Type-Level Programming 101

The Subspace of Scala

Joe Barnes

Follow along at http://192.168.183.88:8080

Powered by

Joe Barnes

@joescii

prose :: and :: conz

Primarily Java from 2004-2013

Started Scala late 2012

Started Scala at Mentor Graphics late 2013

Lift committer in July 2014

So... Type programming...

What do *you* know about *that*, Joe??

Not much...

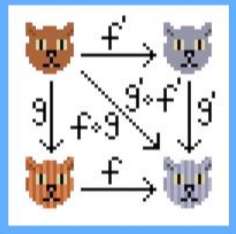
So... Type programming...

What do *you* know about *tha*

I've made zero contributions to shapeless

Not much...

Also no contributions to Cats



Type-Level Programming 101

The Subspace of Scala

Joe Barnes

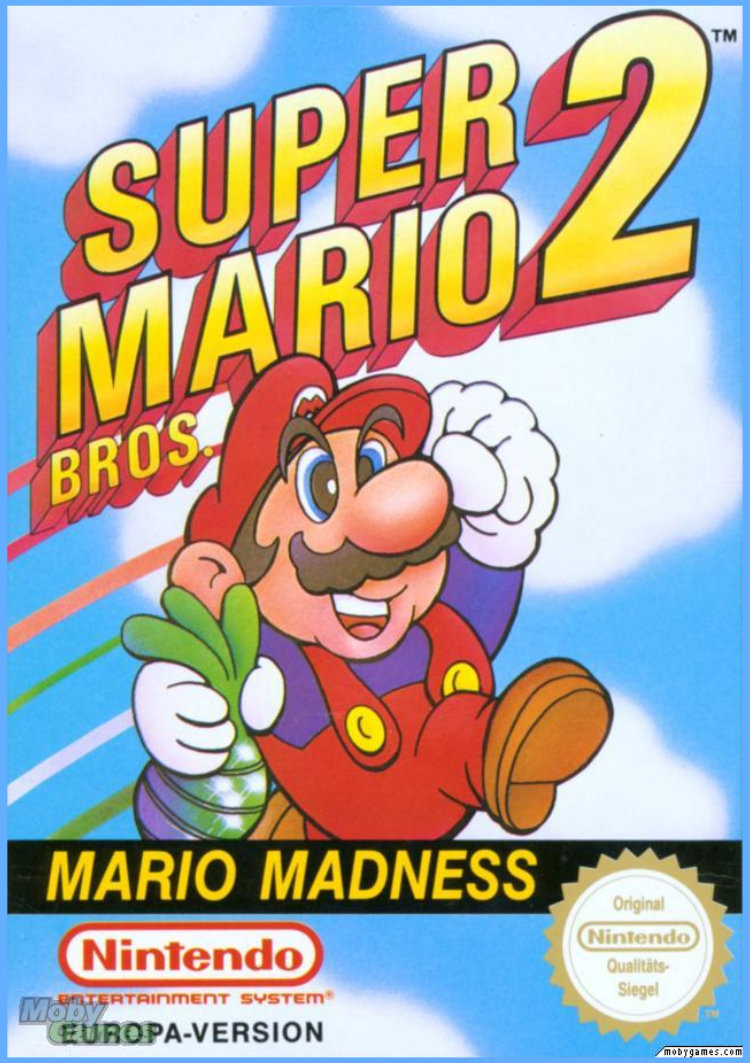
Follow along at http://192.168.183.88:8080

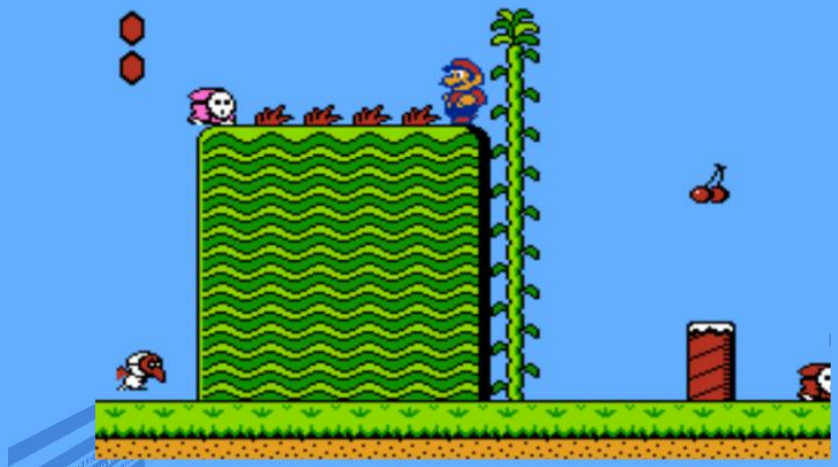
Powered by Lift

Although I have contributed plenty of cat memes!

I'm not here to share expertise on the subject,
but rather my *Aha!* moment.

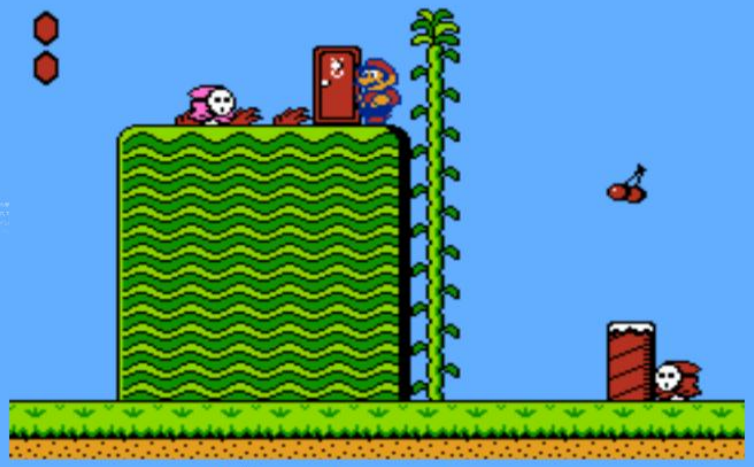
Programming in Scala is like...


SUPER MARIO 2
BROS.
MARIO MADNESS
Nintendo
ENTERTAINMENT SYSTEM
EUROPA-VERSION
Original
Nintendo
Qualitäts-
Siegel
mobygames.com



Normal value programming

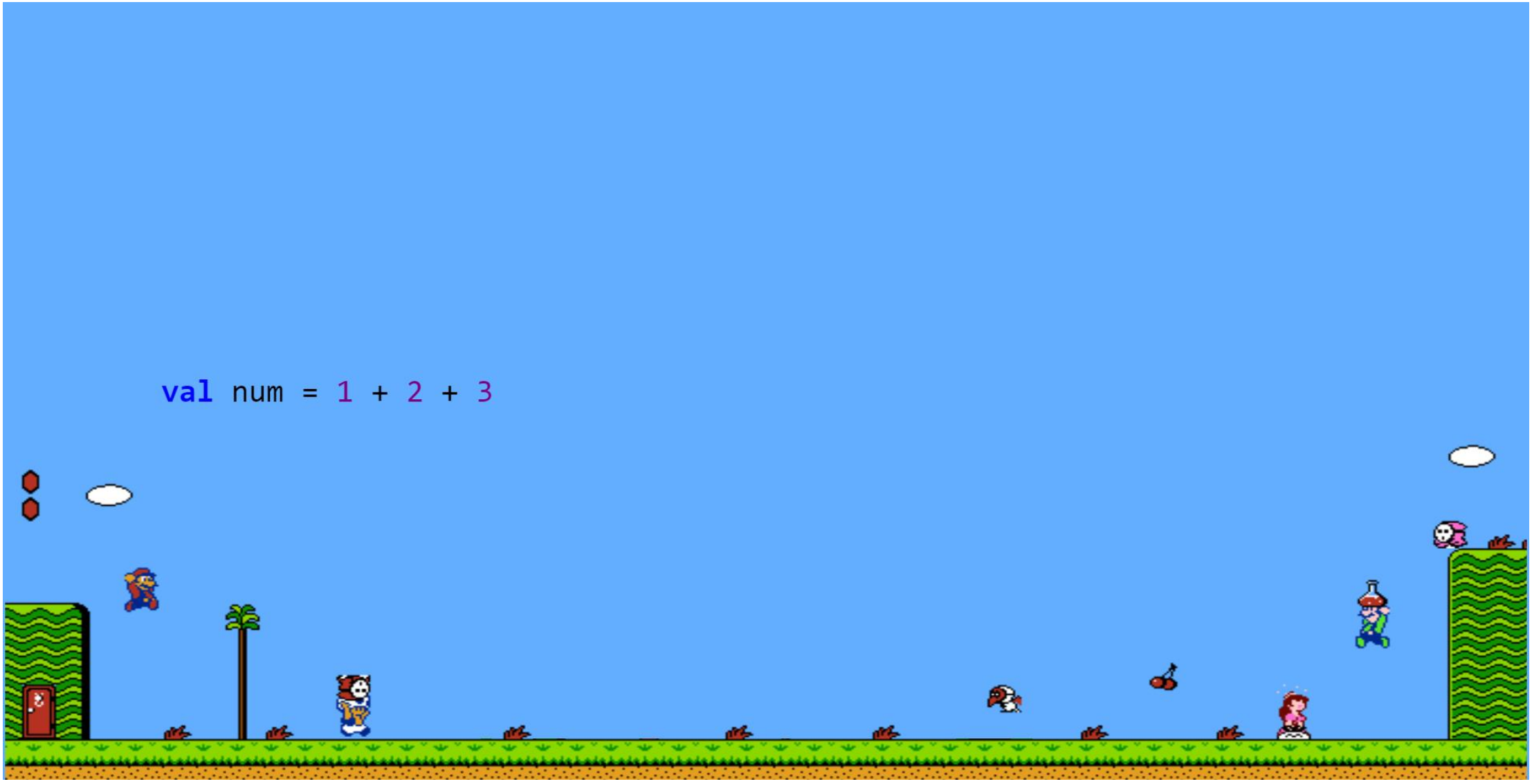
A flask of type programming!



Where does this go?

SUBSPACE!

```
val num = 1 + 2 + 3
```



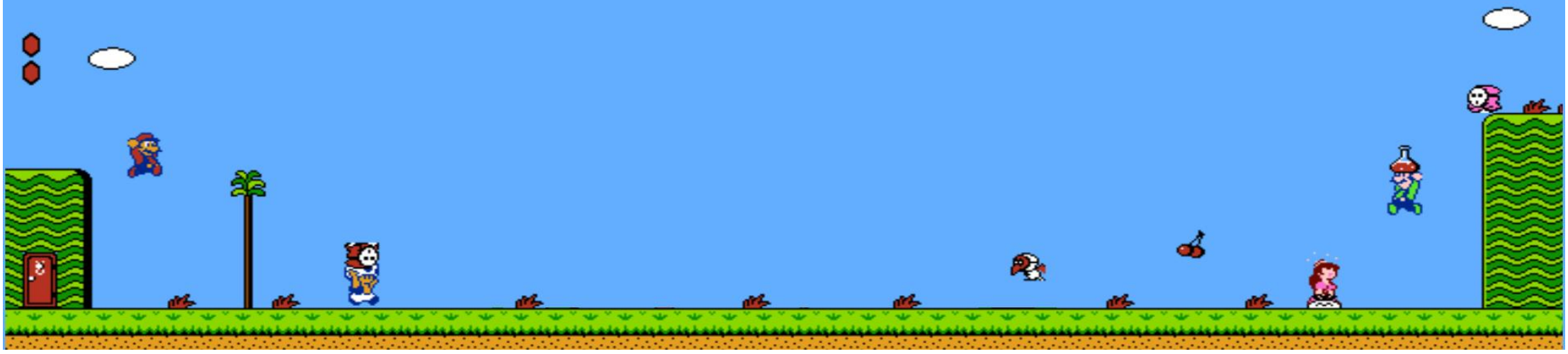
```
val num = 1 + 2 + 3
```

```
lazy val str = "a"+"b"+"c"
```

```
val num = 1 + 2 + 3
```

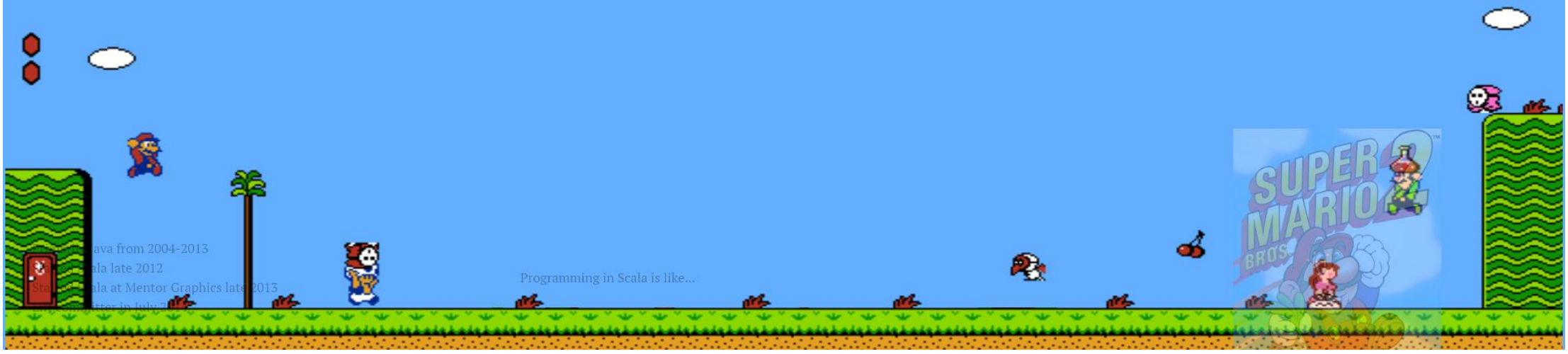
```
lazy val str = "a"+"b"+"c"
```

```
def now = new java.util.Date
```

