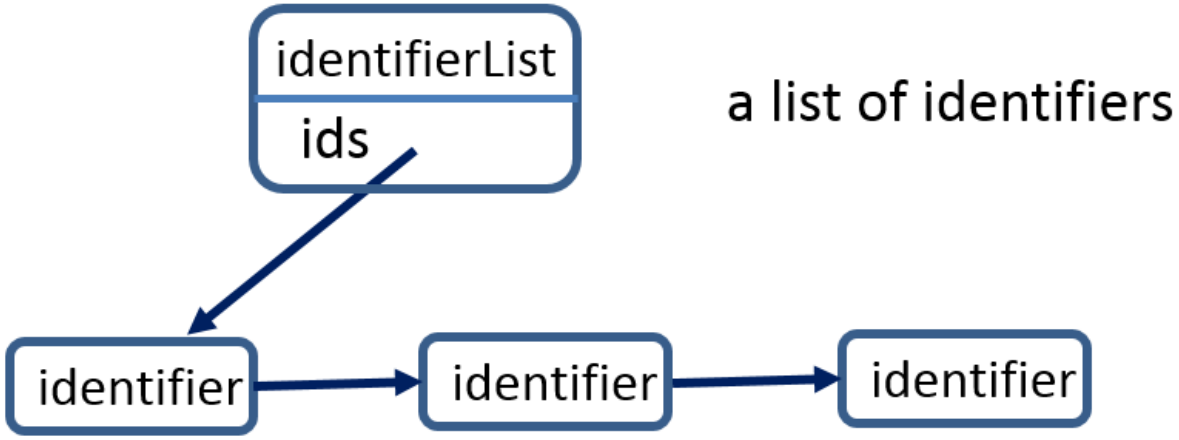


Figure 4: AST for a list of identifiers.

```
procedure Visit(VariableListDeclaring vld)
   typeVisitor := new TypeVisitor()
   call vld.typeName.Accept(typeVisitor)
   foreach id in vld.idlist do
       if currentSymbolTable.DeclaredLocally(id.name)
       then call error("This name has already been declared.")
            id.type           := errorType
            id.attributesRef  := null
       else id.type           := vld.typeName.type
            attr              := new Attribtes(variableAttributes)
            attr.kind         := variableAttributes
            attr.variableType := id.type
            id.attributesRef  := attr
            call currentSymbolTable.EnterSymbol(id.name, attr)
end
```

Figure 8.13 Visit method in TopDeclVisitor for VariableListDeclaring

§8.6.2 Handling type names

```
procedure Visit(Identifier id)
   attr := currentSymbolTable.RetrieveSymbol(id.name)
   if attr != null and attr.kind == typeAttributes
   then id.type          := attr.thisType
        id.attributesRef := attr
   else call error("This name is not a type name.")
        id.type          := errorType
        id.attributesRef := null
end
```

Figure 8.14 Visit method in TypeVisitor for Identifier

§8.6.3 Type declarations

Consider a type declaration such as:

```
type foo is array(1 ..  10) of integer
```

The relevant part of the AST is in Figure 8.15:

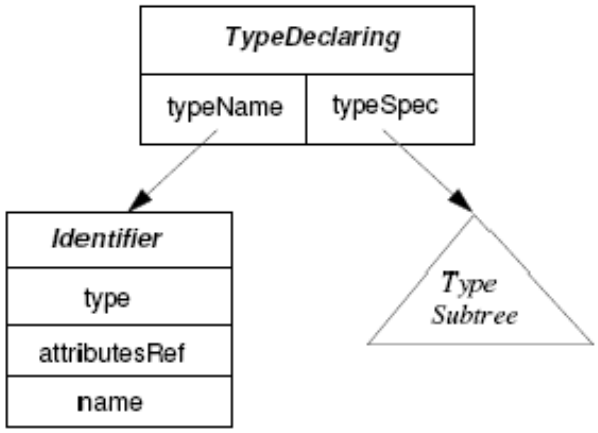


Figure 8.15: Abstract Syntax Tree for Type Declarations

The `Visit` method for a type declaration is in Figure 8.16.

```
procedure Visit(TypeDeclaring td)
   typeVisitor := new TypeVisitor()
   call td.typeSpec.Accept(typeVisitor)
   name := td.typeName.name
   if currentSymbolTable.DeclaredLocally(name)
   then call error("This name is already declared.")
        td.typeName.type          := errorType
        td.typeName.attributesRef := null
   else attr                      := new Attribute(typeAttributes)
        attr.thisType             := td.typeSpec.type // type desc
        call currentSymbolTable.EnterSymbol(name, attr)
        td.typeName.type          := td.typeSpec.type
        td.typeName.attributesRef := attr
end
```

Figure 8.16 Visit method in TopDeclVisitor for TypeDeclaring

As shown in Figure 8.15, `typeSpec` in a `TypeDeclaring` node contains a pointer to the AST of a type.

Note that, in Figure 8.16, `typeSpec.type` is a type descriptor, which is the uniform representation of a type inside a compiler. The type descriptor is constructed during the call

```
td.typeSpec.Accept(typeVisitor)
```

We need to compare the two `Accept(typeVisitor)` calls in Figure 8.13 and 8.16 carefully. There are two cases for declaring a variable:

```
var  x : foo
var  x : array (1 .. 10) of integer
```

For the type parts of the above two declarations, we will always obtain a pointer to a type descriptor. In the formal case (`vld.typeName.type`), the pointer is obtained from the symbol table. In the latter (`td.typeSpec.type`), the pointer is obtained from the construction of a new type descriptor.

Handling errors in type declaration and type reference

Certain errors related to types can be discovered during compilation.

For example, a name could be used as a type in a variable declaration but it is not declared as a type (yet). A compiler would generate an error message for this. Another error message may also be produced every time when the declared variable is used later. In other words, a missing type declaration may result in a series of error messages.

One solution is to stop compilation at the very first error. But this is not good. A better solution is to use a special `errorType`. All further errors related to the `errorType` will be silently skipped.

Example. Consider

```
var a : account;   // issue an error message

. . . a . . .      // no more error message
. . . a . . .      // no more error message
```

Suppose `account` is not declared as a type. Then there is an error in the declaration of `a`.

We should still enter `a` into the symbol table to prevent additional error messages in the uses of `a`.

To enter `a` into symbol table, `a`'s type is `errorType`.

Type compatibility

Consider an assignment statement:

```
var := expression ;
```

When a variable is assigned the value of an expression, we need to decide if the type of the variable is compatible with that of the value (or the expression). Since most programming languages are statically typed, type checking is done during compilation.

Consider

```
a, b : array [ 1 .. 10 ] of integer;
c    : array [ 1 .. 10 ] of integer;
```

Are `a`, `b`, and `c` of the same type?

There are two approaches to type compatibility:

Name equivalence e.g., Ada, Pascal, Modular-2.

`a` and `b` have the same type. `a` and `c` have different types. Name equivalence is enforced by comparing pointers to type descriptors.

Structural equivalence more relaxed.

`a`, `b`, and `c` have the same type.

Type compatibility is an important issue in type checking. Consider an assignment statement

```
a := b;
```

We need to determine whether `a` and `b` have compatible types for the assignment statement.

Name equivalence: application-oriented
Structural equivalence: implementation-oriented

Example. Is the following program meaningful?

```
typedef  length = 1 .. 10;  /* km */
typedef  weight = 1 .. 10;  /* kg */
var      l : length;
var      w : weight;
l := w;
```

Name equivalence is easier to implement—two types are compatible only if they have the same name (equivalently, the same type descriptor)—and more useful.

Structural equivalence is more difficult to implement. There are three methods to implement structural equivalence (similar to compiling Ada imported packages):

1. Compare types during type definition.
2. Compare types when necessary.
3. Compare types when necessary and remember the result of comparison.

Other issues in structural equivalence:

- Do field names matter?
- Does the order of fields matter?

A difficulty for structural equivalence is recursive types.

Example. Are `A` and `B` equivalent types?

```
A = record                B = record
    x : int                   z : int
    y : pointer to A          w : pointer to B
    end                       end
```

A and B are compatible if and only if they are assumed to be compatible initially.

§8.6.4 Variable declarations revisited

A more complex declaration of a list of variables is shown in Figure Figure 8.17. Figure 8.19 is the revised `Visit` method (of Figure 8.13).

```
var a, b, c :  constant private integer = 3
```

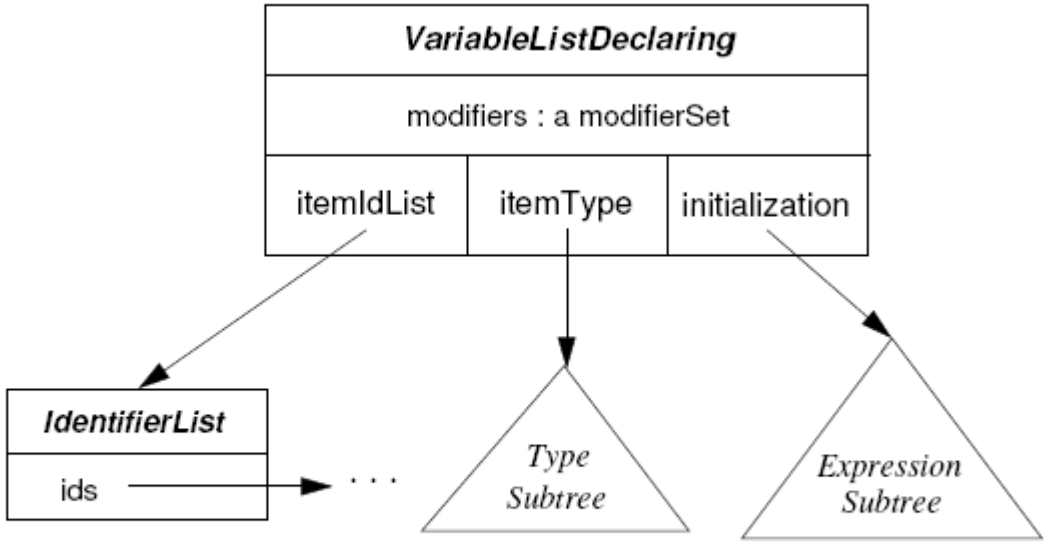


Figure 8.17: AST for Generalized Variable Declarations

```
procedure Visit(VariableListDeclaring vld)
   typeVisitor := new TypeVisitor()
   call vld.itemType.Accept(typeVisitor)
   declType := vld.itemType.type      // type descriptor
   if vld.initialization != null
   then checkingVisitor := new SemanticsVisitor()
        call vld.initialization.Accept(checkingVisitor)
        if not Assignable(vld.initialization.type, declType)
        then call error("initialization not assignable to var type")
   else if const belongs to vld.modifiers
        then call error("initialization missing for constant dcl")

   foreach id in vld.itemIdList do
       if currentSymbolTable.DeclaredLocally(id.name)
       then call error("This name has already been declared.")
            id.type           := errorType
            id.attributesRef  := null
       else attr              := new Attribtes(variableAttributes)
            attr.kind         := variableAttributes
```

```
            attr.variableType := declType    // type descriptor
            attr.modifiers    := vld.modifiers
            call currentSymbolTable.EnterSymbol(id.name, attr)
            id.type           := declType
            id.attributesRef  := attr
end
```

Figure 8.18 Visit method in TopDeclVisitor for VariableListDeclaring

In the call

```
vld.itemType.Accept(typeVisitor)
```

the type could be a simple name (as in Figure 8.13) or a type definition (as in Figure 8.16). It is equivalent to

```
typeVisitor.visit(vld.itemType)
```

The `Assignable` function implements the type equivalence rules.

§8.6.5 Static array types

Consider a simple static array type definition:

```
array[10] of int
array[ a + b ..  c * d ] of array[10] of int
```

The abstract syntax trees are shown in Figure 8.19. The element type could be a type name or could be a general type definition. Either form leads to a reference to a type descriptor.

The number of elements can be specified as a single integer (in this case, the default lower bound could be either 0 or 1) or a pair of integer expressions (as the range of indices).

The `Visit` method for Figure 8.19 (a) is shown in Figure 8.20. It builds a type descriptor for the array type. The `VisitChildren` call processes the subtrees for the size and for the element type, respectively. It will evaluate the constant expressions when necessary with a specialized visitor class. A new `typeDescriptor` object is created and initialized with appropriate `elementType` and `size`.

