CS 321
Programming Languages and Compilers

Names, Scopes, and Bindings
**Binding Time**

• The *binding* of a program element to a particular characteristic or property is the choice of the property from a set of possible properties.

• The time during program formulation or processing when this choice is made is the *binding time*.

• There are many classes of bindings in programming languages as well as many different binding times.

• Also included within the concepts of binding and binding times are the properties of program elements that are determined by the definition of the language or its implementation.
Binding times include:

• Run time (execution time). Two subcategories:
  – On entry to a subprogram or block.
    » Binding of formal to actual parameters
    » Binding of formal parameters to storage locations
  – At arbitrary points during execution
    » binding of variables to values
    » binding of names to storage location in Scheme.
      • e.g. (define size 2)
• Compile time (translation time)
  – Bindings chosen by the programmer
    » Variable names
    » variable types
    » program statement structure
  – Chosen by the translator
    » Relative location of data objects.
  – Chosen by the linker
    » Relative location of different object modules.
• **Language Implementation time (i.e. when the compiler or interpreter is written)**
  - Representation of numbers. Usually determined by the underlying computer, but not always (e.g. Java defines the representation to guarantee portability).
  - Use of implementation-specific features preclude portability.
    - e.g. in Fortran’s expression $x*f(y)$, function $f$ in the does not have to be called when $x$ is 0. If the function has side effects, different implementations produce different results.

• **Language definition time.**
  - Alternative statement forms
  - Data structure types
  - Array storage layout
• Consider $x = x + 10$
  – Type of $X$
    » At translation time in C
    » At run time in Scheme/MATLAB
  – Set of possible values of $X$
    » At implementation time. If $X$ is real it could be
      • the IEEE floating point standard (almost always the choice),
      • the implementation of a specific machine, or
      • a software-based “infinite precision” representation.
  – Value of $X$
    » Changed at run time.
- Properties of operator +
  - At compilation time (depending on the type of the operands because of overloading.
    - If \( x \) is declared integer + means one thing,
    - if \( x \) is declared real means something else.
    - \(+\) can also be overloaded by the programmer. For example, in Fortran 95 it is possible to specify that + operate on intervals and on rational numbers:

```fortran
INTERFACE OPERATOR(+)
  FUNCTION INTEGER_PLUS_INTERVAL(X, Y) 
  ... 
  END 
END INTERFACE
```

```fortran
MODULE PROCEDURE RATIONAL_ADD 
END INTERFACE
```
• Many of the most important and subtle differences between languages involve differences in binding time.

• The trade off is between efficient execution and flexibility.
  – When efficiency is a consideration (Fortran, C) Languages are designed so that as many bindings as possible are performed during translation.
  – Where flexibility is the prime determiner, as in Scheme, most bindings are delayed until execution time so that they may be made data dependent.
• Key events in the life of an object:

- Creation of an object
- Creation of a binding
- Destruction of a binding
- Destruction of an object

Dangling reference if these two times are interchanged
Three storage allocation mechanisms

- Static
- Stack
- Heap
Static Allocation

- Global variables
- Constants
  - manifest, declared (parameter variables in Fortran) or identified by the compiler
- Variables identified as `const` in C can be a function of non constants and therefore cannot be statically allocated.
- Constant tables generated by the compiler for debugging and other purposes.
• In the absence of recursion, all variables can be statically allocated.

• Also, can be statically allocated:
  – Arguments and return values (or their addresses). Allocation can be in processor registers rather than in memory
  – Temporaries
  – Bookkeeping information
    » return address
    » saved registers,
    » debugging information
Static Allocation (Cont.)

