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
   - aliasing
   - point to a location in memory
   - anonymous - new (p)
   - p^ points to location that has no associated name
   - 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.

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
- COMPILATION TIME
- LINK/LOAD TIME
- RUN TIME

<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</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Association of Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>with Identifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initial value &quot;of&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Possible</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meanings of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Representation of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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:
      - `l = 5` -- `l` is of type integer
      - `PI = 3.14` -- `PI` is of type real

- **Advantages:**
  - cheaper
  - safer

- **Disadvantage:**
  - less flexible

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

- More about types later. . .
Type Inferencing

- Type inferred based on surrounding context
- ML (also Miranda/Haskell)
  - `fun circum(r) = 3.14 * r * r`
    - `*` takes matching types
    - `3.14` \( \Rightarrow \) \( r \) is real
    - function returns real
  - `fun times10(x) = 10 * x`
    - `10` \( \Rightarrow \) \( x \) is integer
    - function returns int
  - `fun square(x) = x * x`
    - Cannot infer type
    - Programmer can help…
  - `fun square(x:int) = x * x`
    - \( x \) is int
    - function returns int

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) addr of mem "name"
  ...
  dispose(p)
  here, p is defined but has no value,
  Scope > Lifetime
  end
  ```
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
  ...
end
```

Type Checking

- Ensuring that operands of an operator are of compatible types

- Strong typing
  - A language is strongly typed if type errors can always be detected
    - Compile time:
      - `real := real + char`
    - Runtime:
      - `variant record`

- What popular languages are strongly typed?
  - Fortran: NO (actual/formal parameters not checked)
  - Pascal: NO (variant record without tag)
  - Ada: NO (unchecked conversion)
  - C/C++: NO (union problem)
  - ML: YES (statically declared or inferred)
Type Compatibility

- **name**
  - efficient but restrictive

- **structural**
  - flexible, harder to implement
  - **Issues:**
    - hard to make types incompatible
    - what is considered the same structure?
      - array \([0..10]\) of int
      - array \([1..11]\) of int
    - what about two records?
      - \(a: \text{integer},\)
      - \(b: \text{real};\)
      - and
      - \(x: \text{integer},\)
      - \(y: \text{real};\)