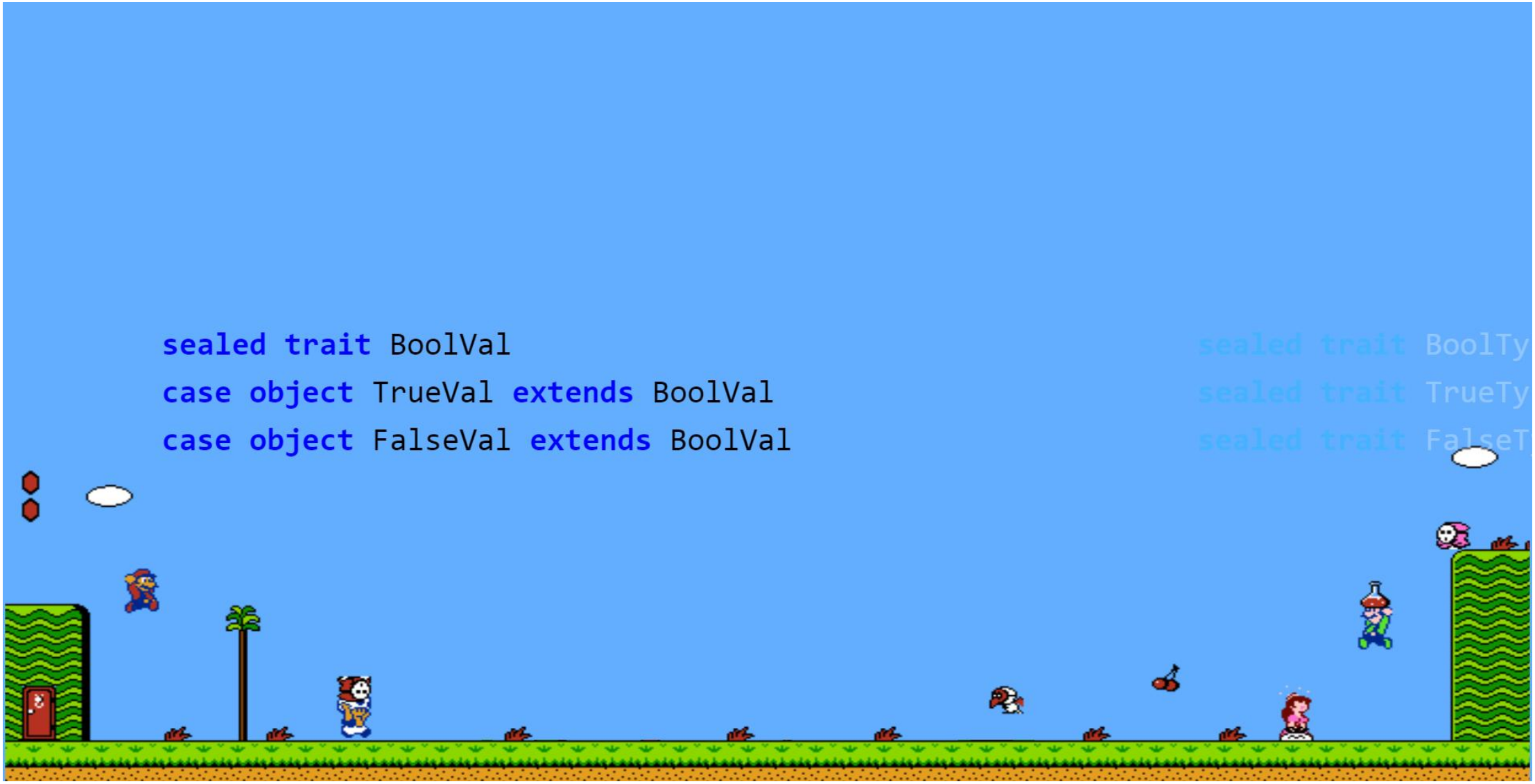


```
type MyMap = Map[Int, String]
```

Let's compare values to types with the simplest domain ever

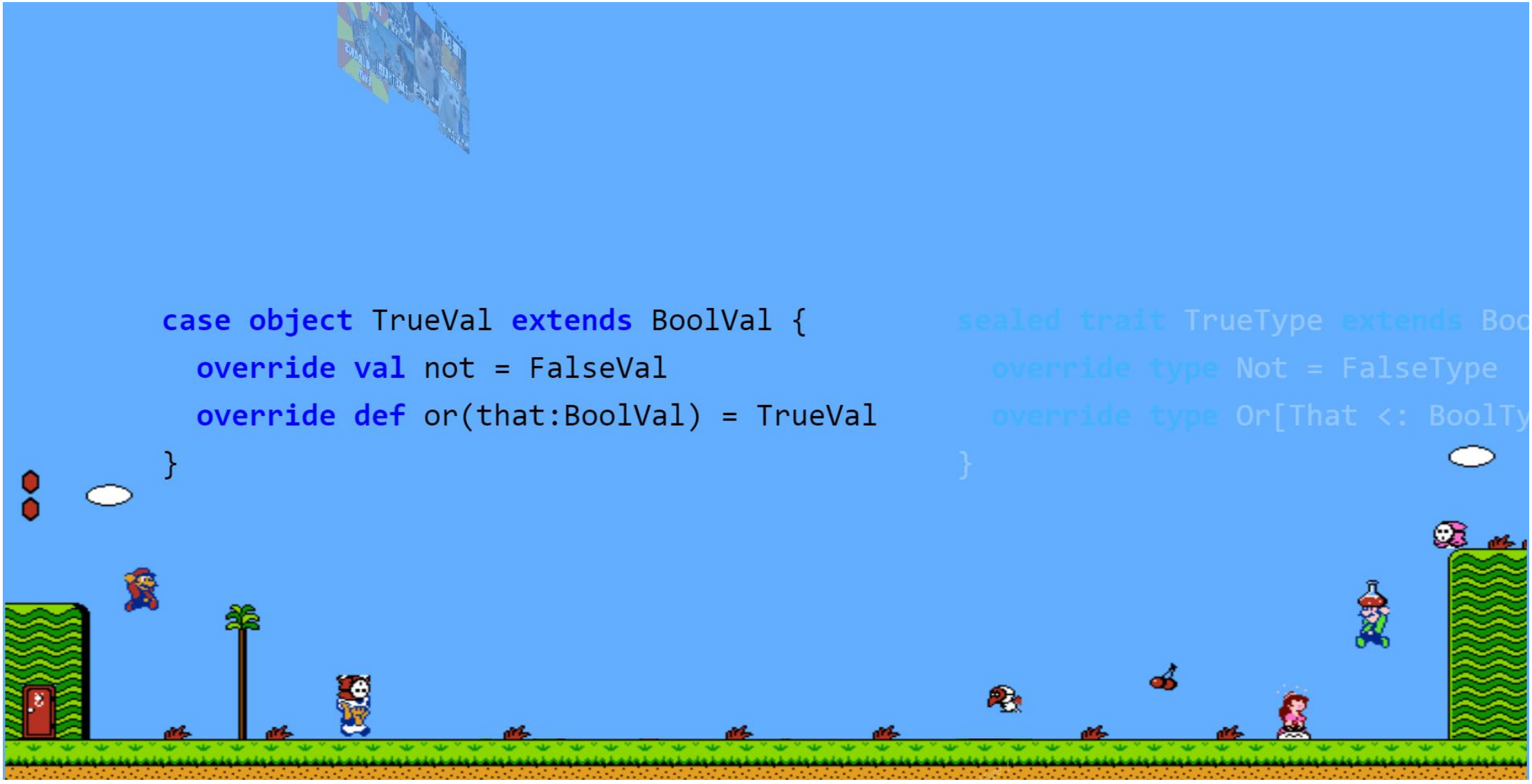


```scala
sealed trait BoolVal
case object TrueVal extends BoolVal
case object FalseVal extends BoolVal
```

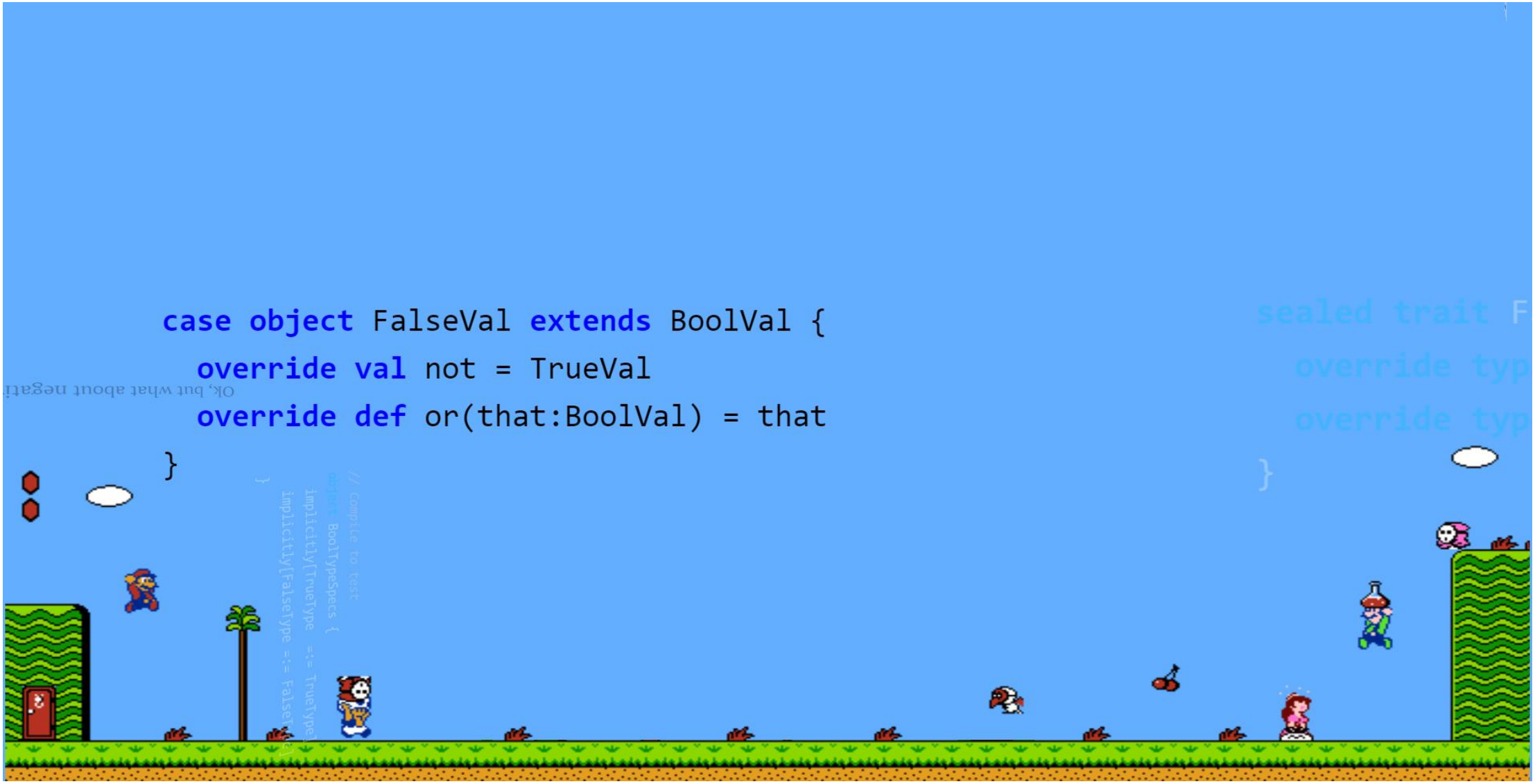


```scala
sealed trait BoolVal {
  def not:BoolVal
  def or(that:BoolVal):BoolVal
}
```

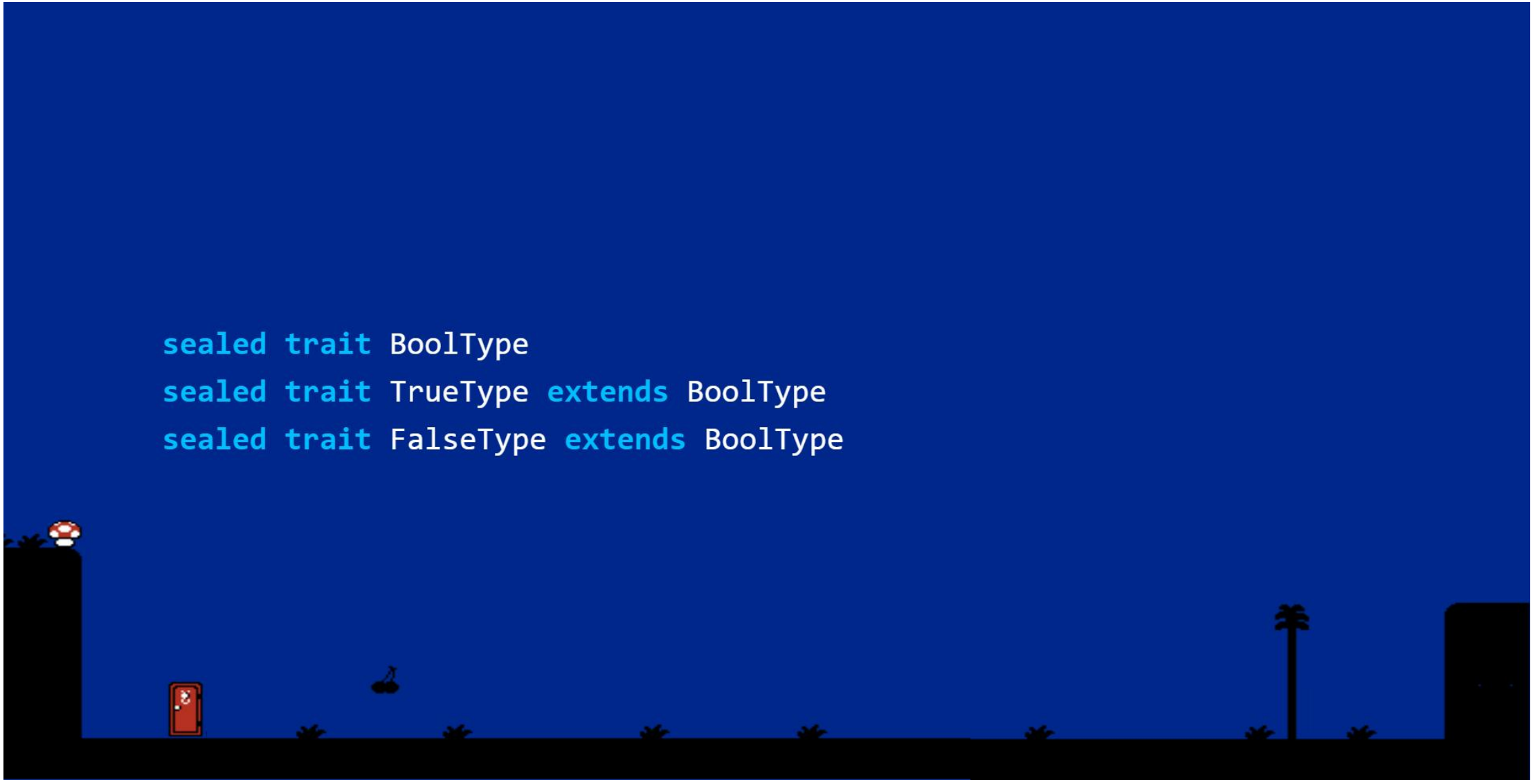
```scala
sealed trait BoolType {
  type Not <: BoolType
  type Or[That <: BoolType] <:
}
```


case object TrueVal extends BoolVal {
override val not = FalseVal
override def or(that:BoolVal) = TrueVal
}
sealed trait TrueType extends Boo
override type Not = FalseType
override type Or[That <: BoolTy
}

```
case object FalseVal extends BoolVal {
  override val not = TrueVal
  override def or(that:BoolVal) = that
}
```



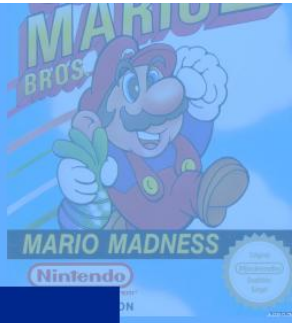
```
sealed trait BoolType
sealed trait TrueType extends BoolType
sealed trait FalseType extends BoolType
```



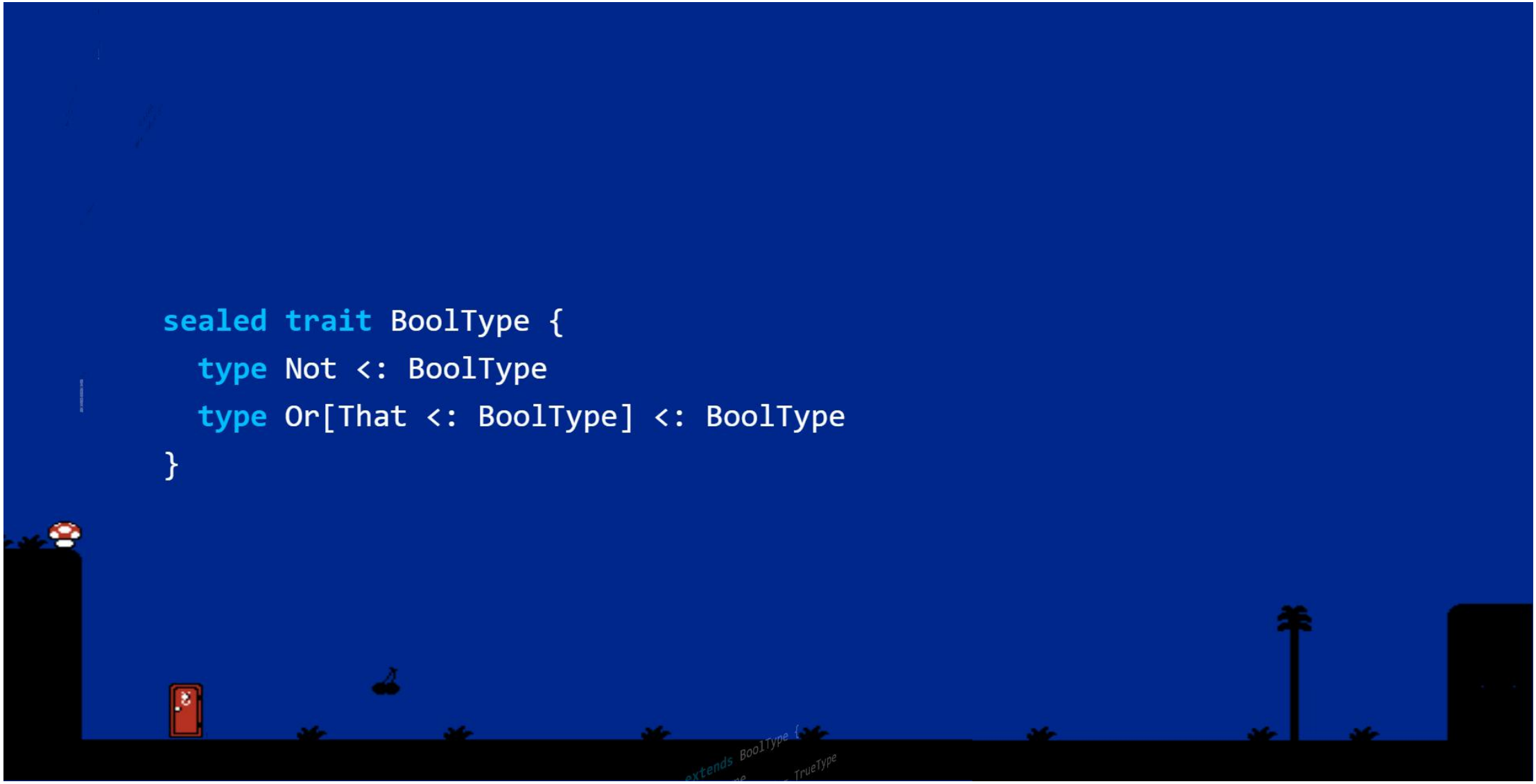
oe Barnes
@joescii
rose :: and :: conz

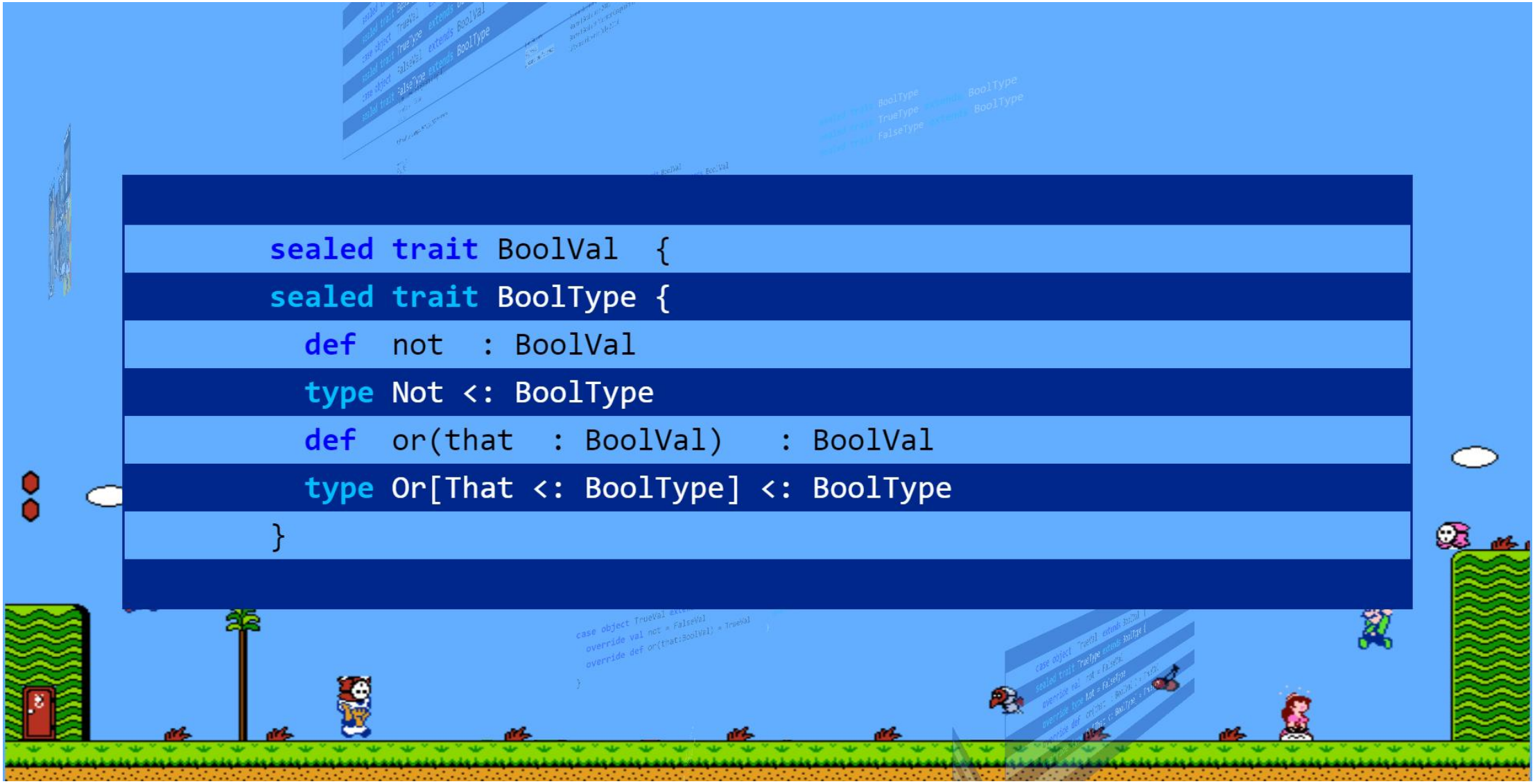
Primarily Java from 2004-2013
Started Scala late 2012
Started Scala at Mentor Graphics late 2013
Lift committer in July 2014

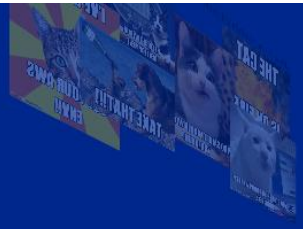
Programming in Scala is like...



