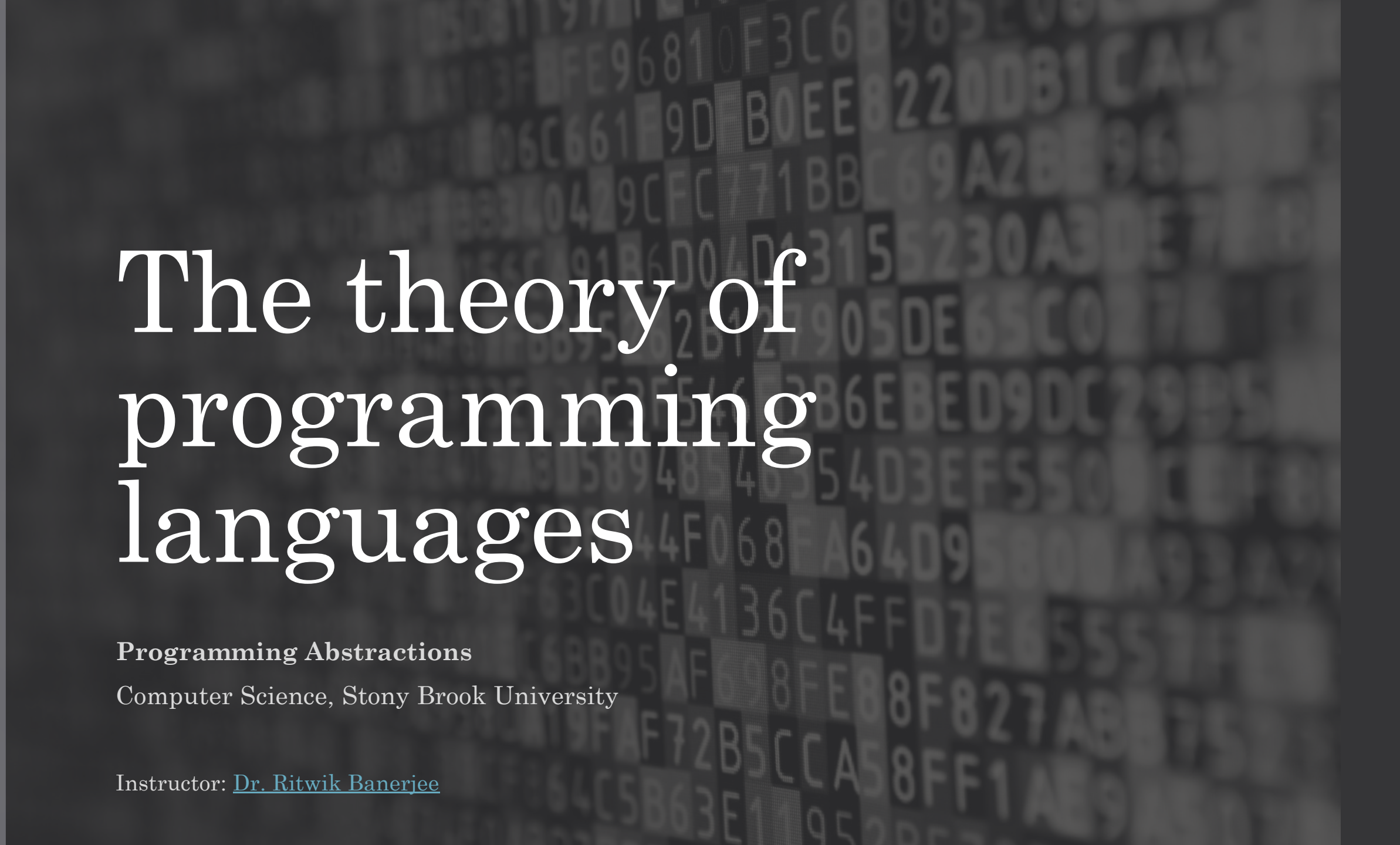


The theory of programming languages

Programming Abstractions

Computer Science, Stony Brook University

Instructor: Dr. Ritwik Banerjee

What is a programming language?

Let us look at some of the simplest computations, viz., arithmetic expressions:

For example, an expression like 1 + 4 * 4 / 2

We can think about this expression as a program in a programming language.