<table>
<thead>
<tr>
<th>Subroutine 1</th>
<th>Subroutine 2</th>
<th>Subroutine 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporaries</td>
<td>Temporaries</td>
<td>Temporaries</td>
</tr>
<tr>
<td>Local variables</td>
<td>Local variables</td>
<td>Local variables</td>
</tr>
<tr>
<td>Miscellaneous bookkeeping</td>
<td>Miscellaneous bookkeeping</td>
<td>Miscellaneous bookkeeping</td>
</tr>
<tr>
<td>Return address</td>
<td>Return address</td>
<td>Return address</td>
</tr>
<tr>
<td>Arguments and returns</td>
<td>Arguments and returns</td>
<td>Arguments and returns</td>
</tr>
</tbody>
</table>
• Needed when language permits recursion
• It could be useful in languages without recursion because it could save space.
• Each subroutine invocation creates a frame or activation record
  – arguments
  – return address
  – local variables
  – temporaries
  – bookkeeping information
• Stack maintained by
  – calling sequence
  – prologue
  – epilogue
Stack-based Allocation (Cont.)

Direction of stack growth (usually lower addresses)

Subroutine A (called from main program)

Subroutine B

Subroutine C

Subroutine D

Temporary

Local variables

Miscellaneous bookkeeping

Return address

Arguments and returns

.fp (when subroutine C is running)

.fp
Heap-based Allocation

• Region of storage in which blocks of memory can be allocated and deallocated at arbitrary times.
• Because they are not allocated in the stack, the lifetime of objects allocated in the heap is not confined to the subroutine where they are created.
  – They can be assigned to parameters (or to components of objects accessed via pointers by parameters)
  – They can be returned as value of the subroutine/function/method.
• There are several strategies to manage space in the heap.

• An important issue is fragmentation (also an issue in virtual memory management systems)
  – *Internal fragmentation* when space allocated is larger than needed.
  – *External fragmentation* when allocated blocks are scattered through the heap. It could be that the total space available could is more than requested, but no block has the needed size.
Heap-based Allocation (Cont.)

• One approach to maintain the free memory space is to use a free list.

• Two strategies to find a block for a given request
  – First fit. Use the first block in the list that is large enough to satisfy the request
  – Best fit. Search the entire list to find the smallest block that satisfy the request

• The free list could be organized (conceptually) as an array of free lists where each list in the array contain blocks of the same size.
  – Buddy system
  – Fibonacci heap (better internal fragmentation)
Garbage collection

- Programmers can manage memory themselves with explicit allocation/deallocation.
- However, garbage collection can be applied automatically by the run-time system to avoid memory leaks and difficult to find dangling references.
  - Lisp
  - Java
- The disadvantage is cost.
Scope rules

• The region of the program in which a binding is active is its scope.
• Most languages today are lexically scoped
• We will also study dynamic scoping for completeness.
Static Scope

- A single global scope (basic, awk?)
- A separate scope for each program unit (main program, subroutines, functions) in FORTRAN.
- We will discuss two classes of Fortran objects
  - variables
  - common blocks
- Common blocks are blocks of storage that can be shared by several program units.
• Common block example

subroutine first
    real b(2)
    logical flag
    complex c
    type coordinates
        sequence
            real x, y
        logical z_0
    end type coordinates
    type (coordinates) p
    common /reuse/ b,c,flag,p
    ...
    subroutine second
    integer I(8)
    common /reuse/ i

<table>
<thead>
<tr>
<th></th>
<th>b(1)</th>
<th>b(2)</th>
<th>c</th>
<th>flag</th>
<th>x</th>
<th>y</th>
<th>z_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reuse</td>
<td>i(1)</td>
<td>i(2)</td>
<td>i(3)</td>
<td>i(4)</td>
<td>i(5)</td>
<td>i(6)</td>
<td>i(7)</td>
</tr>
</tbody>
</table>
• Lifetime of statically allocated variables and common blocks is the duration of the program.
• Most Fortran 77 implementations do all allocations statically.
• If default allocation is not static, variables and common blocks can be saved (i.e. declared as save: save /reuse/, w, r) to force their static allocation.
• Variables can only be saved in the program unit where they are declared.
• If a common block is saved, it has to be saved in all program units where it appears.
The default is that common blocks can go away when there are no active program units that access them.

Saved variables and saved common blocks may cause collisions in parallel executions, but private entities enable the creation of new copies with each invocation.
Static Scope - Nested Subroutines

• In most languages any constant, type, variables or subroutines declared within a subroutine are not visible outside the subroutine.

