Lecture 11 – Metaprogramming

SWS121: Secure Programming

Jihyeok Park

2024 Spring

Recall

- **Contextual Abstractions**
  - Context Parameters
  - Implicit Conversions
  - Extension Methods
  - Given Imports
  - Type Classes

Metaprogramming

Metaprogramming is a programming technique that computes code at **compile-time** to generate code that will be executed at **runtime**.

Metaprogramming

Metaprogramming is a programming technique that computes code at **compile-time** to generate code that will be executed at **runtime**.

It is a powerful technique that can be used to:

- Optimize code
- Generate boilerplate code
- Implement domain-specific languages

Metaprogramming

Metaprogramming is a programming technique that computes code at **compile-time** to generate code that will be executed at **runtime**.

It is a powerful technique that can be used to:

- Optimize code
- Generate boilerplate code
- Implement domain-specific languages

Scala provides several metaprogramming features:

- Inline
- Macros
- Reflection

Metaprogramming

PLRG

Metaprogramming is a programming technique that computes code at **compile-time** to generate code that will be executed at **runtime**.

It is a powerful technique that can be used to:

- Optimize code
- Generate boilerplate code
- Implement domain-specific languages

Scala provides several metaprogramming features:

- Inline
- Macros
- Reflection

You can see what happens at **compile-time** by using the `-Xprint:extmethods` flag for the Scala compiler.

```
$ scalac -Xprint:extmethods MyFile.scala
```

Contents

Contents

Inline

Inlining is a common **compile-time metaprogramming** technique, typically used to achieve performance optimizations.

Inline

Inlining is a common **compile-time metaprogramming** technique, typically used to achieve performance optimizations.

It is a good entry point to when learning about metaprogramming.

Inline

Inlining is a common **compile-time metaprogramming** technique, typically used to achieve performance optimizations.

It is a good entry point to when learning about metaprogramming.

It introduces operations that are guaranteed to **evaluate** at **compile-time**.

Inline

Inlining is a common **compile-time metaprogramming** technique, typically used to achieve performance optimizations.

It is a good entry point to when learning about metaprogramming.

It introduces operations that are guaranteed to **evaluate** at **compile-time**.

Let's learn the following inlining features:

- Inline Constants
- Inline Methods
- Inline Parameters
- Inline Matches
- Transparent Inline Methods

Inline Constants

Inline constants are the simplest form of inlining.

Inline Constants

Inline constants are the simplest form of inlining.

We can define **inline constants** using the `inline` (soft) keyword:

```
inline val Pi = 3.14
```

Inline Constants

Inline constants are the simplest form of inlining.

We can define **inline constants** using the `inline` (soft) keyword:

```
inline val Pi = 3.14
```

Then, all the occurrences of `Pi` will be replaced by `3.14` at **compile-time**

```
val radius = 4.0

val area = Pi * radius * radius
// compiled to `val area = 3.14 * radius * radius`
```

Inline Constants

Inline constants are the simplest form of inlining.

We can define **inline constants** using the `inline` (soft) keyword:

```
inline val Pi = 3.14
```

Then, all the occurrences of `Pi` will be replaced by `3.14` at **compile-time**

```
val radius = 4.0

val area = Pi * radius * radius
// compiled to `val area = 3.14 * radius * radius`
```

It also computes the **expression** containing `Pi` at **compile-time**.

```
val circ1 = 2 * Pi * radius
// compiled to `val circ1 = 6.28 * radius`
```

It is often called **constant folding**.

Inline Constants

However, it does **not** support **commutativity** of the multiplication:

```
val circ2 = Pi * radius * 2
// compiled to `val circ2 = 3.14 * radius * 2`
```

Inline Constants

PLRG

However, it does **not** support **commutativity** of the multiplication:

```
val circ2 = Pi * radius * 2
// compiled to `val circ2 = 3.14 * radius * 2`
```

If the `radius` is also an inline constant, all the expressions containing `Pi` and `radius` will be computed at **compile-time**:

```
inline val Pi = 3.14
inline val radius = 4.0

val area = Pi * radius * radius
// compiled to `val area = 50.24`

val circ1 = 2 * Pi * radius
// compiled to `val circ1 = 25.12`

val circ2 = Pi * radius * 2
// compiled to `val circ1 = 25.12`
```

Inline Constants

Only **literal constant types** are allowed for inline constants.

Inline Constants

Only **literal constant types** are allowed for inline constants.

