The current topic: Types and values

- ✓ Introduction
- ✓ Object-oriented programming: Python
- ✓ Functional programming: Scheme
- ✓ Python GUI programming (Tkinter)
- Types and values
- Logic programming: Prolog
- Syntax and semantics
- Exceptions

Announcements

- Reminder: Term Test 2 is on Monday November 3rd in **GB405**, *not in the regular lecture room*.
  - 50 minutes (11:10 – 12:00).
  - You're allowed to have one double-sided aid sheet for the test. You must use standard letter-sized (that is, 8.5" x 11") paper. The aid sheet can be produced however you like (typed or handwritten).
  - **Bring your TCard.**
  - What's covered?
    - Everything from September 29 up to and including October 24.
    - Lab 2.
  - An old Term Test 2 has been posted.
  - The exercises at the end of each lecture are also good practice.

- Office hours next week:
  - The office hour on Wednesday (November 5th) is **cancelled**. Instead, there will be an office hour on Thursday (November 6th), 1:00-2:00, in SF3207.

Basic topics: types, values, scopes

- Names
- Storage
- Types
- Scopes and referencing environments
- Functions as parameters

These concepts are useful in understanding most programming languages (and programs).

- Reference: Sebesta, chapter 5

Names

- A name identifies a variable (or other thing)

- The identified variable has attributes:
  - name
  - memory address
  - type
  - value
  - scope
  - lifetime

Memory addresses

- Usually considered unchangeable – a variable can't be moved to a different address.

```
int i, j, *p;
p = &i;
p = &j;
```

  - The variable p isn't moving; it's stays in the same memory location but stores different values (its **values** are addresses).
- But if you use the same variable name in different functions, the name means different addresses in different places.
- Some languages allow names to be redefined.
  - Scheme: define, set
  - Python: depends on the way we look at things
    - All variables really store references, and the variables themselves can't be moved.
    - But a variable can be made to refer to a different object (in a different memory location).

Terminology: static vs dynamic

- *Static*: "during compilation"
  - more generally, before the program begins to run
  - "compile-time"
- *Dynamic*: "while the program is running"
  - "run-time"
- Some things that can be either static or dynamic:
  - binding
  - errors
  - storage allocation
- The "static" keyword in C, C++, Java is related but still different.
  - "just once"
  - "invisible"
  - "class-related"

Binding

- *Binding* attaches an attribute to a variable
- Names are a little different from other attributes: we think of names and what they denote as being the same.
- A name is bound to a variable *statically* in most of the languages we're used to.
- In other languages, the name-variable binding is *dynamic*.
  - The same name can be re-bound to a different variable.
  - e.g. Scheme

Dynamic name binding

- Dynamic binding: While a program is running, you can redefine a name to mean a different variable.
- Can't be done in C or Java

```
int i = 5;
i = 6;
```

  - The name denotes the same variable, though the value changes.
- C++'s reference variables provide an alternative name for the same variable.

```
int i = 0;
int& k = i
k = 6  // Now i == 6 too.
```

  - And they would allow dynamic name binding if you could do this:

```
int j = 2;
k = j&;  // trying to make k be a reference to j.
```

  - But in fact you **can't** change what k refers to.

Dynamic name binding

- Here we have dynamic name binding in pseudo-code:

```
define i : int
i = 5
define i : string
i = "hi"
```

  - The same name comes to denote some different thing (where "thing" really means a memory location).

- Languages like Scheme and ML allow this.

Dynamic binding to storage

- We said binding of a variable to a memory address is "Usually considered unchangeable".

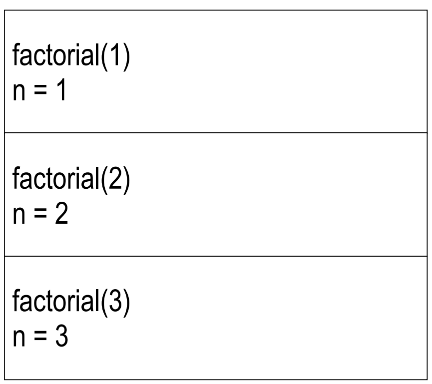
- But that doesn't mean that storage binding is always static.
  - Stack-dynamic: e.g. local variables in functions
  - Heap-dynamic: nameless "variables" (we usually think of them as blocks of memory rather than as variables) allocated at run-time using `malloc`, `new`, or implicitly allocated at run-time during object creation (e.g. in Python).