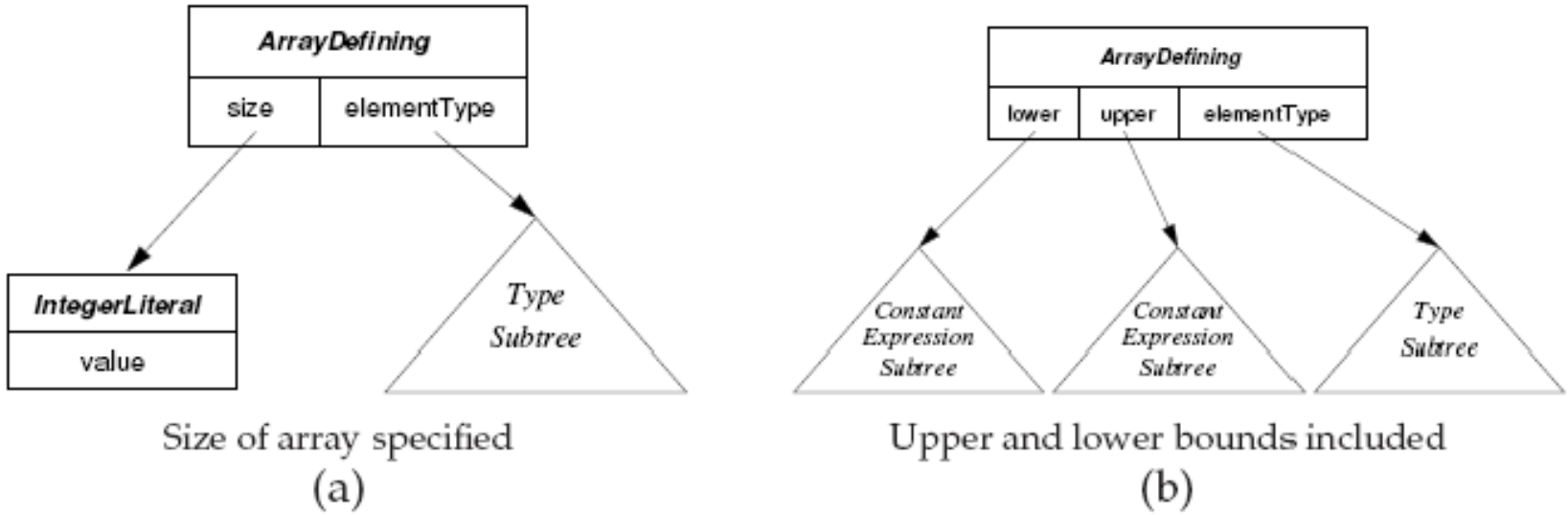


Figure 8.19: Abstract Syntax Trees for Array Definitions

```
procedure Visit(ArrayDefining arraydef)
   call VisitChildren(arraydef)
   arraydef.type := new typeDescriptor(arrayType)
   arraydef.type.elementType := arraydef.elementType.type
   arraydef.type.arraysize   := arraydef.size.value
   // the result is an arrayTypeDescriptor stored in arraydef.def.
end
```

Figure 8.20 The Visit method in TypeVisitor for ArrayDefining.

Some languages supports *dynamic arrays*:

```
function foo(int max) return boolean; {
  var qoo: array[ 1 .. max+1 ] of integer;
  ...
}
```

More examples:

```
function foo(int p) return array [ 1 .. 10 ] of integer; {
  ...
}

... foo(5)[3] := 10;
... foo(7)[9] + 22 ...
```

§8.6.6 Struct and record types

```
record
    a, b, c: integer
    d, e   : real
    f, g, h: boolean
end
```

Most programming languages allow a programmer to declare a structure (or a record). The AST for a structure definition is shown in Figure 8.21. In a `StructDefining` node, there is a `fieldList`, which points to a list of `FieldDeclaring` nodes. Each `FieldDeclaring` node is very similar to a `VariableListDeclaring` node (§8.6.4).

A structure defines a new scope for the fields. The fields can be accessed with qualified names, such as `a.b.c`. Thus, the symbol table for a structure will NOT be pushed onto the stack of currently open scopes. The `TypeDescriptor` for a structure, shown in Figure 8.10 (p. 91), includes a pointer (i.e., `fields`) that points to the symbol table for

the structure.

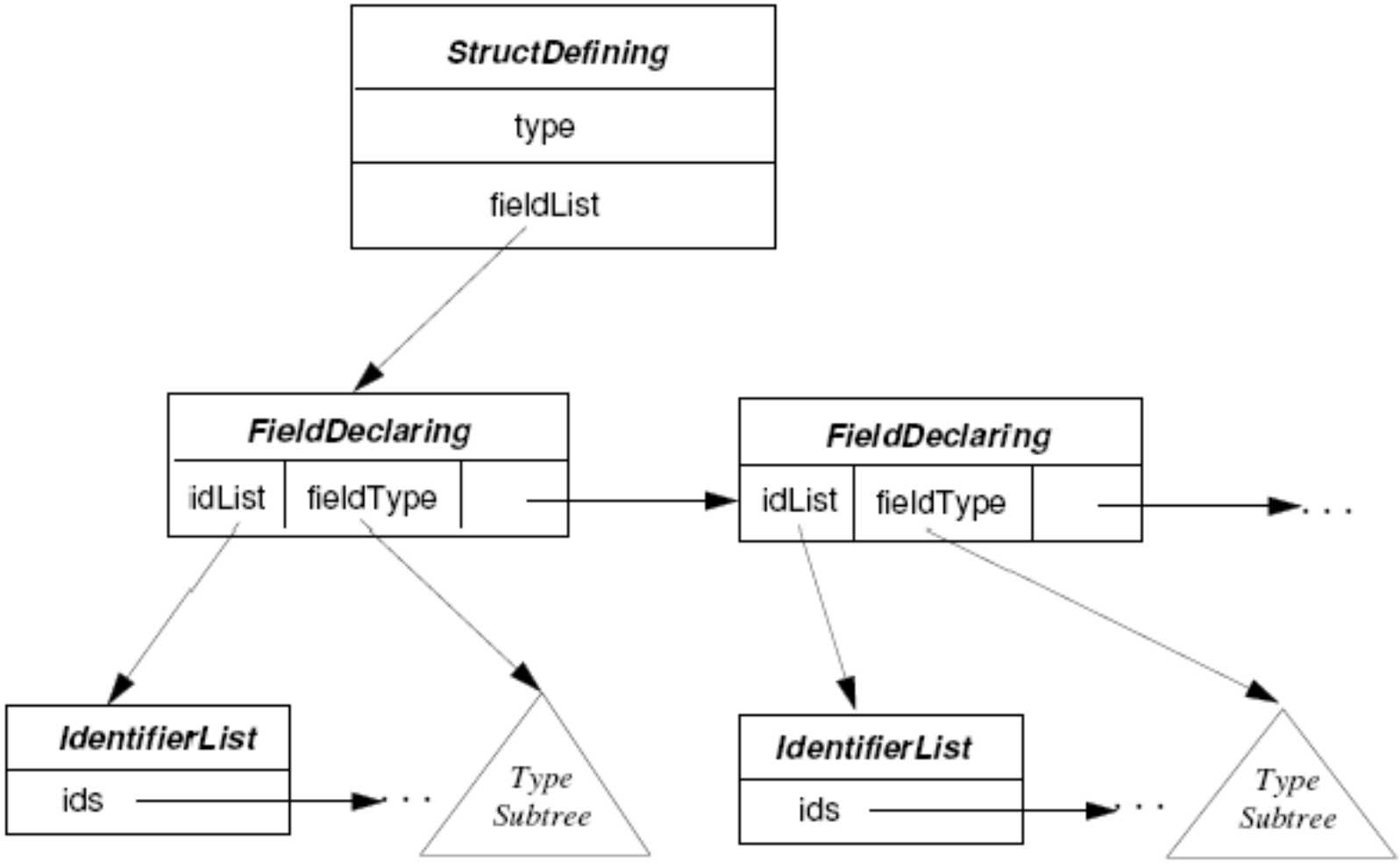


Figure 8.21: Abstract Syntax Tree for a Struct Definition

```
procedure Visit(StructDefining structdef)
   typeRef         := new typeDescriptor(structType)
   typeRef.fields := new SymbolTable() // a new, empty table
   foreach decl in structdef.fieldList do
      call decl.fieldType.Accept(this)
      foreach id in decl.idList do
         if typeRef.fields.DeclaredLocally(id.name)
         then call error("Field names cannot be re-declared.")
              id.type           := errorType
              id.attributesRef := null
         else attr              := new Attributes(fieldAttributes)
              attr.kind         := fieldAttributes
              attr.fieldType    := decl.fieldType.type // type desc
              call typreRef.fields.EnterSymbol(id.name, attr)
              id.type           := decl.fieldType.type
              id.attributesRef := attr
   structdef.type := typeRef  // type descriptor for the struct
end
```

Figure 8.22 The Visit method in TypeVisitor for StructDefining.

The `Visit` method processes each field declaration in turn. In each field declaration, there might be a list of identifers. These identifiers are checked for duplicate declarations within the structure. The final result of the `Visit` method is the type descriptor for the structure type.

Further issues:

• Calcaulte the offset of every field in a structure. Insert padding if necesssary.

• Rearange the order of fields if necessary (say, to reduce memory usage).

• Sometimes we need to add paddings to a structure to satisfy some alignment requirements.

• Calcualte the size of a structure.

• Some languages supports the *union types* (a.k.a. *variant record* in Pascal).

• Split a structure if profitable.

§8.6.7 Enumeration types

An enumeration type contains an ordered list of enumeration constants:

```
( Su, Mo, Tu, We, Th, Fr, Sa )
( red, orange, yellow, green, blue )
( jan, feb, mar, apr, ...  , oct, nov, dec )
( bin, oct, dec, hex )
```

The enumeration constants are usually represented as 0, 1, 2, ... internally. The AST for an enumeration type is shown in Figure 8.24.

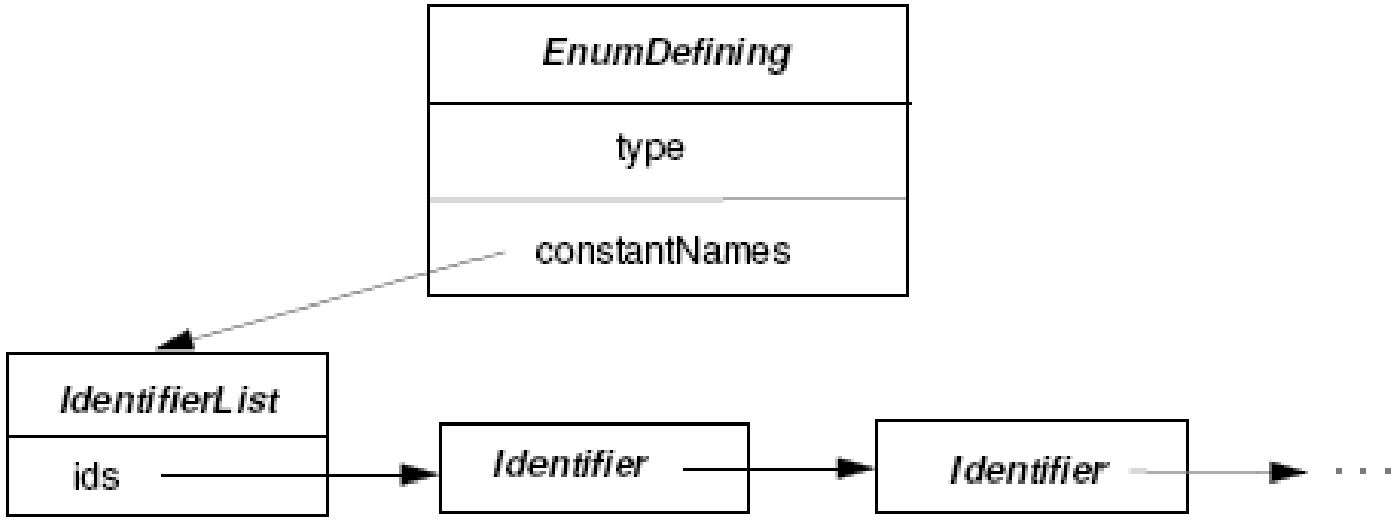


Figure 8.24: Abstract Syntax Tree for an Enumeration Type

The TypeDescriptor and Attributes descriptor for the enumeration types are shown in Figure 8.25.

