CS 320 : Scope

Marco Gaboardi

MSC 116

gaboardi@bu.edu

Announcements

- ***Programming Assignment #5*** is due on Friday, April 17.

- ***Grading Policy for Spring 2020***: Read the article in BU Today, University to Offer Students Credit/No Credit Option

- All ***Zoom meetings*** are recorded (by default), and their recordings available for download shortly after the live meetings.

- All ***Zoom links*** for CS 320 are at the bottom of the *Resources* webpage on *Piazza*.

In the preceding lecture, someone said that the BNF grammar that I proposed for `<expr>` is ambiguous. This grammar is repeated on the two slides following this one.

In fact, as I will explain, it is ambiguous only if parentheses are not inserted – either explicitly as terminal symbols in the grammar, or implicitly in the process of generating an instance of `<expr>`.

Arithmetical expressions, a little more general than the grammar in slides 23, 24, 26 of Lecture 17:

<expr> ::= <term> | (<expr> <addop> <expr>)

<addop> ::= add | minus

<term> ::= <var> | <val>

<var> ::= ...

<val> ::= ...

Examples of arithmetical expressions generated by this BNF grammar:

(((3 add 5) minus 6) add (2 minus 1))

(((X add 5) minus Y) add (2 minus Z))

We can omit the parentheses, as terminal symbols, in the preceding BNF grammar to obtain a more abstract BNF grammar

$$\langle expr \rangle ::= \langle term \rangle \mid \langle expr \rangle \; \langle addop \rangle \; \langle expr \rangle$$

$$\langle addop \rangle ::= \cdots$$

$$\langle term \rangle ::= \cdots$$

but then the parentheses have to be re-introduced in generation:

$$\begin{aligned}
\langle expr \rangle &\Rightarrow (\langle expr \rangle \; \langle addop \rangle \; \langle expr \rangle) \\
&\Rightarrow ((\langle expr \rangle \langle addop \rangle \langle expr \rangle) \langle addop \rangle \langle expr \rangle) \\
&\Rightarrow (((\langle expr \rangle \langle addop \rangle \langle expr \rangle) \langle addop \rangle \langle expr \rangle) \langle addop \rangle \langle expr \rangle) \\
&\Rightarrow (((\langle expr \rangle \langle addop \rangle \langle expr \rangle) \langle addop \rangle \langle expr \rangle) \langle addop \rangle (\langle expr \rangle \langle addop \rangle \langle expr \rangle)) \\
&\Rightarrow \cdots \\
&\Rightarrow (((3 \text{ add } 5) \text{ minus } 6) \text{ add } (2 \text{ minus } 1))
\end{aligned}$$

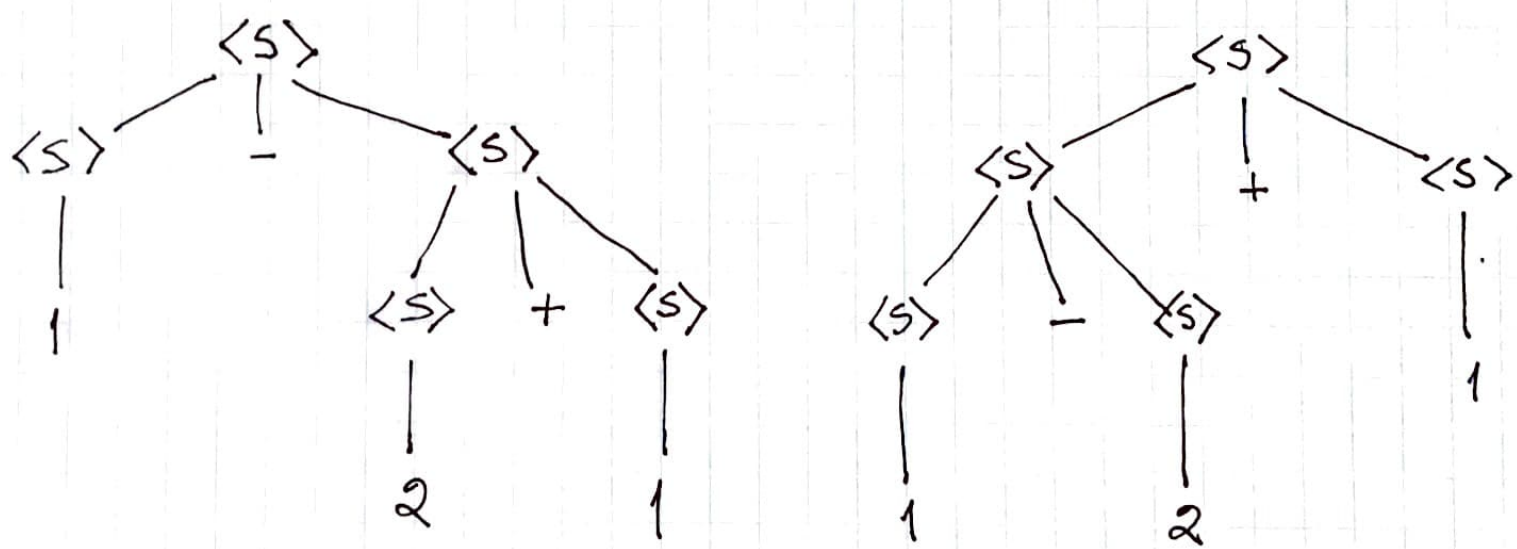
Simplifying the notation a little:

$$\langle S \rangle ::= \langle S \rangle + \langle S \rangle \mid \langle S \rangle - \langle S \rangle \mid 0 \mid 1 \mid \cdots \mid 9$$

Without parentheses, explicitly or implicitly, the grammar is ambiguous: It can generate the same expression according to ~~the~~ two distinct derivations:

$$\langle S \rangle \Rightarrow \langle S \rangle - \langle S \rangle \Rightarrow \langle S \rangle - \langle S \rangle + \langle S \rangle \Rightarrow \cdots \Rightarrow 1 - 2 + 1$$

$$\langle S \rangle \Rightarrow \langle S \rangle + \langle S \rangle \Rightarrow \langle S \rangle - \langle S \rangle + \langle S \rangle \Rightarrow \cdots \Rightarrow 1 - 2 + 1$$



Inserting parentheses in the course of the derivation, i.e. implicitly in the grammar: is .

$$\langle S\rangle \Rightarrow (\langle S\rangle - \langle S\rangle) \Rightarrow (\langle S\rangle - (\langle S\rangle + \langle S\rangle)) \Rightarrow \cdots \Rightarrow (1-(2+1))$$

$$\langle S\rangle \Rightarrow (\langle S\rangle + \langle S\rangle) \Rightarrow ((\langle S\rangle - \langle S\rangle) + \langle S\rangle) \Rightarrow \cdots \Rightarrow ((1-2)+1)$$

Interpretation of $(1-(2+1)) \neq$ Interpretation of $((1-2)+1)$

We can also disambiguate the grammar explicitly, by inserting parentheses as terminal symbols in the rules:

$$\langle S\rangle ::= (\langle S\rangle + \langle S\rangle) \mid (\langle S\rangle - \langle S\rangle) \mid 0 \mid 1 \mid \cdots \mid 9$$

TopHat Q1 - Q2

Variables

Variables

- Functional languages use variables as names (where the association name-value is stored in an environment).
  - We can remember the association, or read the value, but we cannot change it.
- Imperative languages are abstractions of von Neumann architecture
  - A variable abstracts the concept of memory location
- Understanding how variables are managed is an important part to understand the semantics of a programming language.

Mutable vs Immutable Variables

- When we consider variables as names we are working with immutable variables (e.g. the part of OCaml we studied)
- When we consider variables as memory locations we are working with mutable variables (e.g. Python, c, etc.)
- Understanding how variables are managed is an important part to understand the semantics of a programming language.

Learning Goals for today

- Understanding the concept of scope and how languages can differ in their scoping rules.
- Understanding the concept of binding and binding-time.

Scoping rules

Variable names

How shall we evaluate this expression?

Is it referring to this definition?


```
let k=3 in (let k=2 in k+5)
```


Is it referring to this definition?

What is the value of k here?

Scope of a variable

- The scope of a variable is the range of statements over which it is visible
- The scope rules of a language determine how references to names are associated with variables


```
let k=3 in (let k=2 in k+5)
```

OCaml scoping rule says that a variable name is statically associated with the closest definition in the abstract syntax tree.

