Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-16/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 19

- Static Scoping
  - Declarations and Definitions
  - Modules
  - Local symbol tables during compilation
  - Syntax-Directed Translation of three-address code in Scope
  - LeBlanc & Cook data structure and lookup function

Declaration order and use of bindings

- Scope of a binding
  1) In the whole block where it is defined
  2) From the declaration to the end of the block
- Use of binding
  a) Only after declaration
  b) In the scope of declaration
- Many languages use **2–a**. **Java** uses **1–b** for methods in a class. **Modula** uses **1–b** also for variables!
- Some combinations produce strange effects: **Pascal** uses **1) – a)**.

```
const N = 10;
...
procedure foo;
const
    M = N;        (* static semantic error! *)
   var
       A : array [1..M] of integer;
       N : real;        (* hiding declaration *)
```

Reported errors: "N used before declaration"
"N is not a constant"

Declarations and definitions

- "Use after declaration" would forbid mutually recursive definitions (procedures, data types)
- The problem is solved distinguishing ***declaration*** and ***definition*** of a name, as in **C**
- **Declaration**: introduces a name
- **Definition**: defines the binding

```
struct manager;          // Declaration only
struct employee {
    struct manager *boss;
    struct employee *next_employee;
    ...
};
struct manager {         // Definition
    struct employee *first_employee;
... };
```

Nested Blocks

C

```
{ int t = a;
  a = b;
  b = t;
}
```

Ada

```
declare t:integer
begin
  t := a;
  a := b;
  b := t;
end;
```

C++
Java
C#

```
{ int a,b;
  ...
  int t;
  t=a;
  a=b;
  b=t;
  ...
}
```

- In several languages local variables are declared in a block or compound statement
  - At the beginning of the block (Pascal, ADA, …)
  - Anywhere (C/C++, Java, …)
- Blocks can be considered as subroutines that are called where they are defined
- Local variables declared in nested blocks in a single function are all stored in the subroutine frame for that function (most programming languages, e.g. C/C++, Ada, Java)

Out of Scope

- Non-local objects can be *hidden* by local name-to-object bindings
- The scope is said to have a *hole* in which the non-local binding is temporarily inactive but not destroyed
- Some languages, like Ada, C++ and Java, use qualifiers or scope resolution operators to access non-local objects that are hidden
  - P1.X in Ada to access variable X of P1
  - ::X to access global variable X in C++
  - this.x or super.x in Java

Out of Scope Example

```
procedure P1;
var X:real;
  procedure P2;
  var X:integer
  begin
    ... (* X of P1 is hidden *)
  end;
begin
  ...
end
```

- P2 is nested in P1
- P1 has a local variable X
- P2 has a local variable X that hides X in P1
- When P2 is called, no extra code is executed to inactivate the binding of X to P1

Modules

- Modules are the main feature of a programming language that supports the construction of large applications
  - Support *information hiding* through *encapsulation*: explicit import and export lists
  - Reduce risks of *name conflicts;* support *integrity of data abstraction*
- Teams of programmers can work on separate modules in a project
- No language support for modules in C and Pascal
  - Modula-2 ***modules***, Ada ***packages***, C++ ***namespaces***
  - Java ***packages***

Module Scope

- Scoping: modules encapsulate variables, data types, and subroutines in a package
  - Objects inside are visible to each other
  - Objects inside are not visible outside unless ***exported***
  - Objects outside are visible [*open scopes*], or are not visible inside unless *imported* [*closed scopes*], or are visible with "qualified name" [*selectively open scopes*] (eg: **B.x**)
- A module interface specifies exported variables, data types and subroutines
- The module implementation is compiled separately and implementation details are hidden from the user of the module

Module Types, towards Classes

- Modules as abstraction mechanism: collection of data with operations defined on them (sort of *abstract data type)*
- Various mechanism to get module *instances*:
  - Modules as manager: instance as additional arguments to subroutines (**Modula-2**)
  - Modules as types (**Simula**, **ML**)
- Object-Oriented: Modules (classes) + inheritance
- Many OO languages support a notion of Module (packages) independent from classes

