Week 4

Brendon Faleiro

Scoping

- Scoping defines which *variable declaration* does a particular *variable usage* refers to.
- Environment: Set of variables and their values.
- Every time you create a new variable the variable name and its value is added to the environment.
- Each time you exit a scope, all the variables initialized in that scope are deleted from the environment.

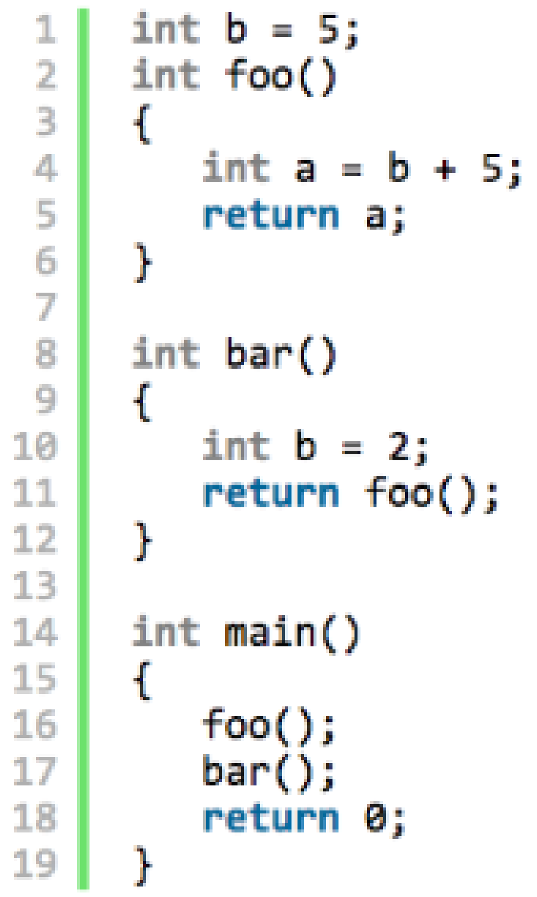
Static Scoping:

- Also called Lexical Scoping
- The value of the variable is decided on the basis of the location of the source code and the lexical context, which is defined by where the named variable or function is defined.
- For static scoping to work, we pass the static environment along with the function every time the function is called.

Dynamic scoping:

- Variable value is defined by the current state when the name is encountered which is determined by the execution context or calling context.
- For dynamic scoping to work, we simply look at the current environment at the time of the function call.

Static vs dynamic scoping



```
int b = 5;
int foo()
{
    int a = b + 5;
    return a;
}

int bar()
{
    int b = 2;
    return foo();
}

int main()
{
    foo();
    bar();
    return 0;
}
```

Statically:
foo() returns 10
bar() returns 10

Dynamically:
foo() returns 10
bar() returns 7

Source: https://msujaws.wordpress.com/2011/05/03/static-vs-dynamic-scoping/

How to typecheck?

- Look up the type of the function.
- Compute the type of input.
- Check that we find an instantiation of the type variable with some type that makes the actual and formal parameters types to be equal.
- The type of the whole thing is the result type of the function.

Type Inference

- let f list =
```
let rec aux n = function list -> match list with
| [] -> n
| _::t -> aux (n+1) t
in aux 0 list;;
```

- What does this function do?

- What is the type of the function?

```
val f : 'a list -> int = <fun>

# let g a b c = c b a;;
  val g : 'a -> 'b -> ('b -> 'a -> 'c) -> 'c = <fun>
```

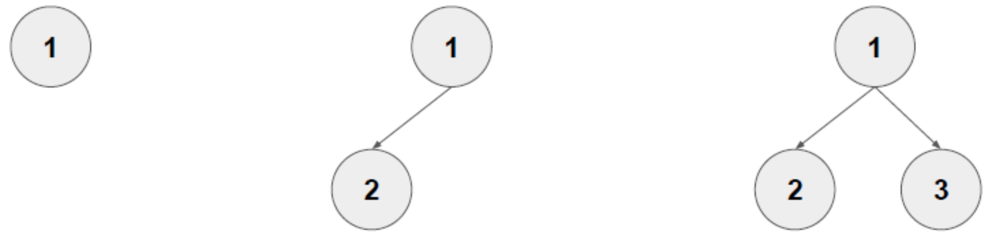
User defined types

```
Syntax: type name = typedef;;
# type ‘a my_list = ‘a list;;

# type value = Int of int | String of string;;
  type value = Int of int | String of string

# [Int 3; String "hello"];;
  - : value list = [Int 3; String "hello"]

# type binTree = Leaf | Node of binTree * int * binTree;;
type binTree = Leaf | Node of binTree * int * binTree
```



