

CS421 Lecture 16: Names and Variables¹

Mark Hills

`mhills@cs.uiuc.edu`

University of Illinois at Urbana-Champaign

July 17, 2006

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Names and Variables

Binding

Binding Categories

Scope

Objectives

By the end of this lecture, you should

- ▶ understand the attributes of variables in a programming language, including concepts such as type and value
- ▶ understand the distinction between static and dynamic properties of variables
- ▶ be familiar with the concept of scope, specifically differences between static and dynamic scope

Names

Def: A *name* is a string of characters used to identify some entity in a program²

- ▶ Names (or *identifiers*) give us a way to name program concepts, such as types, values, classes, functions, etc.
- ▶ Different languages have different rules for names
 - ▶ Case-sensitive (C, C++, Java) or not (Pascal, Lisp)?
 - ▶ Are there special words in a language that are *reserved words*, or are there just *keywords*
 - ▶ Naming requirements/conventions
 - ▶ Name length and allowed characters

²Sebesta, 2nd edition, p.207

Variables

A variable is just an abstraction of memory. We can define a variable as a sextuple of attributes:

- ▶ Name
- ▶ Address
- ▶ Value
- ▶ Type
- ▶ Lifetime
- ▶ Scope

Variables: Names

Remember, names are defined with certain characters and conventions

- ▶ In Haskell and ML, variables start with lowercase letters
- ▶ In Haskell and ML, type constructors start with uppercase letters
- ▶ C-family languages have no similar restrictions, but generally use conventions (Java camel-case, etc)
- ▶ Older Fortran had a 6-character limit, all uppercase names, and “type inference” for variables based on the first letter

Most variables have names, although some do not (heap-allocated variables may not, more on this later)

Variables: Address

The address is the machine memory address that is associated with the variable. This can be trickier than it sounds:

- ▶ Multiple variables may refer to the same address – *aliasing*
 - ▶ pointers and references
 - ▶ Equivalence-type statements and unions
 - ▶ function calls
- ▶ The same name may have multiple addresses at different times
 - ▶ OS-based – different physical address, same virtual address
 - ▶ language-based – variables in functions may have different addresses on each call

Sometimes called an **I-value**, since a variable's address is used on the left-hand side of an assignment.

Variables: Types

The type of a variable determines what kinds of values the variable can refer to. We covered this in Lecture 6.

Variables: Values

The value is what we typically think of when we talk about variables. This is just the data associated with the variable.

- ▶ We will think in terms of *abstract* memory cells to hold the value – that way we don't have to worry about representational issues (word size, packing, bit formats, etc)
- ▶ The value is sometimes called the **r-value**, since it is used on the right-hand side of an assignment; we must know the l-value to get the r-value

Binding

When we talk about associating attributes with variables, we are talking about binding.

- ▶ **Def:** In the general sense, a **binding** is an association between two items, such as variables and attributes
- ▶ **Def:** The time this association occurs is the **binding time**

Binding Times

Binding can take place at:

- ▶ Language design time (identifying * as multiplication)
- ▶ Language implementation time (an int takes 4 bytes)
- ▶ Compile time (variable x is an int)
- ▶ Load time (variable x is in storage location 10)
- ▶ Link time (function f is in load module m)
- ▶ Run time (x has the value 10)

Example

What are the bindings here, and when do they occur?

```
count = count + 5;
```

Static vs. Dynamic Bindings

Bindings are either *static* or *dynamic*:

- ▶ Static bindings first occur before run time and remain unchanged during program execution
- ▶ Dynamic bindings either occur during run time or occur earlier but can be changed during run time

We've already seen one example of this in Lecture 6 – dynamic typing, where the type is either not computed prior to execution or can change during execution.

An Aside: Constants

Constants are similar to variables except the value, once assigned, cannot be changed.

- ▶ Usually they can be allocated statically
- ▶ Some languages allow constants that are defined once at runtime but then not changed (sometimes called readonly)
- ▶ Literals can be thought of as constants (i.e. 5, "hello", etc.)

Declarations

We can use a *declaration* to specify one or more of the attributes for a variable.

- ▶ An **explicit** declaration requires explicitly mentioning the attribute value in the declaration – such as `int x`
- ▶ An **implicit** declaration allows the attribute value to be determined based on context and conventions (e.g. in Fortran, variables starting with I, J, K, L, M, N were considered to be integers)

Declarations typically give a variables name and type, but may provide other attributes (initial value, scope, lifetime).

Storage and Lifetime

- ▶ **Def:** Taking memory from the pool of free memory and assigning it to a variable is **allocation**
- ▶ **Def:** Returning this memory to the free pool is **deallocation**
- ▶ The **lifetime** of a variable is the time between the initial allocation and the future deallocation

In some languages, assigning storage for a variable also provides a starting value. In others, the variable has a random starting value, based on the existing memory contents.