Syntax-directed translation of three-address code with names and scopes

- The three-address code generated by the syntax-directed definitions shown in a previous lesson is simplistic
- It assumes that the names of variables can be resolved by the back-end in global or local variables, which is unrealistic
- We need ***local symbol tables*** to record global declarations as well as local declarations in procedures, blocks, and records (structs) to resolve names

Implementing Static Scoping

- The language implementation must keep trace of current bindings with suitable data structures:
  - Static scoping: *symbol tables* **at compile time**
- **Symbol table** main operations: ***insert***, ***lookup***
  - because of nested scopes, the compiler must handle several bindings for the same name with LIFO policy
  - new scopes (not LIFO) should be created for records and classes
  - Other operations: ***enter_scope, leave_scope***
- The symbol table might be needed at runtime for ***symbolic debugging***
  - The debugger must resolve names in high-level commands by the user
  - Symbol table are saved in portion of the target program code

Symbol Tables for Scoping

```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void somefunc()
{ …
  swap(s.a, s.b);
  …
}
```

We need a symbol table for the *fields* of struct **S**

Need symbol table for *global* variables and functions

Need symbol table for *arguments* and *locals* for each function

Check: **s** is global and has fields **a** and **b**
Using symbol tables we can generate code to access **s** and its fields

Offset and Width for Runtime Allocation

```
struct S
{ int a;
  int b;
} s;
```

The fields **a** and **b** of struct **S** are located at *offsets* 0 and 4 from the start of **S**

a	(0)
b	(4)

The *width* of **S** is 8

```
void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}
```

Subroutine frame holds arguments **a** and **b** and local **t** at *offsets* 0, 4, and 8

Subroutine frame

fp[0]=	**a**	(0)
fp[4]=	**b**	(4)
fp[8]=	**t**	(8)

```
void somefunc()
{ ...
  swap(s.a, s.b);
  ...
}
```