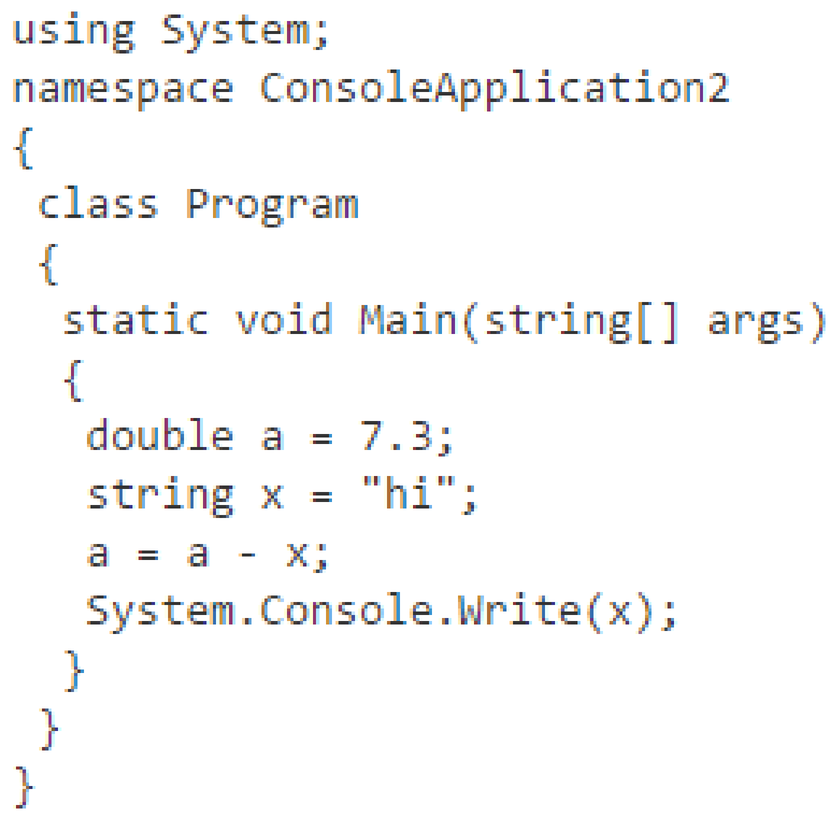
Strongly Typed vs Weakly Typed

Strong Typing:

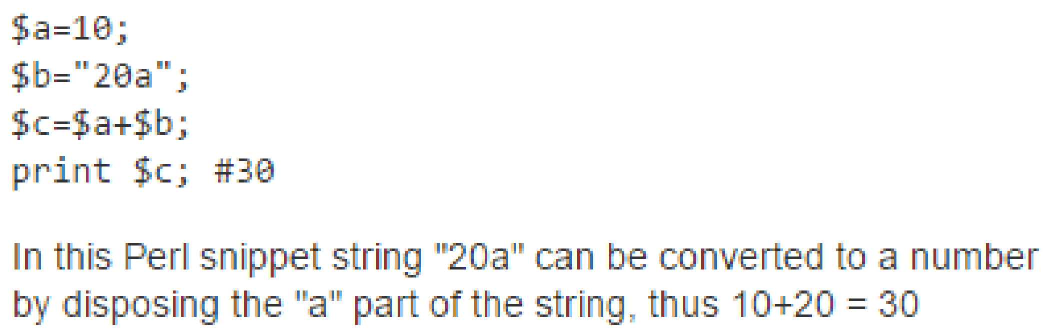
- Almost all modern languages are strongly typed.
    - Exception: C, C++
- You cannot accidently treat one type as another.