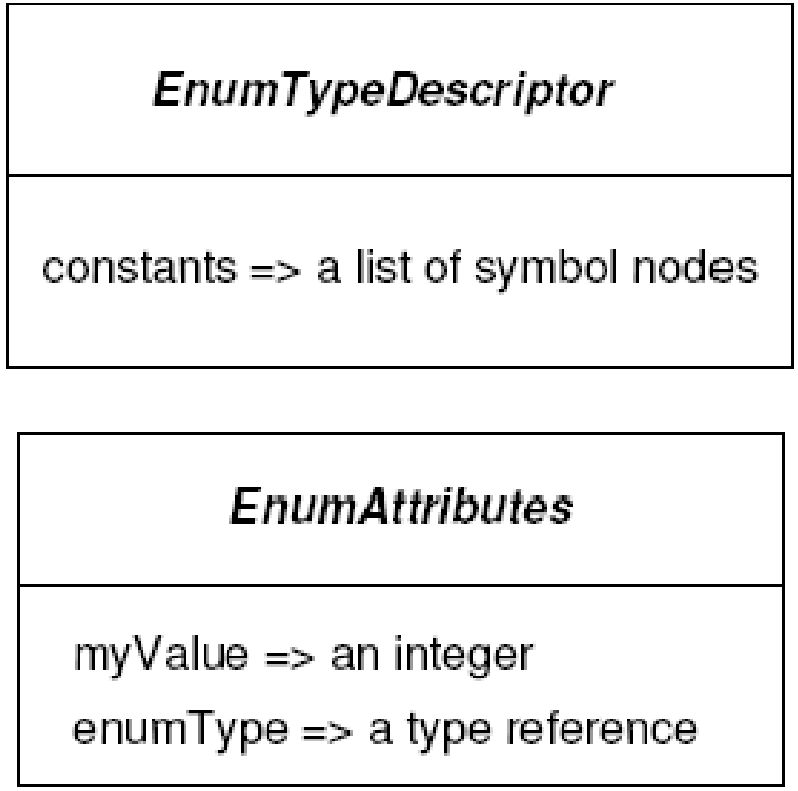


Figure 8.25: Type and Attribute Descriptor Objects for Enumerations

```
procedure Visit(EnumDefining enumdef)
   typeRef := new typeDescriptor(enumType)
   nextval := 0
   foreach id in enumdef.constantNames do
      if currentSymbolTable.DeclaredLocally(id.name)
      then call error("Name cannot be redeclared.")
           id.type           := errorType
           id.attributesRef := null
      else attr              := new Attributes(enumAttributes)
           attr.enumType     := typeRef  // type descriptor
           attr.myVlaue      := nextval
           nextval           := nextval + 1
           call currentSymbolTable.EnterSymbol(id.name, attr)
           id.type           := typeRef
           id.attributesRef := attr
           call typeRef.AppendToConstList(id.name, attr)
   enumdef.type := typeRef
end
```

Figure 8.23 The Visit method in TypeVisitor for EnumDefining.

The `Visit` method for an enumeration type is shown in Figure 8.23.

The final result of the `Visit` method is the type descriptor `typeRef` for the enumeration type, shown in Figure 8.25. In addition the enumeration constants are put into the current symbol table. Each such entry includes an integer value for the constant and a pointer to the type descriptor of the enumeration type. The type descriptor is a list of identifiers together with their respective `Attributes`.

Figure 8.26 is the type descriptor for the enumeration type

```
( Red, Yellow, Blue, Green )
```

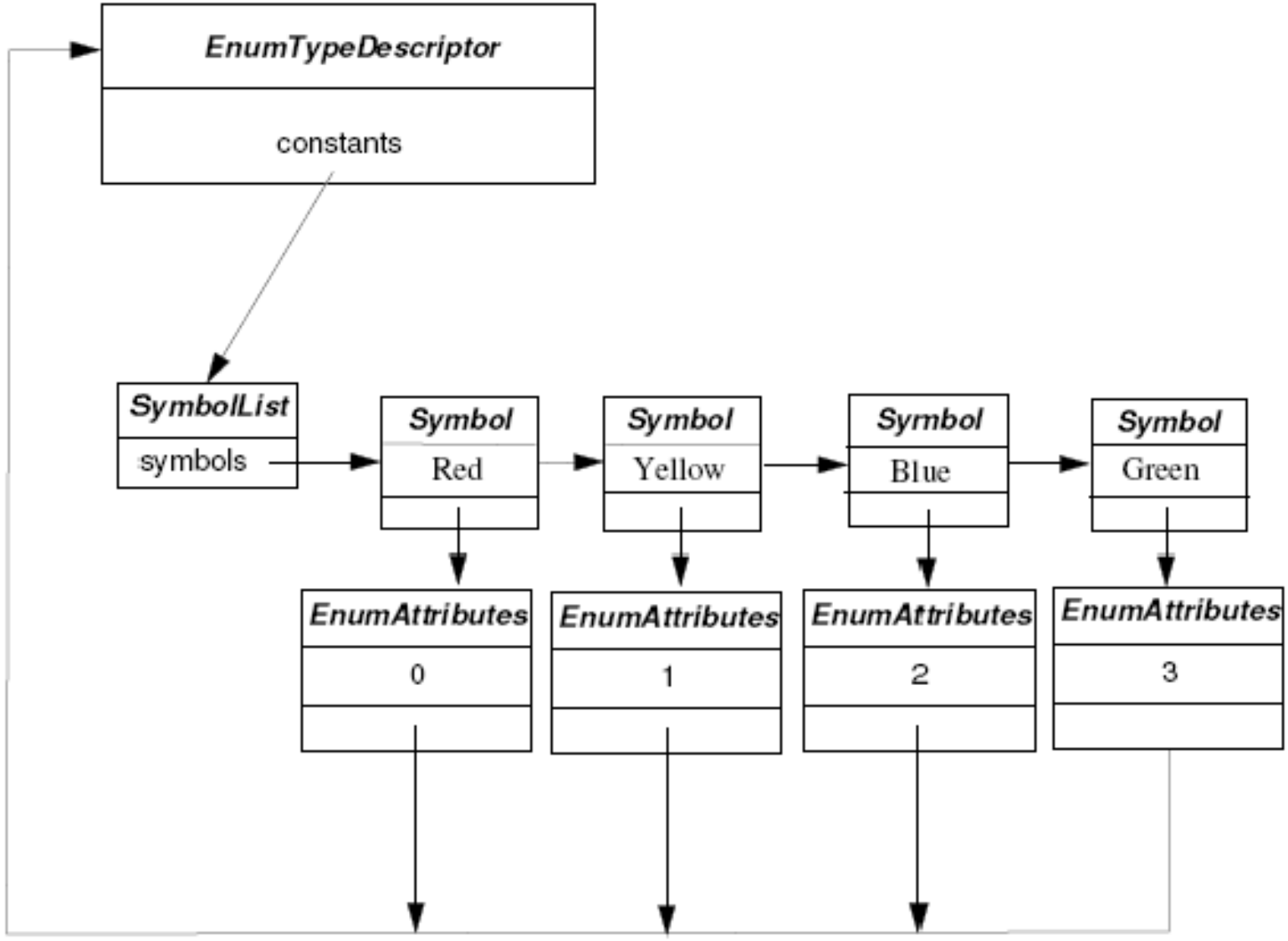


Figure 8.26: Representation of an Enumeration Type

§8.7 Class and method declarations

```
class A extends B implements C, D, E { .  .  .  }
```

A class definition in Java or other similar languages is very similar to a structure definition, with additional complexity.

Though a class is a type, when defining a class, a programmer must provide a name[a] (in contrast, a structure need not have a name):

```
var a :  struct { b :  integer; }
class ccc { .  .  .  }
```

Thus, for a class definition, a compiler needs to enter the class name into the symbol table in addition to creating a type descriptor for the class.

[a] Java allows anonymous classes.

Unlike a structure, a class may include method declarations. A method needs a *signature*, which comprises the types of the parameters and of the return value. The signature is constructed when a method declaration is processed. It is stored in the entry for the method name in the symbol table.

Example. The signature of a method: `foo(III)I`.

The signatures are used in checking the types of expressions. Type checking is performed with a visit through the AST. During type checking we need to know the class and the method that is currently being compiled. The information is readily available in the AST. We assume the following public, global functions that provide the necessary information:

1. `procedure SetCurrentClass(ClassAttributes c)`
2. `procedure GetCurrentClass returns ClassAttributes`
3. `procedure SetCurrentMethod(MethodAttributes m)`
4. `procedure GetCurrentMethod returns MethodAttributes`
5. `procedure SetCurrentConstructor(MethodAttributes m)`
6. `procedure GetCurrentConstructor returns MethodAttributes`
7. `procedure GetRefToObject returns reference`

§8.7.1 Processing class declarations

The AST for a class declaration is shown in Figure 8.27. It includes a name, modifiers, a list of variable declarations, a list of constructor declarations, a list of method declarations, and a parent class (assuming single inheritance). There could also be *destructors*. Possible modifiers for a class in Java include `abstract` and `final`. A class may also implement a list of *interfaces*, which are not included in Figure 8.27.

```
class A implements foo, listener { .  .  .  }
abstract class A { .  .  .  }
final class A { .  .  .  }
```

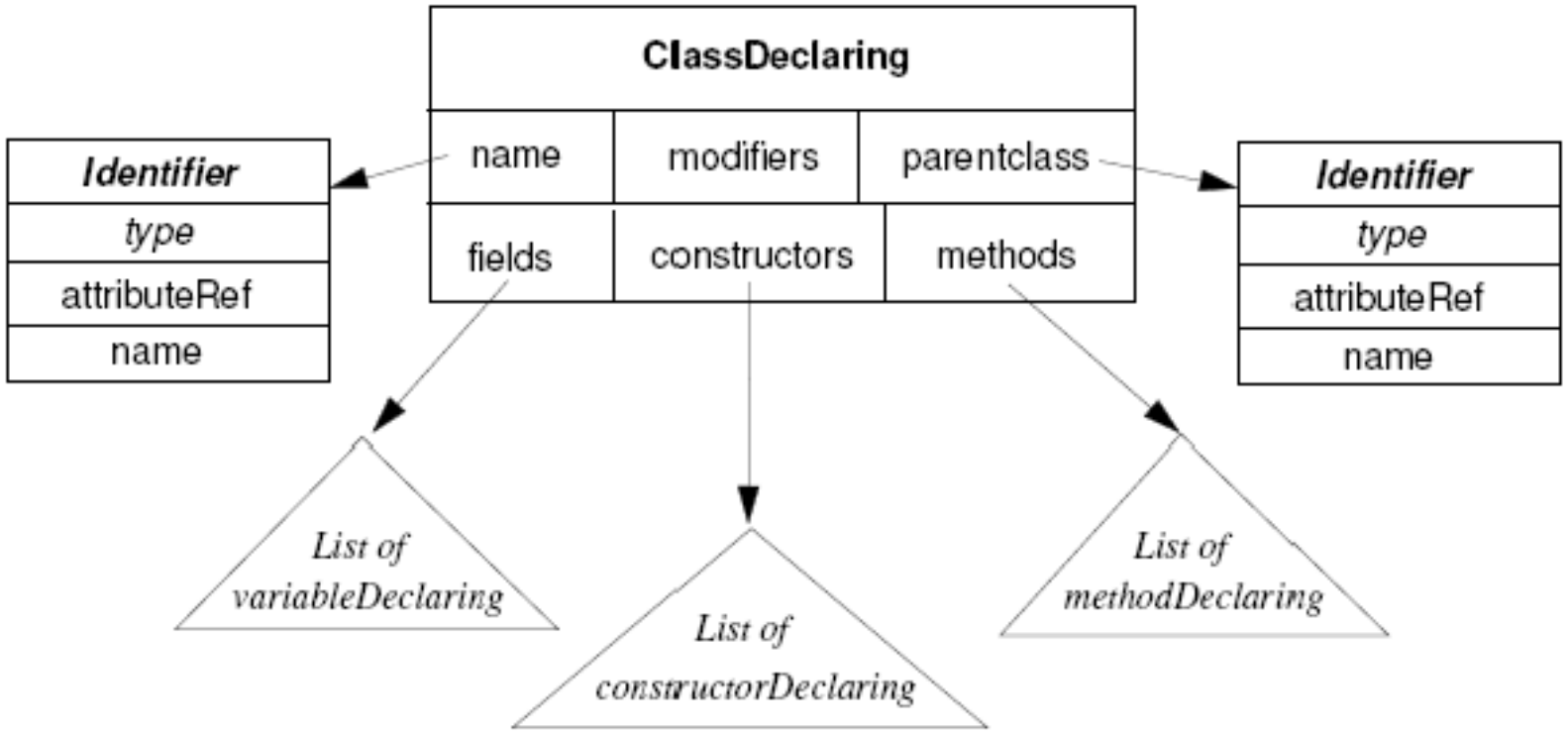


Figure 8.27: Abstract Syntax Tree for a Class Declaration

The `Visit` method in Figure 8.29 attempts to build the attributes and type descriptor for a class declaration, which is shown in Figure 8.28 from the AST.