Here, the program is a single expression that evaluates to a single value (in this case, 9).

To describe this language, we need to understand two aspects of the language.

Its **syntax**, i.e., the *structure* of such expressions.

Its **semantics**, i.e., how such expressions are evaluated, or in other words, what is actual *meaning* of such an expression.

Syntax

- To describe the syntax of a language, we use a **grammar**.
  - A grammar is, simply put, a set of recursive rules that define what *forms* or *shapes* are permitted in the expressions of the language.
  - For example, a general grammar rule in English is subject-verb-object and not subject-object-verb,
  - "I listen to music", and not "I music listen to".
- A very simple grammar for our language of arithmetic expressions:

$$\text{Integer } n ::= \cdots -3 \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid 3 \mid \cdots$$
$$\text{BinOp } \oplus ::= + \mid - \mid * \mid /$$
$$\text{Expression } e ::= n \mid e_1 \oplus e_2$$

  - This grammar defines three types of syntactic constructs : integers, binary operators, and expressions.
  - Each type is further defined in terms of terminals (e.g., 0, 1, +) and non-terminals, which are variables like n, e_1, etc.

Syntax

- We can think of a grammar as something that defines how to decompose a program into its components.
  - $1 + 4 * 4 / 2 = e_1 \oplus e_2$ where $e_1 = 1$ and $e_2 = 4 * e_3$, and $e_3 = 4 / 2$.
  - Can you spot an obvious issue with the grammar we have so far?

    There is ambiguity in our grammar!

- We can also think of a grammar as something that *generates*, or inductively defines all valid expressions.
  - A language is modeled as a set of strings. The grammar of a language can generate *all* of them.

    Integer $n ::= \cdots -3 | -2 | -1 | 0 | 1 | 2 | 3 | \cdots$

    BinOp $\oplus ::= + | - | * | /$

    Expression $e ::= n \mid e_1 \oplus e_2$

  - This notation we have used to describe the grammar is taken from the Backus-Naur Form (BNF). This is an example of a **metalanguage**, or a language to describe languages.

Semantics

- Syntax tells us the *form* of valid programs, but what is the intrinsic *meaning* of such a program?
- For example, the *meaning* of 1 + 4 * 4 / 2 is what it evaluates to, which is 9. But how do we go from the initial expression to its final value?
  - Note that the final is also an expression – specifically, a terminal expression.
- An expression that can be further evaluated is called a **reducible expression**.
- We want to keep reducing until there are no more reducible expressions. The process of doing this is called the **semantics** of a language.
  - The semantics of a language are split into two categories: **small-step semantics**, which describe how the individual steps of a computation take place, and **big-step semantics**, which describe how the overall results are obtained.
  - We will usually be focusing on small-step semantics.
- In order to understand how semantics are formally defined, you will need to know elementary boolean logic and binary relations.

Small-step Semantics of arithmetic expressions

We will need logical rules to derive meaning from our programs (in this case, an arithmetic expression). We will use two relations that establish facts:

1. "e val" is a unary relation establishing that e is a value.
2. "$e_1 \mapsto e_2$" is a binary relation establishing that e_1 *steps-to* e_2 in the computational process.

Remember that we want to establish semantics so that there is a logical path from reducible expressions to irreducible values.

- From an expression like 1 + 4 * 4 / 2, we want to take small steps to eventually reach 9.

The way such a path is defined is in terms of **inference rules**. For example:

$$e_1 \mapsto e_2 \Longrightarrow e_1 \oplus e_3 \mapsto e_2 \oplus e_3$$

Note that the variables are not defined anywhere. There is an *implicit* universal quantification. That is, what we really mean is that

$$\forall\, e_1, \forall\, e_2, \qquad e_1 \mapsto e_2 \Longrightarrow e_1 \oplus e_3 \mapsto e_2 \oplus e_3$$

Small-step Semantics of arithmetic expressions

1. $\mapsto n$ val (unconditionally, any number n is a value)
2. $e_1 \mapsto e_2 \Longrightarrow e_1 \oplus e_3 \mapsto e_2 \oplus e_3$
3. $(e_1 \text{ val}) \wedge (e_2 \mapsto e_2') \Longrightarrow e_1 \oplus e_2 \mapsto e_1 \oplus e_2'$
4. $n_1 = n_2 \oplus n_3 \Longrightarrow n_2 \oplus n_3 \mapsto n_1$