Back to our example

This is the first declaration we find

Start from here


```
let k=3 in (let k=2 in k+5)
```


To find the value of k we look/search declarations, first locally, then in increasingly larger enclosing scopes

Another example

This is the first declaration we find

Start from here


```
let k=3 in (let z=2 in k+5)
```


To find the value of k we look/search declarations, first locally, then in increasingly larger enclosing scopes

Another example

This is the first declaration we find

Start from here


```
let k=3 in k + (let k=2 in k+5)
```


This is the first declaration we find

Start from here

Another example

This is the first declaration we find

Start from here


```
let k=3 in (let k=2 in k+5) + k
```


This is the first declaration we find

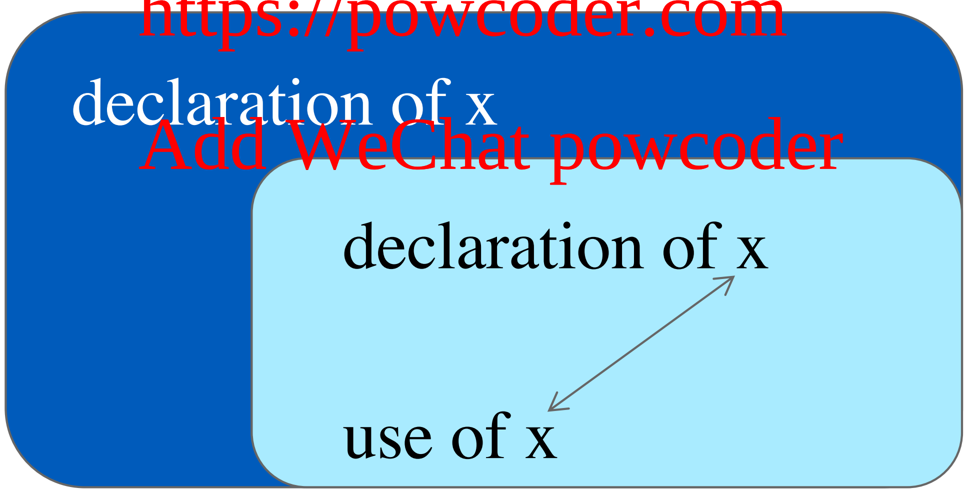
Start from here

Static Scope

- Based on program text
- To connect a name reference to a variable, we (or the compiler) must find the declaration
- Some languages allow nested subprogram definitions, which create nested static scopes
- Search process:
  search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name

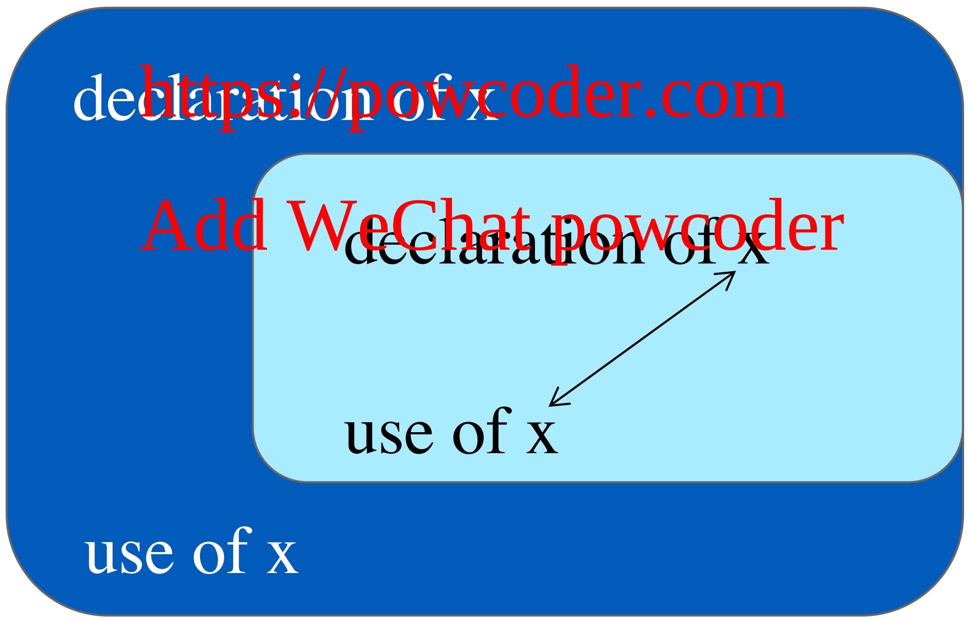
Static Scope

- Variables can be hidden from a unit by having a "closer" variable with the same name



Static Scope

- Variables can be hidden from a unit by having a "closer" variable with the same name



Static Scope

- Search process:
  search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Most of the modern languages are statically scoped: Python, Java, Scala, etc.