- If variables are appearing and disappearing, then their connection to a memory location is dynamic.
  - But while the variable exists, this connection is unchangeable (in C and similar languages).

```
int f() {
  int i; // different calls of f() have different i's
  ...
}
```


Storage on the stack

- Suppose `factorial` is a recursive function. In each call of `factorial` on the stack, there is a distinct variable n, and each such variable n has a fixed location (while it exists).



Storage on the heap

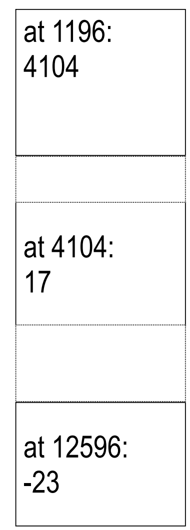
- A heap storage example (in C):

```
int *p;
// p is on the stack at location 1196

p = malloc(sizeof(int));
// now p == 4104
*p = 17

p = malloc(sizeof(int));
// now p == 12596
*p = -23;
```

- Observe that the above code creates two heap-dynamic variables (each having the same size as an integer).



The lifetime of a variable

- Stack-dynamic storage is allocated on function entry, and deallocated on function exit.
  - Storage management is always automatic, and not left to the programmer.

- Heap-dynamic storage is allocated on demand. It is freed either on demand or as needed:
  - Freed on demand
    - If heap freeing is managed by programmers, as in C and C++.
  - Freed when needed
    - If heap freeing is managed by an automatic *garbage collector*, as in Scheme, Java, Python.

Types

- "X is a *strongly typed* language."
  - Where X might be Java, Python, …

- Strong typing may not be clearly defined, but Sebesta's description helps:
  - A language is strongly typed if "type errors are always detected."
  - This implies that "the types of all operands can be determined, either at compile time or at run time."
  - Sebesta, pg. 219.

- Strongly typed languages include:
  - Python
  - Java
  - C#

- What about C/C++?
  - Not strongly typed, since there is no type checking when using *union* types.

Static vs. Dynamic typing

- This is separate from the issue of strongly vs not-strongly typed.

- Static typing:
  - Types are declared by the programmer.
  - Or types are deduced from code by the compiler. For example, in the language ML:

    ```
    fun square (x : int) = x*x;
    ```
  - Obviously `fun` returns an int, and you don't have to say so because the ML compiler can figure it out.

- Dynamic typing:
  - Types are determined at run-time.
  - e.g. Scheme, Python

Types in records

(A *record* is a C-type struct.)

```
struct { int a; char b; };
struct { int x; char y; };
```

- Are these the **same** type?
  - When using *name type compatibility*: no
    - Name type compatibility: Types are only the same if they have the same name.
    - Used by C++, Java, Python.

  - When using *structure type compatibility*: yes
    - Structure type compatibility: Types are the same as long as they have the same structure.

Scope

- *Scope*: the part of a program where a variable can be referred to.
- In *block-structured* languages, scope generally consists of:
  - The block where the variable is declared,
    plus
  - blocks contained by that block.

Blocks in C

```
int f(...) {
  int x;
  ... // can refer to x here

  for (...) {
    int y;
    int x; // another x; this declaration "hides" the old x

    ... // can refer to x (the new x) and y here
  }

  ... // can refer to x (the original x) here
}
```


Blocks in Python

```
def outer():
    x = 3;
    def sub1():
        # We can't modify x but we can refer to it.
        print 'sub1: x =', x
    def sub2():
        # Or we can define another x which "hides" the old one.
        x = -5
        print 'sub2: x =', x
    sub1()
    sub2()
    print 'outer: x =', x

outer()
```

Output:

```
sub1: x = 3
sub2: x = -5
outer: x = 3
```


Dynamic caller, static ancestor

- Caller: the procedure that at run time initiated the execution of the current procedure.
  - This is dynamic, not fixed at compile-time.
- Ancestor: the procedure which encloses the code of the current procedure, in the text that the programmer wrote.
  - This ***is*** fixed at compile-time.
  - Just read the code to find out.
- In a block-structured language:
  - Static scoping: what names can be used is determined by ancestry.
    - Used in most languages.
  - Dynamic scoping: what names can be used is determined by calling history.
    - Not used very much. One example: The original LISP.

Caller vs ancestor

```
procedure A:
  var x : integer;
  x := 2;

  procedure sub1:
    x := x + 1;
  end sub1;

  procedure sub2:
    var x : integer;
    x := 1;
    sub1; // Which x is changed by this call?
  end sub2;

end A;
```

- Using static scoping, `A`'s `x` is changed.
- Using dynamic scoping, `sub2`'s `x` is changed.