- The above four rules define the small-step semantics of our arithmetic expression language.
- Using these, we can evaluate any valid expression, i.e., any expression that is *syntactically* correct in this language.
- Note that we are being a bit lazy here in defining the syntax and semantics. Strictly speaking, we are relying on the *external* language and theory (of arithmetic) to use symbols like '=' and the *metalanguage* of logic to use symbols like $\wedge, \forall$, etc.
- Since we will be studying multiple languages in this course, it is very important to pay attention to what concepts come from elsewhere, and what concepts are *internal* to the language we are studying.

Programs as proofs

- Early work on computability was not about understanding computers as we do today. Rather, it was about formalizing what an *effective procedure* should even mean!

- This resulted in the distinction between
  a) **constructive proofs**, where the proof explicitly shows how to obtain an object with some desired property, and
  b) **non-constructive proofs**, where simply inferring that such an object must exist was enough (e.g., proof by contradiction).

A procedure can be seen as a constructive proof of a proposition:

- Proposition: given certain input data (say, "1 + 4 * 4 / 2"), a certain output exists (correspondingly, "9").
- Constructive proof: run the procedure and provide the output.
- For example:
  - Euclid's algorithm is a constructive proof that given any two non-negative integers, there is a non-negative integer that is their greatest common divisor.
  - Our arithmetic language program is a constructive proof that the expression 1 + 4 * 4 / 2 is 9.

Early development of the imperative and functional paradigms

- Before the age of electronic computers, individual research on formalizing the notion of a program took place in the 1930s.
- Over time, it was shown that many of these different formalisms were equivalent in terms of computability, *i.e.*, anything that could be computed using one model could also be computed using another.

Two main models emerged:

1. The **Turing machine**, a pushdown automata with a storage tape, developed by Alan Turing. This model is the inspiration behind imperative programming.
2. **Lambda calculus**, developed by Alonzo Church, based on parameterized expressions.
   - Each expression was denoted by the Greek letter lambda (λ). Hence the name.
   - This is the basis of all functional programming.

Lambda Calculus

The arithmetic language we just saw is

- not very useful, because it can only do basic arithmetic
- boring, because it can only do basic arithmetic
- too complex for something that does so little
  - defines three different syntactic constructs
  - needs to use external language and theory to define evaluation

Let's look at the theory behind the functional programming paradigm, **lambda calculus**

- and see how it is a simple yet powerful tool to define the syntax and semantics of functional programming.

Lambda Calculus Examples

- It is a language with three constructs:
  1. functions,
  2. function applications, and
  3. variables.
- There is nothing else. No strings, for-loops, numbers, etc.
- A very simple example of a function is $\lambda x.x$
  - The letter λ denotes a "function".
  - The first variable (in this case, "x") is the argument to the function.
  - The dot is a separator, after which is the body of the function.
  - So this function is simply the identity function that returns its input.
- In lambda calculus, a function always has one input and one output.
  - *This is not such a hard restriction. How many things can a Java function return, for instance?*

Lambda Calculus Examples

Functions can be nested.

$$\lambda x.\, \lambda y.\, (xy)$$

This is a function that

1. takes an input x,
2. then returns a function which, in turn, takes an input y, and
3. then calls the function x with y as its argument.

Consider, for example, the addition of two numbers given by the function **`add x y`**:

1. The function **`add`** is the 'lambda' here. It takes the first argument **`x`** and returns a function that adds **`x`** to *its* argument.
2. The variable **`y`** is the argument to the **`add x`** function. It's body adds **`y`** to **`x`**. The final result, of course, is **`y + x`**.

$$
\begin{aligned}
&(\lambda x.\, \lambda y.\, xy)\ (\lambda z.\, z)\ w \\
\mapsto\ &(\lambda y.\, (\lambda z.\, z)\ y)\ w \\
\mapsto\ &(\lambda z.\, z)\ w \\
\mapsto\ &w
\end{aligned}
$$

- Calling a function is equivalent to replacing every instance of the argument variable with the argument expression.
  - E.g., we replaced x with $\lambda z.\, z$

Syntax conventions of lambda calculus:

1. Function application is left-associative: $x\ y\ z$ is equivalent to $(xy)z$ and **not** $x(yz)$.
2. Function application has higher precedence than function definition: $\lambda x.\, \lambda y.\, xy$ is equivalent to $\lambda x.\, \big(\lambda y.\, (xy)\big)$.

