

Names, Binding, Scope

*Her name was McGill and she called herself Lil
But everyone knew her as Nancy.*

-- Lennon & McCartney

■ Names

 4.2, 4.3

- ▲ abstraction
- ▲ variables
- ▲ aliasing

■ Binding

 4.4

- ▲ binding time
- ▲ static vs. dynamic binding
- ▲ variable bindings

■ Scope

 4.8-4.10

- ▲ blocks, visibility
- ▲ static vs. dynamic scoping

87

What's in a Name?

- A *name* is a *handle* associated with some *entity*. When we want to use the entity (create it, change it, destroy it, etc.), we only need to refer to its name.
- An entity's name is an *abstraction*:
 - ▲ it allows us to use the entity without regard for its internal complexities



What internal complexities are hidden by names in programs?

88

What Needs a Name?

- constants, variables, objects, atoms
- operators
- labels
- types, classes
- procedures, functions, predicates, rules
- modules, libraries, programs, packages
- files, disks, peripherals
- commands, menu items, macros
- computers, networks, hosts, domains
- users, user groups

89

Identifiers

A name is denoted by a sequence of characters, called an *identifier*.

- not all sequences of characters are legal identifiers in a given language; they obey naming rules
- Hmmm... sequences, rules, language... Grammar!

```
<identifier> ::= <letter> { <idchar> }  
<idchar>    ::= <letter> | <digit> | _  
<letter>    ::= A | B | C | ... | a | b | ... | z  
<digit>     ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

90

Keywords, Reserved Words

A *keyword* in a programming language is a name that is recognized as having a particular meaning in the language (for all programs written in that language).

A *reserved word* in a programming language is a keyword that may *not* be used by the programmer for naming other things.

- Example reserved words:

```
if C then S1 else S2 end;
```

- Other possibilities:

```
{ C -> S1; S2 }
```

```
kama C kwa hiyo S1 ingina S2 mwisho;
```

```
если C тогда S1 еще S2 конец;
```



Note that keywords have a functional role, and an aesthetic one.

91

Variables

In the lower virtual machines (e.g., machine language) data is merely the value stored in a memory location (or group of locations).

- a *variable* is a higher level abstraction of a memory location (or group of locations).
- in the *imperative* paradigm, program variables are complex entities, consisting of six parts:
<name, address, value, type, lifetime, scope>
- these six *attributes* are essential to the interpretation of a variable, but hidden behind the first attribute (the name)

92

The Variable Sextuplet...

- Name
 - ▲ *almost* every variable has a name
 - ▲ a variable name must be unique within its scope and lifetime
- Address
 - ▲ the location of the variable in memory
 - ▲ a variable's address is sometimes called its *l-value*
 - ▲ for *composite variables* determining the address can be complicated
- Value
 - ▲ the contents of the memory location at the address of a variable
 - ▲ the value may span more than one physical memory cell
 - ▲ a variable's value is sometimes called its *r-value*

93

More The Variable Sextuplet

- Type
 - ▲ a restriction on the kinds of values a variable can store
 - ▲ a variable's type determines how much memory it requires
 - ▲ the type also determines what operations can be done to a variable
- Lifetime
 - ▲ a variable may only be accessible for a part of program execution
 - ▲ variables can be created and destroyed at various times
- Scope
 - ▲ a variable may be visible to some parts of a program and invisible to other parts
 - ▲ some variables block the visibility of other variables

94

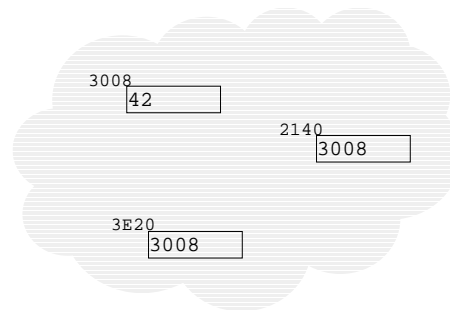
Aliasing...

It is possible for two names to refer to the same memory location. Such names are called *aliases*.

■ pointers

- ▲ a pointer is a variable whose value is the address of another variable
- ▲ pointers with the same *value* are aliases for the memory location they point to

| Name | Address |
|--------|---------|
| myvar | 3008h |
| myptr1 | 2140h |
| myptr2 | 3E20h |
| ... | ... |



95

More Aliasing...

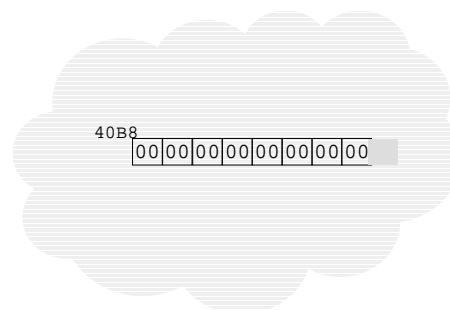
■ unions, variant records