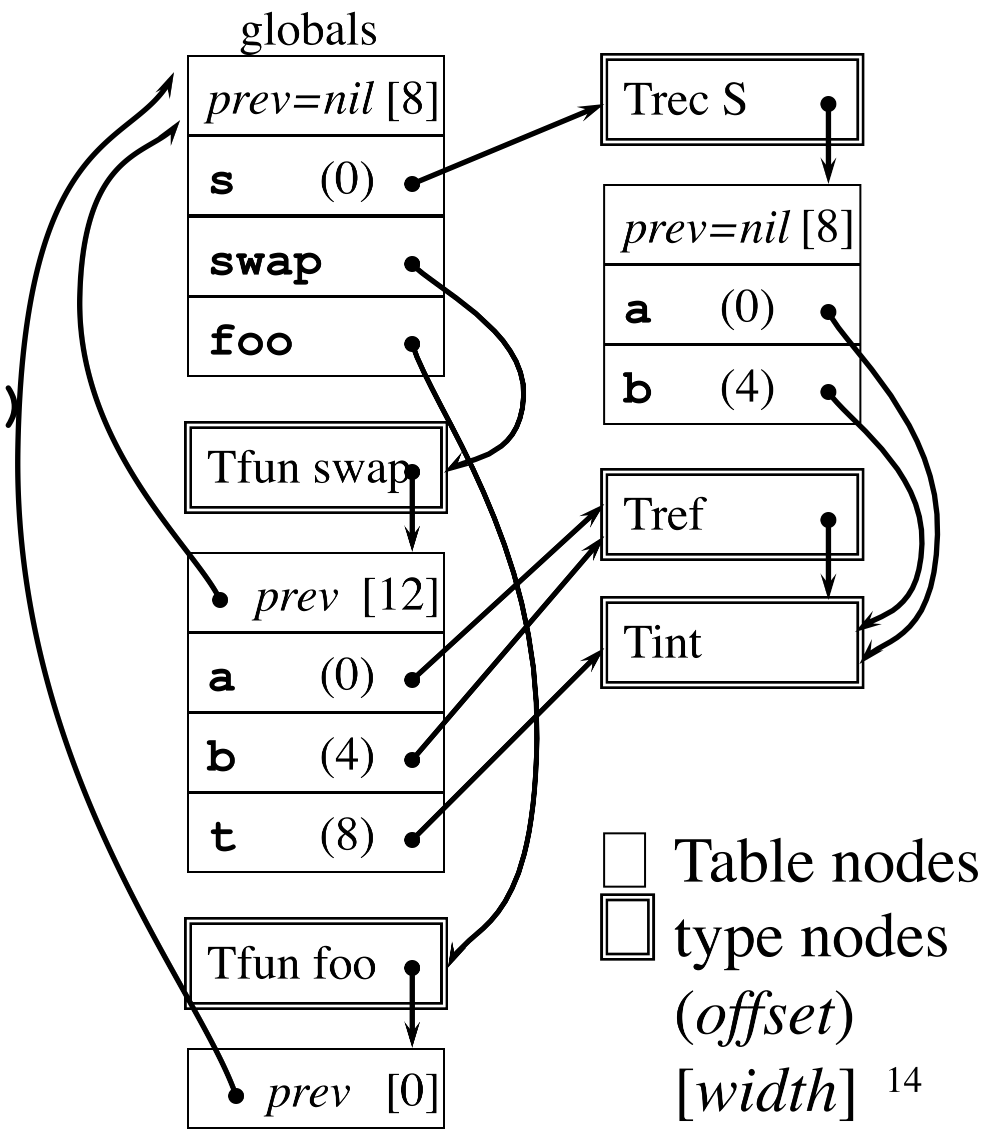
The *width* of the frame is 12

Symbol Tables for Scoping

```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void foo()
{ …
  swap(s.a, s.b);
  …
}
```



Hierarchical Symbol Table Operations

- ***mktable*(*previous*)** returns a pointer to a new (empty) table that is linked to a previous table in the outer scope
- ***enter*(*table, name, type, offset*)** creates a new entry in *table*
- ***addwidth*(*table, width*)** accumulates the total width of all entries in *table*
- ***enterproc*(*table, name, newtable*)** creates a new entry in *table* for procedure with local scope *newtable*
- ***lookup*(*table, name*)** returns a pointer to the entry in the table for *name* by following linked tables

Syntax-Directed Translation: Grammar and Attributes

Productions

$P \rightarrow D$ **;** S
$D \rightarrow D$ **;** D
  | **id :** T
  | **proc id ;** D **;** S
$T \rightarrow$ **integer**
  | **real**
  | **array [num] of** T
  | **^** T
  | **record** D **end**
$S \rightarrow S$ **;** S
  | **id :=** E
  | **call id (** A **)**

Productions *(cont'd)*

$E \rightarrow E$ **+** E
  | E ***** E
  | **-** E
  | **(** E **)**
  | **id**
  | E **^**
  | **&** E
  | E **. id**
$A \rightarrow A$ **,** E
  | E

Synthesized attributes:

***T*.type** pointer to type (ex.: '*integer*', *array*(2, '*real*'), *pointer*(*record*(*Table*)), …)
***T*.width** storage width of type (bytes)
***E*.place** name of temp holding value of *E*

Global data to implement scoping:

tblptr stack of pointers to tables
offset stack of offset values

Syntax-Directed Translation of Declarations in Scope

$P \rightarrow$ { *t* := *mktable*(nil); *push*(*t*, *tblptr*); *push*(0, *offset*) }
D **;** *S*

$D \rightarrow$ **id :** *T* ***enter*(*table, name, type, offset*)**

{ *enter*(*top*(*tblptr*), **id**.name, *T*.type, *top*(*offset*));
top(*offset*) := *top*(*offset*) + *T*.width }

$D \rightarrow$ **proc id ;**

{ *t* := *mktable*(*top*(*tblptr*)); *push*(*t*, *tblptr*); *push*(0, *offset*) }
D_1 **;** *S*
{ *t* := *top*(*tblptr*); *addwidth*(*t*, *top*(*offset*));
pop(*tblptr*); *pop*(*offset*); ***enterproc*(*table, name, newtable*)**
enterproc(*top*(*tblptr*), **id**.name, *t*) }

$D \rightarrow D_1$ **;** D_2

Syntax-Directed Translation of Declarations in Scope (cont'd)