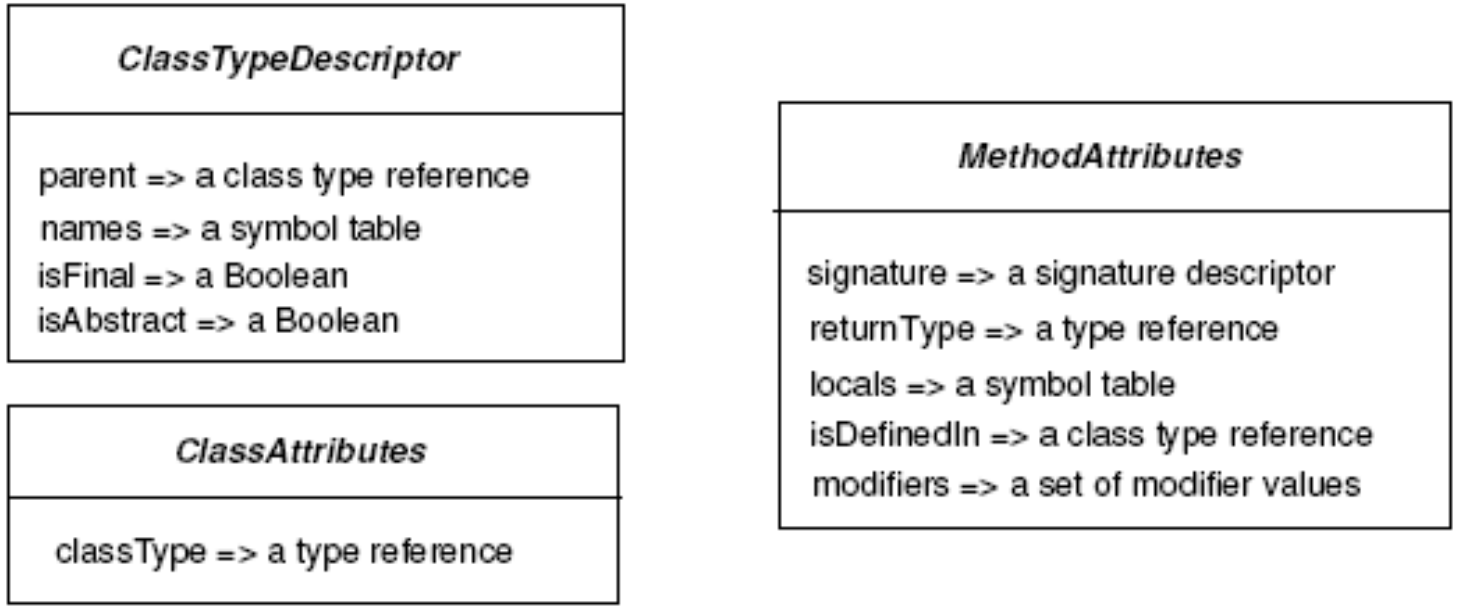


Figure 8.28: Attributes and Type Descriptor for Class Declarations

```
proceudre Visit(ClassDeclaring cd)
   typeRef              := new typeDescriptor(classType)
   typeRef.names        := new SymbolTable()
   attr                 := new Attributes(ClassAttributes)
   attr.classType       := typeRef
   call currentSymbolTable.EnterSymbol(cd.name.name, attr)
   call SetCurrentClass(attr)
   if cd.parentclass == null
   then cd.parentclass := GetRefToObject() // the Object class
   else typeVisitor    := new TypeVisitor()
        call cd.parentclass.Accept(typeVisitor)
   if cd.parentclass.type == errorType
   then attr.classtype := errorType
   else if cd.parentclass.type.kind != classType
        then attr.classtype      := errorType
             callError(cd.parentclass.name, "is not a class name.")
        else // check the parent class is not a final class
          typeRef.parent      := cd.parentclass.attributeRef
          typeRef.isFinal     := MemberOf(cd.modofiers, final)
```

```
            typeRef.isAbstract  := MemberOf(cd.modofiers, abstract)
            call typeRef.names.Incorporate(cd.parentclass.type.names)
            call OpenScope(typeRef.names)
            call cd.fields.Accept(this)
            call cd.constructors.Accept(this)
            call cd.methods.Accept(this)
            call CloseScope()
      call SetCurrentClass(null)  // assume classes are not nested.
      // check all methods are defined for non-abstract classes
      // check all interfaces implemented by this class
end
```

Figure 8.29 The Visit method in TopDeclVisitor for ClassDeclaring

The `Visit` method first creates a new empty type descriptor for the class. The type descriptor includes an empty small symbol table for the names declared in `this` class. The name of the class is entered into the current symbol table.

The parent class of the current class is also determined. The `Object` class serves as the parent if no parent class is specified by the programmer. If a parent class is specified, a new `typeVisitor` is created to visit the parent class. If anything is wrong during the visit, the parent class is set to `errorType`.

The `Incorporate` method makes the names declared in the parent class visible when the body of the current class is processed (say, by adding the names of the parent's symbol table to the current symbol table).

The compiler will set up the `isFinal` and `isAbstract` fields of the class.

The `OpenScope` procedure makes the symbol table of the current class as the new scope.

After a new scope is opened, the compiler will process the fields, constructors, and the methods declared in the class.

When everything is done, the current symbol table is popped off the stack of symbol tables (by the `CloseScope()` call). The `CurrentClss` variable is set to null. On the other hand, if the programming language allows nested classes, the `CurrentClss` variable is reset to a previous value.

The modifiers (`final`, `abstract`) add restrictions to the usage of class names. For instance, a compiler has to check if a parent class is declared as `final`. Also the name of an `abstract` class is not used in a constructor.

There is a separate symbol table for each scope. The symbol tables are arranged in a stack, shown in Figure 8.3. This small-table approach is more appropriate for object-oriented languages.

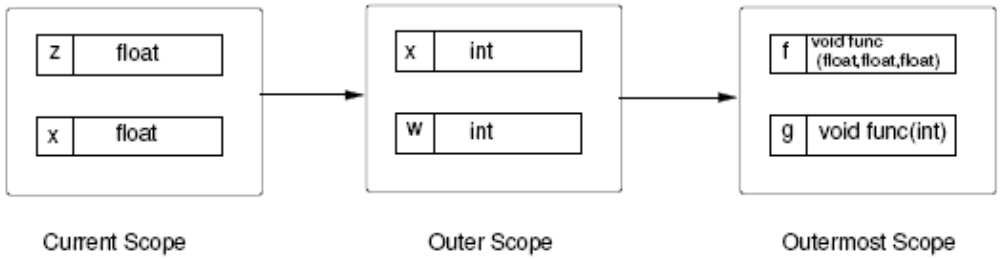


Figure 8.3: A stack of symbol tables, one per scope

To look for a name, we search the symbol tables from the stack top down to the bottom. Furthermore, due to the `Incorporate` method for the class inheritance, after searching the symbol table of a class, we also need to search the symbol tables of the ancestor classes along the inheritance chain.

Multiple inheritance complicates things even further.

```
class A extends B, C implements D, E { .  .  .  }
```

A class is associated with a list of *names* of the interfaces implemented by the class in Java. Each interface has some declarations, which are collected in a separate symbol table. After the declarations in a class are all processed, the (symbol tables of all) interfaces must be checked one by one in order to ensure that all the interfaces are fully implemented by the class.

The list of implemented interfaces must be a field of the class so that further type-checking may be performed when an instance of the class is used as an instance of the interface.

§8.7.2 Processing method declarations

The AST for a method is shown in Figure 8.30. A method has a name, a list of parameters, a body, an optional return type, and some modifiers.

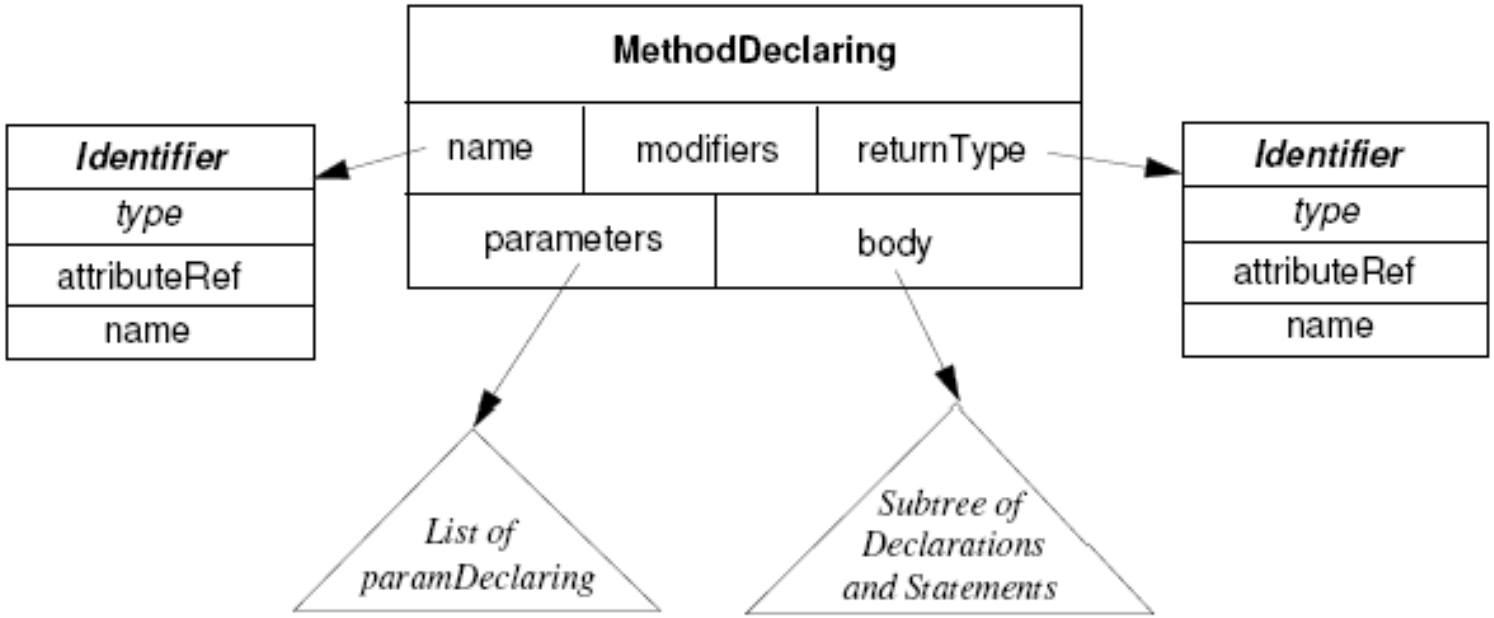


Figure 8.30: Abstract Syntax Tree for a Method Declaration

The method name is entered into the current symbol table.

The local variables of a method are collected in a new symbol table (for the method).

The return type is processed by the `Accept` call with an instance of the `TypeVisitor`. This `Accept` method searches the current symbol table for the `identifier` node and sets the `type` field.

The return type is also stored in the `Attribute` of the method name.

An `Attributes` descriptor for the method is created, with the `returnType`, `modifiers`, `isDefinedIn`, and `locals` fields.

Because methods may be nested, the `Visit` method saves the `currentMethod` and sets up a new `currentMethod`. The `currentMethod` will be re-stored at the end of the `Visit` method.

The `Visit` method for a `ParameterDeclaring` node will handle a list of parameter names and their types. The parameter names are entered into the symbol table. A type descriptor is created for the type of the parameter. Finally the return type is added to the signature of the

method. The signature is recorded in the `Attributes` for the method.

The body of the method consists of a list of local variable declarations and a list of statements. In the call

```
md.body.Accept(this)
```

which is roughly equivalent to

```
this.visit(md.body)
```

when a declaration is encountered the appropriate `Visit` method declared within `TopDeclVisitor` will be invoked.

The `TopDeclVisitor` is a subclass of `SemanticsVisitor`. The `Visit` methods declared in `SemanticsVisitor` for a statement AST node will be invoked when the statement node is visited.

In Java, a method may also carry a list of exceptions that may be thrown in the method.

```
public Object pop() throws EmptyStackException, PException,
                           QException { . . . }
```

In a `MethodDeclaring` node, we should add a field for the list of exceptions and a `Visit` method for the exception list. In addition, in an `Attributes` node for a method, we should add a `declaredThrowList` field for the `Visit` method. See §9.1.7.