Lambda Calculus Examples

Syntax of lambda calculus

Expression $e ::= x$ variable

| $\lambda x. e$ function definition

| $e_1 e_2$ function application

These are the three constructs of the syntax.

A few things to note:

- Strictly speaking, we should also define a grammar rule for "variable", but for now we are assuming that a variable follows the same syntactic conventions of normal mathematics.
- In this grammar, x is a *meta-variable*.
  - That is, it is a placeholder for other variables.

Semantics of lambda calculus

1. $\mapsto \lambda x.e$ val
   functions are values
2. $e_1 \mapsto e_1' \Longrightarrow e_1\ e_2 \mapsto e_1'\ e_2$
   reduce the left-expression until it becomes a function
3. $(\lambda x.e_1)\ e_2 \mapsto [x \to e_2]e_1$
   substitute all occurrences of x in e_1 with e_2

- The substitution in the third rule above is a new construct.
- To understand how it works, we need to carefully understand the notion of **scope**.

Names, Scopes, & Binding

- Lexical scope
- Bound and free variables
- Shadowing or Masking

Names

- A name is a mnemonic (i.e., assists memory) string used to represent something.
  - Usually, they are alphanumeric tokens.
  - They allow programmers to refer to variables, constants, operations, types, etc.
- Without names, we would have to use low-level concepts like addresses for everything!
- Names are absolutely crucial for abstraction:
  - Programmer can associate a name with a potentially complex fragment of a program. This name usually reflects the 'purpose' or 'meaning' of that fragment of code.
  - It hides low-level details and reduces the conceptual complexity for a programmer.

Names

Subroutines have names. They offer *control abstractions* – the programmer can put a complex code fragment inside a subroutine with a meaningful name.

- The name should typically convey to the programmer what the subroutine does.

Classes are *data abstractions* – the programmer can put the details of data representation inside a class, and relatively simple operations to deal with the data.

- Typically, these operations are for access and/or modification (e.g., getters and setters in Java).

Names & Bindings

There are three fundamental questions one might ask about names.

1. How does the name get associated with the thing it represents?
2. Can it change through the course of a program?
3. If it can change, then what are the limitations and rules, if any?

The association in the first question is known as **binding**.

Binding & Binding Times

The time at which the binding is created is known as **binding time**. This can be created in different times:

1. **Language design time:** the design of specific program constructs (syntax), primitive types, and meaning (semantics), etc. are decided when the language is designed.
2. **Language implementation time:** many issues are left to the implementer. These may include numeric precision (i.e., the number of bits), run time memory sizes, etc.
3. **Program writing time:** e.g., the choice of algorithms, data structures, names.
4. **Compile time:** compilers choose (i) how to map high-level constructs to machine code, and (ii) the memory layout for things used in the program.
5. **Link time:** the time at which multiple object codes (machine code files) and libraries are combined into one executable. For complex programs, there may be names in one module that refer to things in another module. The binding, then, cannot be done until link time.
6. **Load time:** the time at which the OS loads the executable into memory so that it can run.
   - OS usually distinguishes between physical/virtual addresses.
   - Virtual addresses are chosen at link time, but physical addresses can change at run time: the processor's memory management hardware translates the virtual addresses into physical addresses during individual instructions at run time.
7. **Run time:** many language-specific decisions may be taken during run time; e.g., the binding of values to variables may occur at run time.

LANGUAGE	FEATURE	BINDING TIME
C	syntax: `if (a>0) b:=a;`	language design
	reserved keywords: **`main`**	
	primitive types: **`float, struct`**	
	calls to static library routines: **`printf`**	link
	specific type of a variable	compile
Java	reserved keywords: **`class`**	language design
Any	internal representation of literals (e.g. **`3.14`** or **`"foo"`**)	language implementation
	non-static allocation of space for variables	run time

Binding & Binding Times

Binding & Binding Times

Early binding time leads to greater efficiency.

