Names

What can we name?

- mutable variables
- values
- functions
- types
- type constructors (e.g., list or vector)
- classes
- modules/packages
- execution points (labels)
- execution points with environment (continuation)
Binding times

A *binding* is an association of two things. The first is usually a name.

*Binding time* is the time at which the association is made.

Binding times:

- Language design time: semantics of most language constructs
- Language implementation time: implementation dependent semantics
- Compile time
- Link time
- Run time

*Static* means before run time, *dynamic* means during run time.
Binding in C++

```cpp
1 class Base {
2   public:
3       virtual void value() { cout << "base class"; return; }
4   };
5 class Child : public Base {
6   public:
7       virtual void value() { cout << "child class"; return; }
8   };
9
10 int main() {
11    Base x;
12    Child y;
13    x = y;
14    x.value();  // static binding
15    Base *xp = new Child();
16    Base &xr = y;
17    xp->value();  // runtime binding
18    xr.value();  // runtime binding
19    return 0;
20 }
```
Scope and lifetime

**Scope**: the region of program text where a binding is active.

**Lifetime**: the period of time between the creation of an entity and its destruction.

Note that these talk about two different things. Scope is a *place* (or many places), whereas lifetime is a *time span*. 
Lifetimes

For objects residing in memory, there are typically three areas of storage, corresponding to different lifetimes:

- **static** objects: lifetime of entire program execution
  - globals, **static** variables
- **stack** objects: from the time the function or block is entered until the time it is exited
  - local variables
- **heap** objects: arbitrary lifetimes, not corresponding to the entrance or exit of a function or block
  - dynamically allocated objects, e.g., with **new**
Scopes

Two major scoping disciplines:

- **static**: binding of a name is given by its declaration in the innermost enclosing block
  - Most languages use some variant of this
  - *Closest nested scope* rule usually applies.

- **dynamic**: binding of a name is given by the most recent declaration encountered at runtime
  - Used in Lisp, Snobol, APL
Scoping example

```javascript
var x = 1;

function f () { print x; }

function g () { var x = 10; f(); }

function h () { var x = 100; f(); }

f(); g(); h();
```

<table>
<thead>
<tr>
<th>Scoping</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>1 1 1</td>
</tr>
<tr>
<td>Dynamic</td>
<td>1 10 100</td>
</tr>
</tbody>
</table>
program A;
var I:integer;
   K:char;

procedure B;
var K:real;
   L:integer;

procedure C;
var M:real;
begin
   (*scope A+B+C*)
   end;

   (*scope A+B*)
end;

   (*scope A*)
end.
Static scoping variations

What is the scope of \( x \)?

```plaintext
{
  statements1;
  var x = 5;
  statements2;
}
```

- C++, Ada: `statements2`
- Legacy C: `statements2` (but `statements1` not allowed)
- Javascript: entire block
- Pascal: entire block, but not allowed to be used in `statements1`!
Memory Allocation

- Static: allocated once at compile time (usually in protected memory.)
  Usually include:
  - Strings, constants, static variables.

- Stacks: allocated in frames on a first-in last-out basis. Frames usually store:
  - Actual parameters
  - Temporaries
  - Local variables
  - Bookkeeping information
  - Return address

- Heap: allocated from main memory according to an allocation policy.
  - First-fit
  - Best-fit
Overloading

*Overloading* is a form of ad-hoc polymorphism whereby methods and operators can have several meanings depending on context.

- Functions: normally distinguished by the function signature.
- Custom memory allocation (*C++*: `new` and placement-`new`)
- Operators
  - Some languages can define new operators (*ALGOL 68*, *Fortran*, *F#*, *Smalltalk*)
  - And others can’t. (*ML*, *Prolog*)
  - Some languages will overload only a limited set (*C++*, *Pascal*, *C#*)
  - And others don’t support overloading at all. (*C*, *Java*, *JavaScript*, *BASIC*)

Do not confuse with a similar but distinct concept of *coercion*. 
Control Structures

A *control structure* is any mechanism that departs from the default of straight-line execution.

- **selection**
  - if statements
  - case statements

- **iteration**
  - while loops (unbounded)
  - for loops
  - iteration over collections

- **other**
  - goto
  - call/return
  - exceptions
  - continuations
Hardware Primer

All high level statements must ultimately execute on hardware. Each hardware platform is different, but they share core characteristics.