Weak Typing:

- Why define a language where there exist programs that, by definition, must pass static checking but then when run can set the computer on fire?
- Ease of language implementation: Checks left to the programmer
- Performance: Dynamic checks take time
- Lower level: Compiler does not insert information like array sizes, so it cannot do the checks

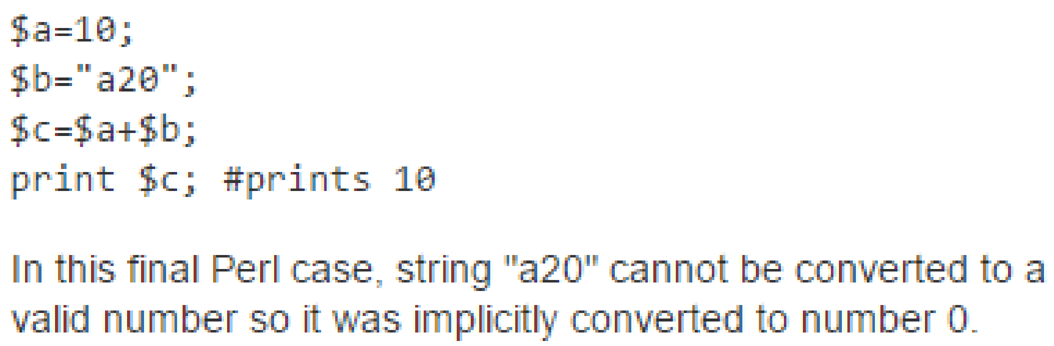


```
using System;
namespace ConsoleApplication2
{
 class Program
 {
  static void Main(string[] args)
  {
   double a = 7.3;
   string x = "hi";
   a = a - x;
   System.Console.Write(x);
  }
 }
}
```



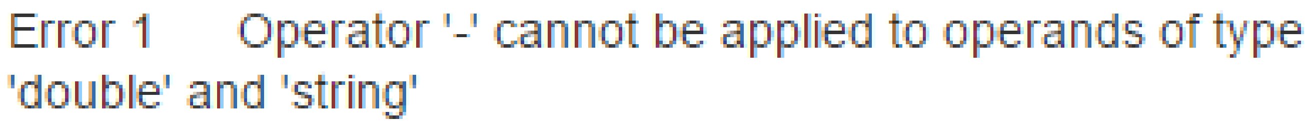
```
$a=10;
$b="20a";
$c=$a+$b;
print $c; #30
```

In this Perl snippet string "20a" can be converted to a number by disposing the "a" part of the string, thus 10+20 = 30



```
$a=10;
$b="a20";
$c=$a+$b;
print $c; #prints 10
```

In this final Perl case, string "a20" cannot be converted to a valid number so it was implicitly converted to number 0.



Error 1 Operator '-' cannot be applied to operands of type 'double' and 'string'

Examples from http://www.i-programmer.info/programming/theory/1469-type-systems-demystified-part2-weak-vs-strong.html

Weak typing leads to insecurity

- Old now-much-rarer saying: “strong types are for weak minds”
    - Idea was humans will always be smarter than a type system (cf. undecidability), so need to let them say “trust me”
    - Reality: humans are really bad at avoiding bugs and we need all the help we can get!

- Reality check: 1 bug in a 30-million line OS written in C can make the whole OS vulnerable

https://www.cs.cornell.edu/courses/cs3110/2014fa/lectures/14/lec14.pdf

Static Typed vs Dynamic Typed

Static Typed

- Type checking is done at compile time (statically).
    - Approach is to give a type to each variable, expression, etc.
    - Purposes include preventing misuse of primitives (e.g., 4/"hi") and avoiding run-time checking
- Static checking is anything done to reject a program...
    - after it (successfully) parses but
    - before it runs
- Nearly all statically typed languages require you to pay for this benefit by writing explicit type declarations for each variable and for the domain and range of each function you define.
- And you must keep these declarations up to date and in sync as you develop and modify your program.

Question:

Which of the following is considered as static checking?

A. Checking that only 7-bit ASCII characters appear in the source code

B. Checking that every left paren is matched by a right paren

C. Checking that all return values of a function have the same type

D. Checking that all pattern matches are exhaustive

E. Checking that the program never causes a division by zero error

Answer

Which is an example of static checking?