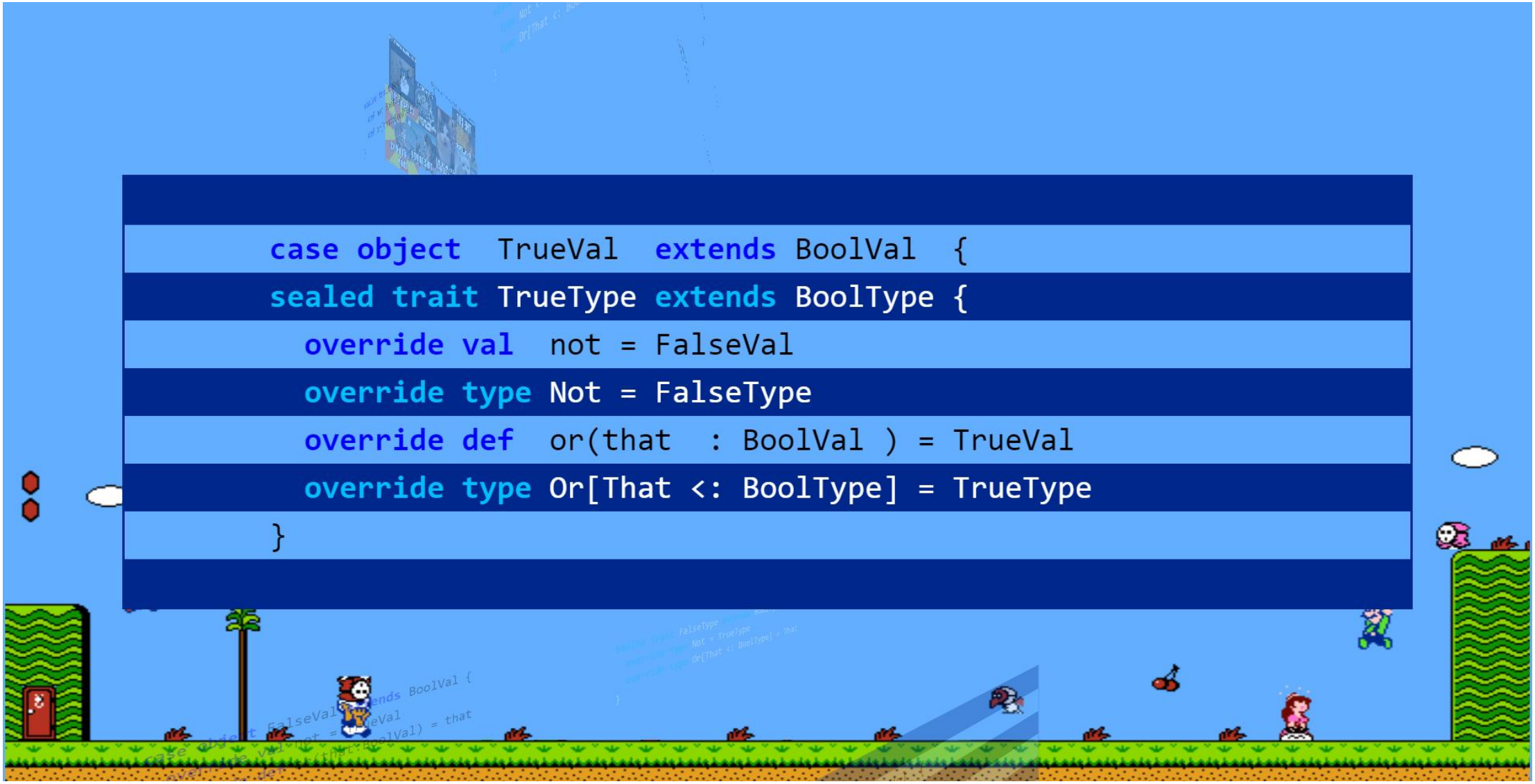

```
sealed trait BoolVal
sealed trait BoolType
case object  TrueVal   extends BoolVal
sealed trait TrueType  extends BoolType
case object  FalseVal  extends BoolVal
sealed trait FalseType extends BoolType
```


sealed trait BoolType {
type Not <: BoolType
type Or[That <: BoolType] <: BoolType
}

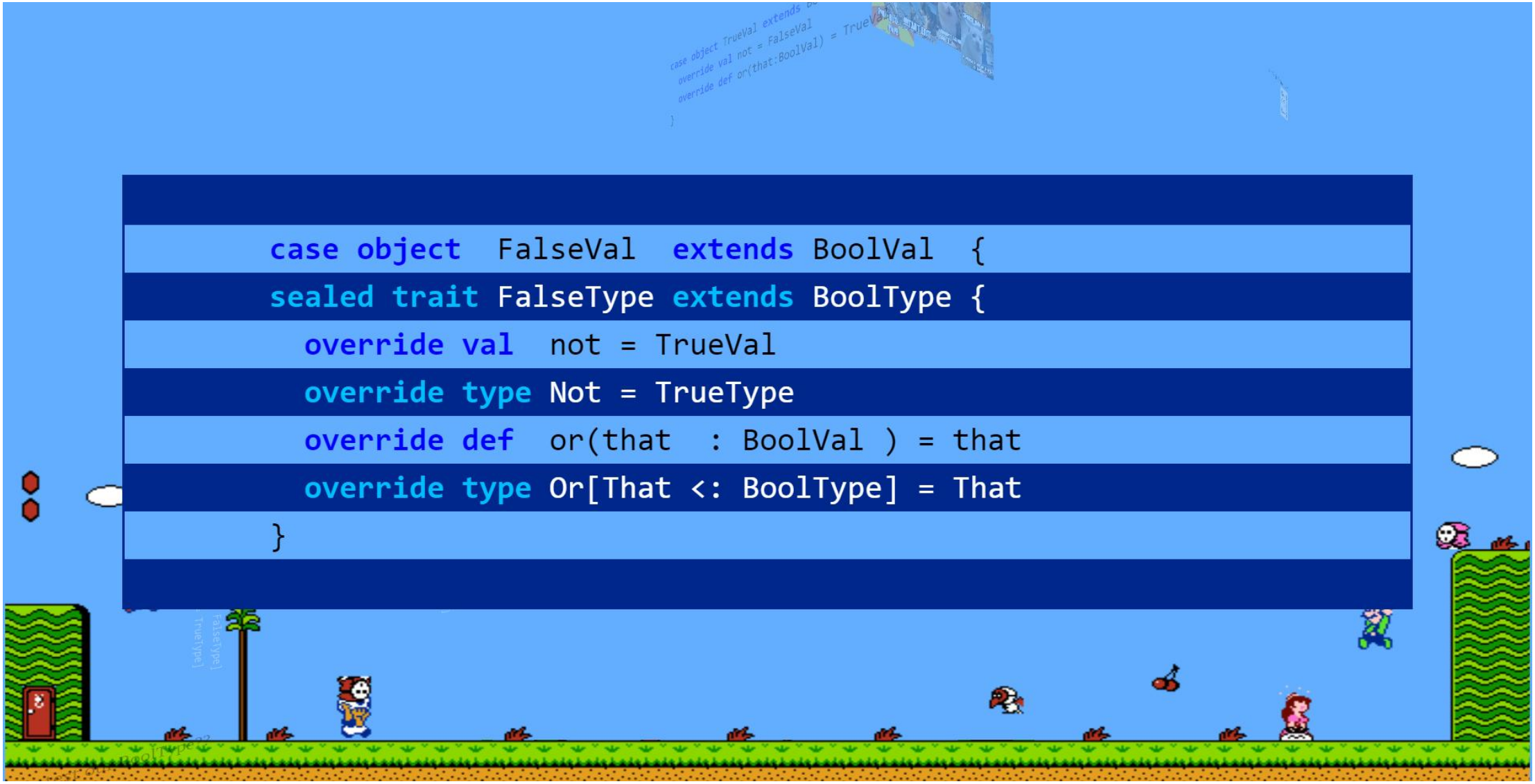

sealed trait BoolVal {
sealed trait BoolType {
def not : BoolVal
type Not <: BoolType
def or(that : BoolVal)   : BoolVal
type Or[That <: BoolType] <: BoolType
}



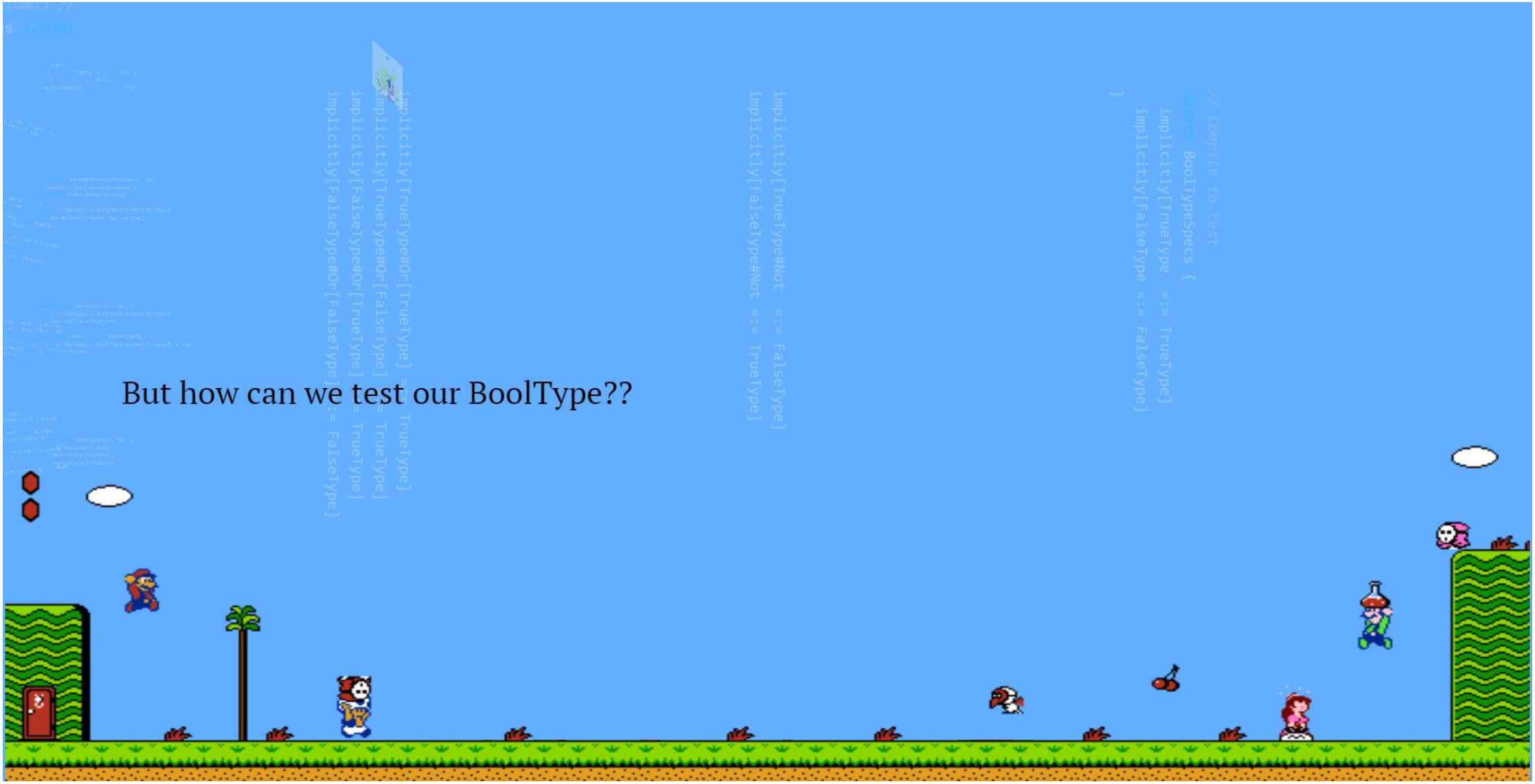
```
sealed trait TrueType extends BoolType {
  override type Not = FalseType
  override type Or[That <: BoolType] = TrueType
}
```


case object TrueVal extends BoolVal {
sealed trait TrueType extends BoolType {
override val not = FalseVal
override type Not = FalseType
override def or(that : BoolVal) = TrueVal
override type Or[That <: BoolType] = TrueType
}

```scala
sealed trait FalseType extends BoolType {
  override type Not = TrueType
  override type Or[That <: BoolType] = That
}
```

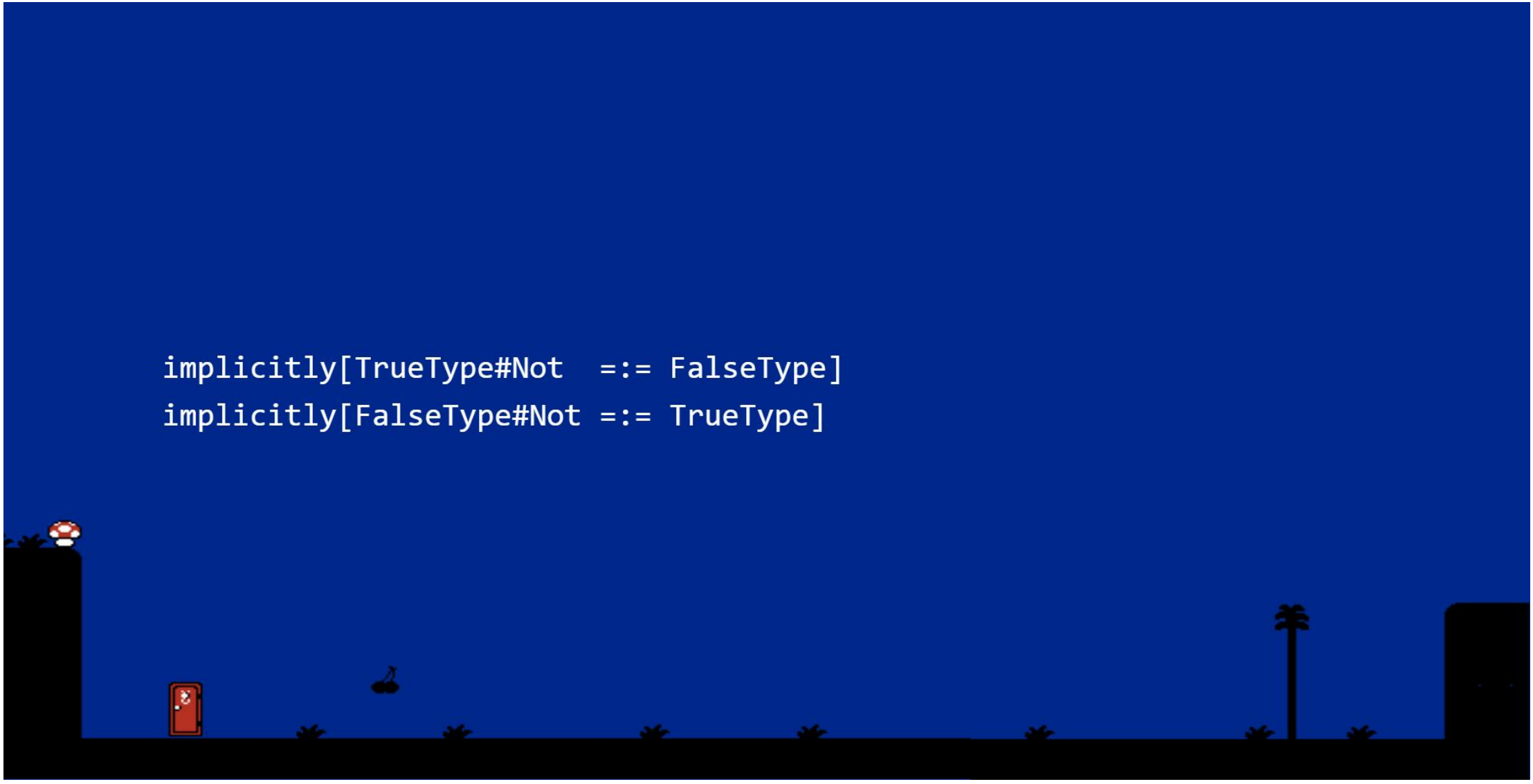

case object FalseVal extends BoolVal {
sealed trait FalseType extends BoolType {
override val not = TrueVal
override type Not = TrueType
override def or(that : BoolVal) = that
override type Or[That <: BoolType] = That
}

But how can we test our BoolType??