- Compilers try to fix decisions that can be taken at compile time to avoid generating code that makes a decision at run time.
- Checking of syntax and static semantics is performed only once at compile time to avoid any run-time overhead.

Later binding time leads to greater flexibility.

- Interpreters allow programs to be modified at run time
- Some languages like Smalltalk and Java allow variable names to refer to objects of multiple types at run time (**runtime polymorphism**).

Things bound *before* runtime are **statically bound**.

- Hence the concept of **static binding**.

Things bound *after* runtime are **dynamically bound**.

- Hence the concept of **dynamic binding**.

Object* Lifetime

An object's lifetime and a binding's lifetime may be different.

An object may retain its value, and may potentially be accessible even when a given name can no longer be used.

For example, when a variable is passed by reference to a subroutine, the lifetime of the binding between the parameter name and the variable that was passed is shorter than the lifetime of the variable itself!

On the other hand, if the binding's lifetime is longer than that of the object itself (i.e., the name exists, but the object doesn't), it's typically a bug in the program! This is called a **dangling reference**.

* By 'object' here, we don't mean how the term is used in object-oriented programming, but just a general 'thing'.

Scope

- The textual region of a program in which a binding is active is its **scope**.
- In most languages, the scope of a binding is determined at compile-time (i.e., *statically* determined).
  - In C, we introduce a new scope when we enter a subroutine. Local object bindings are created.
  - On subroutine exit, the bindings for local variables are destroyed.
- These decisions are made entirely at compile time. Languages where this happens are called **statically scoped** languages. It is also called **lexical scoping**.
- Some languages are dynamically scoped, where the bindings depend on the flow of execution at run time (e.g., early versions of Lisp).
- Often, instead of talking about the scope of a binding, we simply say "scope" (as a noun) by itself without specifying any bindings.
  - This usage of scope is a program region of maximal size in which no bindings are changed or destroyed.
- Typically, a scope is the body of a class, subroutine, etc. These are what we call a **block**, usually delimited with syntax. For example, in C, a block is designated by **`{ ... }`**

Scope and Shadowing

- A variable is said to be **shadowed** when a variable declared within a certain scope has the same name as a variable declared in an outer scope.
  - The outer variable is said to be shadowed by the inner variable.
  - The inner name is said to **mask** the outer name.

```
public class Shadow {
  private int myIntVar = 0;

  private void shadowTheVar() {
    int myIntVar = 5;                    // has the same name as above object instance
                                         // field, so it shadows the above field inside
                                         // this method.
    System.out.println(myIntVar);        // prints 5
    System.out.println(this.myIntVar); // prints 0 ('this' moves the namespace to the
                                         // outer scope)
  }
}
```

Scope and Binding in λ-Calculus

What does the expression $(\lambda x.\lambda x.x)\ y\ z$ evaluate to?

- When a variable is used in an expression under a function with a variable of the *same* name, then the variable is **bound**.
  - In $\lambda x.x$, the inner variable x is bound to the function argument x.
- If a variable is not bound, it is called **free**.
  - E.g., in $\lambda x.y$, the inner variable y is free.
- The variable binding is to the closest argument in its enclosing functions. So, the innermost x **shadows** the x in the next outer scope.
  - In other words, all names are **local** to their definitions.
  - This is the abstract syntactic and semantic equivalent of the shadowing we saw in the Java example earlier.

What are the free and bound variables in $(\lambda x.x)\ (\lambda y.yx)$?

Scope and Binding in λ-Calculus

Formally, a variable x is **free** in an expression under one of the following three conditions:

1. x is free in x.
2. x is free in $\lambda y.e$ if the name $x \neq y$ and x is free in the expression e.
3. x is free in the expression $e_1\ e_2$ if x is free in the expression e_1 or if it is free in the expression e_2.

