Programming Languages

Lecture 4
Benefits of dynamic typing

Not *adapted from Dan Grossman's PL class,*

U. of Washington

Declaring functions in C++ vs Python

C++ uses *static typing*: most code can be checked at compile-time to make sure rules involving types are not violated.

```
int double(int n) {
   return 2 * n;
}
```

Python uses dynamic typing: most code cannot be checked for type errors at compile-time; this has be delayed until run-time.

```
def double(n):
   return 2 * n
```

Dynamic typing

- Racket (like most Scheme or Lisp dialects) is dynamically typed.
- Some characteristics of dynamic typing:
  - Values have types, but variables do not.
    - A variable can refer to different types during its lifetime.
  - Most type-error bugs cannot be found before the program is run, and not until the offending line of code is encountered.
    - Possible to write code with type errors that aren't discovered for a long time, if buried in code that isn't executed often.
  - Traditionally (but not always), dynamically-typed languages are interpreted, whereas statically-typed languages are compiled.

Some good things about dynamic typing

- Enables polymorphism (enabling code to handle any data type).
  - Example: Calculating the length of a list.

```
(define (length lst)
   (if (null? lst) 0 (+ 1 (length (cdr lst)))))
```

versus

```
int length_int_array(int_node* array) {
   if (array->next == NULL) return 0;
   else return 1 + length_int_array(array->next);
}
```

Easier to create flexible data structures

- In Racket, it's easy to create a list that can contain any other kind of data structure:
  - List of integers: '(1 2 3)
  - List of booleans: '(#f #f #t #f #t)
  - List of strings: '("a" "b" "c")
  - List of mixed types: '("a" 42 #f)
  - List of really mixed types: '(17 (3 #f) ("hi") -9 (1 (2 (3) 4 ())))
- Also, all of these lists will work with our length function!

- Mixing types in a single data structure is not easy in statically-typed languages.
- In C++, arrays or vectors must all hold the same type.

"Manual" type-checking

- Dynamically-typed languages often have some way for the programmer to discover the type of a variable.
- In Racket (all of these return #t or #f):
  - `number?`
    - `also integer?, rational?, real?`
  - `list?`
  - `pair?`
  - `string?`
  - `boolean?`
- Enables a single function to do different things depending on the type of an argument.

Length of a list vs length of nested lists

- For "regular" list
  - if empty list, return 0
  - else return 1 + length of the cdr of the list.

- For a list with possible nested lists…
  - if empty list, return 0
  - if the car of the list is a list…   do what?
  - else (car is not a list)…   do what?

Length of a list vs length of nested lists

- For "regular" list
  - if empty list, return 0
  - else return 1 + length of the cdr of the list.

- For a list with possible nested lists…
  - if empty list, return 0
  - if the car of the list is a list
    - return length of the car (which is a list) plus length of cdr
  - else (car is not a list)
    - return 1 + length of the cdr

Length of a list vs length of nested lists

```
(define (length-nested lst)
  (cond ((null? lst) 0)
        ((list? (car lst))
           (+ (length-nested (car lst))
              (length-nested (cdr lst))))
        (#t (+ 1 (length-nested (cdr lst))))))
```

Let's do some practice…

- A "secret" of Racket/Scheme that I haven't told you:
- Function bodies may contain more than one expression.
  - In "pure" functional programming, this isn't true.
  - But it's nice to have this facility at times.
  - For debugging, can use (display <whatever>) and (newline)
- Example:

```
(define (length lst)
   (display lst)
   (newline)
   (if (null? lst) 0 (+ 1 (length (cdr lst)))))
```