For example, the `Double` type is not a literal constant type because it represents all the double-floating values rather than a single value:

```
inline val Pi: Double = 3.14
// error: inline value must have a literal constant type
```

Inline Constants

Only **literal constant types** are allowed for inline constants.

For example, the `Double` type is not a literal constant type because it represents all the double-floating values rather than a single value:

```
inline val Pi: Double = 3.14
// error: inline value must have a literal constant type
```

In the original code snippet, the `Pi` is defined as literal constant type `3.14` whose **only possible value** is `3.14`:

```
inline val Pi: 3.14 = 3.14
```

Inline Constants

PLRG

Only **literal constant types** are allowed for inline constants.

For example, the `Double` type is not a literal constant type because it represents all the double-floating values rather than a single value:

```
inline val Pi: Double = 3.14
// error: inline value must have a literal constant type
```

In the original code snippet, the `Pi` is defined as literal constant type `3.14` whose **only possible value** is `3.14`:

```
inline val Pi: 3.14 = 3.14
```

We can assign a literal constant type to an expression containing `Pi` will be computed to a single value at **compile-time**.

```
inline val radius: 4.0 = 4.0
val area: 50.24 = Pi * radius * radius
val circ1: 25.12 = 2 * Pi * radius
val circ2: 25.12 = Pi * radius * 2
```

Inline Methods

We can also define **inline methods** using the `inline` keyword:

```
inline def inc(n: Int): Int = n + 1
```

Inline Methods

We can also define **inline methods** using the `inline` keyword:

```
inline def inc(n: Int): Int = n + 1
```

Then, all the occurrences of `inc` will be replaced by the expression `n + 1` for a given argument `n` at **compile-time**:

```
val x = inc(42)
// compiled to `val x = 43`
```

Inline Methods

PLRG

We can also define **inline methods** using the `inline` keyword:

```
inline def inc(n: Int): Int = n + 1
```

Then, all the occurrences of `inc` will be replaced by the expression `n + 1` for a given argument `n` at **compile-time**:

```
val x = inc(42)
// compiled to `val x = 43`
```

However, it does not support **multiple applications** of the inline method:

```
val y = inc(inc(42))
// compiled to `val y = { val n = 43;  n + 1 }`
```

Inline Methods

We can also define **inline methods** using the `inline` keyword:

```
inline def inc(n: Int): Int = n + 1
```

Then, all the occurrences of `inc` will be replaced by the expression `n + 1` for a given argument `n` at **compile-time**:

```
val x = inc(42)
// compiled to `val x = 43`
```

However, it does not support **multiple applications** of the inline method:

```
val y = inc(inc(42))
// compiled to `val y = { val n = 43;  n + 1 }`
```

It seems possible to apply **inline methods** multiple times if `n` is also **inline**.

Inline Methods

We can also define **inline methods** using the `inline` keyword:

```
inline def inc(n: Int): Int = n + 1
```

Then, all the occurrences of `inc` will be replaced by the expression `n + 1` for a given argument `n` at **compile-time**:

```
val x = inc(42)
// compiled to `val x = 43`
```

However, it does not support **multiple applications** of the inline method:

```
val y = inc(inc(42))
// compiled to `val y = { val n = 43;  n + 1 }`
```

It seems possible to apply **inline methods** multiple times if `n` is also **inline**.

To do so, we need to utilize **inline parameters**.

Inline Parameters

In inline methods, we can define **inline parameters** using `inline` keyword:

```
inline def inc(inline n: Int): Int = n + 1
```

Inline Parameters

PLRG

In inline methods, we can define **inline parameters** using `inline` keyword:

```
inline def inc(inline n: Int): Int = n + 1
```

Then, the given argument `n` will be replaced by the expression at **compile-time** for all the occurrences of `inc`:

```
val x = inc(42)
// compiled to `val x = 43`

val y = inc(inc(42))
// compiled to `val y = 44`

val z = inc(inc(inc(inc(inc(42)))))
// compiled to `val z = 47`
```

Inline Parameters

Inline methods handle each kind of parameters differently:

- **By-value parameters.** The compiler generates **val** bindings.
- **By-name parameters.** The compiler generates **def** bindings.
- **Inline parameters.** The compiler inlines the expression.