Implementing scope

- Function calls are recorded on the stack.
  - Therefore, the structure of the stack records calling history, not ancestry.
- Stack frame for a function call must include a *static link* to the enclosing scope in the program text.
  - The static link records ancestry.
- The "enclosing scope" is some other function, so the static link is usually a pointer to a lower stack frame.
  - Is this always the case?

Scope vs lifetime

- *Scope* is the part of the program text where you can refer to a variable.
- *Lifetime* is the time period of the program's execution when the variable exists.
- A variable may exist during the execution of code that is not within its scope:

```
void b() {
  ... /* x isn't visible here */
}
void a() {
  int x;
  b(); /* x exists while this call is running */
}
```

- Can a variable's scope include parts of the program that are active (at run time) when the variable does not exist?

Referencing environment

- The *referencing environment* is the set of names that can be used at a particular point in a program
- Determining the referencing environment is simple in a language with static scope.
  - The set of usable names is the set of names that are in-scope and not hidden.
  - This can be found from the program text.
- Determining the referencing environment is harder in a language with dynamic scope.
  - This can only be found by understanding execution, since it's depends on the calling history.

Referencing environments in C and Java

- C: two sets of names are available:
  - names defined locally within a function
  - names defined globally, externally to functions
    - some of these invisible, through "static" declarations (in other files).
- Java: many sets of names
  - names defined locally within a method
  - instance variables for the current object
  - class variables for the current object's class
  - public instance variables for other objects
    - These objects must themselves be part of the referencing environment.
  - public class variables for other classes
- Java has no set of globally-defined names like C's.

A dynamic-scope example

- In a C-like language with dynamic scoping:

```
void f1() { int a, b; 1 ... }
void f2() { int b, c; 2 ... f1(); }
void main() { int c, d; 3 ... f2(); }
```

- Which variables are accessible at points 1, 2 and 3?
  - At point 1: `f1`'s `a`, `f1`'s `b`, `f2`'s `c`, `main`'s `d`.
  - At point 2: `f2`'s `b`, `f2`'s `c`, `main`'s `d`.
  - At point 3: `main`'s `c`, `main`'s `d`.

"First-class" status for a type

- A *first-class type* is a type with values that can be:
  - created at run time
  - assigned to variables
  - returned from functions
  - passed as an argument
  - exist without a name
- You can think of values of a first-class type as "things" in your program.
  - "Things" that you can work with.
- Are functions first-class?

A C example

```
double integrate(double *f(double), double a, double b)
    { ... sum += f(x); ... }

double myFun (double x)
    { return 3*x*x + 2*x + 1; }

int main(void) {...
  printf(integrate(myFun, 0.5, 12));
  ...
}
```

Are functions things in C?

- C functions can be:
  - passed as parameters
  - returned as function results
  - assigned to variables
  - ***but,*** only in the form of pointers to statically-defined functions

- They *cannot* be created at run time.
- They *cannot* exist without a name.

- That is, C doesn't have anything like a lambda expression.

"Functions" in Java

- Functions in Java are methods of objects or classes.

- They cannot be referred to except by calling them.

- They can be "created" at run time by instantiating an anonymous class:

```
MyFile x = new MyFile(){int getFileType() { return 3;}};
```

  - This defines (and instantiates) a nameless class that extends `MyFile`.
  - But the code for the method exists at compile time.

- Unlike C functions, Java methods cannot be passed as parameters.

So, are functions first-class?

- Not in C or Java.

- Yes, in functional languages: Scheme, Lisp, ML

- Almost yes, in Python:
  - Recall that lambda expressions in Python can only consist of a single expression.
    - This restricts our ability to create functions at run time.

- Prolog (which we'll be looking at next) doesn't exactly have functions, but its terms can behave like functions and can be created at run time.

Referencing in parameter functions

- If you pass a function as a parameter, what names can it refer to?

- *Shallow binding*: the names available where the function is actually called.

- *Deep binding*: the names available where the function was defined.
  - This is what is done in Python.

- This is not the same as the distinction between dynamic and static binding!

- Reference: Sebesta, Section 9.6.

Example: shallow vs deep

- Looks like C but isn't:

```
int x;
f2( ) { printf(x); }
f3( ) { int x = 3; f4(f2); }
f4(void f( )) { int x = 4; f( ); }
x = 1;
f3( );
```

- Shallow: prints ____
- Deep: prints ____
- See the next slide for answers...

Answers

- Shallow: 4
- Deep: 1
- And 3? That's *ad-hoc* binding.
  - Ad-hoc binding: The names that are available are those available in the function that *passes* (as opposed to receives) the parameter.