A constructor is similar to a method, but without a return type. A `Visit` method for a `ConstructorDeclaring` node looks very similar to a `Visit` method for a `MethodDeclaring` node.

```
procedure Visit(MethodDeclaring md)
   typeVisitor           := new TypeVisitor()
   call md.returnType.Accept(typeVisitor)
   attr                  := new Attributes(MethodAttributes)
   attr.returnType       := md.returnType.type
   attr.modifiers        := md.modifiers
   attr.isDefinedIn      := GetCurrentClass()
   attr.locals           := new SymbolTable()
        // Enter method name into currentSymbolTable.
   call currentSymbolTable.EnterSymbol(md.name.name, attr)
   md.name.attributeRef := attr
   call OpenScope(attr.locals)
   oldCurrentMethod      := GetCurrentMethod()
   call SetCurrentMethod(attr)
   call md.parameters.Accept(this)
   attr.signature        :=
        md.parameters.signature.AddReturn(attr.returnType)
   call md.body.Accept(this)
   call SetCurrentMethod(oldCurrentMethod)
```

```
    call CloseScope()
end
```

Figure 8-31 The Visit method in TopDeclaring for MethodDeclaring

§8.8 An introduction to type checking

When translating statements and expressions, a compiler (actually, a `SemanticsVisitor`) needs to check the types using the information in the symbol table. The information is collected when declarations are processed (see §8.6 and §8.7).

Consider assignment statements such as:

```
id := expr ;
myarray[ expr ] := expr ;
mystruct.myfield := expr ;
```

Its AST is shown in Figure 8.32. The left subtree denotes the target of assignment, which could be a simple variable, an element of an array, or a field in a structure. The right subtree denotes an expression whose value is assigned to the target. Uniform visitors will traverse both subtrees to determine the types.

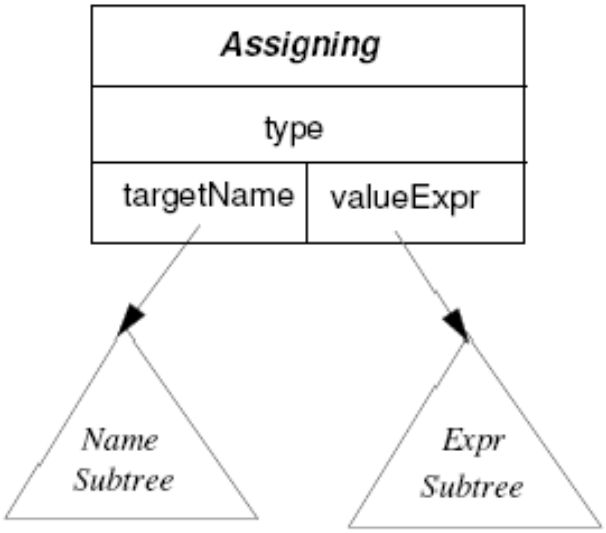


Figure 8.32: Abstract Syntax Tree for an Assignment

Left values and right values

A simple assignment, such as

```
a := b ;
```

Here `a` would denote the address of the variable `a` while `b` will denote the value of the variable `b`. The address is called the *left value* (L-value) and the value is the *right value* (R-value).

We use a `SemanticsVisitor` to analyze constructs that will produce an R-value. The constructs could be an identifier, a literal, a (binary or unary) operator, an element of an array, a field in a structure, or a function call.

We use a `LHSSemanticsVisitor` to analyze constructs that will produce an L-value. The constructs could be an identifier, an element of an array, or a field in a structure. A `LHSSemanticsVisitor` will check if the target is assignable.

For illegal constructs such as

```
3.14 := b ;        // LHS is a literal.
a + b * 3 := c ;   // LHS is an expression.
```

the *parser* will catch these errors. Thus, it is not necessary for a `LHSSemanticsVisitor` to repeat the same check.

The general structure of `SemanticsVisitor` and `LHSSemanticsVisitor` is shown in Figure 8.33.

```
class NodeVisitor
   procedure VisitChildren(n)
      foreach c in n.GetChildren() do call c.Accept(this)
   end
end

class SemanticsVisitor extends NodeVisitor
   procedure Visit(Identifier id) ... // 8.8.1

   procedure Visit(IntLiteral intlit)
      intlit.type := integertype
   end

   procedure Visit(Assigning assign) ... // 8.8.2
```

```
    procedure Visit(BinaryExpr bexpr)
       call VisitChildren(bexpr)
       bexpr.type := BinaryResultType(bexpr.operator,
             bexpr.leftExpr.type, bexpr.rightExprt.type)
    end

    procedure Visit(UnaryExpr uexpr)
       call VisitChildren(uexpr)
       uexpr.type := UnaryResultType(uexpr.operator,
             uexpr.subExpr.type)
    end

    procedure Visit(ArayReferencing ar) ... // fig 8.37

    procedure Visit(StructReferencing sr) ... // fig 8.38

end
```

```
class LHSSemanticsVisitor extends SemanticsVisitor
   procedure Visit(Identifier id)
      semvisitor := new SemanticsVisitor()
      call id.Accept(semvisitor)    // resolve the id
      if not IsAssignable(id.attributeRef)
      then call error("id is not assignable.")
           id.type          := errorType
           id.attributeRef := null
   end

   procedure Visit(ArrayReferencing ar)
      call ar.arrayName.Accept(this)
      // the whole array is assignable
      semvisitor := new SemanticsVisitor()
      call ar.Accept(semvisitor)
   end
```

```
procedure Visit(StructReferencing sr)
   semvisitor := new SemanticsVisitor()
   call sr.Accept(semvisitor)
   if sr.type != errorType
   then call sr.objectName.Accept(this)
        // the whole obj is assignable
        st := sr.objectName.type.fields
        attributeRef := st.RetreiveSymbol(fieldName.name)
        if not IsAssignable(attributeRef)
        // Is the field assignable?
        then call error("field is not an assignable field.")
end
```

Figure 8.33 Type checking visitors

The class hierarchy is as follows:

NodeVisitor → SemanticsVisitor → LHSSemanticsVisitor

§8.8.1 Simple identifiers and literals

Here is the `Visit(Identifier)` procedure in the `SemanticsVisitor`.
It handles situations as

```
. . . := b ;
```

```
procedure Visit(Identifier id) ... // in SemanticVisitor 8.8.1
   id.type          := errorType
   id.attributeRef  := null
   attributeRef     := currentSymbolTable.RetrieveSymbol(id.name)
   if attributeRef == null
   then call error("Identifier is not declared.")
   else if IsDataObject(attributeRef)
        then id.attributeRef := attributeRef
             id.type         := id.attributeRef.variableType
        else call error("Identifier does not name a data object")
end
```

The `Visit` method for an identifier looks up the identifier in the symbol table.

The identifier must be found in the symbol table and it must denote a data object otherwise an error message is issued. If everything goes well, `id.attributeRef` and `id.type` will be set up properly.

Note that all expressions (which have values) are represented by *data objects*.

Identifiers on the left-hand side

The `Visit` method for `identifier` in `LHSSemanticsVisitor` (Figure 8.33) will invoke the `Visit` method in `SemanticsVisitor` first (though `id.Accept(visitor)`).

Then it will check if the `id.attributeRef` is assignable.

For example, if `id.attributeRef` is null (that is, `id` is not found in the current symbol table) or if `id` denotes a constant/type/method/class, it is not assignable.

The `Visit` method for a `literal` is quite trivial.

```
class LHSSemanticsVisitor extends SemanticsVisitor
   procedure Visit(Identifier id)
      semvisitor := new SemanticsVisitor()
      call id.Accept(semvisitor)    // resolve the id
      if not IsAssignable(id.attributeRef)
      then call error("id is not assignable.")
           id.type          := errorType
           id.attributeRef := null
   end
```

Figure 8.33 Type checking visitors

§8.8.2 Assignment statements See Figure 8.32 (p. 160).

```
procedure Visit(Assigning assign) // in the SemanticsVisitor class
   // process left child of the assignment
   lhsVisitor := new LHSSemanticsVisitor()
   call assign.targetname.Accept(lhsVisitor)
   // process right child of the assignment
   call assign.valueExpr.Accept(this)
   // perform type checking for assignment
   if Assignable(assign.valueExpr.type, assign.targetName.type)
   then assign.type := assign.targetName.type
   else call error("assignment type error.")
        assign.type := errorType
end
```

For an assignment statement, a `LHSSemanticsVisitor` infers the type of the LHS (i.e., the target of assignment) and a `SemanticsVisitor` infers the type of the RHS expression. Finally `Assignable` checks if the RHS can be assigned to the LHS according to the type compatibility rule.

§8.8.3 Checking expressions

Complex expressions are built from simpler ones with binary and unary operators, such as

```
a + b
  - c
```

We will consider binary and unary expressions, whose AST are shown in Figure 8.35.

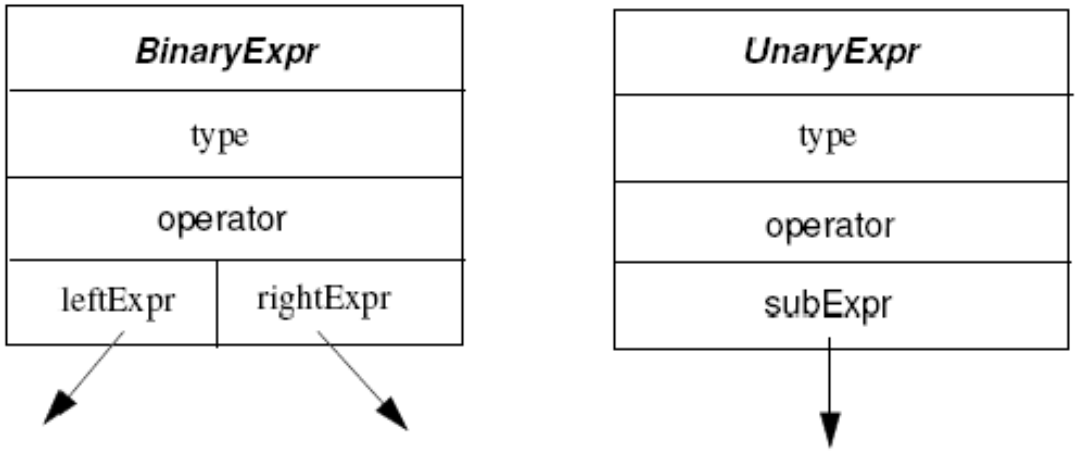
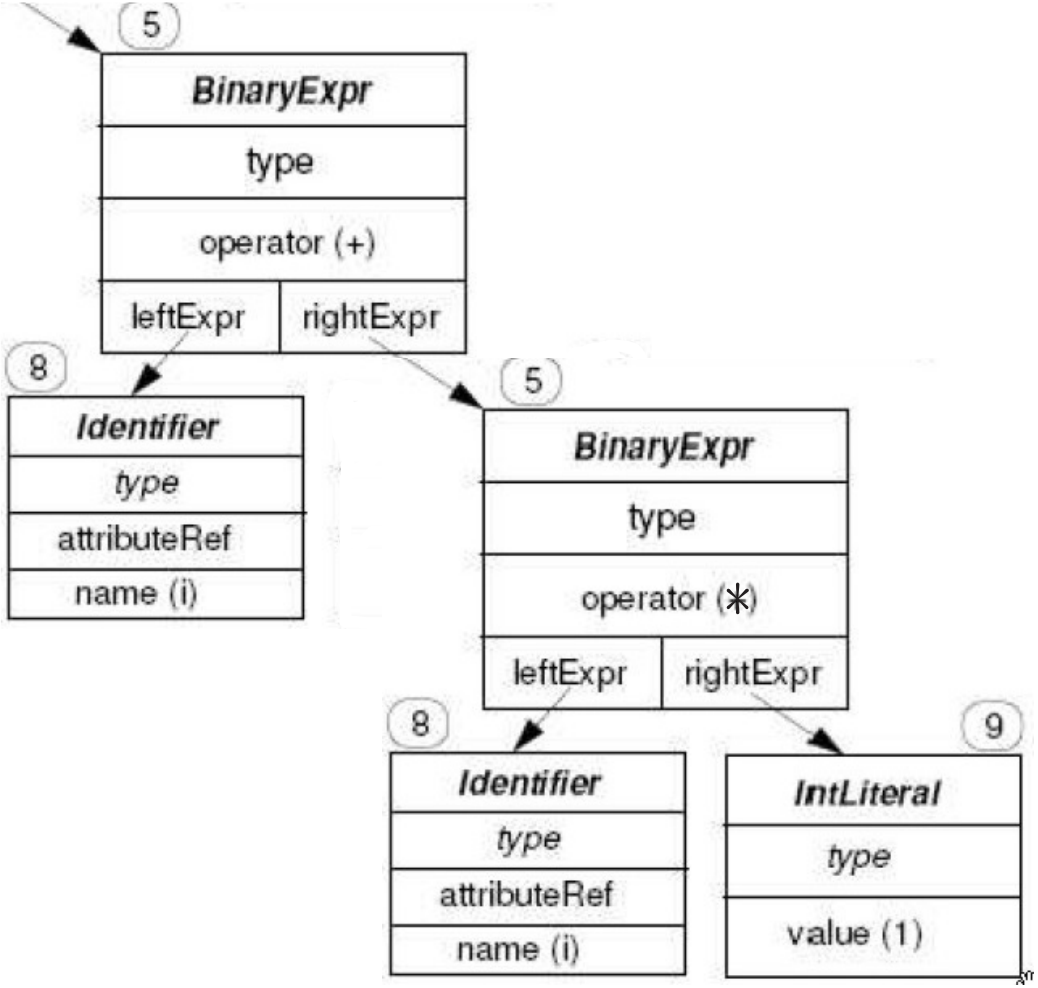


Figure 8.35: Abstract Syntax Tree Representations for Unary and Binary Expressions

Example. The AST for the expression "`i + i * 1`" is shown in Figure 8.39.