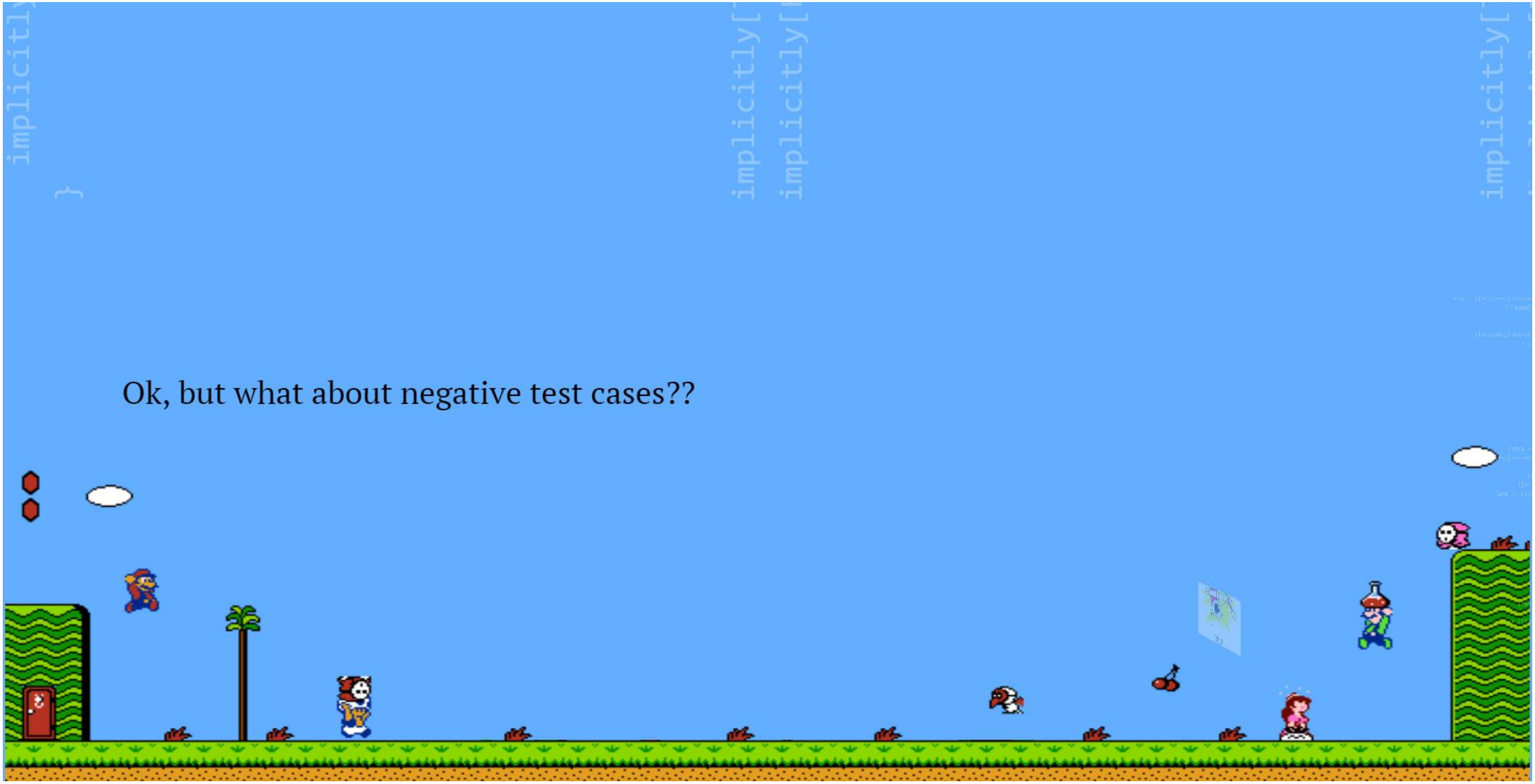


```scala
// Compile to test
object BoolTypeSpecs {
  implicitly[TrueType  =:= TrueType]
  implicitly[FalseType =:= FalseType]
}
```

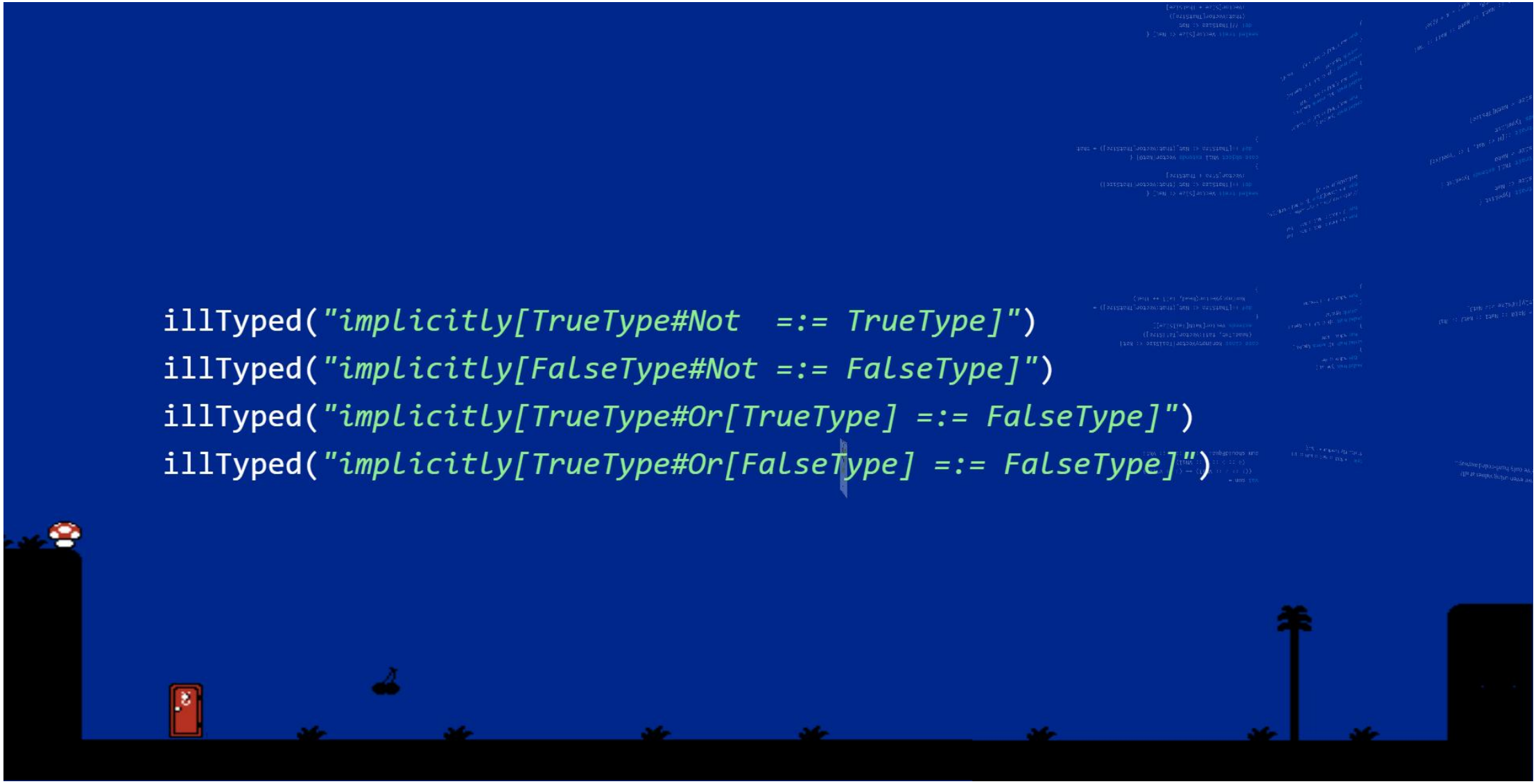
```
implicitly[TrueType#Not  =:= FalseType]
implicitly[FalseType#Not =:= TrueType]
```



```
implicitly[TrueType#Or[TrueType] =:= TrueType]
implicitly[TrueType#Or[FalseType] =:= TrueType]
implicitly[FalseType#Or[TrueType] =:= TrueType]
implicitly[FalseType#Or[FalseType] =:= FalseType]
```


Ok, but what about negative test cases??


import shapeless.test.illTyped
// Compiles only if string DOESN'T compile
illTyped("implicitly[TrueType =:= FalseType]")
illTyped("implicitly[FalseType =:= TrueType]")

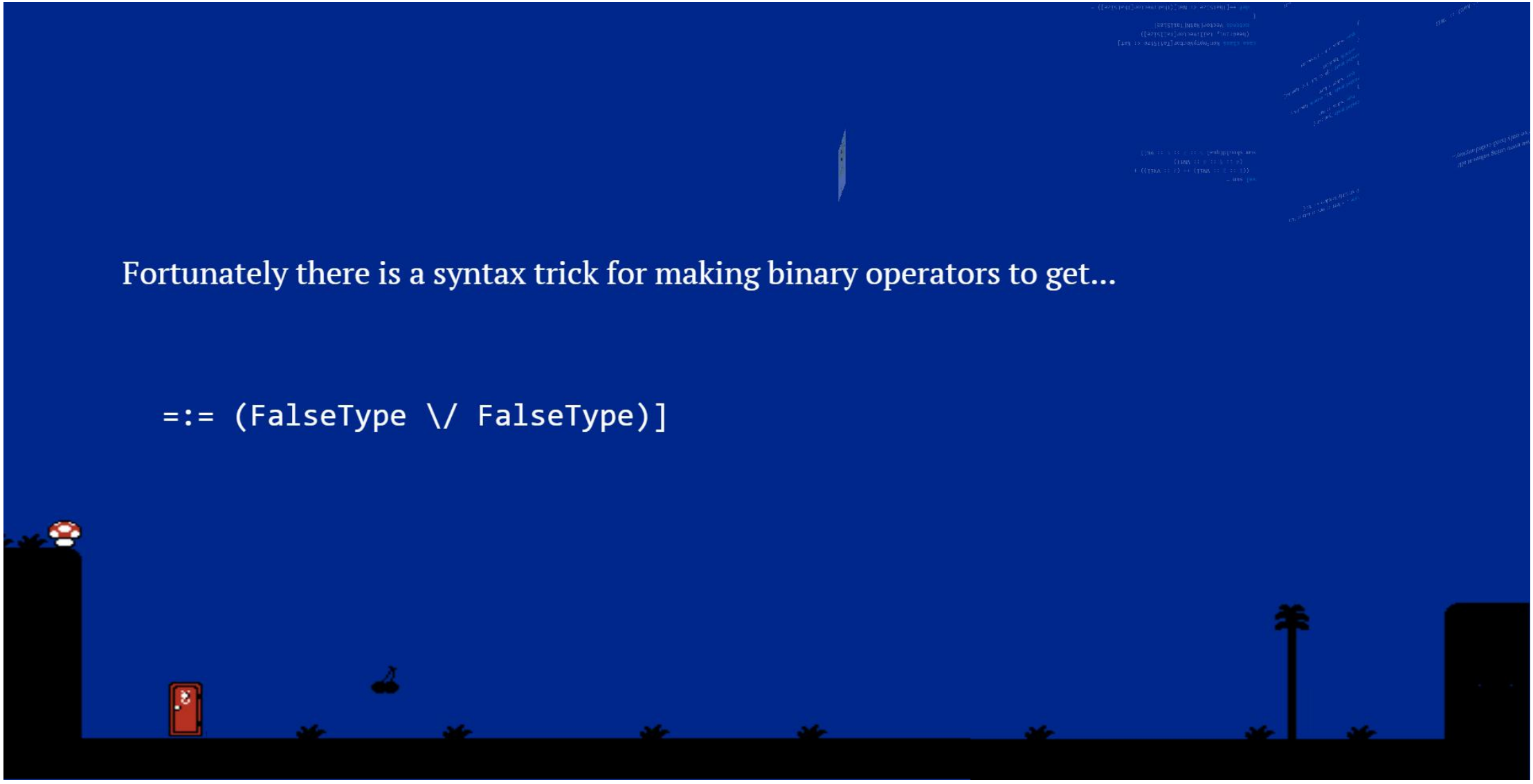

illTyped("implicitly[TrueType#Not =:= TrueType]")
illTyped("implicitly[FalseType#Not =:= FalseType]")
illTyped("implicitly[TrueType#Or[TrueType] =:= FalseType]")
illTyped("implicitly[TrueType#Or[FalseType] =:= FalseType]")

That's cool and everything, but this is kinda ugly for Scala...

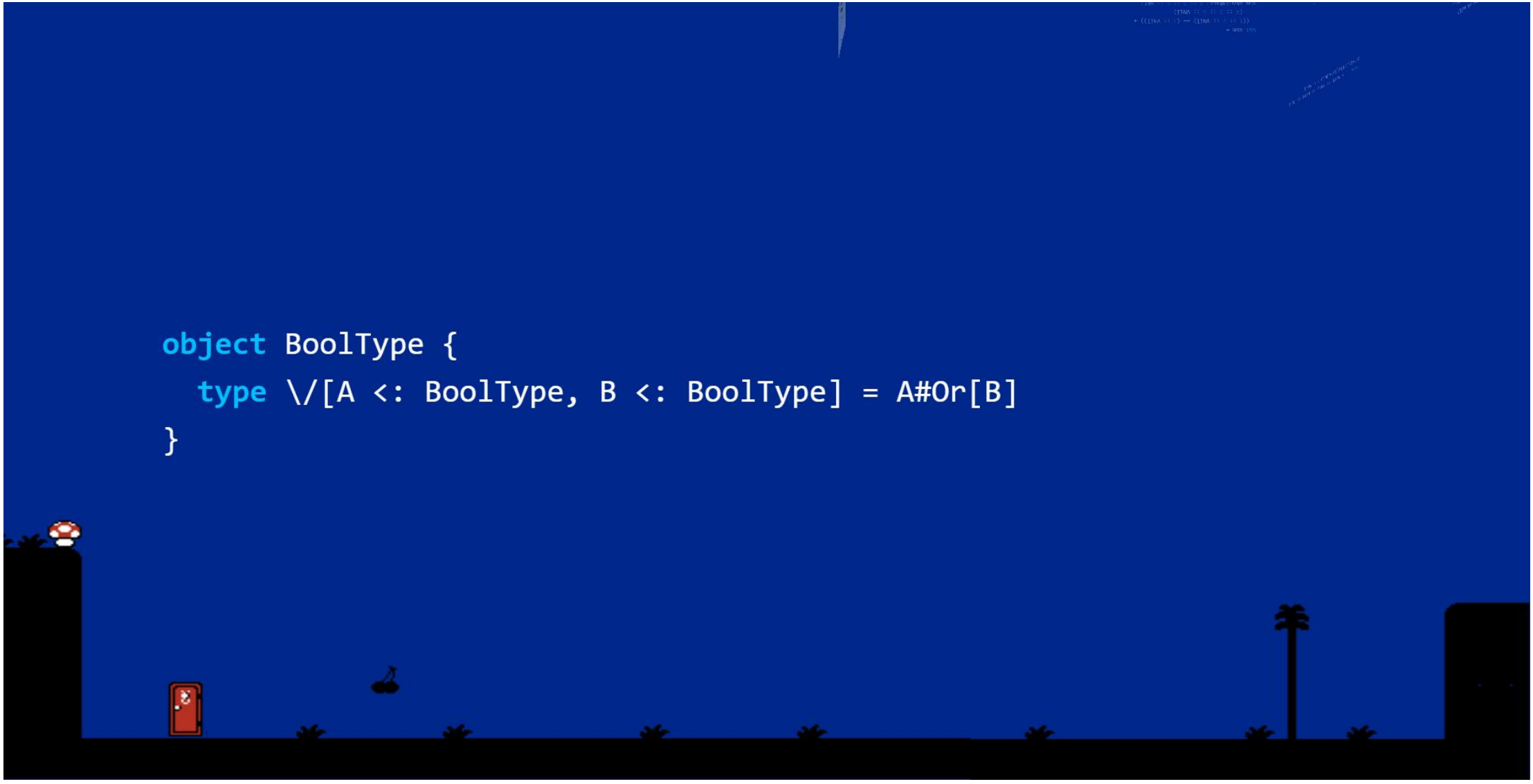
```
implicitly[TrueType#Or[FalseType] =:= TrueType]
```

Fortunately there is a syntax trick for making binary operators to get...

```
=:= (FalseType \/ FalseType)]
```



```
object BoolType {
  type \/[A <: BoolType, B <: BoolType] = A#Or[B]
}
```