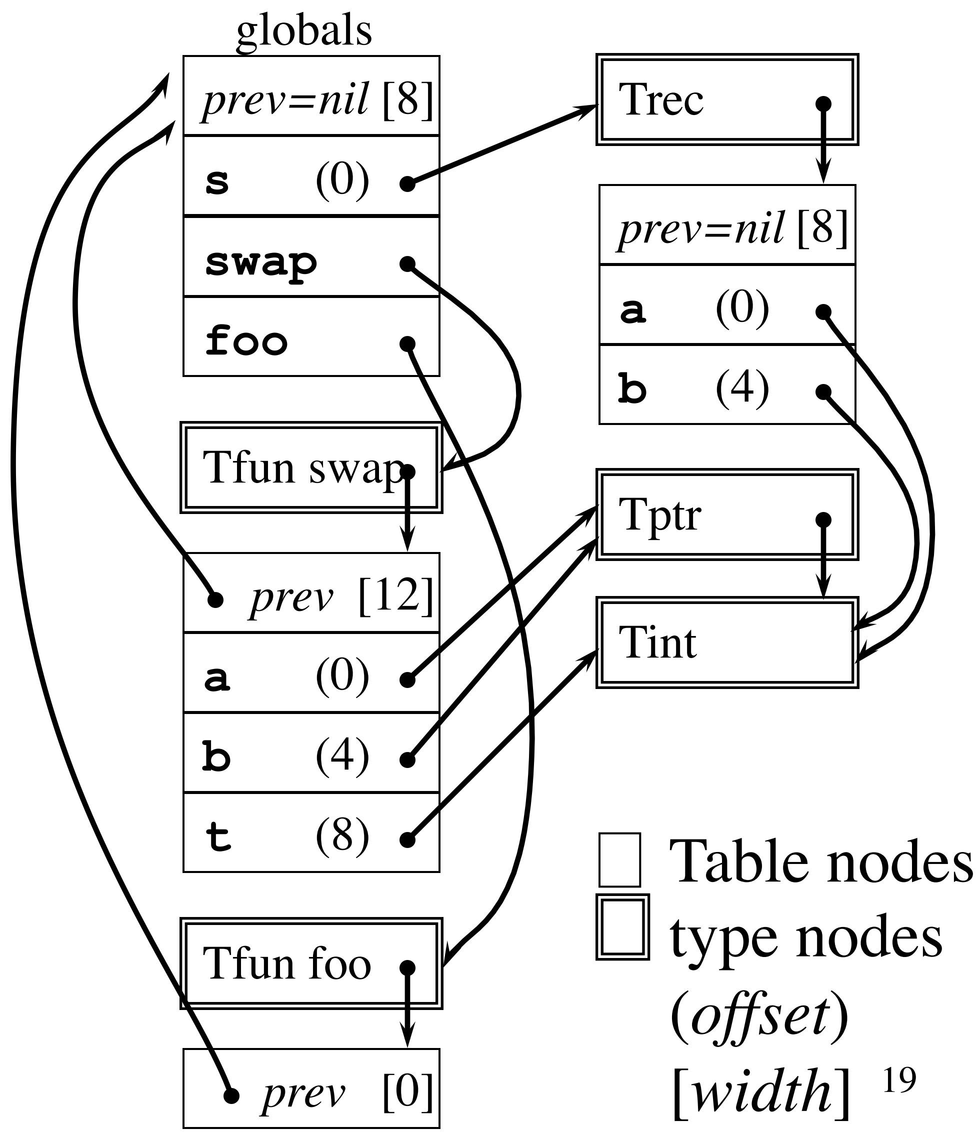
$T \rightarrow$ **integer** { T.type := '*integer*'; T.width := 4 }
$T \rightarrow$ **real** { T.type := '*real*'; T.width := 8 }
$T \rightarrow$ **array [num] of** T_1
{ T.type := *array*(**num**.val, T_1.type);
T.width := **num**.val * T_1.width }
$T \rightarrow$ **^** T_1
{ T.type := *pointer*(T_1.type); T.width := 4 }
$T \rightarrow$ **record**
{ *t* := *mktable*(nil); *push*(*t*, *tblptr*); *push*(0, *offset*) }
D **end**
{ T.type := *record*(*top*(*tblptr*)); T.width := *top*(*offset*);
addwidth(*top*(*tblptr*), *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*) }

Example

```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void foo()
{ …
  swap(s.a, s.b);
  …
}
```