5
BinaryExpr
type
operator (+)
leftExpr
rightExpr
8
Identifier
type
attributeRef
name (i)
5
BinaryExpr
type
operator (*)
leftExpr
rightExpr
8
Identifier
type
attributeRef
name (i)
9
IntLiteral
type
value (1)

```
class SemanticsVisitor extends NodeVisitor

   procedure Visit(BinaryExpr bexpr)
      call VisitChildren(bexpr)
      bexpr.type := BinaryResultType(bexpr.operator,
            bexpr.leftExpr.type, bexpr.rightExprt.type)
   end

   procedure Visit(UnaryExpr uexpr)
      call VisitChildren(uexpr)
      uexpr.type := UnaryResultType(uexpr.operator,
            uexpr.subExpr.type)
   end

end
```

Figure 8.33 Type checking visitors for binary and unary operators

The `Visit` methods for the binary and unary expressions are shown in Figure 8.33. They first visit the children and call the `BinaryResultType` and `UnaryResultType` methods, respectively, to determine the type of the whole expression.

The `BinaryResultType` and `UnaryResultType` methods enforce the type rules of the programming language. For instance, adding two integers results in an integer, while adding an integer and a float results in a float (if the language allows implicit type conversion) or `errorType` otherwise. On the other hand, adding an integer and a boolean result in `errorType`. Comparing two integers results in a boolean.

For a complex instruction, such as the one in the previous figure, we will have recursive calls:

`Visit(BinaryExpr)` → `VisitChildren(BinaryExpr)` → `BinaryExpr.Accept(this)` → `Visit(BinaryExpr)` → `VisitChildren(BinaryExpr)` → `Identifier.Accept(this)` → `Visit(Identifier)` → `currentSymbolTable.RetrieveSymbol(id.name)`

The recursive calls return from bottom to the root. This agrees with our common understanding that an expression is built from simpler expressions.

Exercise. How to extend the above class in order to handle ternary operators, such as "`?:`" in C? Design a suitable AST for ternary operators.

§8.8.4 Checking complex names

It is common to reference array elements and fields in a structure:

```
myarray[i+5] + myrec.myfield.foo
```

Their AST are shown in Figure 8.36. Note that the `arrayName` and `objectName` need not always be an `Identifier` node.

The operators for array elements and fields are left-associative. For example,

- `a[b][c]` is interpreted as `(a[b])[c]`.
- `a.b.c` is interpreted as `(a.b).c`.
- `a[b].c.d[e]` is interpreted as `(((a[b]).c).d)[e]`.

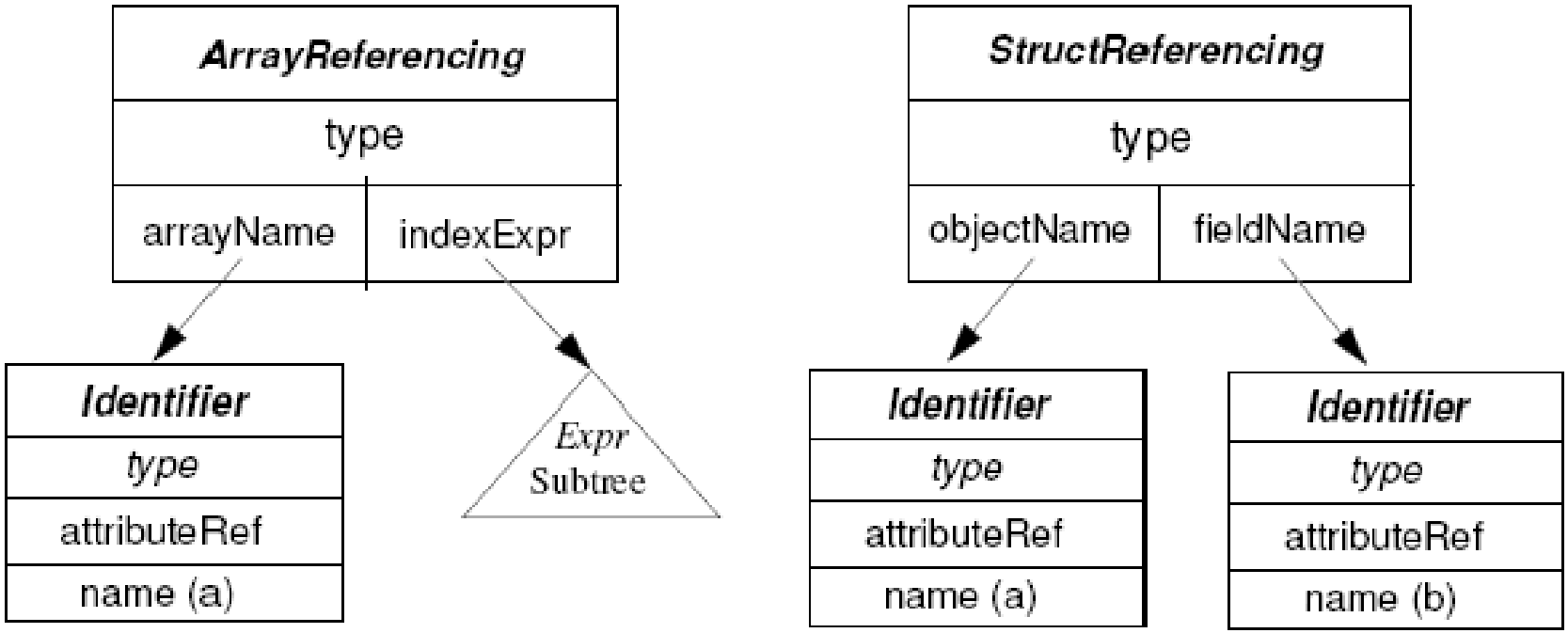


Figure 8.36: Abstract Syntax Trees for Array and Struct References

For example, assume

```
var qoo:  array [ 1 ..  10 ] of struct account;
```

then

```
qoo:  array of struct account
```

```
i:  int
```

```
qoo[i]:  struct account
```

```
procedure Visit(ArrayReferencing ar) // in SemanticsVisitor
   call VisitChildren(ar)
   if ar.arrayName.type == errorType
   then ar.type := errorType
   else if ar.arrayName.kind != arrayTypeDescriptor
        then call error("The name is not an array.")
             ar.type := errorType
        else ar.type := ar.arrayName.type.elementType
   if ar.indexExpr.type != errorType and
      ar.indexExpr.type != integer
   then call error("Index is not an integer")
end
```

Figure 8.37 Checking an array reference in SemanticsVisitor

The `VisitChildren(ar)` call determines the types of the `arrayName` and the `indexExpr`. After checking that the `arrayName` refers to an array and the `indexExpr` is an integer expression, the type of the element reference is the type of the element with the statement

```
ar.type := ar.arrayName.type.elementType
```

The code generation phase will generate code to check the index is within the proper bounds.

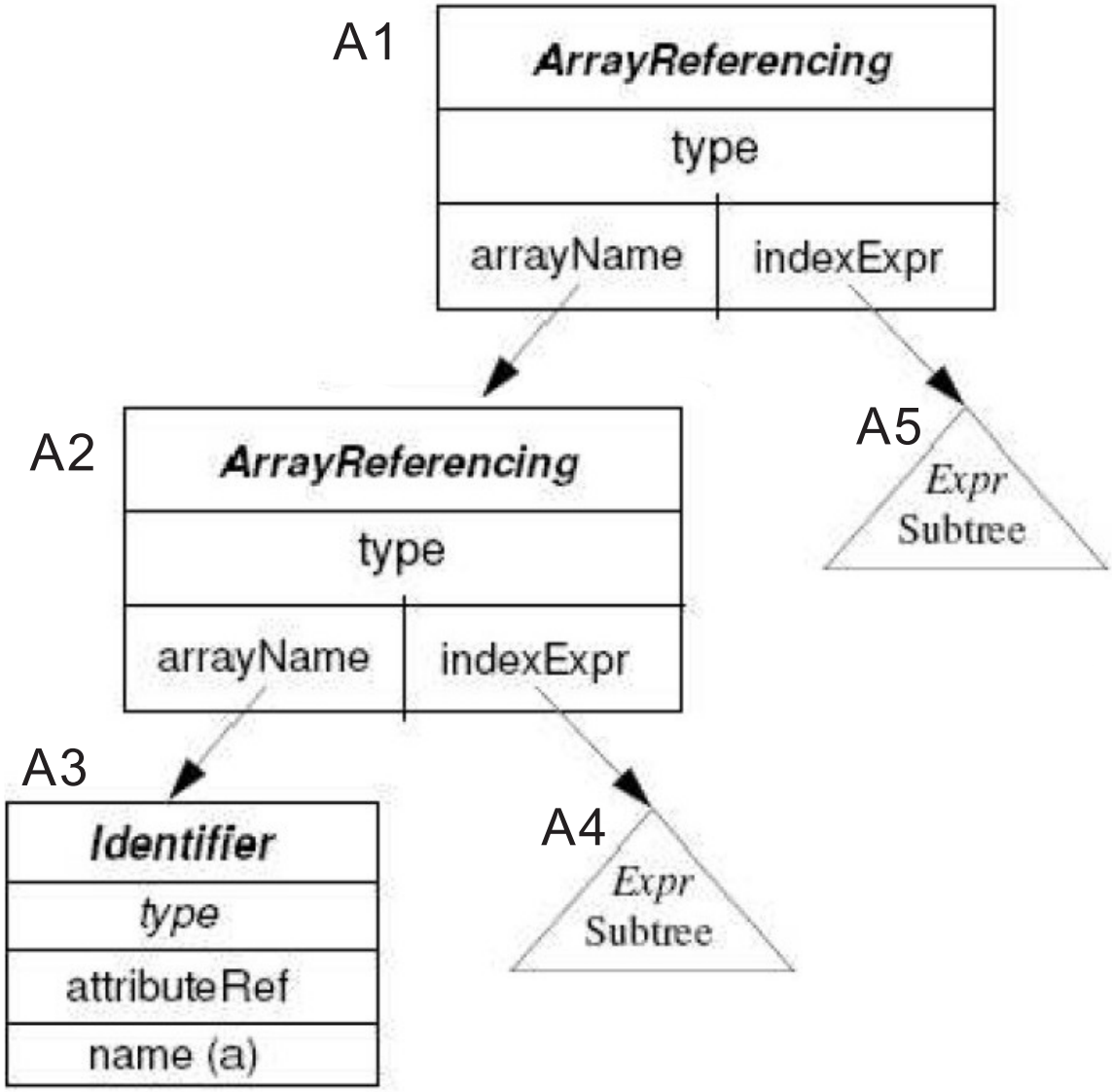
Some languages may allow array indices of other types.

In the above code, we assume the index must have the integer type. Some programming languages allow more flexible enumeration types for indices. The above code must be modified accordingly.