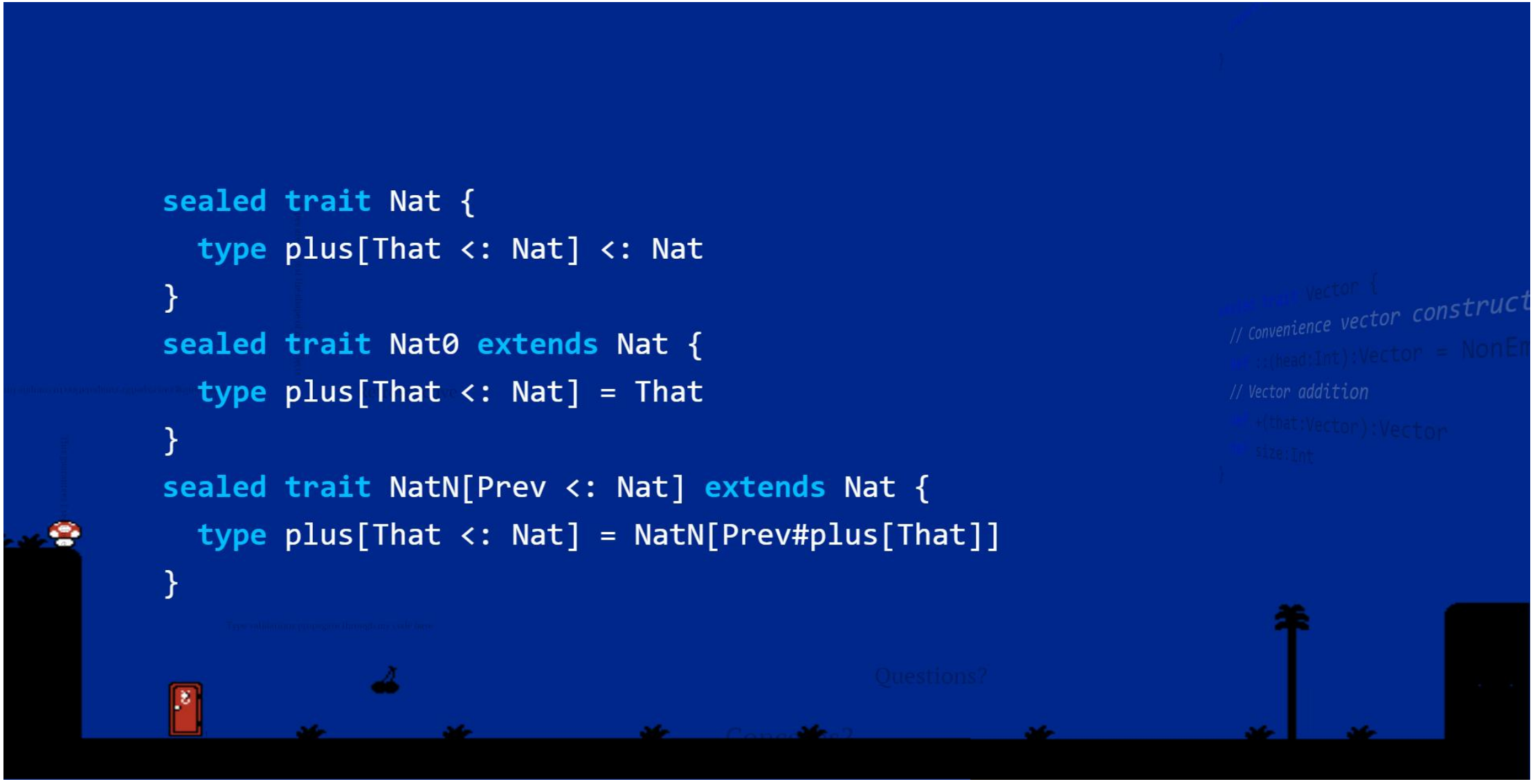



// A type with two parameters
// can be written in infix notation:
implicitly[\/[FalseType, FalseType]
=:= (FalseType \/ FalseType)]

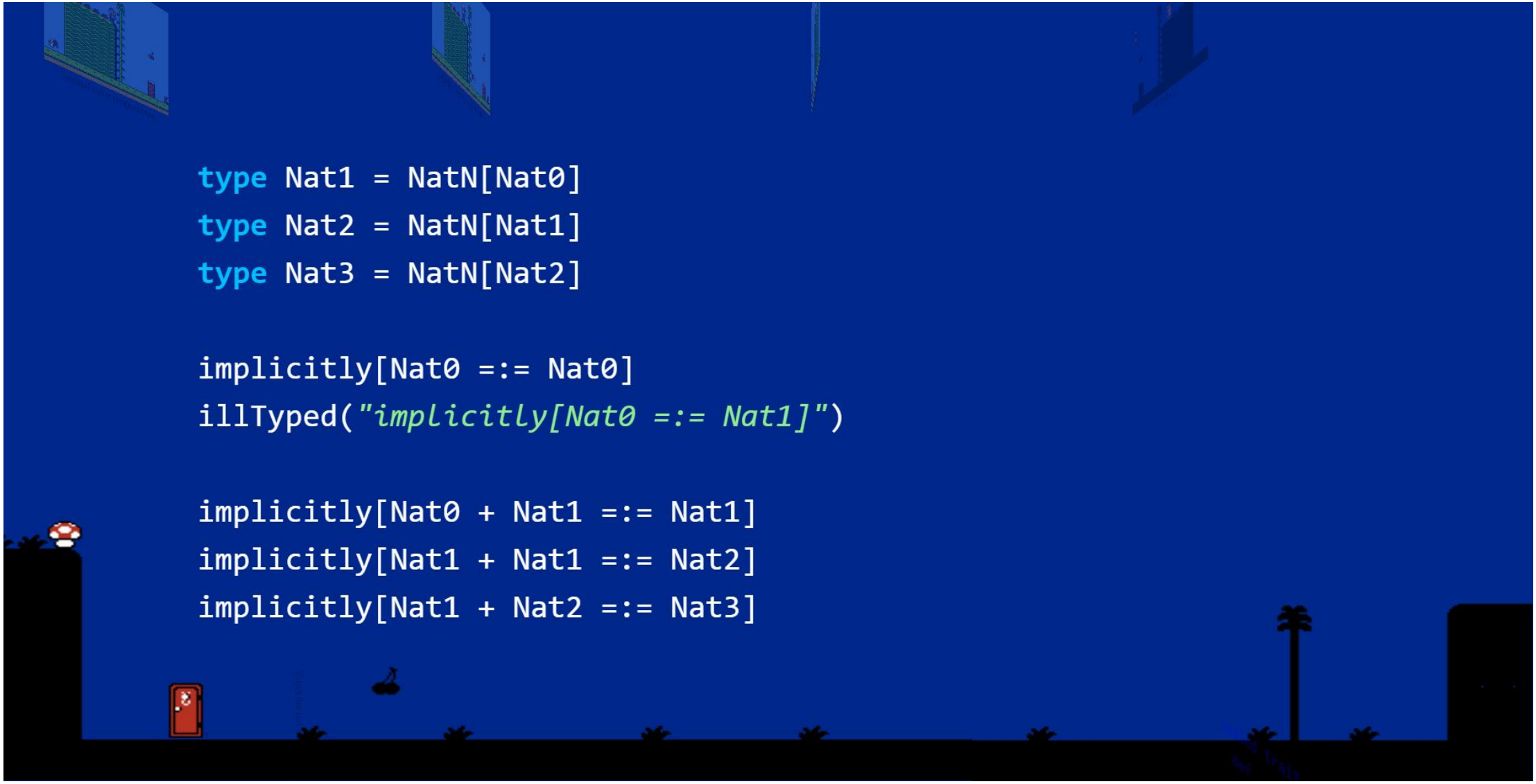

MARIO MADNESS
Nintendo
type Nat
type Nat
type Nat
implicit
illTyped
implicit
Well that was easy... It's just two types, really...


sealed trait Nat {
def +(that:Nat):Nat
}
case object Nat0 extends Nat {
override def +(that:Nat) = that
}
case class NatN(prev:Nat) extends Nat {
override def +(that:Nat) =
NatN(prev + that)
}


val nat1 = NatN(Nat0)
val nat2 = NatN(nat1)
val nat3 = NatN(nat2)
Nat0 + nat1 shouldEqual nat1
nat1 + nat1 shouldEqual nat2
nat1 + nat2 shouldEqual nat3


sealed trait Nat {
type plus[That <: Nat] <: Nat
}
sealed trait Nat0 extends Nat {
type plus[That <: Nat] = That
}
sealed trait NatN[Prev <: Nat] extends Nat {
type plus[That <: Nat] = NatN[Prev#plus[That]]
}

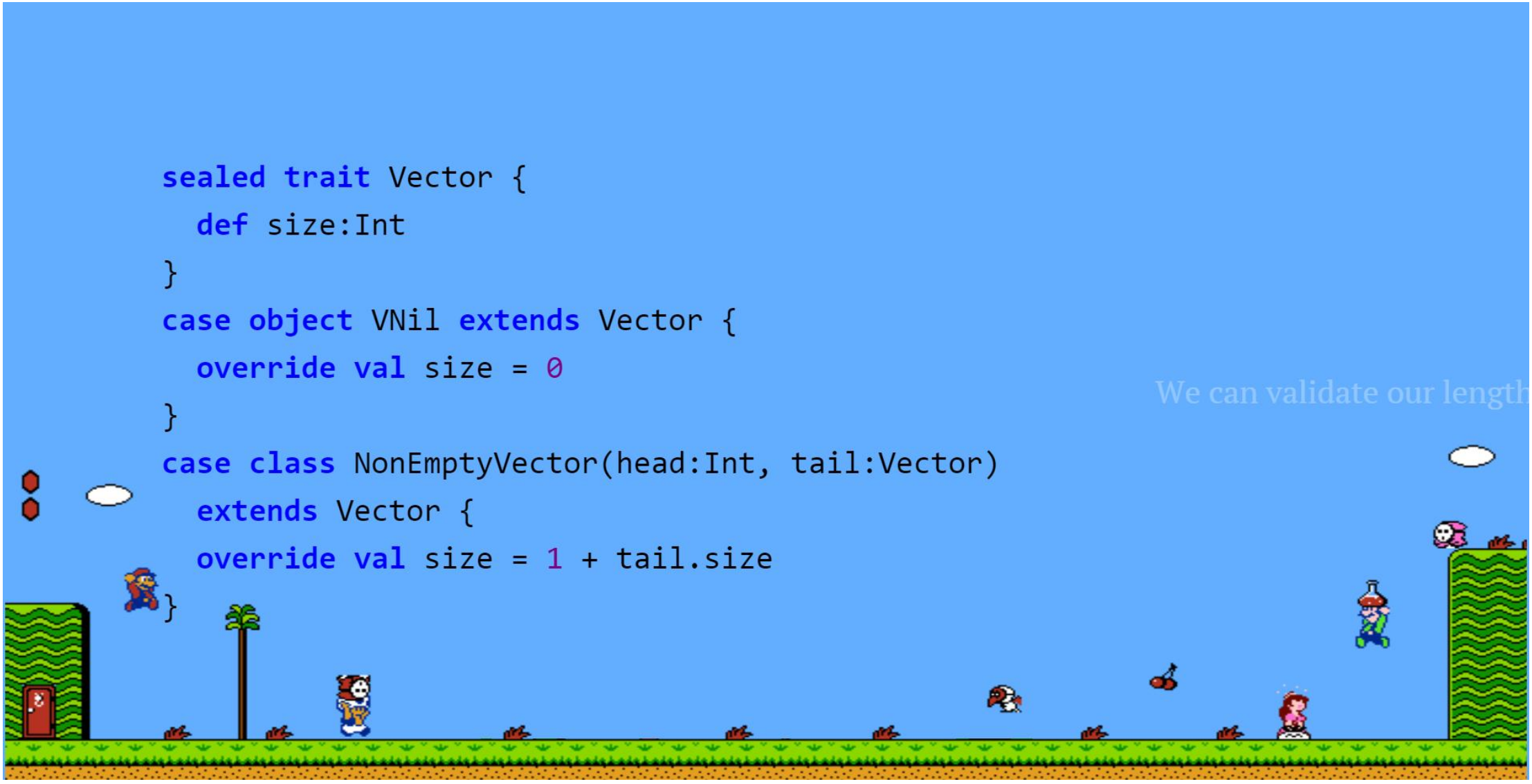

object Nat {
type +[A <: Nat, B <: Nat] = A#plus[B]
}


type Nat1 = NatN[Nat0]
type Nat2 = NatN[Nat1]
type Nat3 = NatN[Nat2]
implicitly[Nat0 =:= Nat0]
illTyped("implicitly[Nat0 =:= Nat1]")
implicitly[Nat0 + Nat1 =:= Nat1]
implicitly[Nat1 + Nat1 =:= Nat2]
implicitly[Nat1 + Nat2 =:= Nat3]


Types, types, and more types, blah, blah, blah. What good is it??

```scala
sealed trait Vector {
  def size:Int
}
case object VNil extends Vector {
  override val size = 0
}
case class NonEmptyVector(head:Int, tail:Vector)
  extends Vector {
  override val size = 1 + tail.size
}
```

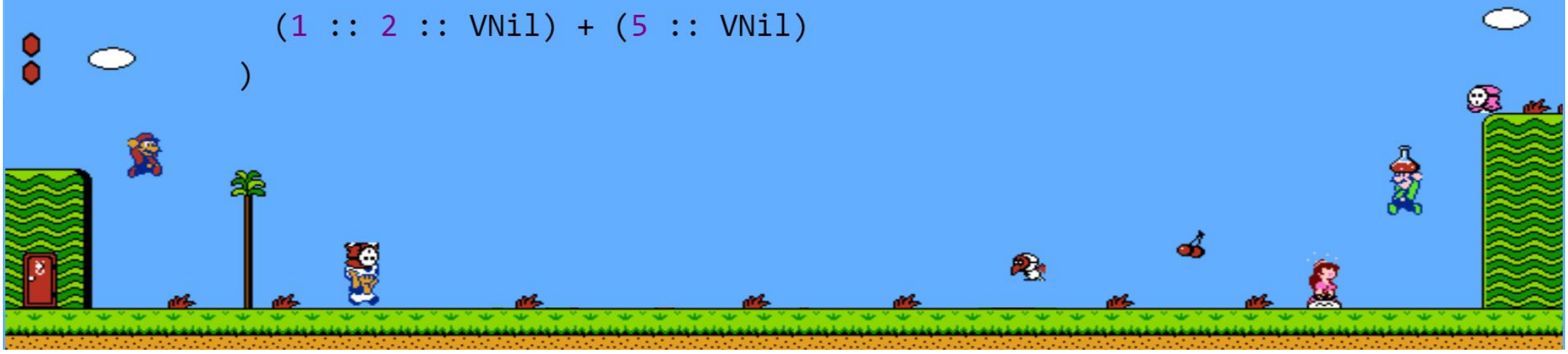
We can validate our length



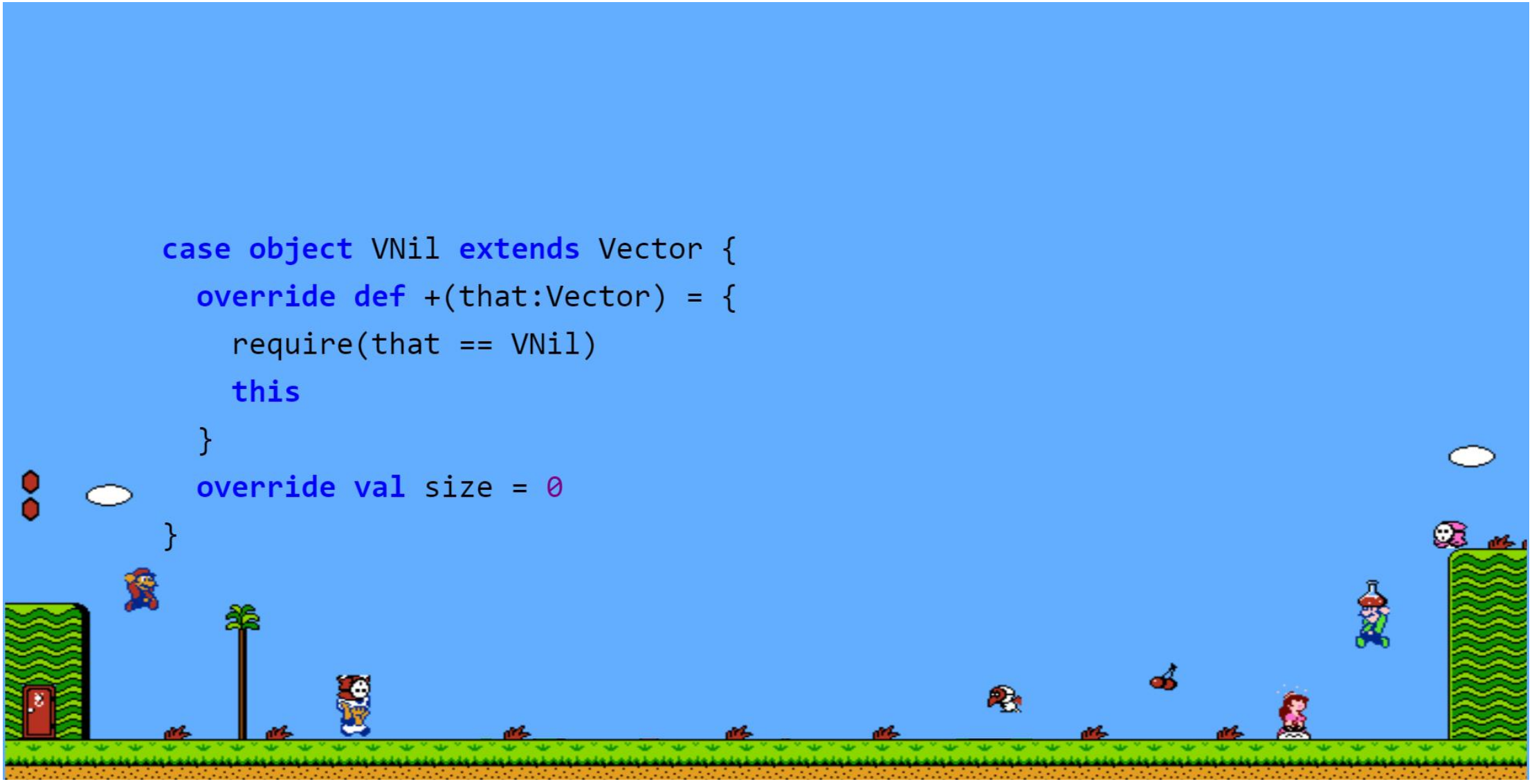
```scala
sealed trait Vector {
  // Convenience vector constructor
  def ::(head:Int):Vector = NonEmptyVector(head, this)
  // Vector addition
  def +(that:Vector):Vector
  def size:Int
}
```

```scala
val sum = (1 ::
sum shouldEqual

illTyped("(1 ::
```