Inline Parameters

Inline methods handle each kind of parameters differently:

- **By-value parameters.** The compiler generates **val** bindings.
- **By-name parameters.** The compiler generates **def** bindings.
- **Inline parameters.** The compiler inlines the expression.

```
inline def add(a: Int, b: => Int, inline c: Int): Int =
  a + a + b + b + c + c

def x: Int = 42

val y: Int = add(x + 1, x * 2, x / 3)
```

Inline Parameters

Inline methods handle each kind of parameters differently:

- **By-value parameters.** The compiler generates **val** bindings.
- **By-name parameters.** The compiler generates **def** bindings.
- **Inline parameters.** The compiler inlines the expression.

```
inline def add(a: Int, b: => Int, inline c: Int): Int =
  a + a + b + b + c + c

def x: Int = 42

val y: Int = add(x + 1, x * 2, x / 3)
```

Then, the definition of y will be compiled to:

```
val y: Int =
  val a: Int = x + 1                  // for by-value parameter `a`
  def b: Int = x * 2                  // for by-name parameter `b`
  a + a + b + b + (x / 3) + (x / 3) // for inline parameter `c`
```

Inline Parameters

Since **inline parameters** are inlined for its all occurrences, it might evaluated **multiple times** similar to by-name parameters.

```
def foo(name: String): Int = { println(s"Parameter: $name"); 42 }
val z: Int = add(foo("a"), foo("b"), foo("c"))
```

Inline Parameters

Since **inline parameters** are inlined for its all occurrences, it might evaluated **multiple times** similar to by-name parameters.

```
def foo(name: String): Int = { println(s"Parameter: $name"); 42 }
val z: Int = add(foo("a"), foo("b"), foo("c"))
```

Then, the definition of z will be compiled to:

```
val z: Int =
  val a: Int = foo("a")
  def b: Int = foo("b")
  a + a + b + b + foo("c") + foo("c")
// Parameter: a
// Parameter: b
// Parameter: b
// Parameter: c
// Parameter: c
```

Inline Parameters

PLRG

Since **inline parameters** are inlined for its all occurrences, it might evaluated **multiple times** similar to by-name parameters.

```
def foo(name: String): Int = { println(s"Parameter: $name"); 42 }
val z: Int = add(foo("a"), foo("b"), foo("c"))
```

Then, the definition of z will be compiled to:

```
val z: Int =
  val a: Int = foo("a")
  def b: Int = foo("b")
  a + a + b + b + foo("c") + foo("c")
// Parameter: a
// Parameter: b
// Parameter: b
// Parameter: c
// Parameter: c
```

While **by-name parameters** evaluated by calling generated functions (e.g., `def b = foo("b")`), **inline parameters** have **no function call overhead** since they are evaluated by inlining the expression (e.g., `foo("c")`).

Inline Parameters – Example

PLRG

We can define **inline method** `power` that computes the **power** of a given integer `x` with an integer `n` as an exponent:

```
inline def power(inline x: Int, inline n: Int): Int =
  if (n == 0) 1
  else if (n % 2 == 1) x * power(x, n - 1)
  else power(x * x, n / 2)
```

Inline Parameters – Example

We can define **inline method** `power` that computes the **power** of a given integer `x` with an integer `n` as an exponent:

```
inline def power(inline x: Int, inline n: Int): Int =
  if (n == 0) 1
  else if (n % 2 == 1) x * power(x, n - 1)
  else power(x * x, n / 2)
```

Then, all the occurrences of `power` will be replaced by the expression for a given arguments `x` and `n` at **compile-time**:

```
val x = power(2, 10)
// compiled to `val x = 1024`

def k: Int = 2
val y = power(k, 10)
// compiled to `val y = k * k * k * k * k * k * k * k * k * k * 1`
```

Inline Parameters – Example

PLRG

We can define **inline method** `power` that computes the **power** of a given integer `x` with an integer `n` as an exponent:

```
inline def power(inline x: Int, inline n: Int): Int =
  if (n == 0) 1
  else if (n % 2 == 1) x * power(x, n - 1)
  else power(x * x, n / 2)
```

Then, all the occurrences of `power` will be replaced by the expression for a given arguments `x` and `n` at **compile-time**:

```
val x = power(2, 10)
// compiled to `val x = 1024`

def k: Int = 2
val y = power(k, 10)
// compiled to `val y = k * k * k * k * k * k * k * k * k * k * 1`
```

It requires **ten multiplications** for `y`.