- Microprocessor, bus, memory, ICs, peripherals.
- Main memory. Harvard vs. Von Neumann architecture.
- Registers: general, special-purpose (e.g., flags, program counter).
- Instructions: opcode plus operands.
- Frequent operations:
  - Move information between memory and registers.
  - Execute instructions (e.g., math, bit manipulation, compare)
  - Branch, conditionally or unconditionally.
- Cache.
  - Stores frequently accessed data and instructions.
  - Tries to predict what the hardware will do next.
  - Biased toward small blocks and backward branching.
The Infamous GoTo

- In machine language, there are no if statements or loops.
- We only have branches, which can be either unconditional or conditional (on a very simple condition).
- With this, we can implement loops, if statements, and case statements. In fact, we only need
  1. increment
  2. decrement
  3. branch on zero

  to build a universal machine (one that is Turing complete).

- We don’t do this in high-level languages because unstructured use of the goto can lead to confusing programs. See “Go To Statement Considered Harmful” by Edgar Dijkstra.
Selection

- if Condition then Statement – Pascal, Ada
- if (Condition) Statement – C/C++, Java
- To avoid ambiguities, use end marker: end if, “}”
- To deal with multiple alternatives, use keyword or bracketing:

if Condition then
    Statements
elsif Condition then
    Statements
else
    Statements
end if;
if Condition1 then
    if Condition2 then
        Statements1
    end if;
else
    Statements2
end if;
Statement Grouping

- Pascal introduces begin-end pair to mark sequence
- C/C++/Java abbreviate keywords to `{ }`
- Ada dispenses with brackets for sequences; keywords for the enclosing control structure are sufficient
  ```
  for J in 1..N loop ...
  end loop
  ```
  - More writing but more readable

- Another possibility – make indentation significant (e.g., ABC, Python, Haskell)
Short-circuit evaluation

if \( x/y > 5 \) then \( z := \ldots \) -- what if \( y = 0? \)
if \( y /= 0 \) and \( x/y > 5 \) then \( z := \ldots \)

But binary operators normally evaluate both arguments.

Solutions:

- a lazy evaluation rule for logical operators (Lisp, C)
  
  \[
  C_1 \&\& C_2 \quad // \text{don’t evaluate } C_2 \text{ if } C_1 \text{ is false}
  \]
  
  \[
  C_1 \|\| C_2 \quad // \text{don’t evaluate } C_2 \text{ if } C_1 \text{ is true}
  \]

- a control structure with a different syntax (Ada)
  
  -- don’t evaluate \( C_2 \)
  
  \[
  \text{if } C_1 \text{ and then } C_2 \text{ then} \quad -- \quad \text{if } C_1 \text{ is false}
  \]
  
  \[
  \text{if } C_1 \text{ or else } C_2 \text{ then} \quad -- \quad \text{if } C_1 \text{ is true}
  \]
Multiway selection

Case statement needed when there are many possibilities “at the same logical level” (i.e., depending on the same condition)

case Next_Char is
  when 'I' => Val := 1;
  when 'V' => Val := 5;
  when 'X' => Val := 10;
  when 'C' => Val := 100;
  when 'D' => Val := 500;
  when 'M' => Val := 1000;
  when others => raise Illegal_Numeral;
end case;

Can be simulated by sequence of if-statements, but logic is obscured.
The Ada case statement

- no flow-through (unlike C/C++)
- all possible choices must be covered
  - if all choices not covered explicitly, default action is mandatory
- no inaccessible branches:
  - no duplicate choices (C/C++, Ada, Java)
- choices must be static (Ada, C/C++, Java, ML)
- in many languages, type of expression must be discrete (e.g., no floating point, no string)
Implementation of case

A possible implementation for C/C++/Java/Ada style case:
(If we have a finite set of possibilities, and the choices are computable at compile-time.)

- build table of case handlers, one entry for each case
- transform input value to table index
- branch to that address
- execute
- branch to end of case statement (if break keyword used)

This is not the typical implementation for a ML/Haskell style case.
Complications

case (n+1) is
  when integer’first..0 ⇒ Put_Line("negative");
  when 1 ⇒ Put_Line("unit");
  when 3 | 5 | 7 | 11 ⇒ Put_Line("small prime");
  when 2 | 4 | 6 | 8 | 10 ⇒ Put_Line("small even");
  when 21 ⇒ Put_Line("house wins");
  when 12..20 | 22..99 ⇒ Put_Line("manageable");
  when others ⇒ Put_Line("irrelevant");
end case;

Implementation would be a combination of tables and if statements.
Consider: Serial Copy

```c
void send (int *to, int *from, int count) {
    do { /* precondition: count > 0 */
        *to++ = *from++;
    } while (--count > 0);
}
```

This is called serial copy. It requires a test and branch after each copy.