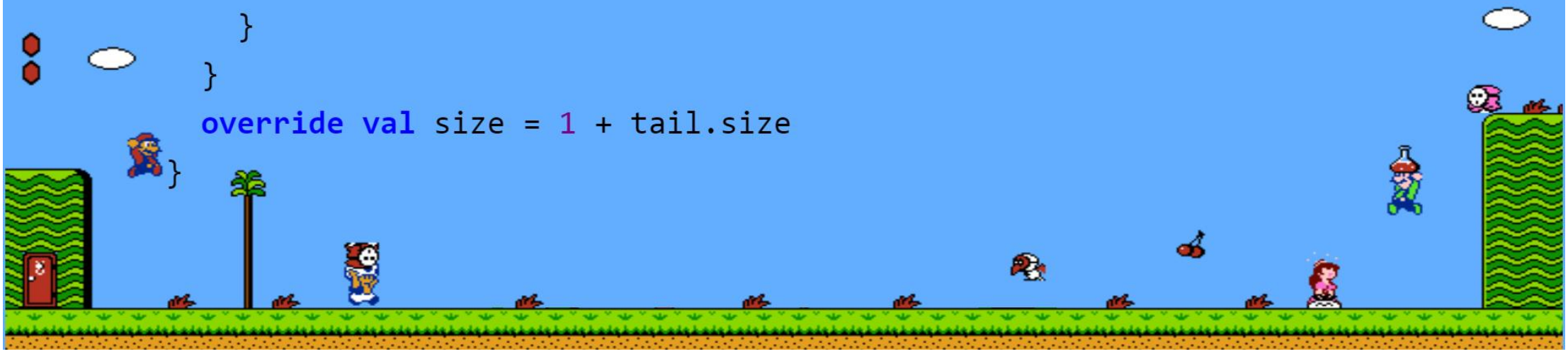
```
val sum = (1 :: 2 :: VNil) + (3 :: 4 :: VNil)
sum shouldEqual 4 :: 6 :: VNil

intercept[IllegalArgumentException](
  (1 :: 2 :: VNil) + (5 :: VNil)
)
```



```
case object VNil extends Vector {
  override def +(that:Vector) = {
    require(that == VNil)
    this
  }
  override val size = 0
}
```

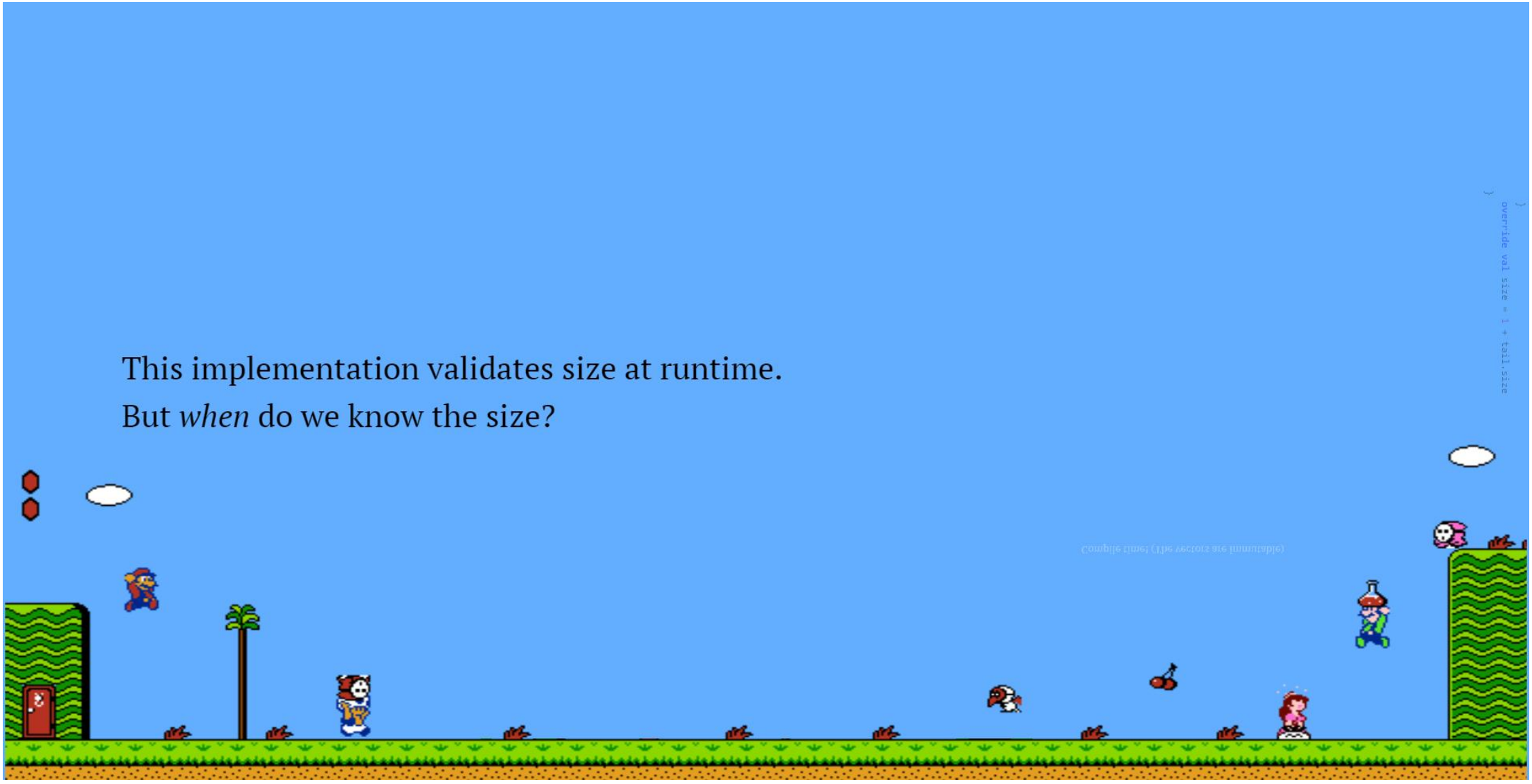


```
case class NonEmptyVector(head:Int, tail:Vector)
  extends Vector {
  override def +(that:Vector) = {
    require(that.size == size)
    that match {
      case NonEmptyVector(h, t) => (head + h) :: (tail + t)
    }
  }
  override val size = 1 + tail.size
}
```

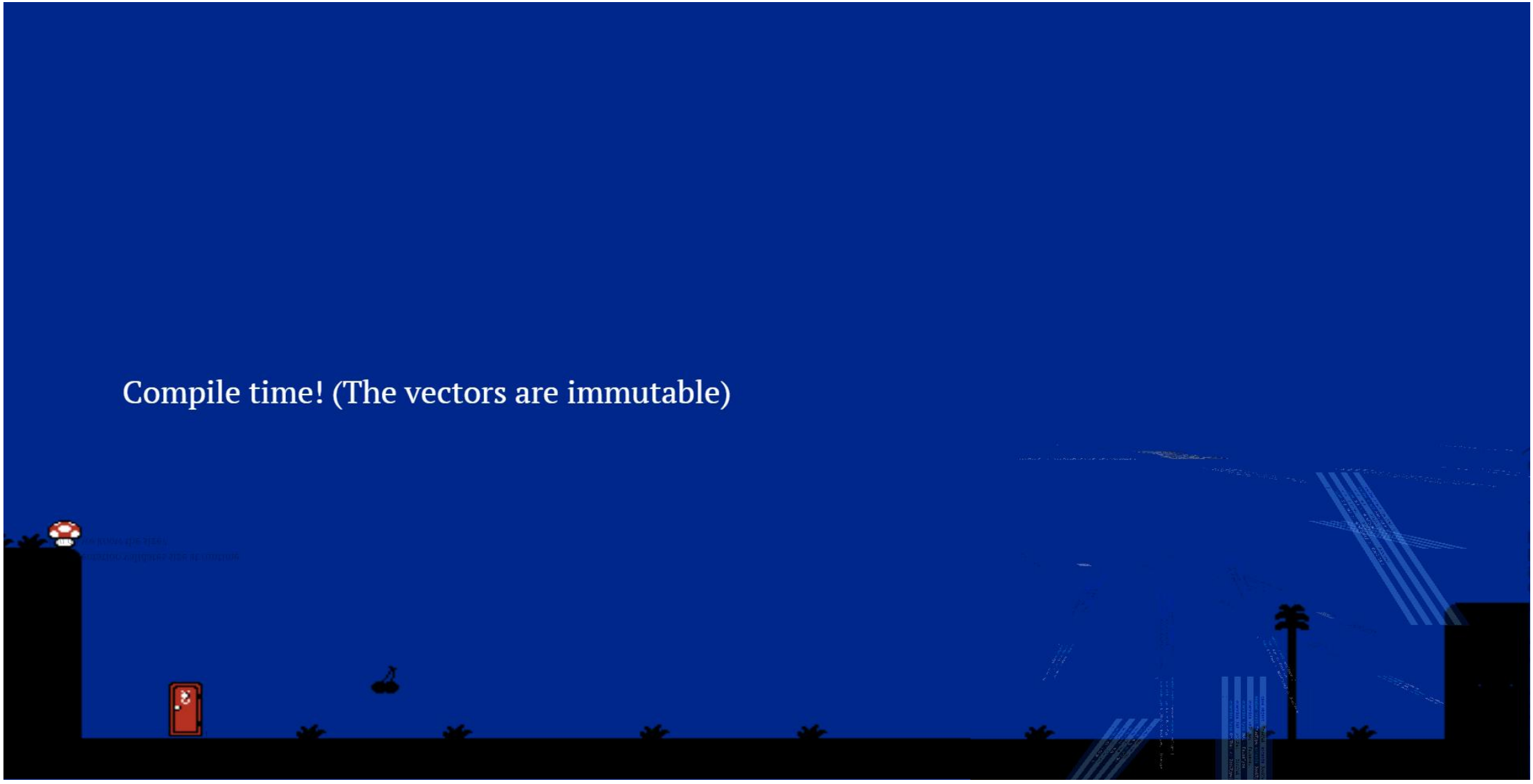


This implementation validates size at runtime.

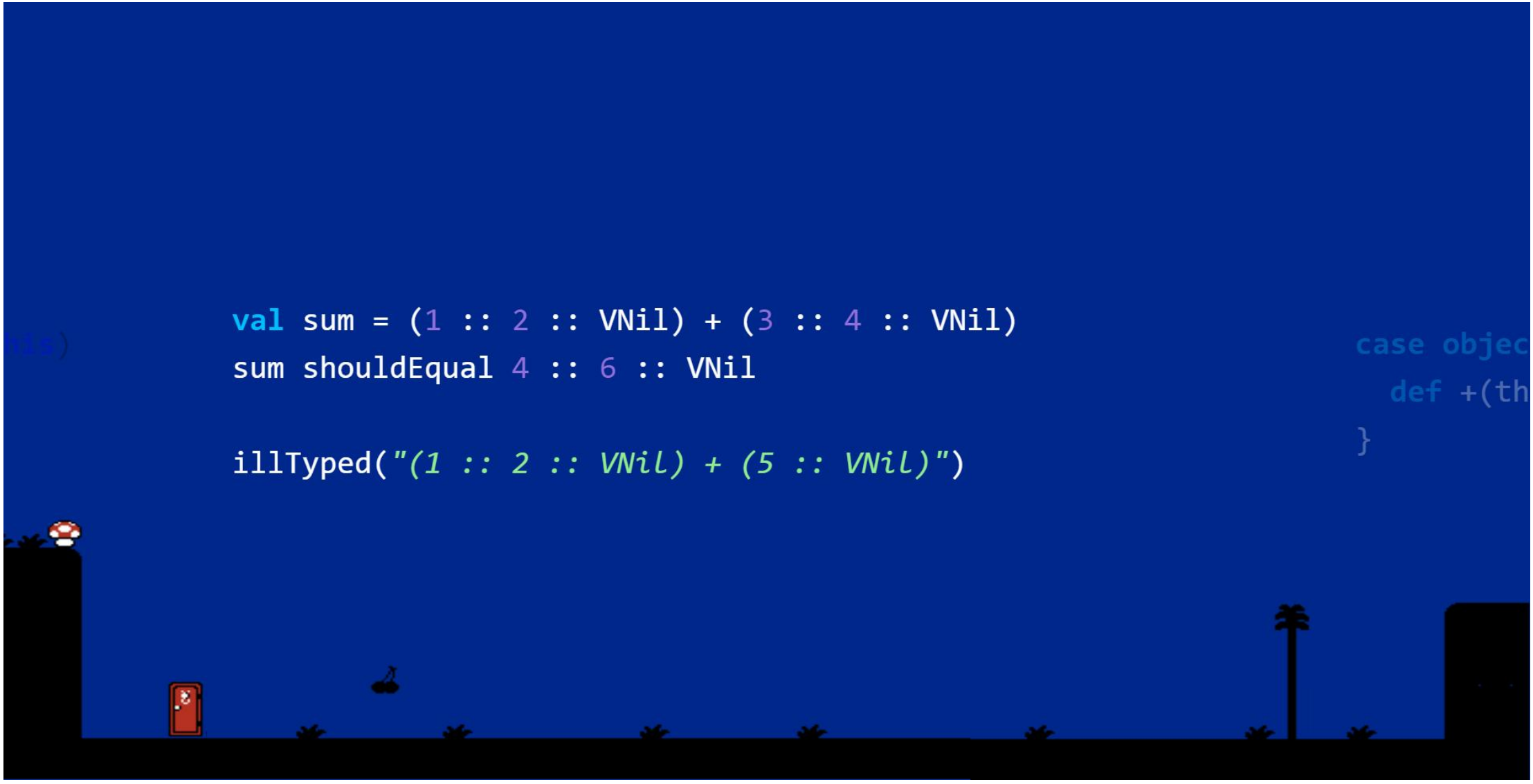
But *when* do we know the size?



Compile time! (The vectors are immutable)



We can validate our lengths without a runtime IllegalArgumentException!

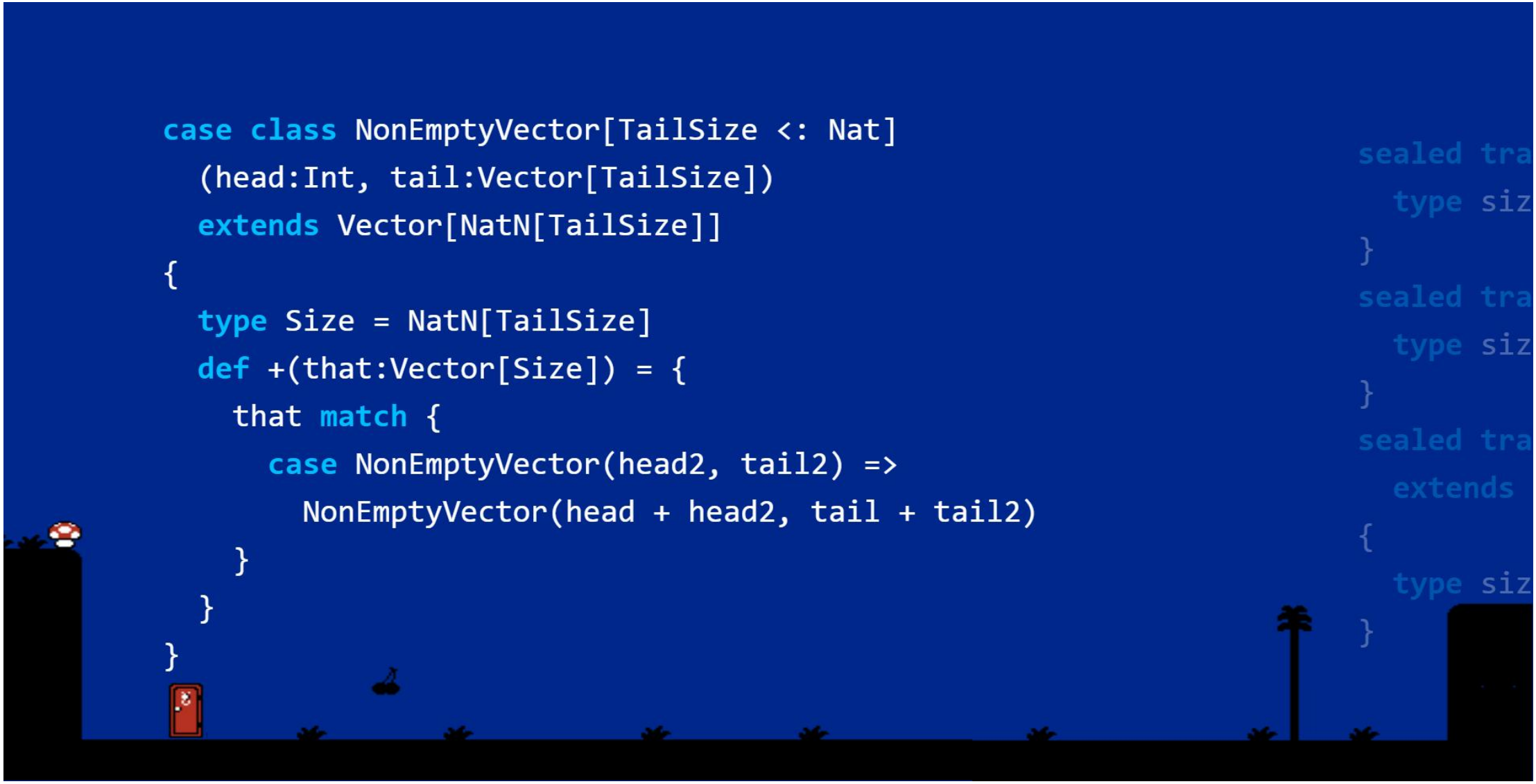

val sum = (1 :: 2 :: VNil) + (3 :: 4 :: VNil)
sum shouldEqual 4 :: 6 :: VNil
illTyped("(1 :: 2 :: VNil) + (5 :: VNil)")

```
sealed trait Vector[Size <: Nat] {
  def ::(head:Int):Vector[NatN[Size]] =
    NonEmptyVector(head, this)
  def +(that:Vector[Size]):Vector[Size]
}
```