Inline Parameters – Example

We can optimize it by defining `x` as **by-value** parameter:

```
inline def power(x: Int, inline n: Int): Int =
  if (n == 0) 1
  else if (n % 2 == 1) x * power(x, n - 1)
  else power(x * x, n / 2)
```

Inline Parameters – Example

PLRG

We can optimize it by defining `x` as **by-value** parameter:

```
inline def power(x: Int, inline n: Int): Int =
  if (n == 0) 1
  else if (n % 2 == 1) x * power(x, n - 1)
  else power(x * x, n / 2)
```

```
val x = power(2, 10)
// compiled to `val x = 1024`

def k: Int = 2
val y = power(k, 10)
// compiled to
// val y =
//   val x0: Int = k * k       // k^2
//   val x1: Int = x0 * x0     // k^4
//   val x2: Int = x1 * x1     // k^8
//   x0 * x2 * 1               // k^10
```

This optimized version requires only **five multiplications** for `y`.

Inline Matches

We can redefine the **inline method** `power` using **inline matches** using the `inline` keyword:

```
inline def power(x: Int, inline n: Int): Int = inline n match
  case 0 => 1
  case _ if n % 2 == 1 => x * power(x, n - 1)
  case _ => power(x * x, n / 2)
```

Inline Matches

We can redefine the **inline method** `power` using **inline matches** using the `inline` keyword:

```
inline def power(x: Int, inline n: Int): Int = inline n match
  case 0 => 1
  case _ if n % 2 == 1 => x * power(x, n - 1)
  case _ => power(x * x, n / 2)
```

```
val x = power(2, 10)
// compiled to `val x = 1024`

def k: Int = 2
val y = power(k, 10)
// compiled to `val y = ...`
```

Inline Matches

PLRG

We can redefine the **inline method** `power` using **inline matches** using the `inline` keyword:

```
inline def power(x: Int, inline n: Int): Int = inline n match
  case 0 => 1
  case _ if n % 2 == 1 => x * power(x, n - 1)
  case _ => power(x * x, n / 2)
```

```
val x = power(2, 10)
// compiled to `val x = 1024`

def k: Int = 2
val y = power(k, 10)
// compiled to `val y = ...`
```

Note that if we use the normal `match` expression, it cannot be compiled because of **exceeding the maximum number of inlines**.

Transparent Inline Methods

Consider the following inline method `default` that returns the default value of a given type name in `String`:

```
inline def default(name: String): Any = inline name match
  case "Int"     => 0
  case "String"  => ""
  case "Boolean" => false
  case _         => ???
```

Transparent Inline Methods

PLRG

Consider the following inline method `default` that returns the default value of a given type name in `String`:

```
inline def default(name: String): Any = inline name match
  case "Int"     => 0
  case "String"  => ""
  case "Boolean" => false
  case _         => ???
```

Then, we want to get the result value as its corresponding type:

```
val x: Int     = default("Int")     // `Int` expected but `Any`
// compiled to `val x: Int = (0: Any)`

val y: String  = default("String")  // `String` expected but `Any`
// compiled to `val y: String = ("": Any)`

val z: Boolean = default("Boolean") // `Boolean` expected but `Any`
// compiled to `val z: Boolean = (false: Any)`
```

Transparent Inline Methods

Transparent inline methods allow for an inline piece of code to **refine** the **return type** based on the precise type of the inlined expression.

```
transparent inline def default(name: String): Any = inline name match
  case "Int"     => 0
  case "String"  => ""
  case "Boolean" => false
  case _         => ???
```

Transparent Inline Methods

Transparent inline methods allow for an inline piece of code to **refine** the **return type** based on the precise type of the inlined expression.

```
transparent inline def default(name: String): Any = inline name match
  case "Int"     => 0
  case "String"  => ""
  case "Boolean" => false
  case _         => ???
```

Now, the return type of the `default` method will be refined based on the type of the inlined expression:

```
val x: Int     = default("Int")
// compiled to `val x: Int = 0`

val y: String  = default("String")
// compiled to `val y: String = ""`

val z: Boolean = default("Boolean")
// compiled to `val z: Boolean = false`
```

Transparent Inline Methods – Example

PLRG

Let's define an **inline method** `evenOrOdd` that returns a string `"even"` if a given integer `n` is even, otherwise `"odd"`:

```
transparent inline def evenOrOdd(
  inline number: Int
): String = inline (number % 2) match
  case 0 => "even"
  case _ => "odd"
```

Transparent Inline Methods – Example

Let's define an **inline method** `evenOrOdd` that returns a string `"even"` if a given integer `n` is even, otherwise `"odd"`:

```
transparent inline def evenOrOdd(
  inline number: Int
): String = inline (number % 2) match
  case 0 => "even"
  case _ => "odd"
```

It is a **transparent inline method** that refines the return type based on the type of the inlined expression.

