Name/Identifiers and their 6 Attributes

→ 1. Name
   = identifier
   length, legal chars, case-sensitivity, special words
   can be one-one, many-one, or none-one mapping to memory

→ 2. Address
   pointing to a location in memory
   may vary dynamically

→ 3. Type
   range of values + legal operations
   variable, constant, label, pointer, program, ...

→ 4. Representation/Value
   interpreted contents of the location
   l-value (address)
   r-value (value)

→ 5. Scope
   Range of statements over which the variable is visible.
   Static/dynamic

→ 6. Lifetime
   Time during which the variable is bound to a storage location.
Name/Value Graph

- name
  - possesses
  - attributes: type scope
  - denotes
  - address
    - denotes
    - physical cell
      - contains
      - contents
        - interprets
        - representation
          - input/output
          - value
A number is an adjective and thus can only be represented!!
Name/Value Graph - Example II

- EVAL
  - possesses
  - attributes:
    - type: program
    - scope: OS
  - denotes
  - 5930F2
    - denotes
    - (memory)
      - contains
      - 0100001010100100100
      - 0100100010001000100
      - 0011100010001001010
  - interprets
  - (the algorithm)
    - input/output
  - (program)
Name/Value Graph - Example III

- **P** possesses attributes:
  - type: pointer
  - scope: BLIP

- **830FF** denotes (memory)

- **153A4** contains

- **address 153A4** interprets input/output

\( ^\wedge X \)
Binding

• How and when are attributes bound to identifiers?
  
  → Static
    
    occurs before runtime (compile time, link/load time)
    
    constant throughout program execution
  
  → Dynamic
    
    occurs or can change during runtime

• In many ways, the various binding times determine the flavor of a language.

• As binding time gets earlier:
  
  → efficiency goes up
  
  → safety goes up
  
  → flexibility goes down
BINDING TIMES

- LANGUAGE DESIGN
- COMPILER DESIGN/IMPLEMENTATION
- COMPILE TIME
- LINK/LOAD TIME
- RUN TIME
## BINDING TIMES

<table>
<thead>
<tr>
<th></th>
<th>Language Design</th>
<th>Compiler Design</th>
<th>Compiler Time</th>
<th>Link/Load Time</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set of objects in Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Association of Type with Identifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial value “of” identifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Possible Meanings of Operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Representation of Values</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Type Binding

When is type bound to variable?

How is binding specified?

- **Static typing** (before runtime)
  - explicit declaration
    
    var x: integer
  
  - implicit declaration by prior agreement
    
    e.g., Fortran:
    
    I = 5 First mention of I -- integer
    PI = 3.14 First mention of PI -- real

- **Advantages:**
  
  - cheaper
  
  - safer

- **Disadvantage:**
  
  - less flexible
Type Binding

- **Dynamic Binding** (after compile time)
  - Identifier gets type of value assigned to it as needed.
    
    ```
    x := 5  -- x is of type integer
    ...
    x := "foo"  -- x is of type string
    ```

- **Advantage:**
  - flexibility

- **Disadvantages:**
  - runtime overhead -- extra work at run-time
  - poor error detection -- type checking is replaced by type changing
  - you will lose 5 points in an homework assignment for reusing identifiers

- More about types later...
Scope

- **Static (lexical) scope**
  - Scope of a identifier is determined by the textual layout of the program.
  - In block structured languages, scope of an identifier is
    - the unit in which it is defined, plus
    - all units immediately nested inside the declaring unit (excluding those in which the variable is redeclared), plus
    - all units within which the declaring unit is nested.
  - To find the declaration of an identifier,
    1. lift all declarations to the top of the unit
    2. look through the statically enclosing units until a declaration is found.

- **Dynamic Scope**
  - Scope of an identifier depends on program execution, and therefore changes dynamically.
  - To find declaration, look up through the call chain.
Example (evaluate both ways)

```pascal
program foo;
var x: integer;

procedure f;
begin
  print(x);
end f;

procedure g;
var x: integer;
begin
  x := 2;
  f;
end g;

begin
  x := 1;
  g;
end foo.
```
Lifetime (= extent)

- The lifetime of a variable is the interval of time during which it is bound to a specific memory location.

- Static identifiers
  - bound to memory cells before execution (load time)
  - retain same binding throughout execution
  - efficient, inflexible
  - allow history-sensitivity
  - do not support recursion
  - Ex: FORTRAN variables

- Semidynamic identifiers
  - storage allocated when unit is called
  - storage deallocated when unit returns
  - allows recursion
  - Ex: Variables declared in Pascal Procedures
Lifetime (continued)

- Explicit Dynamic Identifiers
  - storage allocated and deallocated by programmer
  - *new, dispose* in Pascal
  - flexible and efficient, but dangerous
  - Ex: Pointer variables

- Implicit Dynamic Identifiers
  - automatically bound to storage as needed
  - storage automatically reclaimed when no longer needed
  - flexible, safe, less efficient
  - Ex: lists in LISP, Prolog
Scope ≠ Lifetime

- lifetime > scope: storage that can't be accessed through that variable.
  
  ```pascal
  var p: ^integer;
  begin
    ...
    new(p)
    ...
  end
  here, storage is still allocated but p is not defined, Lifetime > Scope
  
  ```

- scope > lifetime: variable without storage.
  
  ```pascal
  var p: ^integer;
  begin
    ...
    new (p)
    ...
    dispose(p)
    end
  here, p is defined but has no value, Scope > Lifetime
  ```
Scope ≠ Lifetime (continued)

- Also, scope has "holes" during execution, but lifetime does not.

```pascal
procedure f;
var x: real;
begin
  ...
end

procedure g;
var x: integer;
begin
  ...
  f
  ...
end
```

out of x's scope during execution of f (assuming static scope), but x's lifetime persists.
Variable Initialization

- static or dynamic

- once for static variables, at each allocation for dynamic variables

- many possible methods:
  
  sum: integer := 0; (Ada)

  int first := 10 (Algol 68 initialization)

  int first = 10 (Algol 68 constant declaration)

- unavailable in Pascal

- default initializations
Aliasing

- Two variables are aliases if they share the same storage.

```pascal
var x,y: ^integer; var x,y: integer;
begin
  new (x);
  x^ := 5;
  y := x;
  y := x;
  x^ := 10;
  writeln (y);
  writeln (y^);
end
```

- Also results from *var* parameters:

```pascal
procedure p (var x,y: integer);
...

p (a,b);
p (x1,x1);
```