TopHat Q3-Q10

Scope Blocks

A method of creating static scopes inside program units (ALGOL 60)


```
void sub() {
   int count;
   while (...) {
    int count;
        count++;
         ...
     }
    ...
 }
```

Program constructs (“blocks”) create scopes

Scope Block Example:

```
int main()
{
  int x=5;
  {
    int x=4;
    printf("The value of x in the block is %d\n", x);
  }
  printf("The value of x in outside the block is %d",
x);
  return 0;
}
```


In C we can write a program like the one above

Scope Block Example:


```
int main()
{
  int x=5;
  {
    int x=4;
    printf("The value of x in the block is %d\n", x);
  }
  printf("The value of x in outside the block is %d",
x);
  return 0;
}
```

```
main.c: In function 'main':
main.c:17:57: error: 'x' undeclared (first use in this function)
     printf("The value of x in outside the block is %d", x);
                                                          ^
```

Dynamic Scope

- Based on calling sequences of program units, not their textual layout,
- You can think about it more as temporal rather than spatial,
- References to variables are connected to declarations by searching back through the chain of subprogram calls that brought execution to this point.

Dynamic Scope Example

```
function big() {
  function sub1(){
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```


big calls sub1
sub1 calls sub2
sub2 uses x

- Static scoping -- Ref to x in sub2 is to big's x
- Dynamic scoping-- Ref to x in sub2 is to sub1's x

Dynamic Scope Example in bash

```
$ x=1
$ function g () { echo $x ; x=2 ; }
$ function f () { x=3 ; g ; }
$ f # does this print 1, or 3?
$ g # does this print 1, 2 or 3?
$ echo $x # does this print 1,2 or 3?
```


`echo $x` corresponds to printing the value of the variable `x`.

What does this program print?

Another Example in bash

```
$ x=1
$ function h () { x=2 ; echo $x ; }
$ function g () { x=3 ; echo $x ; h;}
$ function f () { x=4 ; echo $x; g ; }
$ f # What does this print?
$ g # What does this print?
$ h # What does this print?
$ echo $x # What does this print?
```


What does this program print?

TopHat Q11 - Q13

Variables classification

- The local variables of a program unit are those that are declared in that unit
- The nonlocal variables of a program unit are those that are visible in the unit but not declared there
- Global variables are a special category of nonlocal variables

Local Variables in Ocaml - examples

```
let k=3 in 6 + k
```

The variable k is local to the body of the let


```
fun k -> 8 * k
```


The variable k is local to the body of the function

```
let x=3 in (let k=2 in k + x) + x
```

The variable k is local to the body of the internal let, and the variable x is local to the body of the external let.

Do we have global variables in OCaml?

Global Variables in Ocaml - examples


```
let foo = fold_left (*) 0;;
```


When we create a variable at the top level, we can think about it as global.

TopHat Q14 - Q15

Bindings

The Concept of Binding

- A binding is an association between an entity and an attribute. Examples:
  - a variable and its type,
  - a variable and its value,
  - a function and its name,
  - an operation and its symbol.

The Concept of Binding