Transparent Inline Methods – Example

PLRG

Let's define an **inline method** `evenOrOdd` that returns a string `"even"` if a given integer `n` is even, otherwise `"odd"`:

```
transparent inline def evenOrOdd(
  inline number: Int
): String = inline (number % 2) match
  case 0 => "even"
  case _ => "odd"
```

It is a **transparent inline method** that refines the return type based on the type of the inlined expression.

We can even assign the **literal constant type** to the return type:

```
val x: "even" = evenOrOdd(42)
// compiled to `val x: "even" = "even"`
val y: "odd" = evenOrOdd(37)
// compiled to `val y: "odd" = "odd"`
```

Contents

Macros

With a **macro**, we can treat **programs** as **values**, which allows us to analyze and generate them at compile time.

Macros

With a **macro**, we can treat **programs** as **values**, which allows us to analyze and generate them at compile time.

A Scala expression with type `T` is represented by an instance of the type `scala.quoted.Expr[T]`.

Macros

With a **macro**, we can treat **programs** as **values**, which allows us to analyze and generate them at compile time.

A Scala expression with type `T` is represented by an instance of the type `scala.quoted.Expr[T]`.

The following macro implementation prints the expression of the provided argument at compile-time in the standard output of the compiler process:

```
import scala.quoted.*
def inspectCode(x: Expr[Any])(using Quotes): Expr[Any] =
  println(s"Expr: ${x.show}")
  x
```

Macros

PLRG

With a **macro**, we can treat **programs** as **values**, which allows us to analyze and generate them at compile time.

A Scala expression with type `T` is represented by an instance of the type `scala.quoted.Expr[T]`.

The following macro implementation prints the expression of the provided argument at compile-time in the standard output of the compiler process:

```
import scala.quoted.*
def inspectCode(x: Expr[Any])(using Quotes): Expr[Any] =
  println(s"Expr: ${x.show}")
  x
```

After printing the argument expression, we return the original argument as a Scala expression of type `Expr[Any]`.

Macros

With a **macro**, we can treat **programs** as **values**, which allows us to analyze and generate them at compile time.

A Scala expression with type `T` is represented by an instance of the type `scala.quoted.Expr[T]`.

The following macro implementation prints the expression of the provided argument at compile-time in the standard output of the compiler process:

```
import scala.quoted.*
def inspectCode(x: Expr[Any])(using Quotes): Expr[Any] =
  println(s"Expr: ${x.show}")
  x
```

After printing the argument expression, we return the original argument as a Scala expression of type `Expr[Any]`.

The `show` method of the `Expr` class returns the string representation of the expression.

Macros

Inline methods provide the entry point for macro definitions:

```
inline def inspect(inline x: Any): Any = ${ inspectCode('x) }
```

Macros

Inline methods provide the entry point for macro definitions:

```
inline def inspect(inline x: Any): Any = ${ inspectCode('x) }
```

A **quoted code block** `'{ ... }` casts a Scala expression of type `T` to an instance of `scala.quoted.Expr[T]`.

Macros

Inline methods provide the entry point for macro definitions:

```
inline def inspect(inline x: Any): Any = ${ inspectCode('x) }
```

A **quoted code block** `'{ ... }` casts a Scala expression of type `T` to an instance of `scala.quoted.Expr[T]`.

`${ ... }` is a **splice** that evaluates the code within the splice and places the result in the generated code.

Macros

Inline methods provide the entry point for macro definitions:

```
inline def inspect(inline x: Any): Any = ${ inspectCode('x) }
```

A **quoted code block** `'{ ... }` casts a Scala expression of type `T` to an instance of `scala.quoted.Expr[T]`.

`${ ... }` is a **splice** that evaluates the code within the splice and places the result in the generated code.

The macro `inspect` takes an argument `x` and returns the result of the `inspectCode` macro applied to the argument `x`.

Macros

Inline methods provide the entry point for macro definitions:

```
inline def inspect(inline x: Any): Any = ${ inspectCode('x) }
```

A **quoted code block** `'{ ... }` casts a Scala expression of type `T` to an instance of `scala.quoted.Expr[T]`.

`${ ... }` is a **splice** that evaluates the code within the splice and places the result in the generated code.

The macro `inspect` takes an argument `x` and returns the result of the `inspectCode` macro applied to the argument `x`.