Syntax-Directed Translation of Statements in Scope

$S \rightarrow S$ **;** S

$S \rightarrow$ **id :=** E

{ *p* := *lookup*(*top*(*tblptr*), **id**.name);

if *p* = nil **then**

error()

else if *p*.level = 0 **then** *// global variable*

emit(**id**.place ':=' *E*.place)

else *// local variable in subroutine frame*

emit(fp[*p*.offset] ':=' *E*.place) }

Globals

s	(0)
x	(8)
y	(12)

Subroutine frame

fp[0]=	**a**	(0)
fp[4]=	**b**	(4)
fp[8]=	**t**	(8)
…		

Syntax-Directed Translation of Expressions in Scope

$E \rightarrow E_1$ **+** E_2 { E.place := *newtemp*();
  emit(E.place ':=' E_1.place '+' E_2.place) }

$E \rightarrow E_1$ ***** E_2 { E.place := *newtemp*();
  emit(E.place ':=' E_1.place '*' E_2.place) }

$E \rightarrow$ **-** E_1 { E.place := *newtemp*();
  emit(E.place ':=' 'uminus' E_1.place) }

$E \rightarrow$ **(** E_1 **)** { E.place := E_1.place }

$E \rightarrow$ **id** { p := *lookup*(*top*(*tblptr*), **id**.name);
  if p = nil **then** *error*()
  else if p.level = 0 **then** *// global variable*
    emit(E.place ':=' **id**.place)
  else *// local variable in frame*
    emit(E.place ':=' fp[p.offset]) }

Syntax-Directed Translation of Expressions in Scope (cont'd)

$E \rightarrow E_1$ **^** { E.place := *newtemp*();
    emit(E.place ':=' '*' E_1.place) }
$E \rightarrow$ **&** E_1 { E.place := *newtemp*();
    emit(E.place ':=' '&' E_1.place) }
$E \rightarrow$ **id**$_1$ **.** **id**$_2$ { p := *lookup*(*top*(*tblptr*), **id**$_1$.name);
    if p = nil **or** p.type != Trec **then** *error*()
    else
      q := *lookup*(p.type.table, **id**$_2$.name);
      if q = nil **then** error()
      else if p.level = 0 **then** // *global variable*
          emit(E.place ':=' **id**$_1$.place[q.offset])
      else // *local variable in frame*
          emit(E.place ':=' fp[p.offset+q.offset])}