```

union {
  int mynum;
  char myletter;
  float bunch[10];
};
enum fieldtype {i,c,f};

type myunion = record
  case fieldtype: (i,c,f) of
    i : (mynum : integer);
    c : (myletter : char);
    f : (bunch : array[1..10] of real)
  end;
```

| Name | Address |
|----------|---------|
| mynum | 40B8h |
| myletter | 40B8h |
| bunch | 40B8h |
| ... | ... |



96

More Aliasing

Aliasing also occurs:

- as a result of certain kinds of *parameter passing*
 - ▲ more in section 8: *Procedural Abstraction*
- as a result of *macro expansion*
 - ▲ more in section 8: *Procedural Abstraction*
- through *explicit* aliasing mechanisms in some languages
 - ▲ Fortran

In general, aliasing is considered a “bad thing”!

97

Binding

Binding is the association of an attribute with an entity. When we associate an attribute to an entity, we are *binding* the attribute to the entity.

- when we associate a type with a variable, we are *binding* the type to the variable
- when we associate a particular memory location with a variable, we are *binding* the address to the variable
- etc.

In programming languages, we have many different things to bind and many different times when we can bind them.

98

Binding Time

Recall that a variable can be thought of as a *sextuple* (consisting of six attributes). Each of the six attributes is *bound* to the variable at some time or other.

- *compile time*
 - ▲ binding occurs during translation
 - ▲ applies to interpreters too! (*interpretation time* or *translation time*)
- *load time*
 - ▲ binding occurs during the load/link phase
 - ▲ when libraries and modules are linked in
 - ▲ when *external references* are resolved
- *run time*
 - ▲ binding occurs sometime between the start of execution of a program and its termination

99

Static Binding vs. Dynamic Binding

- *static binding*
 - ▲ binding occurs before program execution and is permanent during execution (does not change while the program is running)
 - ▲ compile time binding
 - ▲ load time binding
- *dynamic binding*
 - ▲ binding *occurs* during program execution
 - or
 - ▲ binding *expires* during program execution
 - or
 - ▲ binding *changes* during program execution

100

Variable Bindings...

- variable → name
 - compile time
 - ▲ bound by variable declarations
- variable → address
 - load time
 - ▲ normal variables
 - run time
 - ▲ stack variables
 - ▲ dynamic memory allocation
 - ▲ subprogram parameters
- variable → type
 - compile time
 - ▲ most imperative languages
 - run time
 - ▲ weakly-typed languages
 - ▲ implicitly-typed languages

101

More Variable Bindings


- variable → value
 - run time
 - ▲ assignment, etc.
 - load time
 - ▲ initialization
- variable → lifetime
 - compile time
 - ▲ lifetime is bound statically, even for implicit dynamic variables
 - run time
 - ▲ for *some* explicit dynamic variables
- variable → scope
 - compile time
 - ▲ expressed by location of declarations
 - run time
 - ▲ rare (for dynamic scoping only)

102

Variable → Address

An address is bound to a variable when it is allocated memory.

- *static variables*
 - ▲ memory is allocated *once*, before program execution (load time)
- *dynamic variables*
 - ▲ memory is allocated after program execution begins (run time)
 - ▲ *explicit allocation/deallocation*
 - programmer is responsible for memory allocation
 - new/release, malloc/free, etc.
 - ▲ *implicit allocation/deallocation*
 - stack variables (local to a *block*)
 - some subprogram parameters

 If a language has *static variables* only (like Fortran),
is recursion possible?

103

Scope

We already defined the *scope* of a name as the range of statements in a program to which the name is *visible*.

More rigorously:

- the *scope* of a name N denoting an entity E is the set of all statements in a program for which N refers uniquely to E
- the definition implies that there are places in a program where N does not refer uniquely to E
 - ▲ there may be places where N refers to something other than E
 - ▲ there may be places where N does not refer to anything


104

Blocks

Some imperative languages are called *block-structured languages* because programs are organized as nested blocks.

- A *block* is a *delimited* group of statements, possibly named.
 - ▲ a group of statements is *delimited* if there are markers indicating the beginning and end of the group
 - ▲ blocks can have their own local variable declarations; these variables can only be used within the block
 - ▲ many block-structured languages allow the programmer to associate a *name* with the marker indicating the beginning of a block; examples of named blocks are *procedures* and *functions*
 - ▲ a block without a name is called an *anonymous block*

 Is Pascal a block-structured language?

 Would you consider C a block structured language?

105

Anonymous Block Examples

Ada

```
cylDiam: REAL := 2.8;
cylDepth: REAL := 8.3;
cylVolume: REAL;

put("The volume is: ");
declare
  pi: constant REAL := 3.142;
  circArea, cylRadius: REAL;
begin
  cylRadius := cylDiam/2;
  circArea := pi*cylRadius**2;
  cylDepth := circArea*cylDepth;
end;
put(cylVolume);
```

C