Array references on the left-hand side

An array reference could appear on the left-hand side of an assignment statement and could be more complex:

```
myarray[3][5] := 7 ;
```


A1
ArrayReferencing
type
arrayName
indexExpr
A2
ArrayReferencing
type
arrayName
indexExpr
A5
Expr
Subtree
A3
Identifier
type
attributeRef
name (a)
A4
Expr
Subtree

```
class LHSSemanticsVisitor extends SemanticsVisitor

   procedure Visit(ArrayReferencing ar)
      call ar.arrayName.Accept(this)
      // the whole array is assignable
      semvisitor := new SemanticsVisitor()
      call ar.Accept(semvisitor)
   end
```

Figure 8.33 Type checking visitors

In Figure 8.33, the `Visit(ArrayReferencing ar)` method in the `LHSSemanticsVisitor` class first uses the `ar.arrayname.Accept(this)` call to call the `Visit(ArrayReferencing ar)` method (in Figure 8.33) in the `LHSSemanticsVisitor` class.

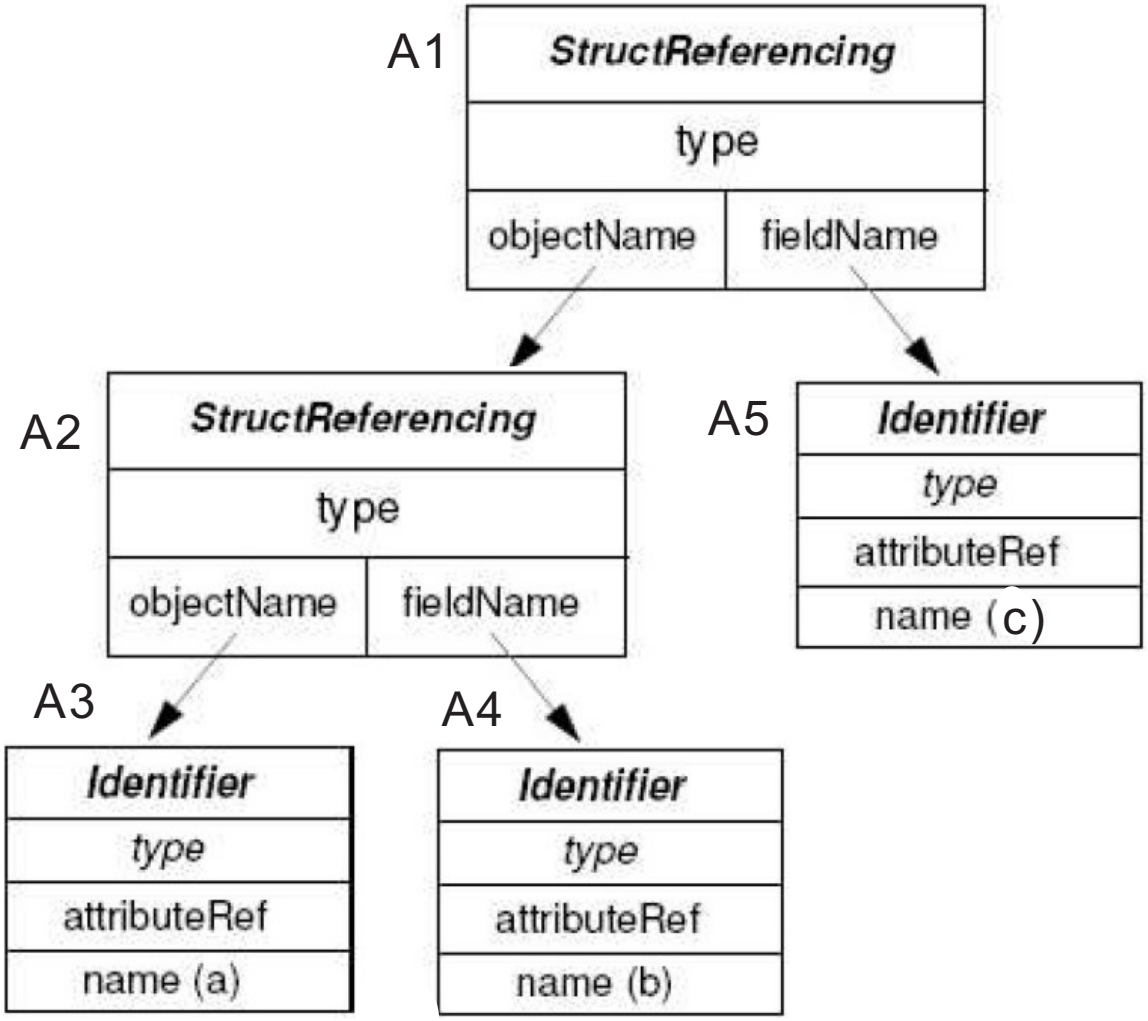
```
A1.arrayName == A2
A1.arrayName.arrayName == A2.arrayName == A3
```

```
LHS-Visit(A1) →
(P1) A1.arrayname.Accept(this) → LHS-Visit(A2) →
(Q1) A2.arrayname.Accept(this) → LHS-Visit(A3) →
(R1) A3.Accept(visitor) → Visit(Identifier A3) →
currentSymbolTable.RetrieveSymbol(id) (p.  327),
(R2) then check if A3 is assignable.  →
(Q2) A2.Accept(visitor) → Visit(A2) (Fig 8.37) →
→ VisitChildren(A2)
→ A2.type := A2.arrayName.type.elementType →
(P2) A1.Accept(visitor) → Visit(A1) (Fig 8.37) →
→ VisitChildren(A1)
→ A1.type := A1.arrayName.type.elementType
```

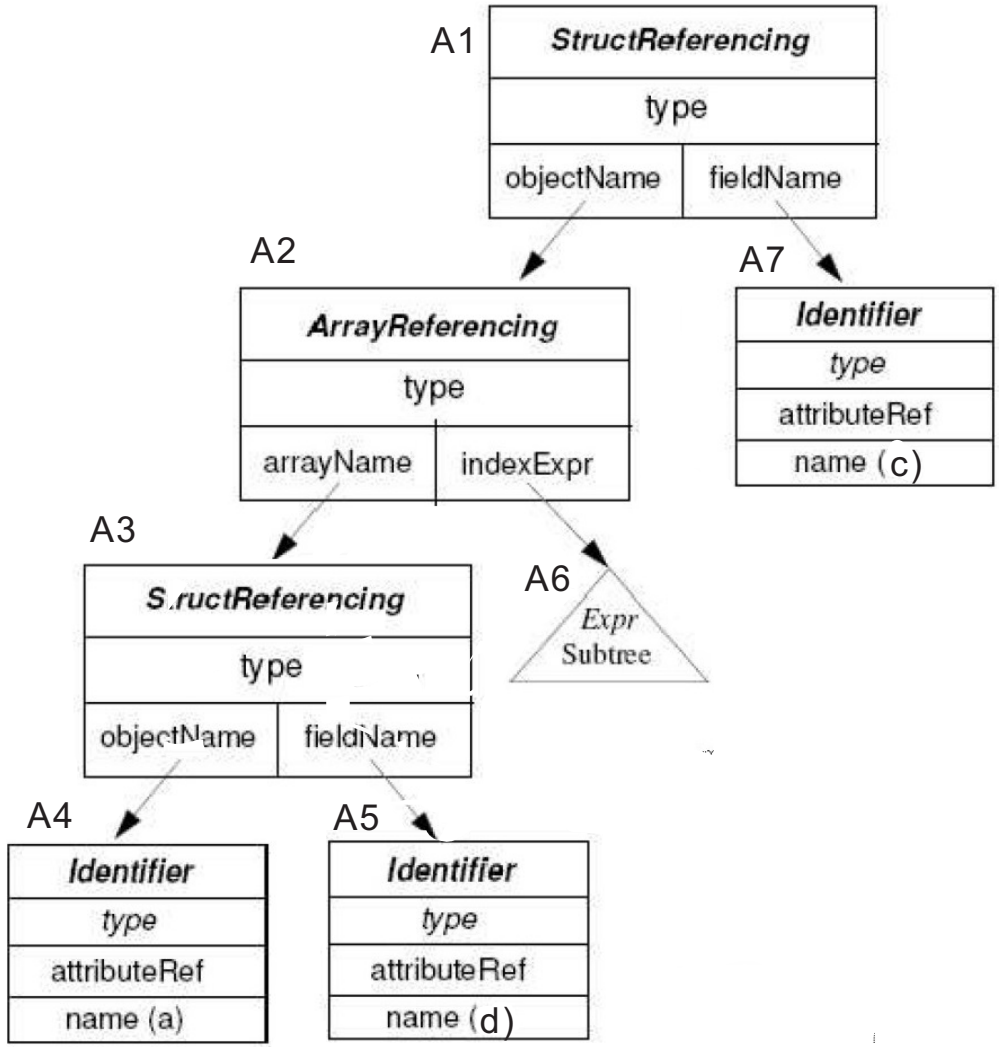
The `Vist(StructReferencing)` is similar. A structure reference could be more flexible, such as

```
a.b.c
a.d[3].c
```

```
(a.b).c
```


A1
StructReferencing
type
objectName
fieldName
A2
StructReferencing
type
objectName
fieldName
A5
Identifier
type
attributeRef
name (c)
A3
Identifier
type
attributeRef
name (a)
A4
Identifier
type
attributeRef
name (b)

```
((a.d)[3]).c
```


A1
StructReferencing
type
objectName
fieldName
A2
ArrayReferencing
type
arrayName
indexExpr
A7
Identifier
type
attributeRef
name (c)
A3
StructReferencing
type
objectName
fieldName
A6
Expr
Subtree
A4
Identifier
type
attributeRef
name (a)
A5
Identifier
type
attributeRef
name (d)

Structure referencing on the right-hand side

```
...  := a.b;
```

```
procedure Visit(StructReferencing sr)
   call sr.objectName.Accept(this)
   if sr.objectName.type == errorType
   then sr.type := errorType
   else if sr.objectName.kind != strcutTypeDescriptor
        then call error("The identifier is not a structure.")
             sr.type := errorType
        else st := sr.objectName.type.fields // a symbol table
             attributeRef := st.RetrieveSymbol(fieldName.name)
             if attributeRef == null
             then call error("The name is not a field.")
                  sr.type := errorType
             else sr.type := attributeRef.fieldType
end
```

Figure 8.38 Checking a structure reference in SemanticsVisitor

When referencing a field of a structure, we obtain the symbol table (i.e., `sr.objectName.type.fields`) for the structure (which contains all the fields in the structure).

Then the `st.RetrieveSymbol(fieldName.name)` call looks up the `fieldName.name` in the symbol table and returns an `attributeRef`.

Finally, the type of the whole expression is the type of the field with the statement

```
sr.type := attributeRef.fieldType
```

Structure referencing on the left-hand side

```
class LHSSemanticsVisitor extends SemanticsVisitor

   procedure Visit(StructReferencing sr)
      semvisitor := new SemanticsVisitor()
      call sr.Accept(semvisitor)
      if sr.type != errorType
      then call sr.objectName.Accept(this)
           // the whole obj is assignable
           st := sr.objectName.type.fields
           attributeRef := st.RetreiveSymbol(fieldName.name)
           if not IsAssignable(attributeRef)
           // Is the field assignable?
           then call error("field is not an assignable field.")
   end
```

Figure 8.33 Type checking visitors

Structure referencing on the left-hand side

```
a.b := ...  ;
```

The `Visit(StructReferencing)` method in the `LHSSemanticVisitor` class (Figure 8.33) first calls the `Visit(StructReferencing)` method in the `SemanticVisitor` class (Figure 8.38) via the call

```
visitor := new SemanticVisitor()
sr.Accept(visitor)
```

Then it repeats the work

```
sr.objectName.Accept(this)
st := sr.objectName.type.fields
attributeRef := st.RetrieveSymbol(fieldName.name)
```

Finally, it checks if `attributeRef` is assignable. Repeating the work is necessary in order to make sure that all the relevant components in a complex structure reference, such as

```
p.q[5].r.s[3].t := ...
```

are assignable.