For example, the following function call prints the string representation of the expression `Expr: "abc".repeat(3)` and returns the result `"abcabcabc"`.

```
inspect("abc".repeat(3))
// Expr: "abc".repeat(3)
// "abcabcabc"
```

Inlining vs. Macros

PLRG

A key difference between **inlining** and **macros** is:

- **Inlining** works by rewriting code and performing optimisations based on **rules the compiler knows**.
- On the other hand, a **macro** executes **user-written code** that generates the code that the macro expands to.

Inlining vs. Macros

A key difference between **inlining** and **macros** is:

- **Inlining** works by rewriting code and performing optimisations based on **rules the compiler knows**.
- On the other hand, a **macro** executes **user-written code** that generates the code that the macro expands to.

Technically, compiling the inlined code `${ inspectCode('x) }` calls the method `inspectCode` at compile time (through Java reflection), and the method `inspectCode` then executes as normal code.

Inlining vs. Macros

A key difference between **inlining** and **macros** is:

- **Inlining** works by rewriting code and performing optimisations based on **rules the compiler knows**.
- On the other hand, a **macro** executes **user-written code** that generates the code that the macro expands to.

Technically, compiling the inlined code `${ inspectCode('x) }` calls the method `inspectCode` at compile time (through Java reflection), and the method `inspectCode` then executes as normal code.

To execute `inspectCode`, we need to compile its source code first.

Inlining vs. Macros

PLRG

A key difference between **inlining** and **macros** is:

- **Inlining** works by rewriting code and performing optimisations based on **rules the compiler knows**.
- On the other hand, a **macro** executes **user-written code** that generates the code that the macro expands to.

Technically, compiling the inlined code `${ inspectCode('x) }` calls the method `inspectCode` at compile time (through Java reflection), and the method `inspectCode` then executes as normal code.

To execute `inspectCode`, we need to compile its source code first.

As a technical consequence, we **cannot** define and use a macro in the **same class/file**.

Inlining vs. Macros

PLRG

A key difference between **inlining** and **macros** is:

- **Inlining** works by rewriting code and performing optimisations based on **rules the compiler knows**.
- On the other hand, a **macro** executes **user-written code** that generates the code that the macro expands to.

Technically, compiling the inlined code `${ inspectCode('x) }` calls the method `inspectCode` at compile time (through Java reflection), and the method `inspectCode` then executes as normal code.

To execute `inspectCode`, we need to compile its source code first.

As a technical consequence, we **cannot** define and use a macro in the **same class/file**.

However, it is **possible** to have the macro definition and its call in the **same project** as long as the implementation of the macro can be compiled first.

Macros – Examples

Let's define a **macro** that computes the **power** of a given integer `x` with an integer `n` as an exponent at compile-time:

```
import scala.quoted.*

def unrollPower(x: Expr[Int], n: Int)(using Quotes): Expr[Int] = n match
  case 0 => '{ 1 }
  case 1 => x
  case _ => '{ $x * ${ unrollPower(x, n - 1) } }

def powerCode(x: Expr[Int], n: Expr[Int])(using Quotes): Expr[Int] =
  val code = unrollPower(x, n.valueOrAbort)
  println(s"power(${x.show}, ${n.show}) compiled to `${code.show}`")
  code

inline def power(inline x: Int, inline n: Int) = ${ powerCode('x, 'n) }
```

Macros – Examples

Let's define a **macro** that computes the **power** of a given integer `x` with an integer `n` as an exponent at compile-time:

```
import scala.quoted.*

def unrollPower(x: Expr[Int], n: Int)(using Quotes): Expr[Int] = n match
  case 0 => '{ 1 }
  case 1 => x
  case _ => '{ $x * ${ unrollPower(x, n - 1) } }

def powerCode(x: Expr[Int], n: Expr[Int])(using Quotes): Expr[Int] =
  val code = unrollPower(x, n.valueOrAbort)
  println(s"power(${x.show}, ${n.show}) compiled to `${code.show}`")
  code

inline def power(inline x: Int, inline n: Int) = ${ powerCode('x, 'n) }
```

```
def cubic(x: Int): Int = power(x, 3)
// power(x = x, n = 3) compiled to `x * x * x`
```

Summary

PLRG

1. Inline
    - Inline Constants
    - Inline Methods
    - Inline Parameters
    - Inline Matches
    - Transparent Inline Methods

2. Macros

Next Lecture

- Concurrent Programming

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr