names

(source: Louden, "Programming Languages, Principles and Practices", 2nd Edition, Ch. 5, pp. 125-134)

- p. 126: "A fundamental abstraction mechanism in a programming language is the use of **names**, or **identifiers**, to denote language entities or constructs."
- "In most languages, **variables**, ... [**subroutines**], and **constants** can have names assigned by the programmer."
- "A fundamental step in describing the semantics of a language is to describe the conventions that determine the meaning of each name used in a program."

location and value

(source: Louden, Ch. 5, pp. 125-134)

- other important concepts to understand, related to but separate from names: **location** and **value**
  - **value**: any storable quantity
  - **location**: a place where a value can be stored

a name's meaning

(source: Louden, Ch. 5, pp. 125-134)

- p. 127: "The **meaning** of a name is determined by the properties, or **attributes**, associated with the name."
- what attributes? These can vary by kind of name and by programming language, but some examples include:
  - data type
  - value
  - location

examples of name attributes: example 1

(source: Louden, Ch. 5, pp. 125-134)

- C language example:

```
const int MAGIC = 5;
```

- associates with the name `MAGIC` the following attributes:
  - a data type attribute of 'integer constant'
  - a value attribute of 5
  - (note: in C, the 'constant' attribute is part of the data type -- in other languages, this might not be the case...)
  - (...another example of different languages including different aspects as part of the data type: early Pascal considered an array's size to be part of its type)

examples of name attributes: example 2

(source: Louden, Ch. 5, pp. 125-134)

- another C language example:

```
int quantity;
```

- associates with the name `quantity` the following attributes:
  - an attribute 'variable'
  - a data type attribute of 'integer'

examples of name attributes: example 3

(source: Louden, Ch. 5, pp. 125-134)

- yet another C language example:

```
double functy(int value)
{
    return (value * 2.0)/7.0;
}
```

- associates with the name `functy` the following attributes:
  - an attribute 'function'
  - the number, names, and data types of its parameters (here, one parameter with name `value` and data type 'integer')
  - the body of code to be executed when `functy` is called (here, the return statement with its computation)

examples of name attributes: examples 4, 5

(source: Louden, Ch. 5, pp. 125-134)

- Are declarations the only language constructs that can associate attributes to names? NO;
- for example: (still in C)

```
x = 2;
```

  - this assignment statement associates the new attribute 'value 2' to the variable `x`

```
int* y;
y = new int;
```

  - `y` is a pointer variable
  - the assignment statement allocates memory for an integer variable -- it associates a location attribute to it -- as well as associates a new value attribute to `y`

binding

(source: Louden, Ch. 5, pp. 125-134)

- p. 128: "The process of associating an attribute to a name is called **binding**."
- "An attribute can be classified according to the time during the translation/execution process when it is computed and bound to a name.
  - ...called the **binding time** of the attribute."
- two broad categories of binding times:
  - **static binding**: "occurs prior to execution"
  - **dynamic binding**: "occurs *during* execution"
- **static attribute**: *able* to be bound statically;
- **dynamic attribute**: must be bound dynamically;

binding

(source: Louden, Ch. 5, pp. 125-134)

- p. 128: "The process of associating an attribute to a name is called **binding**."
- "An attribute can be classified according to the time during the translation/execution process when it is computed and bound to a name.
  - ...called the **binding time** of the attribute."
- two broad categories of binding times:
  - **static binding**: "occurs prior to execution"
  - **dynamic binding**: "occurs *during* execution"
- **static attribute**: *able* to be bound statically;
- **dynamic attribute**: must be bound dynamically;

which attributes are which?