```
slope : point -> point -> float option
```



```
let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
    Some s ->
      print_string ("Slope: " ^ string_of_float s)
  | None ->
      print_string "Vertical line.\n"
```

Possible Binding Times

- Binding time is the time at which a binding takes place.
  - Language design time -- E.g. bind operator symbols to operations,
  - Language implementation time -- E.g. bind floating point type to a representation
  - Compile time -- E.g. bind a variable to a type (e.g. in C or Java)
  - Load time -- E.g. bind a static variable to a memory cell (e.g. C or C++)
  - Runtime -- E.g. bind a non-static local variable to a memory cell.

Storing Bindings

- Each binding must be recorded in some specific data structure.
- For example, an environment stores a set of bindings of values to variables:

$$((x_1=v_1),(x_2=v_2),\ldots,(x_n=v_n))$$

- As another example, a typing environment stores a set of bindings of types to variables:

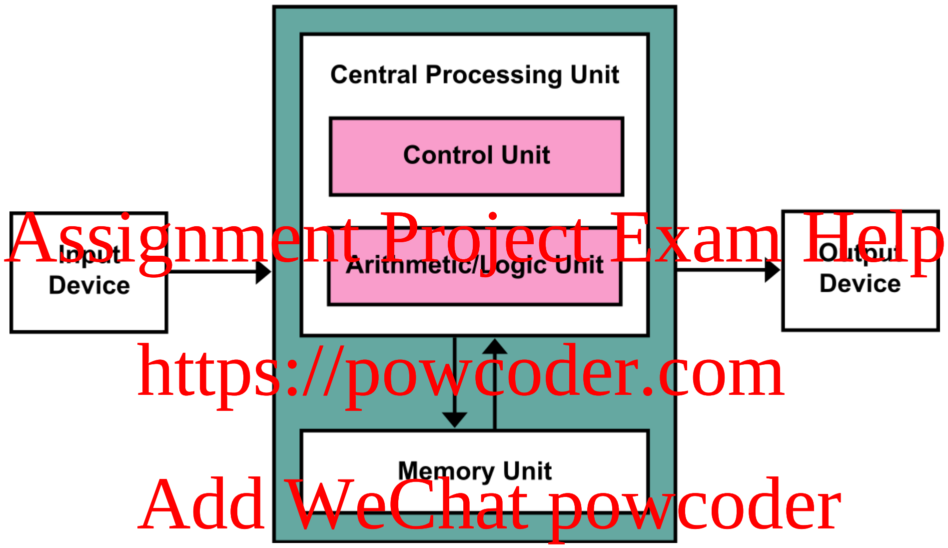
$$((x_1:type_1),(x_2:type_2),\ldots,(x_n:type_n))$$

Static and Dynamic Binding

- A binding is static if it first occurs before run time and remains unchanged throughout program execution.
- A binding is dynamic if it first occurs during execution or can change during execution of the program

TopHat Q16 - Q19

Variable – a low level imperative view

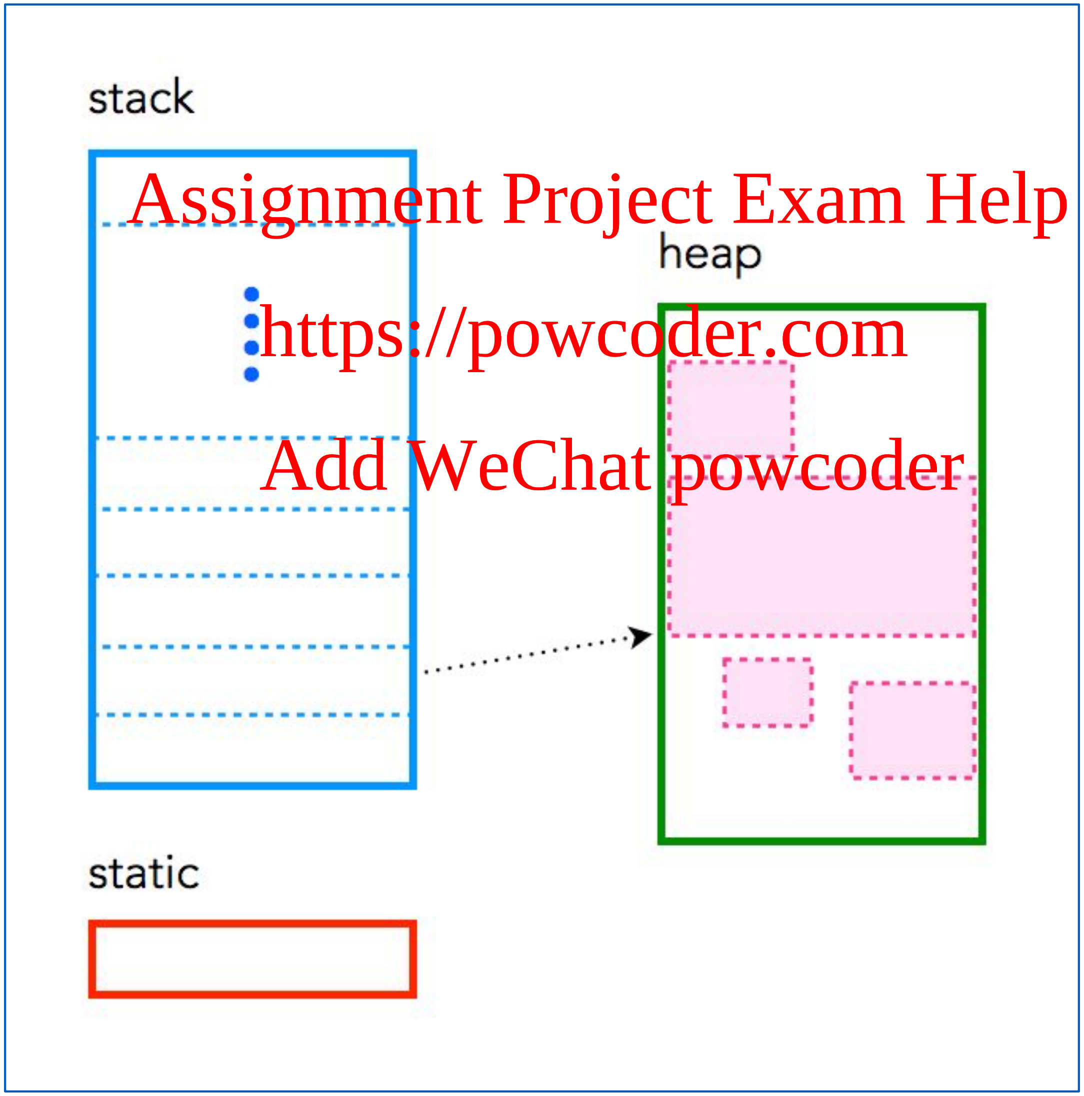


- Storage Bindings & Lifetime
  - Allocation - getting a cell from some pool of available cells
  - Deallocation - putting a cell back into the pool

Variables in imperative languages

- A variable is an abstraction of a memory cell
- Variables can be characterized by multiple attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope
- The lifetime of a variable is the time during which it is bound to a particular memory cell

Categories of Variables by Lifetimes – the C perspective



Categories of Variables by Lifetimes

Static -- bound to memory cells before execution begins and remains bound to the same memory cell throughout execution

- For example, we have static variables in functions in C and C++
- A top-level variable in OCaml can be seen as Static
- Static variables can be efficiently referenced through direct address,
- Impose a rigid programming discipline, not enough to support the need of general recursive functions.

Categories of Variables by Lifetimes

Stack-dynamic -- Bindings are created when their declaration statements are executed.

- Examples: local variables in C or Java subprograms (methods), variable binded by a let in Ocaml.

- We can bind them when we have explicit declarations or when a function is called (binding of actual and formal parameters).