Is there a more efficient way of doing this?
void send (int *to, int *from, int count) {
    register n = (count + 7) / 8;
    switch (count % 8) {
        case 0: do { *to++ = *from++; case 7: *to++ = *from++; case 6: *to++ = *from++; case 5: *to++ = *from++; case 4: *to++ = *from++; case 3: *to++ = *from++; case 2: *to++ = *from++; case 1: *to++ = *from++; }
            while (--n > 0);
    }
}
Duff’s Demystified (How Case Statements Actually Work in C)

if(x==0) goto label_case1;
if(x==1) goto label_case2;
if(x==2) goto label_case3;
if(x==3) goto label_case4;
goto label_finish;

label_case1: do-something();
goto label_finish; /* break */
label_case2: do-something();
goto label_finish; /* break */
label_case3: do-something();
goto label_finish; /* break */
label_case4: do-something();

label_finish:
Indefinite loops

- All loops can be expressed as while-loops
  - good for invariant/assertion reasoning
- condition evaluated at each iteration
- if condition initially false, loop is never executed

```plaintext
while condition loop ... end loop;
```

is equivalent to

```plaintext
if condition then
  while condition loop ... end loop;
end if;
```

if `condition` has no side-effects
Executing while at least once

Sometimes we want to check condition at end instead of at beginning; this will guarantee loop is executed at least once.

- repeat ... until condition; (Pascal)
- do { ... } while (condition); (C)

can be simulated by while + a boolean variable:

```plaintext
first := True;
while (first or else condition) loop
  ...
  first := False;
end loop;
```
Breaking out

A more common need is to be able to break out of the loop in the middle of an iteration.

- break (C/C++, Java)
- last (Perl)
- exit (Ada)

```plaintext
loop
  ... part A ...
  exit when condition;
  ... part B ...
end loop;
```
Breaking way out

Sometimes, we want to break out of several levels of a nested loop

- give names to loops (Ada, Perl)
- use a goto (C/C++)

Outer: while C1 loop ...
   Inner: while C2 loop ...
      Innermost: while C3 loop ...
         exit Outer when Major_Failure;
         exit Inner when Small_Annoyance;
         ...
      end loop Innermost;
   end loop Inner;
end loop Outer;
Definite Loops

Counting loops are iterators over discrete domains:

- for J in 1..10 loop ... end loop;
- for (int i = 0; i < n; i++) { ... }

Design issues:

- evaluation of bounds
- scope of loop variable
- empty loops
- increments other than 1
- backwards iteration
- non-numeric domains
Evaluation of bounds

for J in 1..N loop
    ...
    N := N + 1;
end loop; -- terminates?

Yes – in Ada, bounds are evaluated once before iteration starts.
Note: the above loop uses abominable style.

C/C++/Java loop has hybrid semantics:
for (int j = 0; j < last; j++) {
    ...
    last++; -- terminates?
}

No – the condition “j < last” is evaluated at the end of each iteration.
The loop variable

- is it mutable?
- what is its scope? (i.e., local to loop?)

Constant and local is a better choice:

- *constant*: disallows changes to the variable, which can affect the loop execution and be confusing
- *local*: don’t need to worry about value of variable after loop exits

Count: integer := 17;
...
for Count in 1..10 loop
  ...
end loop;
... -- Count is still 17
Different increments

Algol60:

\[
\text{for } j \text{ from } \text{exp1} \text{ to } \text{exp2} \text{ by } \text{exp3} \text{ do } ...
\]

* too rich for most cases; typically, \text{exp3} is +1 or -1.
* what are semantics if \text{exp1} > \text{exp2} and \text{exp3} < 0?

C/C++:

\[
\text{for (int } j = \text{exp1}; \ j <= \text{exp2}; \ j += \text{exp3}) \text{ do } ...
\]

Ada:

\[
\text{for } J \text{ in } 1..N \text{ loop } ...
\]
\[
\text{for } J \text{ in reverse } 1..N \text{ loop } ...
\]

Everything else can be programmed with a while loop
Non-numeric domains

Ada form generalizes to discrete types:

```
for M in months loop ... end loop;
```

Basic pattern on other data types:

- define primitive operations: `first`, `next`, `more_elements`
- implement `for` loop as:

```java
iterator = Collection.Iterate();

for (element thing = iterator.first;
     iterator.more_elements();
     thing = iterator.next()) {
    ...
}
```