Bindings and Variable Categories

We can separate binding of scalar (unstructured) variables into four categories, to make them easier to organize and compare:

- ▶ Static Variables
- ▶ Stack-Dynamic Variables
- ▶ Explicit Heap-Dynamic Variables
- ▶ Implicit Heap-Dynamic Variables

Static Variables

Static variables are bound to memory cells before execution begins, remaining bound to the same locations until execution ends.

- ▶ This can include globals and specially-designated static variables
- ▶ Allows for history sensitivity and improved performance
- ▶ Prevents recursive calls (if only static supported) and can waste memory

Stack-Dynamic Variables

Stack-dynamic variables are those whose storage binding are created when their declarations are *elaborated* – when the binding process indicated by the declaration is executed. This then occurs at *run-time*.

- ▶ Example: variables at the start of a method declaration are elaborated when the method is invoked and then deallocated when the method exits
- ▶ Storage is taken from the stack
- ▶ This is the default in many languages
- ▶ Ease of allocation and space management are advantages
- ▶ ...while cost of indirect lookups (generally off a stack or frame pointer) is a disadvantage

Explicit Heap-Dynamic Variables

Explicit heap-dynamic variables are nameless variables created explicitly by run-time calls in a program – for instance, with `malloc` or `new`.

- ▶ Explicit heap-dynamic variables are allocated from the heap
- ▶ They do not have names, but instead a pointer or reference to this variable is saved into another variable – note this means we have two variables!
- ▶ Some languages include explicit deallocation as well, while others use garbage collection
- ▶ Advantage: storage requirements do not need to be known before the program starts
- ▶ Disadvantage: heap management is tricky and can be slower

Implicit Heap-Dynamic Variables

Implicit Heap-Dynamic Variables have all attributes (value, type, etc) set on assignment. Languages like JavaScript use implicit heap-dynamic variables.

- ▶ Advantage: extremely flexible
- ▶ Disadvantage: expensive – all information maintained at runtime
- ▶ Disadvantage: hard to check/verify without running the code
- ▶ Plus, this has the heap management disadvantages of explicit heap-dynamic variables

Scope

The *scope* of a variable refers to the part of a program in which that variable is visible – in which it can be referenced.

- ▶ **Def:** Scope is **static** if it can be determined at compile time – scope in this case is a *lexical* property of a program (it is determined from the program text, and not from an execution)
- ▶ **Def:** Scope is **dynamic** if it is based on the execution of a program – if the variables that are visible at any point in time are determined by the execution path through the program

Note: scope and lifetime are **not** the same!

Static Scope

- ▶ Scope computed at compile time
- ▶ Names associated with declarations based on program text
- ▶ Subprograms and blocks generate a hierarchy of scopes – the subprogram or block containing the current subprogram or block is its *static parent*

Finding the Definition

We can usually use the following procedure to find the definition referenced by a name:

- ▶ Is the variable local? If yes, done
- ▶ If nonlocal, is it in the static parent? If yes, done, if no, check the static parent's static parent
- ▶ If no declaration is found, error

Visually, this is like putting your finger on the name and tracing up the page until you find the declaration.

Example

```
1 program main;  
2   var x : integer;  
3   procedure sub1;  
4     var x : integer;  
5     begin (* sub1 *)  
6       ... x ...  
7     end; (* sub1 *)  
8   begin (* main *)  
9     ... x ...  
10  end (* main *)
```

Which x goes which which?

Dynamic Scope

With dynamic scope, we determine which variable a name refers to based on which variable we've seen most recently with that name, based on the path through the program (the sequence of subroutine calls), that is still active (has not been deallocated).

- ▶ Early versions of LISP used this
- ▶ Now thought generally to be a mistake
- ▶ Common LISP now uses static scope, with an option for dynamic
- ▶ Perl also allows variables to have dynamic scope

Example

```

1 program main;
2   var x : integer;
3   procedure sub1;
4     begin (* sub1 *)
5       ... x ...
6     end; (* sub1 *)
7   procedure sub2;
8     var x : integer;
9     begin (* sub2 *)
10      ... call sub1 ...
11    end; (* sub2 *)
12  begin (* main *)
13    ... call sub2 ...
14    ... call sub1 ...
15  end (* main *)
    
```

Which x goes which which?

Referencing Environment

Def: The **referencing environment** of a point in a program is the set of all variables visible to that program point. In a statically scoped language, this consists of all local variables along with all variables in static ancestors, minus those that have been hidden by another declaration.

Example

```

1 program main;
2   var x, y : integer;
3   procedure sub1;
4     var z : integer
5     begin (* sub1 *)
6       ... point1 ...
7     end; (* sub1 *)
8   procedure sub2;
9     var w, x : integer;
10    begin (* sub2 *)
11      ... point2 ...
12    end; (* sub2 *)
13    ... point3 ...
14  end; (* main *)
  
```

- ▶ point1: main.x main.y sub1.z
- ▶ point2: main.y sub2.w sub2.x
- ▶ point3: main.x main.y