- Most modern programming languages support stack-dynamic variables.

Categories of Variables by Lifetimes

Stack-dynamic -- Bindings are created when their declaration statements are executed.

- Stack-allocated variables provide a form of local storage

- Local storage is needed to support recursive functions.

- Supporting stack-dynamic variables require multiple allocation and deallocation.
- Working with stack-dynamic variables is more costly than static variables, and it doesn't support a global view.

Categories of Variables by Lifetimes

Explicit heap-dynamic -- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution.

- Usually are referenced: explicitly or implicitly through pointers or references.
- Examples: dynamic objects in C++ (via new and delete), all objects in Java, references in OCaml.
- Heap-dynamic variables may support an effective management of the storage.
- If the management is too low level it becomes unreliable.

Garbage Collection

- A Garbage Collector (GC) is an algorithm that automatically finds unused objects in the heap-allocated variables of an application and prepares them for reuse

- GC frees programmers from worrying about the exact lifetime of objects and ensures that the heap will not be corrupted by access to previously freed data

- ... but introduces often unpredictable pauses that may be costly and can increase the memory required.

TopHat Q20 - Q22

Type Binding

- Why is the role of a type?
  - Specifies what is the set of possible values,
  - Avoiding errors, providing type safety,
  - Specifies how much space I need for a variable

When does the binding take place?

- If static, the type may be specified by either an explicit or an implicit declaration,
- If dynamic, the type is implicitly declared.

Static Type Binding Explicit/Implicit Declaration

- An explicit declaration is a program statement used for declaring the types of variables
- An implicit declaration is a default mechanism for specifying types of variables through default conventions, rather than declaration statements

Type inference can help to determine types of variables thanks to information provided by the context:

- The initial value can set the type of a variable (e.g. C#)
- The use of the variable can set its type (e.g. OCaml).

Dynamic Type Binding

- Dynamic Type Binding is usually specified through an assignment statement, implicitly associating the variable with the type of the value it is assigned to:


```
x = [2, 4, 6, 8];
x = 17.3;
```


- This way of binding types to variables is used in dynamic typing disciplines (e.g. typing approach of JavaScript, PHP, etc.).
- These often provide more flexibility, but type error are more difficult to detect, and the checking can be more costly.