LeBlanc & Cook Symbol Table

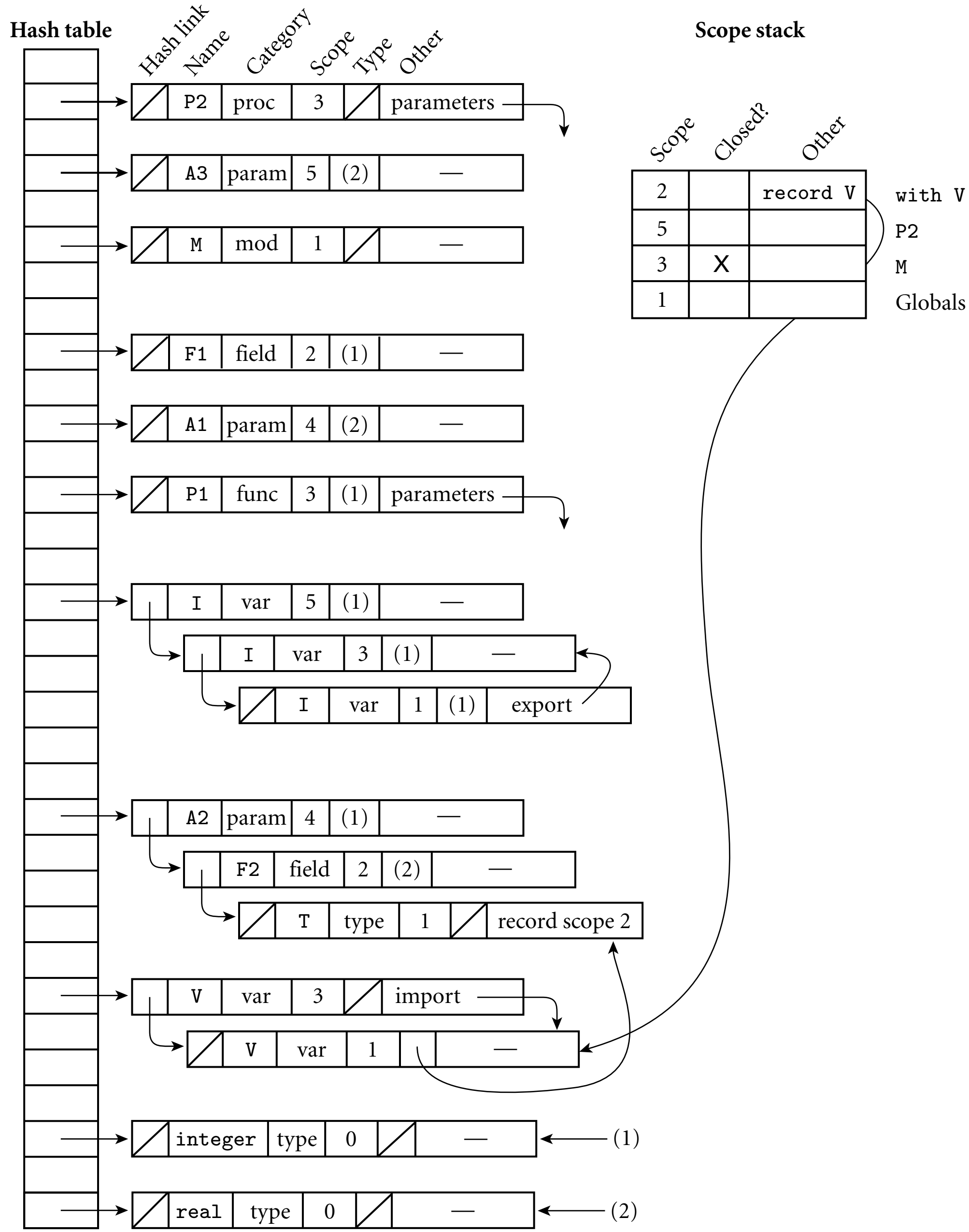
In the translation just shown, *lookup* of a name may require traversing all enclosing symbol tables, from the current one to the global one.

The LeBlanc & Cook Symbol Table implementation for static scoping uses *a hash table* and *a stack*, instead of a tree of symbol tables. This guarantees more efficient lookups.

- Each scope has a **serial number**
  - Predefined names: 0 (**pervasive**)
  - Global names: 1, and so on
- Names are inserted in a **hash table**, indexed by the name
  - Entries contain symbol name, category, **scope number**, (pointer to) type, …
- **Scope Stack**: contains numbers of the currently visible scopes
  - Entries contain scope number and additional info (closed?, …). They are pushed and popped by the semantic analyzer when entering/leaving a scope
- Look-up of a *name*: scan the entries for *name* in the hash table, and look at the scope number *n*
  - If n <> 0 (*not pervasive*), scan the Scope Stack to check if scope *n* is visible
  - Stops at first *closed* scope. Imported/Export entries are pointers.

A Modula2 program

```
type                                    [1]
    T = record
        F1 : integer;                   [2]
        F2 : real;
    end;
var V : T;
...
module M;
    export I; import V;                 [3]
    var I : integer;
    ...
    procedure P1 (A1 : real;
        A2t: integer) : real;
    begin                               [4]
        ...
    end P1;
    ...
    procedure P2 (A3 : real);
    var I : integer;                    [5]
    begin
        ...
        with V do
            ...
        end;
        ...
    end P2;
    ...
end M;
```



LeBlanc & Cook lookup function

```
procedure lookup(name)
    pervasive := best := null
    apply hash function to name to find appropriate chain
    foreach entry e on chain
        if e.name = name -- not something else with same hash value
            if e.scope = 0
                pervasive := e
            else
                foreach scope s on scope stack, top first
                    if s.scope = e.scope
                        best := e      -- closer instance
                        exit inner loop
                    elsif best != null and then s.scope = best.scope
                        exit inner loop -- won't find better
                    if s.closed
                        exit inner loop -- can't see farther
    if best != null
        while best is an import or export entry
            best := best.real entry
        return best
    elsif pervasive != null
        return pervasive
    else
        return null -- name not found
```