• In the closest nested scope rule: a name is known in the scope in which it is declared unless it is hidden by another declaration of the same name.
procedure P1 (A1 : T1);
var X : real;

...  
procedure P2 (A2 : T2);
...
procedure P3 (A3 : T3);
...
begin
    ...
    (* body of P3 *)
end;
...

begin
    ...
    (* body of P2 *)
end;
...

procedure P4 (A4 : T4);
...
function F1 (A5 : T5) : T6;
var l : integer;
...
begin
    ...
    (* body of F1 *)
end;
...

begin
    ...
    (* body of P4 *)
end;
...

begin
    ...
    (* body of P1 *)
end
• To find the frames of surrounding scopes where the desired data is a static link could be used.
Static Scope - Nested Subroutines (Cont.)
{ /* B1 */

    { /* B2 */

        { /* B3 */

            { /* B4 */

            }

        }

    }

}
Static Scope - Nested Subroutines (Cont.)

```c
P1()
{
    /* B1 */
    P2()
    {
        P3()
    }
    /* B2 */
    {
        /* B3 */
        P2()
    }
}

P3()
{
    P3()
}
```
Static Scope - Modules

- Modularization depends on information hiding.
- Functions and subroutines can be used to hide information. However, this is not flexible enough.
- One reason is that persistent data is usually needed to create abstraction. This can be addressed in some cases using statically allocated values.
/*
   Place into *s a new name beginning with the letter \texttt{l} and continuing
   with the ascii representation of an integer guaranteed to be distinct
   in each separate call. \texttt{s} is assumed to point to space large enough
   to hold any such name; for the short ints used here, seven characters
   suffice. \texttt{l} is assumed to be an upper or lower-case letter.
   \texttt{printf} `prints` formatted output to a string.
*/

void gen_new_name (char *s, char l) {
    static short int name_nums[52];
    /* C guarantees that static local variables are initialized
       to zeros */
    int index = (l >= \texttt{a} \&\& l <= \texttt{z}) ? l-'a' : 26 + l-'A';
    name_nums[index]++;
    sprintf (s, \texttt{%c%d\0}, l, name_nums[index]);
}
But modularization often requires a variety of operations on persistent data.
Objects inside a module are visible to each other.

Objects inside can be hidden explicitly (using a keyword like private) or implicitly (objects are only visible outside if they are exported).

In some languages, objects outside need to be imported to be visible within the module.
VAR a, b: CARDINAL;
MODULE M;
    IMPORT a; EXPORT w, x;
VAR u, v, w: CARDINAL;
MODULE N;
    IMPORT u; EXPORT x, y;
    VAR x, y, z: CARDINAL;
    (* x, u, y, z visible here *)
END N;
    (* a, u, v, w, x visible here *)
END M;
    (* a, b, w, x visible here *)

MODULE M;
    VAR a: CARDINAL;
MODULE N1;
    EXPORT b;
    VAR b: CARDINAL;
    (* only b visible here *)
END N1;
MODULE N2;
    EXPORT c;
    VAR c: CARDINAL;
    (* only c visible here *)
end N2;
MODULE N3;
    IMPORT b, c;
    (* b, c visible here *)
END N3;
END M;
CONST stack_size = ... 
TYPE element = ... 
...

MODULE stack; 
IMPORT element, stack_size; 
EXPORT push, pop; 
TYPE 
  stack_index = [1..stack_size]; 
VAR 
  s : ARRAY stack_index OF element; 
  top : stack_index;         (* first unused slot *)

PROCEDURE error; ...

PROCEDURE push (elem : element); 
BEGIN 
  IF top = stack_size THEN 
    error; 
  ELSE 
    s[top] := elem; 
    top := top + 1; 
  END; 
END push; 
END push;

PROCEDURE pop () : element;    (* A Modula-2 function is just a *) 
BEGIN 
  IF top = 1 THEN 
    error; 
  ELSE 
    top := top - 1; 
    RETURN s[top]; 
  END; 
END pop; 
END pop; 

BEGIN 
  top := 1; 
END stack; 

VAR x, y : element; 
...
push (x); 
...
y := pop;
Module polar_coordinates
  type polar
    private
    real rho, theta
  end type polar
  interface operator (*)
    module procedure polar_mult
  end interface
  contains
    function polar_mult(p1,p2)
      type (polar), intent (in) :: p1,p2
      type (polar) polar_mult
      polar_mult= polar(p1%rho*p2%rho, &
        p1%theta+p2%theta)
    end function polar_mult
  ...
end module polar_coordinates
const stack_size := ...

type element : ...