A. Checking that only 7-bit ASCII characters appear in the source code // done before parsing

B. Checking that every left paren is matched by a right paren // done during parsing

C. Checking that all return values of a function have the same type

D. Checking that all pattern matches are exhaustive

E. Checking that the program never causes a division by zero error

// has to be done at run-time

- OCaml is statically typed, with all the benefits that entails (including a good part of the speed of OCaml's compiled code).
- You don't need to write explicit type declarations (unless you want to), because the compiler *infers* your types for you.
- If you use your functions and data inconsistently, OCaml detects this and gives you a type error, at compile time, so that you can correct the problem.
- All with no bookkeeping.

- OCaml static checking prevents these errors from ever occurring at run-time:
  - Using arithmetic on a non-number
  - Trying to evaluate a function application e1 e2 where e1 does not evaluate to a function
  - Having a non-Boolean between if and then
  - Having a pattern-match with a redundant pattern (*static check but not a type check*)

- OCaml static checking does not prevent these errors from ever occurring at run-time:
  - Exceptions (e.g., hd [])
  - Division-by-zero

Instead, these are detected at run-time

Dynamically Typed

- Many languages, and most "scripting languages" give you strong *dynamic* typing, which means that you pay for strong typing at runtime:
  - wasted memory (your data are all tagged with type information).
  - wasted cycles (your data are repeatedly type-checked upon every operation).
- Worse, none of this prevents type errors from occurring at runtime -- it just means that the runtime error will clearly explain that it is in the nature of a type conflict.

So are dynamically typed languages really that bad?

- Not having to obey OCaml or Java’s typing rules can be convenient
  - Maybe arrays can hold anything
  - Maybe everything is true except false and []
  - Maybe don’t need to create a datatype just to pass different types of data to a function
  - By not requiring types, tags, etc., more code can just be reused with data of different types
    - e.g., great libraries for working with lists/arrays in languages like Python and Ruby
    - whereas Java and OCaml collections libraries are often have very complicated static types

Basis of many modern "scripting" languages, e.g., Python, Ruby, etc

Implementation can analyze the code to ascertain whether some checks aren’t needed, then optimize them away

Will this pass type-checking?

```
# let f g = (g 7, g true);;
# f (function x -> (x,x));;
```

Ans: Static-> NO

Dynamic -> YES

```
# let rec pow x y =  if y = 0 then 1 else x * pow (x,y-1);;
```

Ans: Static -> NO

Dynamic -> NO

Static Typing

- Convenience:

Can assume the data has a fixed type without any additional information.

- Restrictive:

Does not allow possibly harmless programs.

Dynamic Typing

- Convenience:

You can return a bool or an int without any problem.

You can build heterogenous lists.

- Restrictive:

Allows useful programs but at the cost of tagging everything at runtime.

Conclusion on Static vs Dynamic

- Controversial topic!
- There are real trade-offs here you should know
- Simply debating "static vs. dynamic typing" isn't useful. Most languages have examples of both ends
- "What should we enforce statically" makes more sense to debate
- Ideally would have flexible languages that allow best-of-both-worlds
- Still an open and active area of research!

Parametric Polymorphism vs static overloading

- Parametric Polymorphism:
  - 1 function, different parameter types.
- Static Overloading:
  - Different functions.
  - Function gets called based on the type of input.
- What if static overloading is not allowed?
  - Change the name!

Polymorphic Functions

- Functions which can accept arguments of different (any) types
- The type signature of the function has one or more type variables (like 'a), for the arguments
- The type of return type is usually dependent on the argument type or is fixed
- Can return type be polymorphic, with the argument type is fixed?
  - No. It would violate strong typing, inevitable run-time error!
  - Only example:-   raise: exn -> 'a   (* which is a run-time exception anyway.*)

Identify the Polymorphic functions

```
# let f x = x + 1
x: int, f: int -> int
-  Not polymorphic
```

```
# let f x = 1
x: 'a (anything)  , f: 'a -> int
-  Polymorphic
```

```
# let f x y = x
x: 'a ,  y: 'b, f: 'a -> 'b -> 'a
-  Polymorphic
```

```
# let f (x, y) = x +. y
x: float, y: float , f: (float * float) ->
float
- Not Polymorphic
```

```
# let f x = match x with
    [] -> 0
  | _::t -> 1 + (f t)
x: 'a list ,  f: 'a list -> int
- Polymorphic
```