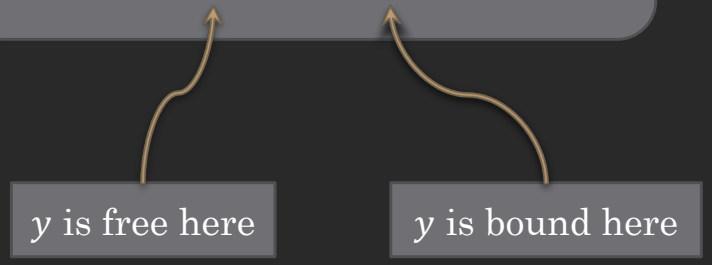
Formally, a variable x is **bound** in an expression under one of the following two conditions:

1. x is bound in $\lambda y.e$ if the name $x = y$ or if the name x is bound in the expression e.
2. x is bound in the expression $e_1\ e_2$ if x is bound in the expression e_1 or if it is bound in the expression e_2.

Scope and Binding in λ-Calculus

A variable can be both free and bound in the same expression.

- E.g., $(\lambda x.xy)(\lambda y.y)$.



Formally, a variable x is **free** in an expression under one of the following three conditions:

1. x is free in x.
2. x is free in $\lambda y.e$ if the name $x \neq y$ and x is free in the expression e.
3. x is free in the expression $e_1\ e_2$ if x is free in the expression e_1 or if it is free in the expression e_2.

Formally, a variable x is **bound** in an expression under one of the following two conditions:

1. x is bound in $\lambda y.e$ if the name $x = y$ or if the name x is bound in the expression e.
2. x is bound in the expression $e_1\ e_2$ if x is bound in the expression e_1 or if it is bound in the expression e_2.

Substitutions

- In standard λ-calculus, we do not name functions. Instead, we write the whole function definition to apply and then evaluate it.
- Consider the identity function applied to itself: $(\lambda x.\, x)(\lambda x.\, x)$.

  Recall that the x in the body of the first expression is independent of the x in the body of the second one.

  Therefore, we can rewrite this function application as

  $$(\lambda x.\, x)(\lambda x.\, x) \equiv (\lambda x.\, x)(\lambda z.\, z)$$

  The second function is the input argument to the first one, so we can substitute the body of the first function $\lambda x.\, x$ with that argument to just obtain:

  $$[x \mapsto \lambda z.\, z]x \equiv \lambda z.\, z$$

  which is the identity function again.

  That is, the identity function applied to itself is just the identity function.

Be careful not to mix up the free occurrences with the bound occurrences!

Consider the expression $(\lambda x.(\lambda y.xy))y$

- The inner function has a bound y, but the rightmost y is free. So, the following is an incorrect substitution:

$$\lambda y.yy$$

- To avoid such confusion, rename the bound variable to get the correct result:

$$(\lambda x.(\lambda t.xt))y \equiv \lambda t.yt$$

- If the function $\lambda x.e_1$ is applied to e_2, we substitute all *free* occurrences of x in e_1 with e_2.
- If the substitution would bring a free variable of e_2 in an expression where the same name is already bound, we rename that variable before performing the substitution.

Errors and ‘stuck’ states

- In our arithmetic language, every expression was reducible to a value *or* an error. But generally, we can get stuck in an undefined error state.
  - For example, the single free variable x. Is this reducible? No. Is this a value? No. So, no rule of semantics can be applied. We are “stuck”! Such a state is different from an explicitly defined error state.
- Can we identify, ahead of evaluations, whether an expression will lead to such a stuck state?
  - No. The proof of this “no” is called the Halting problem, which is beyond the scope of this course.
  - But we can introduce more structure to the basic lambda calculus to avoid getting stuck. This additional structure is the introduction of **types**.

Why bother learning about λ-calculus?

- It pushes the boundary of simplicity while still being an extremely powerful framework to define 'computation'.
  - Much simpler than equivalent definitions like the Turing machine.
- Based on a simple but complete understanding of functions, which are at the core of all programming.
  - One input, one output, and just substitute the input argument in the function body.
  - Other functions – e.g., one that has multiple arguments – can be easily incorporated by introducing the notion of 'lists' or 'tuples'.
  - This is an example of decoupling of the function definition and the data type: functions can be used with or without tuples, and tuples can be used with or without functions.
- The theory of λ-calculus is the basis of all kinds of program analysis.
  - Because of its simplicity, it is much easier to answer questions like, "will this program will ever reach a segmentation fault" in λ-calculus than, say, in C++.