... type stack = module
    imports (element, stack_size)
    exports (push, pop)

type
    stack_index = 1..stack_size

var
    s : array stack_index of element

    top : stack_index

procedure push (elem : element) = ...

function pop returns element = ...

... initially
    top := 1

end stack

var A, B : stack
var x, y : element

... A.push (x)

... y := B.pop
Dynamic scope

• Early lisp systems were implemented so that variables are bound dynamically rather than statically.
• In a language with dynamic binding, free variables in a procedure get their values from the environment in which the procedure is called rather than the environment in which the procedure is defined.
Consider the program

\[
\text{(define (sum-powers a b n)} \backslash \\
\text{ (define (nth-power x)} \backslash \\
\text{ (expt x n))} \backslash \\
\text{(sum nth-power a 1+ b))} \backslash \\
\]

Where sum is defines as follows

\[
\text{(define (sum term a next b)} \backslash \\
\text{(if (> a b)} \backslash \\
\text{ 0)} \backslash \\
\text{ (+ (term a)} \backslash \\
\text{ (sum term (next a) next b))))} \backslash \\
\]
Dynamic scope (Cont.)

- Traditionally Lisp systems have been implemented so that variables are bound dynamically rather than statically.
- In a language with dynamic binding, free variables in a procedure get their values from from which the procedure is called rather than from the environment in which the procedure is defined.
- For example, the free variable \( n \) in \( \text{nth-power} \) would get whatever \( n \) had when \( \text{sum} \) called it.
- In this example, since \( \text{sum} \) does not rebind \( n \), the only definition of \( n \) is still the one from \( \text{sum-powers} \).
**Dynamic scope (Cont.)**

- But if we had used \( n \) instead of \( \text{next} \) in the definition of \( \text{sum} \), then \( \text{nth-power} \)'s free variable would refer to \( \text{sum} \)'s third argument, which is not what we intended.

- This would produce an error, since the value of \( n \) here is not a number, as required by \( \text{nth-power} \).

- As can be seen from this example, dynamic binding violates the principle that a procedure should be regarded as a “black box”, such that changing the name of a parameter throughout a procedure’s definition will no change the procedure behavior.
Dynamic scope (Cont.)
Dynamic scope (Cont.)

• In a statically bond language, the sum-powers program must contain the definition of nth-power as a local procedure.

• If nth-power represents a common pattern of usage, its definition must be repeated as an internal definition in many contexts.
Dynamic scope (Cont.)

• It should be attractive to be able to move the definition of \texttt{nth-power} to a more global context, where it can be shared by many procedures

\begin{verbatim}
(define (sum-powers a b n)
 (sum nth-power a 1+ b))

(define (product-powers a b n)
 (product nth-power a 1+ b))

(define (nth-power x)
 (expt x n))
\end{verbatim}

• The attempt to make this work is what motivated the development of dynamic binding discipline.
• In general, dynamically bound variables can be helpful in structuring large programs.

• They simplify procedure calls by acting as implicit parameters.

• For example, a low-level procedure \texttt{nprint} called by the system \texttt{print} procedure for printing numbers might reference a free variable called \texttt{radix} that specifies the base in which the number is to be printed.

• Procedures that call \texttt{nprint}, such as the system \texttt{print} operation, should not need to know about this feature.
Dynamic scope (Cont.)

• On the other hand, a user might want to temporarily change the `radix`.

• In a statically bound language, `radix` would have to be a global variable.

• After setting `radix` to a new value, the user would have to explicitly reset it. But the dynamic binding mechanism could accomplish this setting and resetting automatically, in a structured way

  (define print-in-new-radix number radix)
  (print number))

  (define (print frob)
   < expressions that involve nprint>)

  (define (nprint number)
   ...
   radix
   ...

47
Names, Scopes, and Bindings
Symbol Tables

- Symbol tables are used to keep track of scope and binding information about names.
- The symbol table is searched every time a name is encountered in the source text.
- Changes occur when a new name or new information about a name is discovered.
- The abstract syntax tree will contain pointers to the symbol table rather than the actual names used for objects in the source text.
Symbol Tables (Cont.)

• Each symbol table entry contains
  – the symbol name,
  – its category (scalar variable, array, constant, type, procedure, field name, parameter, etc.)
  – scope number,
  – type (a pointer to another symbol table entry),
  – and additional, category specific fields (e.g. rank and shape for arrays)

• To keep symbol table records uniform, it may be convenient for some of the information about a name to be kept outside the table entry, with only a pointer to this information stored in the entry.
Symbol Tables (Cont.)

- The symbol table may contain the keywords at the beginning if the lexical scanner searches the symbol table for each name.
- Alternatively, the lexical scanner can identify keywords using a separate table or by creating a separate final state for each keyword.
Symbol Tables (Cont.)

- One of the important issues is handling static scope.
- A simple solution is to create a symbol table for each scope and attach it to the node in the abstract syntax tree corresponding to the scope.
- An alternative is to use an additional data structure to keep track of the scope. This structure would resemble a stack:
procedure new_id(id)
    for index=top to scope_marker(LL - 1) by -1
        if id == symbol_table(additional(index)).name then error()
    k = new_symbol_table_index()
    symbol_table(k).name=id
    additional(top++) = k

procedure old_id(id)
    for index= top to 0 by -1
        if id == symbol_table(additional(index)).name then return additional(index)
    error()

procedure scope_entry()
    scope_marker(LL++)=top

procedure scope_exit()
    top = scope_marker(--LL)
• A hash table can be added to the previous data structure to accelerate the search.

• Elements with the same name are linked from top to bottom.

• Search start at the entry of the hash table and proceeds through the linked list until the end of the list is reached \((old_id)\) or until the link list refers to an element below \(scope\_marker(LL - 1)\) \((new\_id)\)
Symbol Tables (Cont.)

- This approach does not work in some cases.
- Consider the with statement of Pascal and Modula 2.

```pascal
Date = RECORD
day: [1..31];
  mo: month;
  yr: CARDINAL
END

d1: Date;

WITH d1 DO
  day:=10; mo:=Sep; yr:=1981
END

is equivalent to

d1.day:=10; d1.mo:=Sep; d1.yr:=1981
```
Symbol Tables

procedure lookup (name)
  pervasive := best := nil
  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
          elsif best <> nil 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 <> nil
        while best is an import or export entry
          best := best.real_entry
      return best
    elsif pervasive <> nil
      return pervasive
  else
    return nil  -- name not found
Symbol Tables (Cont.)

---

Names, Scopes, and Bindings

---

Symbol Tables (Cont.)
Association Lists and Central Reference Tables

Referencing environment A-list

I  --  Type, location, etc.
J  --  Type, location, etc.
Q  --  Type, location, etc.
P  --  Type, location, etc.
J  --  Type, location, etc.
I  --  Type, location, etc.

(predefined names)

I, J : integer
procedure P (i : integer)
\[ \ldots \]
Procedure Q
   J : integer
   \ldots
   P (j)
   \ldots
   main program
   Q

Central reference table

<table>
<thead>
<tr>
<th></th>
<th>Type, location, etc.</th>
<th>Type, location, etc.</th>
<th>Type, location, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(other names)
The binding of referencing environments

- Shallow binding: the referencing environment of a routine is not created until the subroutine is actually called.
- Deep binding: the program binds the environment at the time the subroutine is passed as a parameter.
- Deep binding is implemented by creating an explicit representation of a referencing environment and bundling it together with a reference to the subroutine. Closure
P1()
{ REAL X
  { /* B1 */
    { /* B2 */
      { /* B3 */
        P2(P3)
      }
      P3()
      {
        x
      }
    }
  }
  P2(PX)
  {
    PX()
  }
}