```
float cylDiam = 2.8;
float cylDepth = 8.3;
float cylVolume;

printf("The volume is: ");
{
  const float pi = 3.142;
  float circArea, cylRadius;

  cylRadius = cylDiam/2;
  circArea = pi*cylRadius*cylRadius;
  cylVolume = circArea*cylDepth;
}
printf("%f", cylVolume);
```

106

Named Blocks and Nesting

Most block-structured languages allow unlimited nesting of blocks.

Consider the Pascal example:

- unlimited nesting of *procedures* (named blocks) is allowed
- names are only *visible* inside the block where they're defined (including inside *subblocks*)

🔍 Where is Z visible?

Pascal

```
program P;
  var X: integer

  procedure A;
    var Y: char;

    procedure B;
      var Z: Boolean
    begin SB end;

  begin SA end;

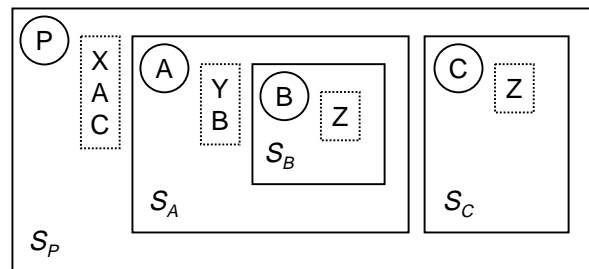
  procedure C;
    var Z: integer;
  begin SC end;

begin SP end.
```

107

Visibility

Consider a slightly more abstract view of our Pascal program (called a *nesting diagram*):

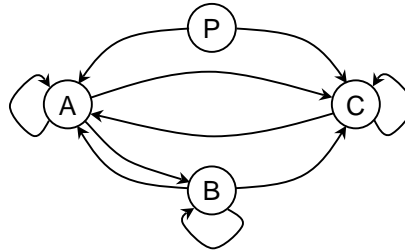


- There are *eight* names in our program (including P):
 - ▲ P X A C Y B Z (in B) Z (in C)
- we can make the names *unique* by specifying their *context*
 - ▲ P P.X P.A P.C P.A.Y P.A.B P.A.B.Z P.C.Z

108

Potential Call Graphs

Here is another kind of diagram called a *potential call graph*. The graph shows which *named blocks* are visible to each other (and could potentially be called).



- recall that a name is *visible* to statements inside the block where the name is defined; e.g., names visible to S_B :
 - ▲ P.A, P.B, P.C, P.X, P.A.Y, P.A.B, P.A.B.Z

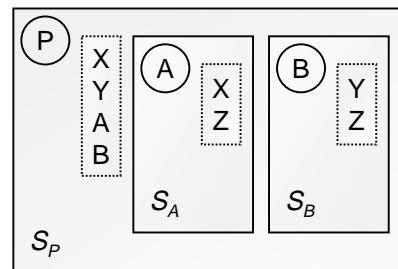
🔍 Why is P not visible to the statements in B (S_B)?

109

Referencing Environments

The *referencing environment* of a statement is the set of variables that are *visible* to the statement.

- referencing environment of S_P :
 - ▲ P.X, P.Y, P.A, P.B
- referencing environment of S_A :
 - ▲ P.A.X, P.A.Z, P.Y, P.A, P.B
- referencing environment of S_B :
 - ▲ P.B.Y, P.B.Z, P.X, P.A, P.B



Note that $P.X$ is visible to S_P and S_B , but *not* to S_A . $P.A.X$ *hides* $P.X$ in S_A ; A is a *hole in the scope* of $P.X$.

- in S_A the name X refers to $P.A.X$
- in S_P and S_B the name X refers to $P.X$

110

Dynamic Scoping

- The scoping discussed so far has been *static scoping* (a.k.a. *lexical scoping*). With static scoping, the scope of a name never changes and can be determined independently of program execution.
 - ▲ a name is visible within the block where it is *defined* in the program (including within subblocks of that block)
- With *dynamic scoping*, the scope of a name depends on program execution, and may change depending on subprogram calls.
 - ▲ a name that is visible within the currently executing block is visible within blocks *called* from within the currently executing block.


111

Static Scoping vs. Dynamic Scoping

This Pascal program shows the difference between static and dynamic scoping.

 What does the name X refer to?

| | static | dynamic |
|---|--------|---------|
| ① | | |
| ② | | |
| ③ | | |

 What does the program output?

| static | dynamic |
|--------|---------|
| | |

```

program P;
  var X: integer;
  procedure A;
  begin
    X := X + 1;
    print(X)
  } ③
end;
  procedure B;
  var X: integer;
  begin
    X := 42;
    A
  } ②
end;
begin
  X := 13;
  B
} ①
end.
  
```

112