A complex name example

Figure 8.39 is the AST for the expression `(s.a)[i+1].f`:

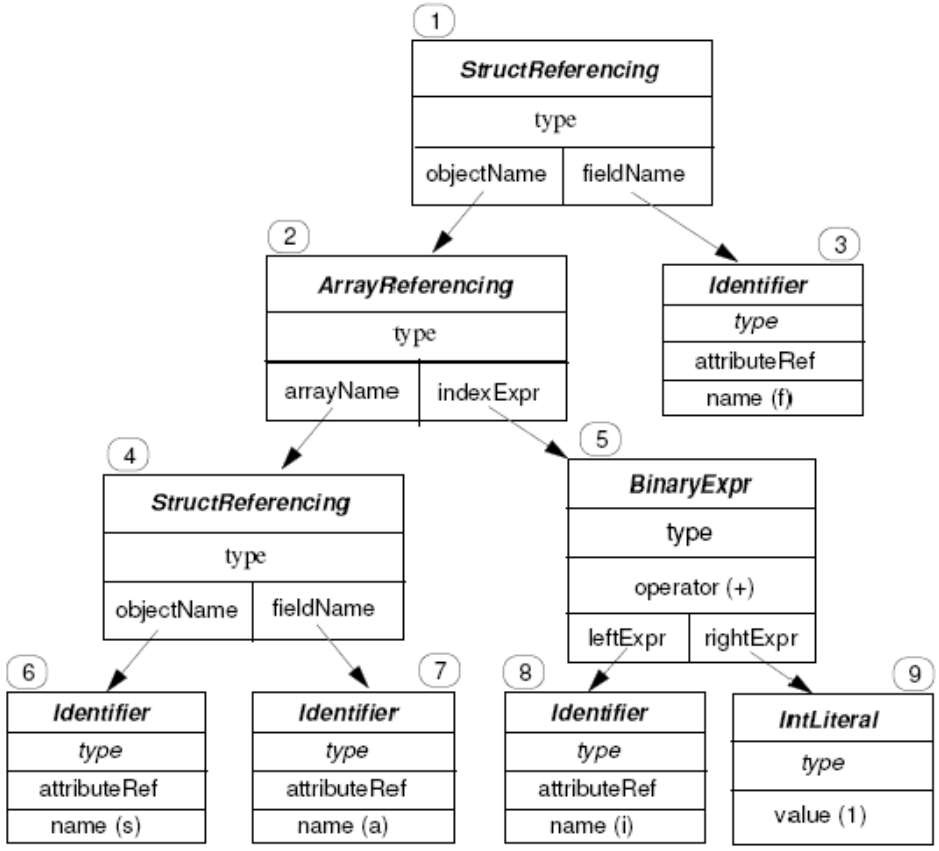


Figure 8.39: AST for Array and Struct Example

Example. Please draw the abstract syntax for the following reference:

```
a.b(c[f.g[k/3]].e).q[m]
```

Here `a` is an objeect, `b` is a member function (which return an object), `c` is an array (of objects), `f` is an object, `g` is an array, `k` is an integer, `e` is an field, `q` is a field having an array type, and `m` is an integer. Assume all these variables are declared properly in the program.

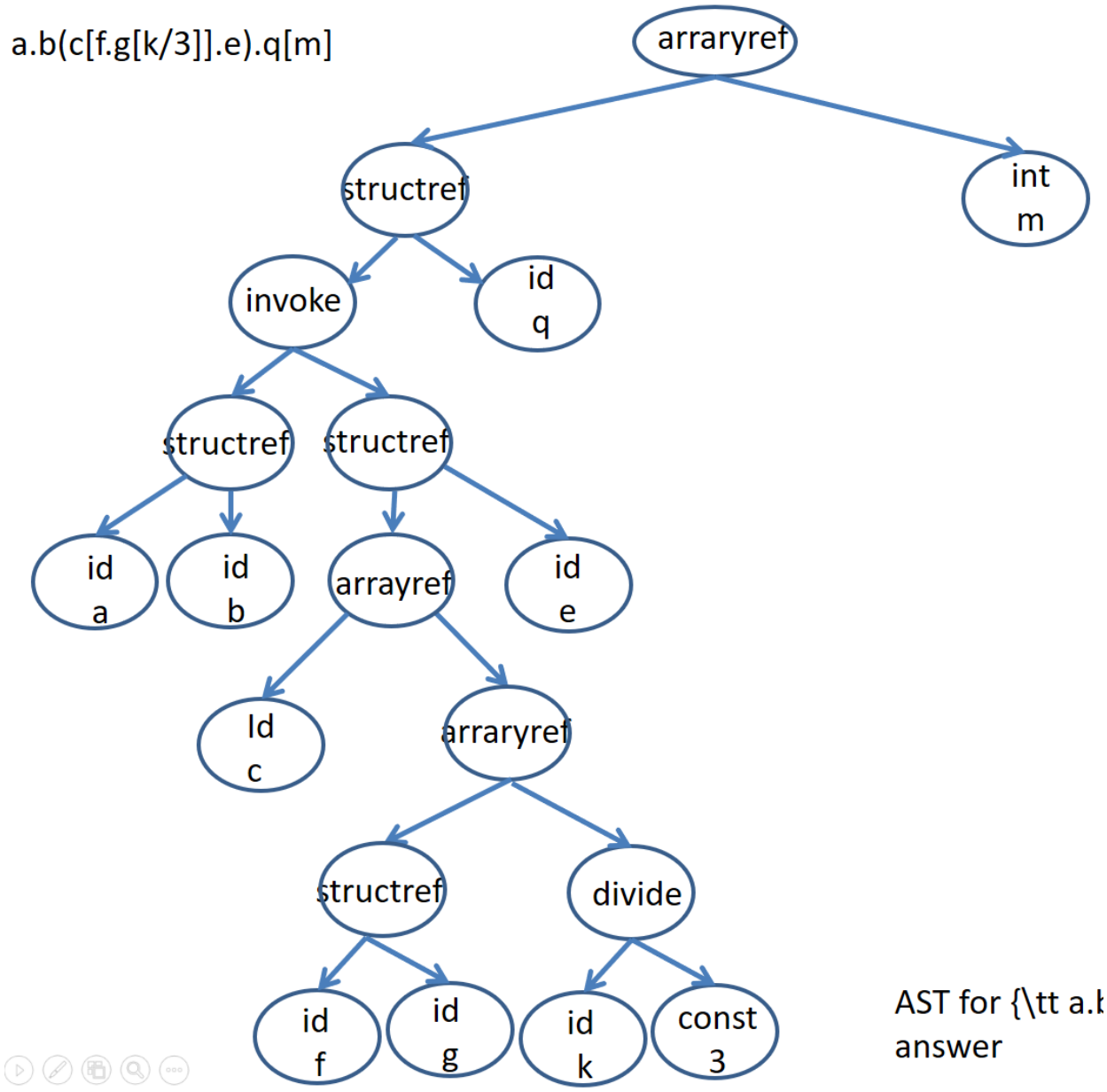


Figure 7: AST for question 5

Finally this is the end of Chapter 8.

Appendix Left value and right value in a compiler

```
int a = 5;
```

`a` has both a left value (its address) and a right value (its contents).

```
const in b = 7;
```

`b` has only a right value.

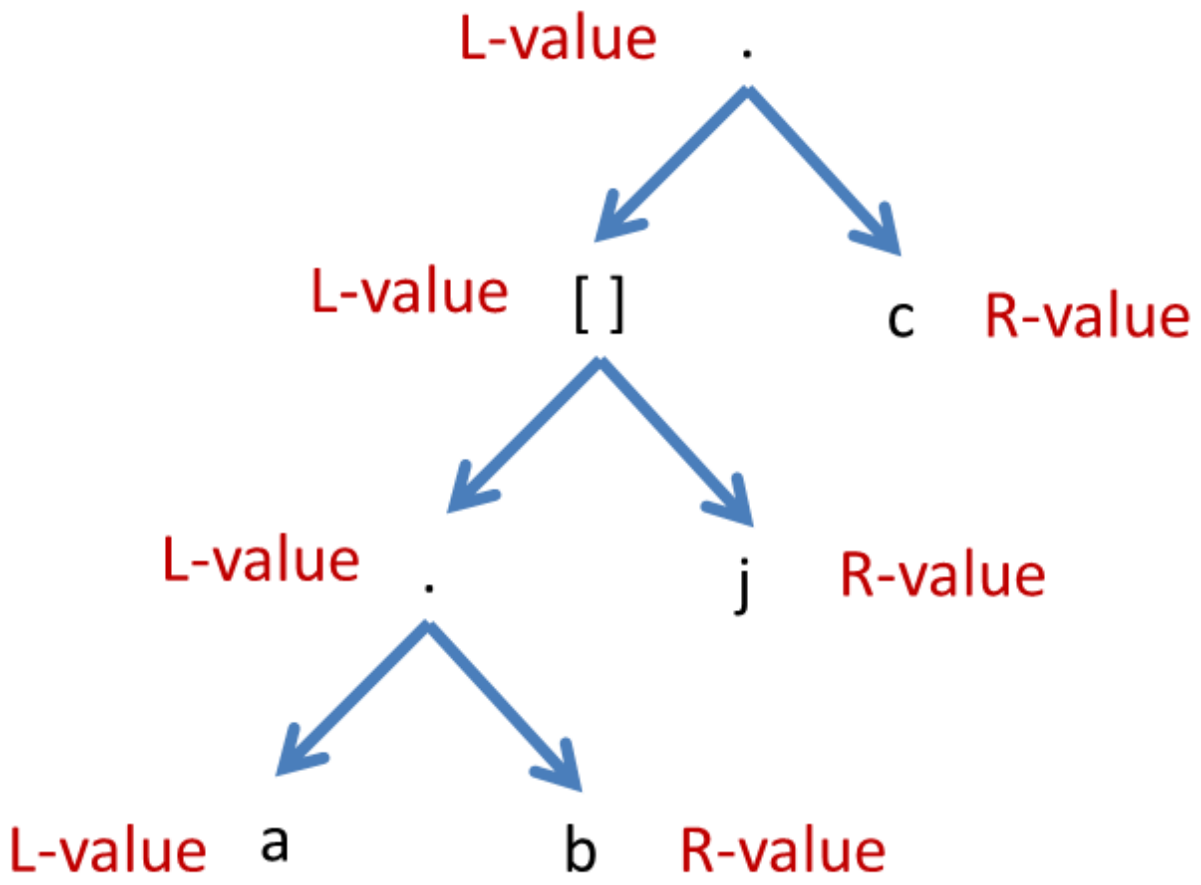
When calling a procedure, all the parameters have right values only. For reference parameter (in C++) and `var` parameters (in Pascal), the parameter has a left value. The return value is a right value. (That is, you cannot assign a value to the return value.)

```
int c[7];
```

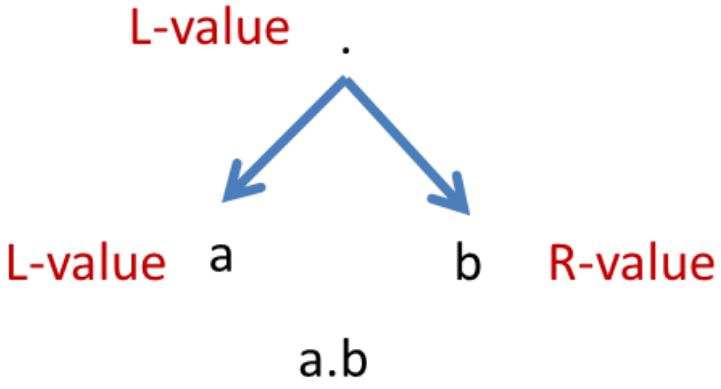
`c` has both a left value (which is usually the starting address of the array) and a right value (which is the contents of the whole array).
`c[i]` has both a left value (which is usually the starting address of the element) and a right value (which is the contents of the element).
`c` generates a left value while `i` generates a right value. The indexing operation `[ ]` generates a left value.

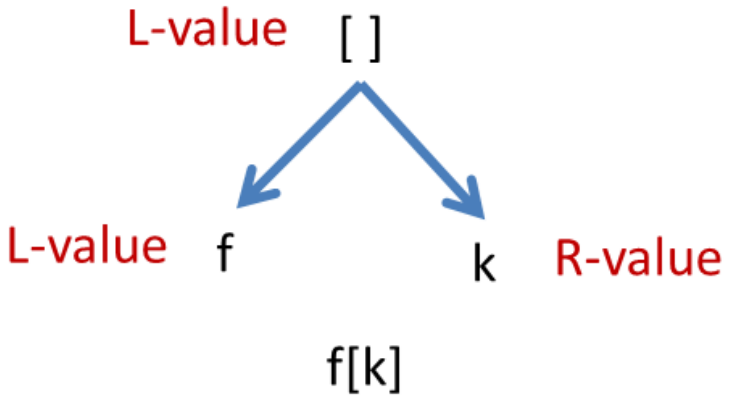
`struct { int e; float f[7]; } g;`
The field, such as `e`, is treated as a constant. It has a right value, which is the offset in the structure. To reference a field, we use `g.e`. Here `g` generates a left value (which is the starting address of the whole structure) and `e` generates a right value (which is the offset). The `.` operator generates a left value.

For a reference such as `g.f[k]`, `g` generates a left value, `f` generates a right value and `g.f` generates a left value. `k` generates a right value. Finally, `g.f[k]` generates a left value.



a.b[j].c


L-value
.
L-value
a
b
R-value
a.b


L-value
[ ]
L-value
f
k
R-value
f[k]