(source: Louden, Ch. 5, pp. 125-134)

- p. 128: "Languages differ SUBSTANTIALLY in which attributes are bound statically and which are bound dynamically"
- binding times may depend on the kind of translator being used, too;
- for example:
  - languages that support the functional programming model often have more dynamic binding
  - interpreters perform most bindings dynamically
  - compilers perform more bindings statically

which attributes are which? part 2

(source: Louden, Ch. 5, pp. 125-134)

- "To make the discussion of attributes and binding **independent** of such translator issues,
  - we usually refer to the binding time of the attribute as the **earliest** time that the **language rules** permit the attribute to be bound."

which attributes are which? examples

(source: Louden, Ch. 5, pp. 125-134)

```
const int MAGIC = 2;
```

- the value 2 is bound **statically** to the name `MAGIC`

```
int val;
```

- the data type 'integer' is bound **statically** to the name `val`

```
val = 2;
```

- binds the value 2 to `val` **dynamically** when the assignment statement is **executed**

```
/* C++ */ y = new int;
```

- **dynamically** binds a storage location to `*y` and assigns that location as the value of `y`

ASIDE: stages of execution

(source: MacLennan, Ch. 2, pp. 43-44)

...when a compiler is your translator...

- the stages for early FORTRAN (still frequently used):

1. Compilation
2. Linking
3. Loading
4. Execution

- **compilation**: translates individual statements into relocatable object code
- **linking**: incorporates references to external, already-compiled subprograms (e.g. libraries)
- **loading**: places (or loads) the program into memory -- converting it from relocatable to absolute format

ASIDE: phases of compilation

(sources: MacLennan, Ch. 2, pp. 43-44, Scott, Ch.1, pp. 27-28 and 33-34)

- the phases of compilation for early FORTRAN (still frequently used):

1. Lexical and syntax analysis
2. Optimization
3. Code synthesis

- **lexical analysis (scanning)**: read characters and group them into tokens
- **syntax analysis (parsing)**: organizes tokens into a parse tree (based on, often, a CFG)
- **optimization**: transforming the result so far to make it more efficient
- **code synthesis**: put together the parts of the object code instructions in relocatable format

subcategories of static binding

(source: Louden, Ch. 5, pp. 128-129)

A static attribute may be bound...

- ...when the language is defined (**language definition** time)
- ...when the language is implemented (**language implementation** time)
- ...during parsing (**translation** time or **compile** time)
- ...during the linking of the program with libraries (**link** time)
- ...during the loading of the program for execution (**load** time)

subcategories of static binding - examples, part 1

(source: Louden, Ch. 5, pp. 128-129)

- ...of **language definition time** binding:
  - predefined identifiers that have their meaning (and thus their attributes) specified by the language definition -- such as when the two type `boolean` values are specified as `true` and `false`
- ...of **language implementation time** binding:
  - when the range of the integer type is determined by the implementation;
- ...of **compile time** binding:
  - in `const int n = 2;`
    can't all of that be bound at compile time?

subcategories of static binding - examples, part 2

(source: Louden, Ch. 5, pp. 128-129)

- ...of **link time** binding:
  - "...the **body** of an externally-defined function will not be bound until link time"
- ...of **load time** binding:
  - "...and the location of a global variable is bound at load time, since its location does not change during the execution of the program."

dynamic binding - examples

(source: Louden, Ch. 5, pp. 128-129)

- methods called in Java or virtual calls in C++
- types of Prolog or Scheme variables
- bindings of values to variables...! 8-)

tradeoffs: earlier vs. later binding times

(source: Louden, Ch. 5, pp. 128-129, and Scott, Ch. 3, p. 113)

- binding times definitely impact the design and implementation of programming languages;
- IN GENERAL, **early** binding times are associated with **more efficiency**
- IN GENERAL, **later** binding times are associated with **more flexibility**

scope of a binding

- (Louden, p. 134) the **scope** of a binding is "the region of the program over which the binding is maintained"
- **static scoping**: (Scott, p. 123) follows the structure of the code as it appears in **written** form;
  - with this, don't have to consider the flow of control at run time to determine a name's scope;
- **dynamic scoping**: (Scott, p. 139) "the bindings between names and objects depend on the flow of control at run time, and in particular on the order in which subroutines are called."
  - the 'current' binding for a given name is the one encountered most recently during execution, and not yet destroyed by returning from its scope"