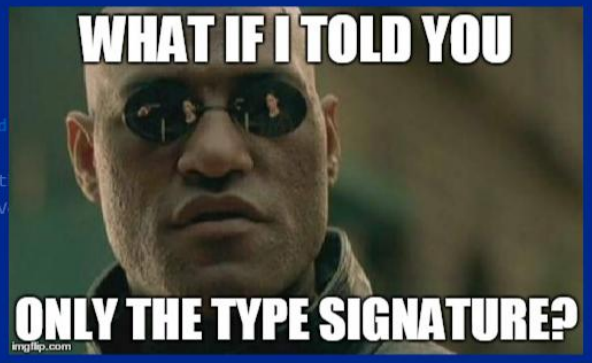
So why are we e
Thus far we've

```scala
case object VNil extends Vector[Nat0] {
  def +(that:Vector[Nat0]) = this
}
```

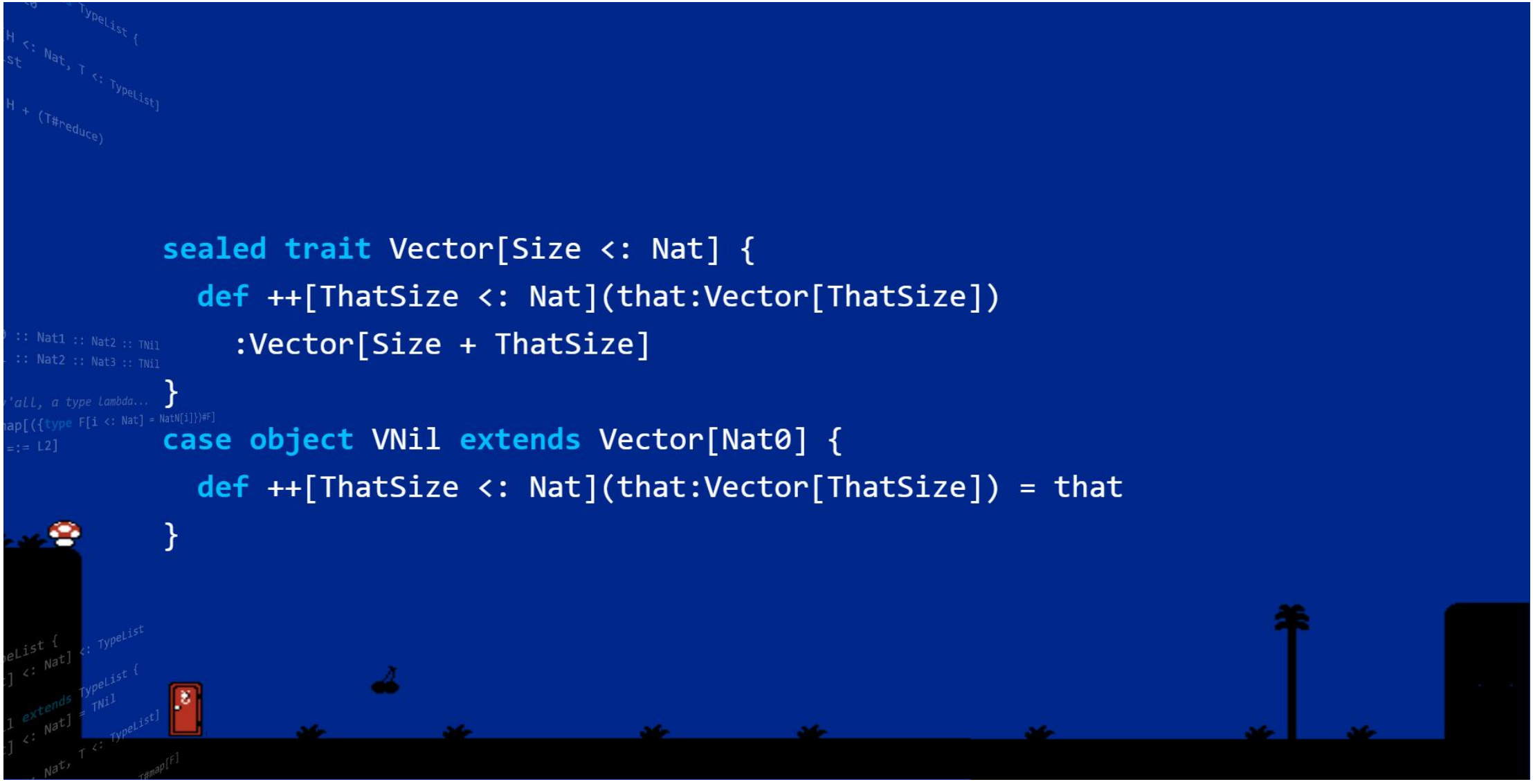
```scala
type L = N
implicitly
```


case class NonEmptyVector[TailSize <: Nat]
(head:Int, tail:Vector[TailSize])
extends Vector[NatN[TailSize]]
{
type Size = NatN[TailSize]
def +(that:Vector[Size]) = {
that match {
case NonEmptyVector(head2, tail2) =>
NonEmptyVector(head + head2, tail + tail2)
}
}
}
sealed tra
type siz
}
sealed tra
type siz
}
sealed tra
extends
{
type siz
}

```
sealed trait Vector[Size <: Nat] {
  def ++[ThatSize <: Nat](that:Vector[ThatSize])
    :Vector[Size + ThatSize]
}
case object VNil extends Vector[Nat0] {
  def ++[ThatSize <: Nat](that:Vector[ThatSize]) = that
}
```




sealed trait Vector[Size <: Nat] {
def ??[ThatSize <: Nat]
(that:Vector[ThatSize])
:Vector[Size + ThatSize]
}


sealed trait Vector[Size <: Nat] {
def ++[ThatSize <: Nat](that:Vector[ThatSize])
:Vector[Size + ThatSize]
}
case object VNil extends Vector[Nat0] {
def ++[ThatSize <: Nat](that:Vector[ThatSize]) = that
}

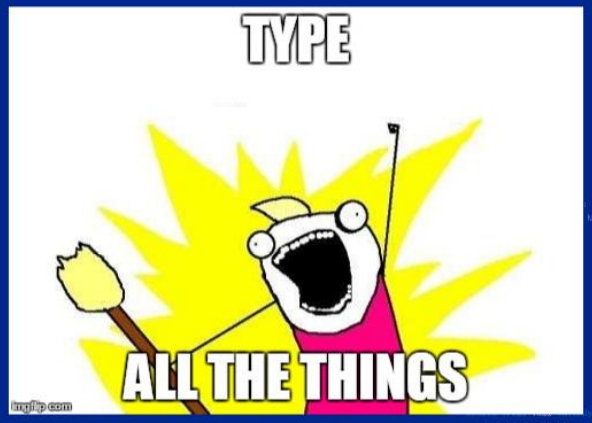

case class NonEmptyVector[TailSize <: Nat]
(head:Int, tail:Vector[TailSize])
extends Vector[NatN[TailSize]]
{
def ++[ThatSize <: Nat](that:Vector[ThatSize]) =
NonEmptyVector(head, tail ++ that)
}

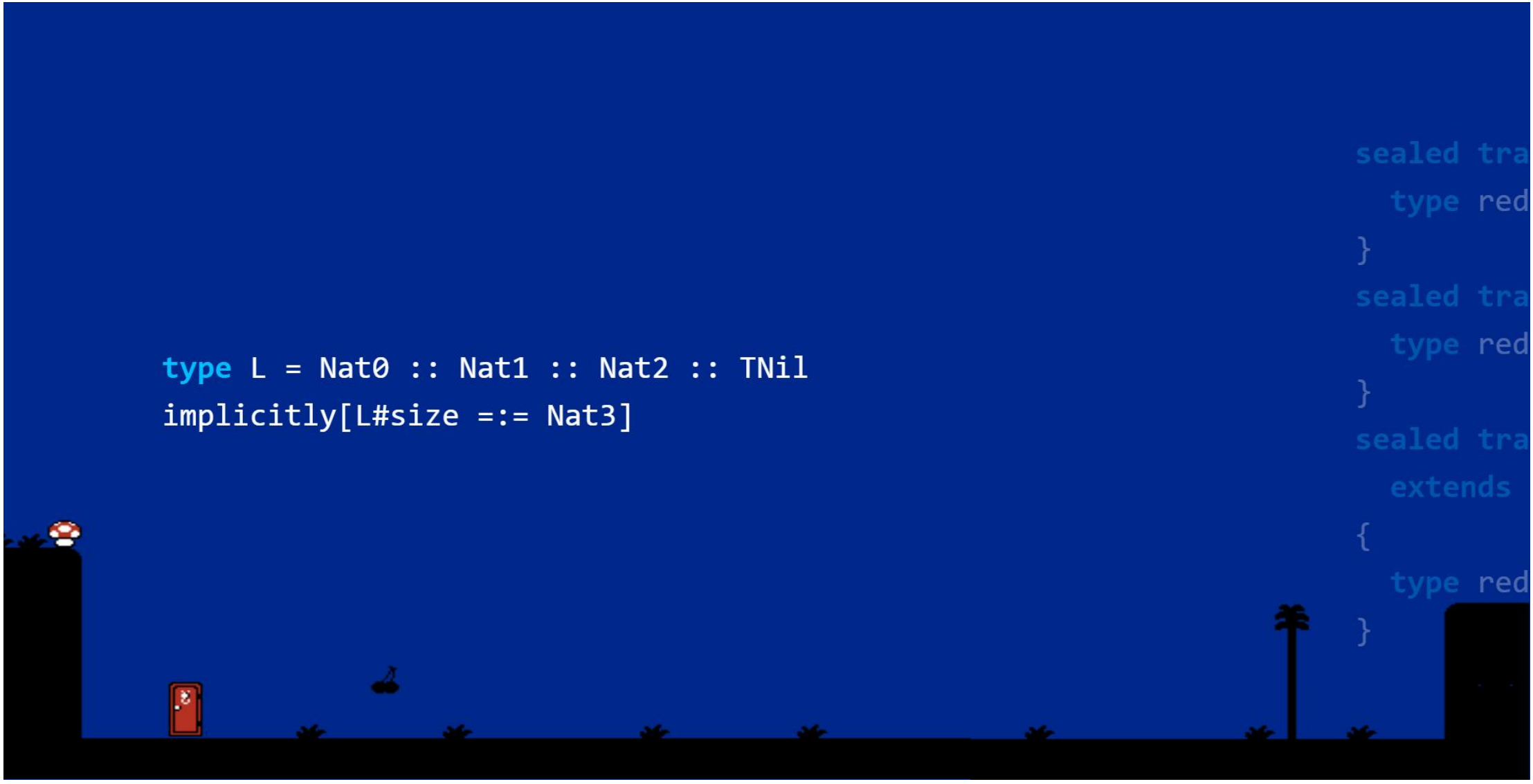

val sum =
((1 :: 2 :: VNil) ++ (3 :: VNil)) +
(4 :: 5 :: 6 :: VNil)
sum shouldEqual 5 :: 7 :: 9 :: VNil

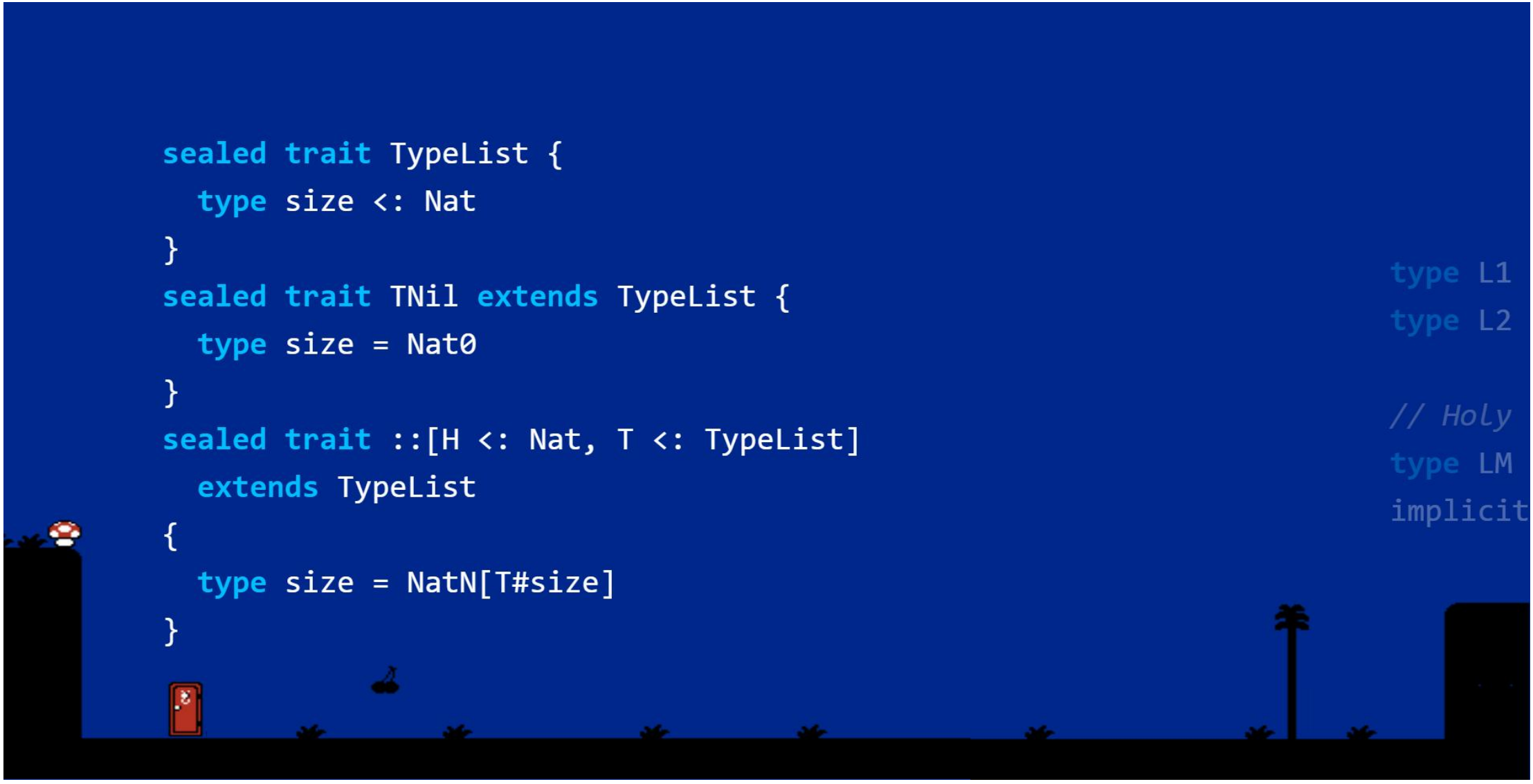
So why are we even using values at all?

Thus far we've only hard-coded anyway...

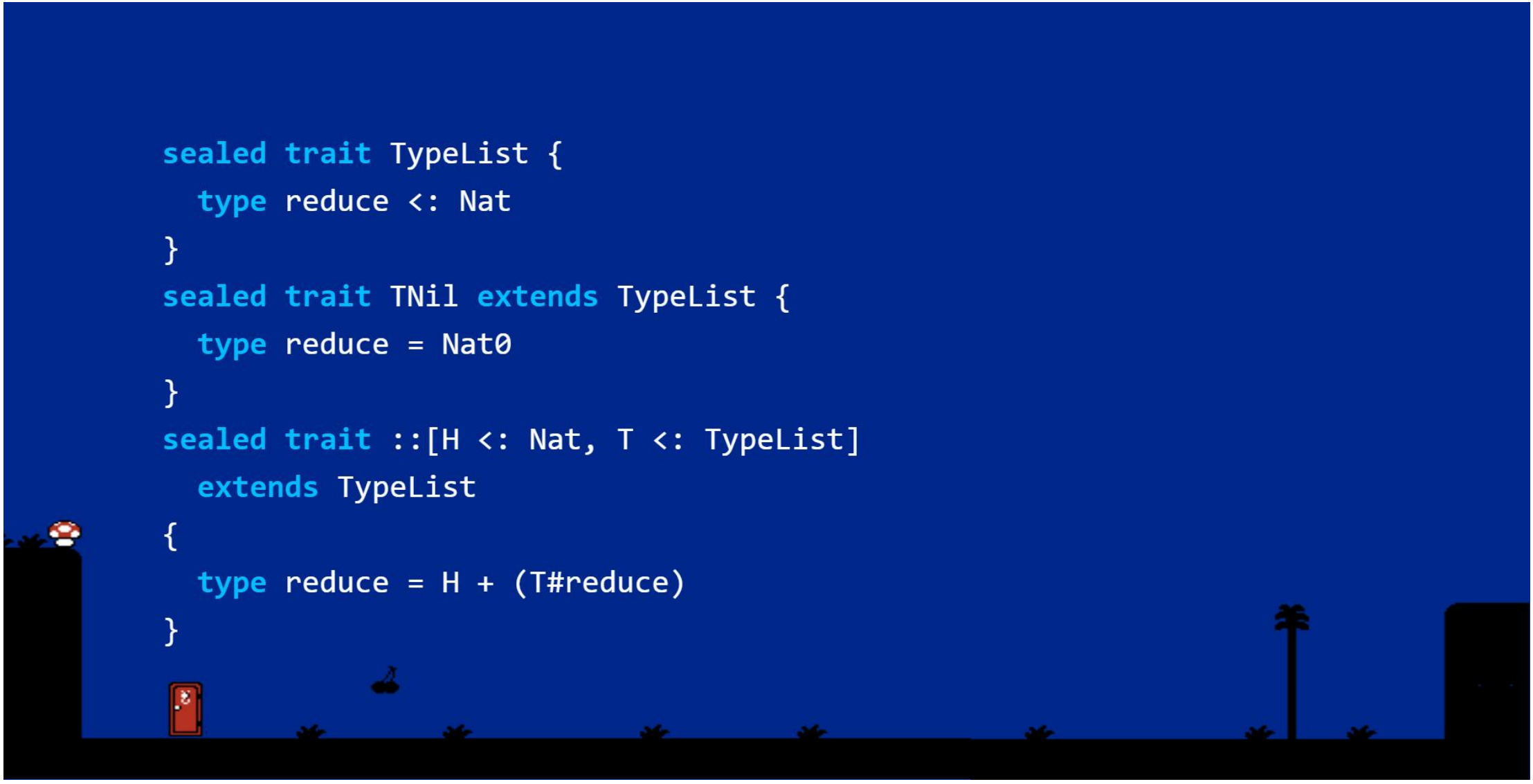
```
type L = N
implicitly
```

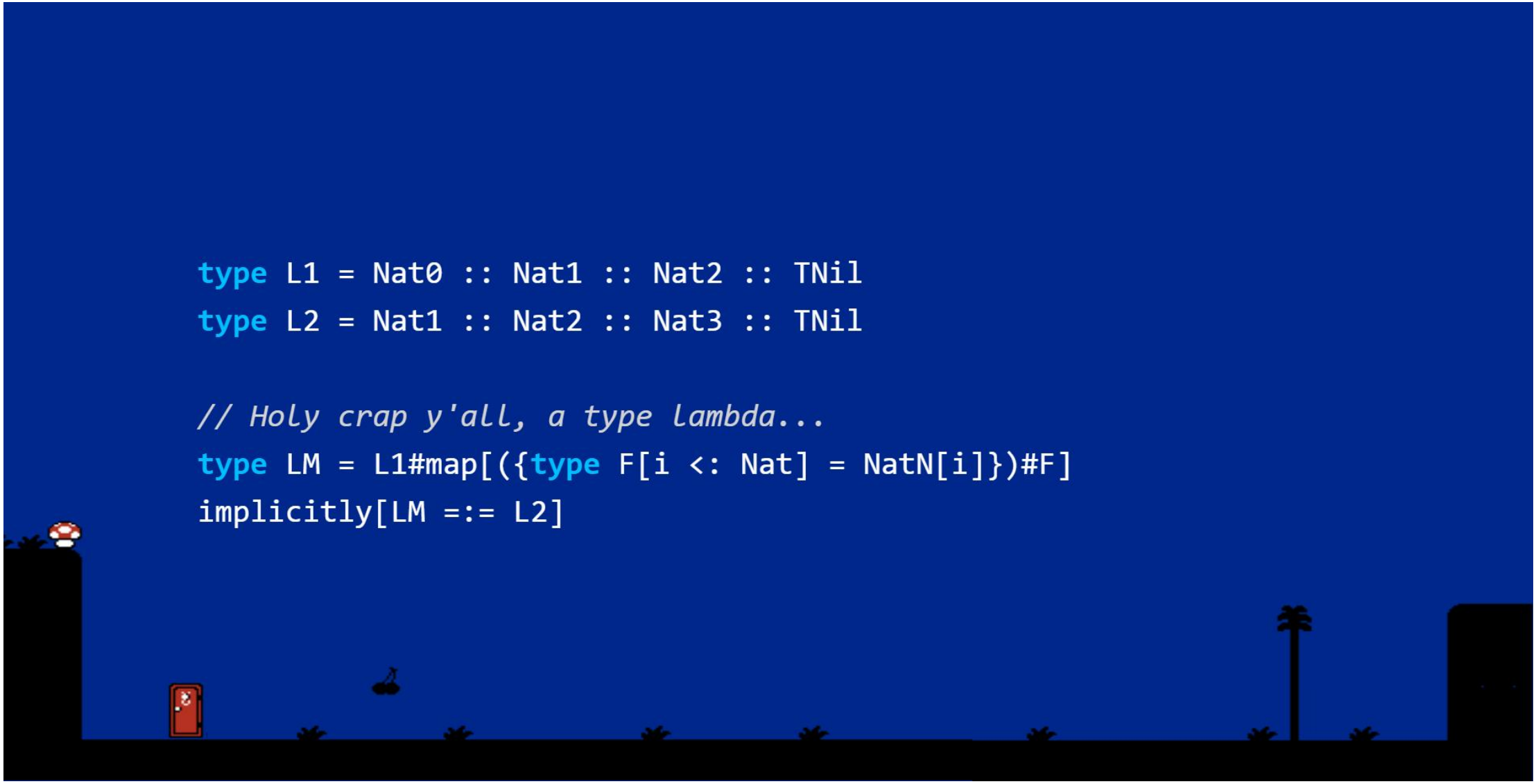

TYPE
ALL THE THINGS
imgflip.com

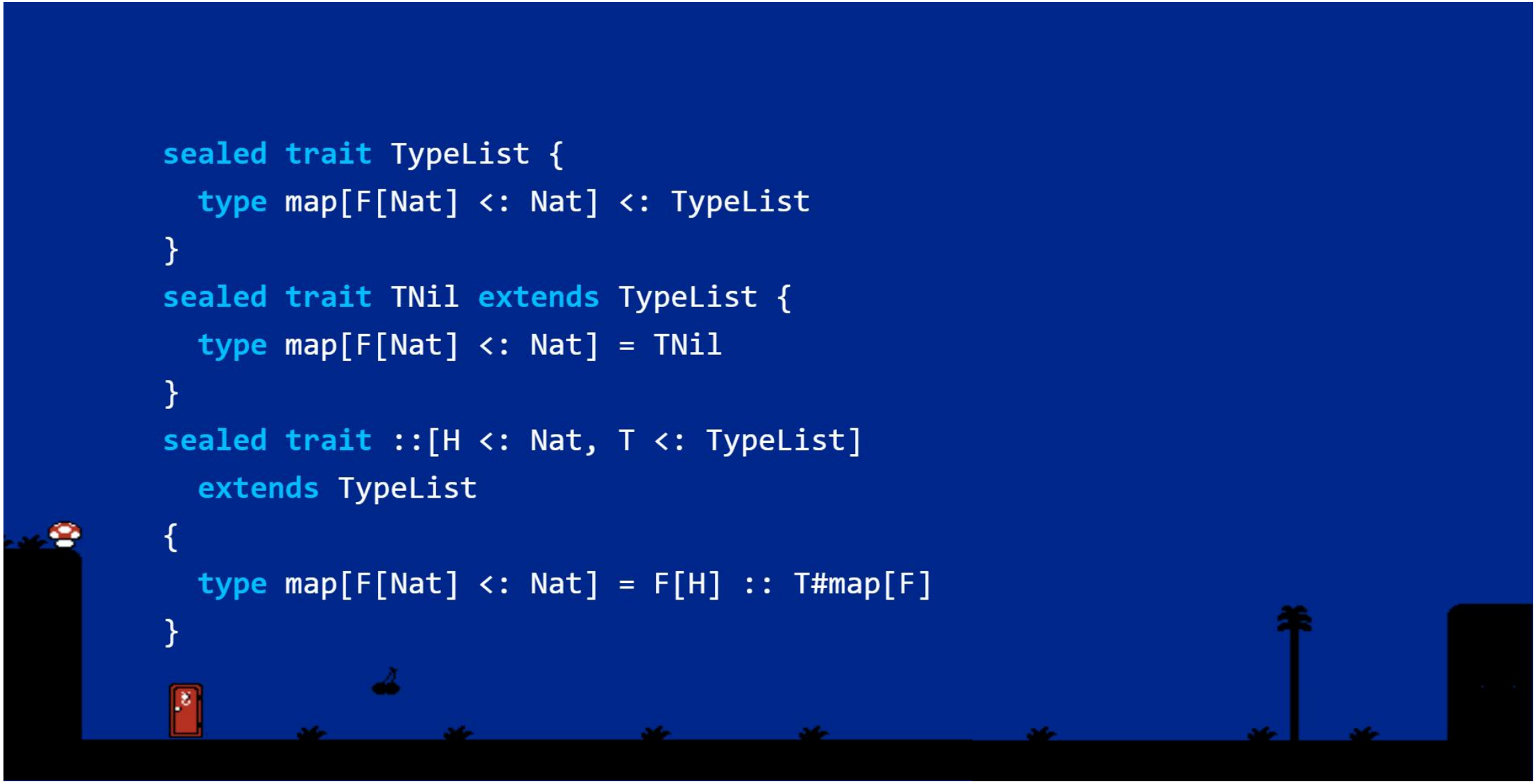

type L = Nat0 :: Nat1 :: Nat2 :: TNil
implicitly[L#size =:= Nat3]
sealed tra
type red
}
sealed tra
type red
}
sealed tra
extends
{
type red
}


sealed trait TypeList {
type size <: Nat
}
sealed trait TNil extends TypeList {
type size = Nat0
}
sealed trait ::[H <: Nat, T <: TypeList]
extends TypeList
{
type size = NatN[T#size]
}
type L1
type L2
// Holy
type LM
implicit

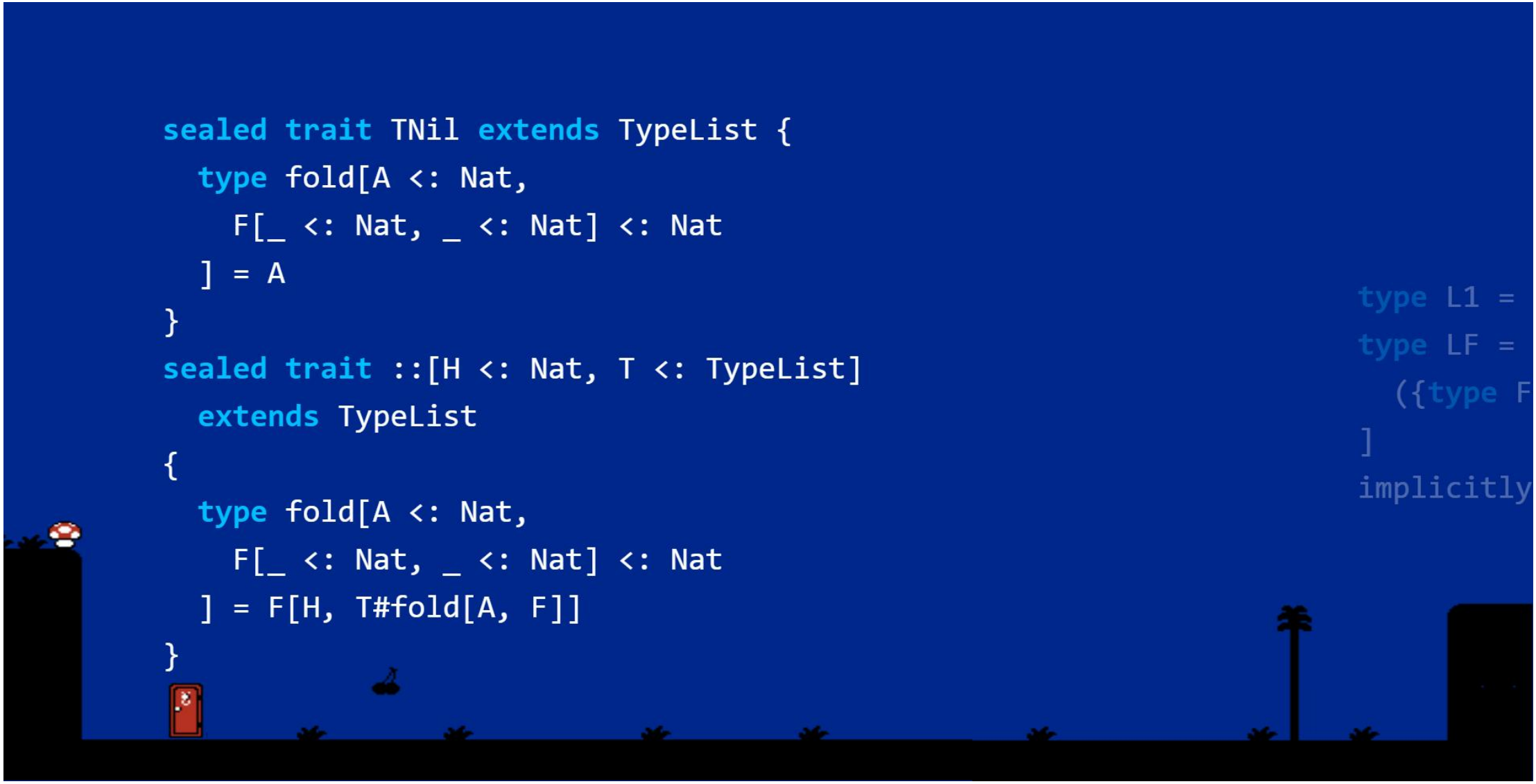
```
type L = Nat1 :: Nat2 :: Nat3 :: TNil
implicitly[L#reduce =:= Nat6]
```


sealed trait TypeList {
type reduce <: Nat
}
sealed trait TNil extends TypeList {
type reduce = Nat0
}
sealed trait ::[H <: Nat, T <: TypeList]
extends TypeList
{
type reduce = H + (T#reduce)
}

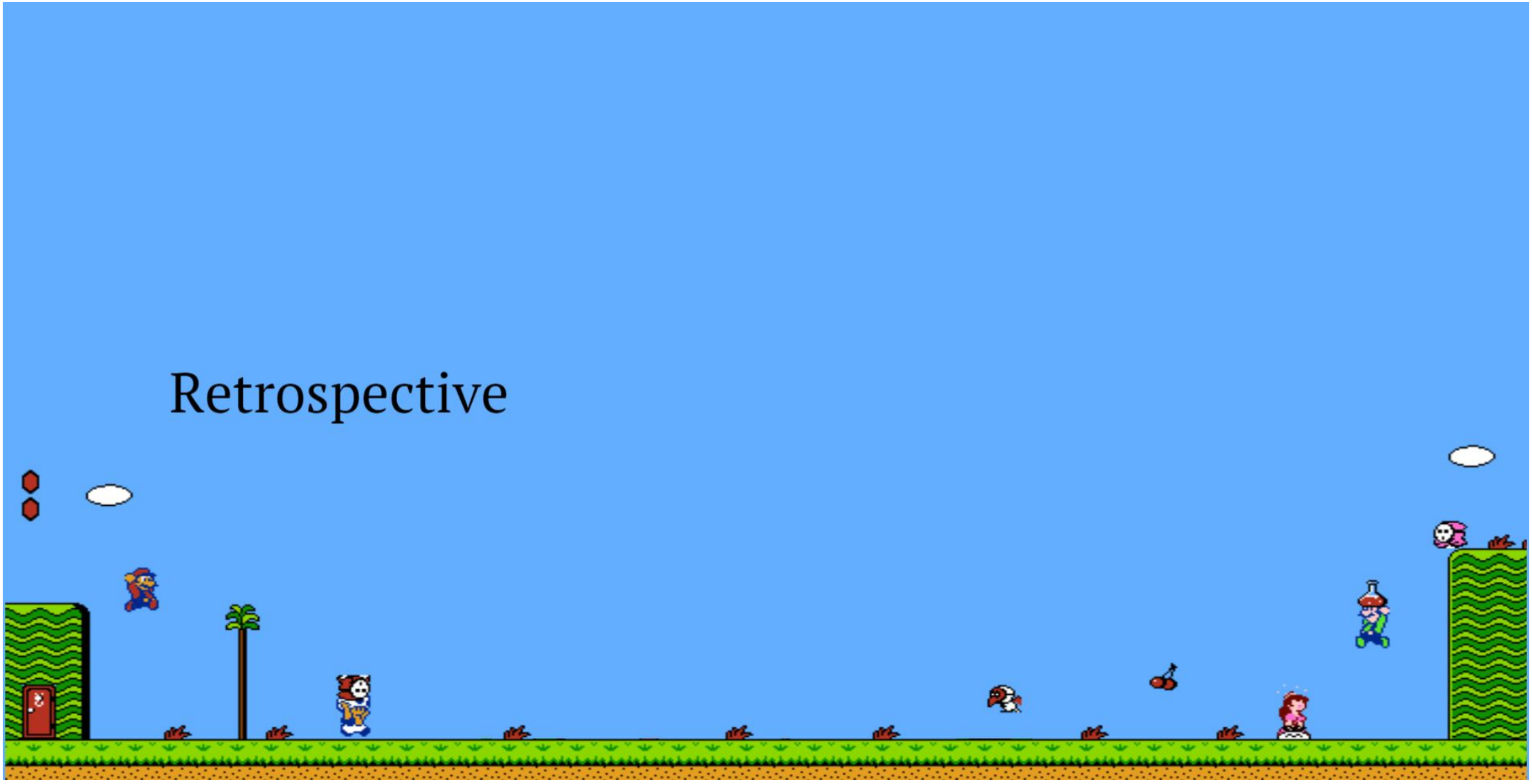

type L1 = Nat0 :: Nat1 :: Nat2 :: TNil
type L2 = Nat1 :: Nat2 :: Nat3 :: TNil
// Holy crap y'all, a type lambda...
type LM = L1#map[({type F[i <: Nat] = NatN[i]})#F]
implicitly[LM =:= L2]


sealed trait TypeList {
type map[F[Nat] <: Nat] <: TypeList
}
sealed trait TNil extends TypeList {
type map[F[Nat] <: Nat] = TNil
}
sealed trait ::[H <: Nat, T <: TypeList]
extends TypeList
{
type map[F[Nat] <: Nat] = F[H] :: T#map[F]
}


type L1 = Nat2 :: Nat1 :: Nat0 :: Nat1 :: TNil
type LF = L1#fold[Nat0,
({type F[A <: Nat, B <: Nat] = A + B})#F
]
implicitly[LF =:= Nat4]
sealed tra
type map
}
sealed tra
type map
}
sealed tra
extends
{
type map
}


sealed trait TNil extends TypeList {
type fold[A <: Nat,
F[_ <: Nat, _ <: Nat] <: Nat
] = A
}
sealed trait ::[H <: Nat, T <: TypeList]
extends TypeList
{
type fold[A <: Nat,
F[_ <: Nat, _ <: Nat] <: Nat
] = F[H, T#fold[A, F]]
}
type L1 =
type LF =
({type F
]
implicitly

Retrospective



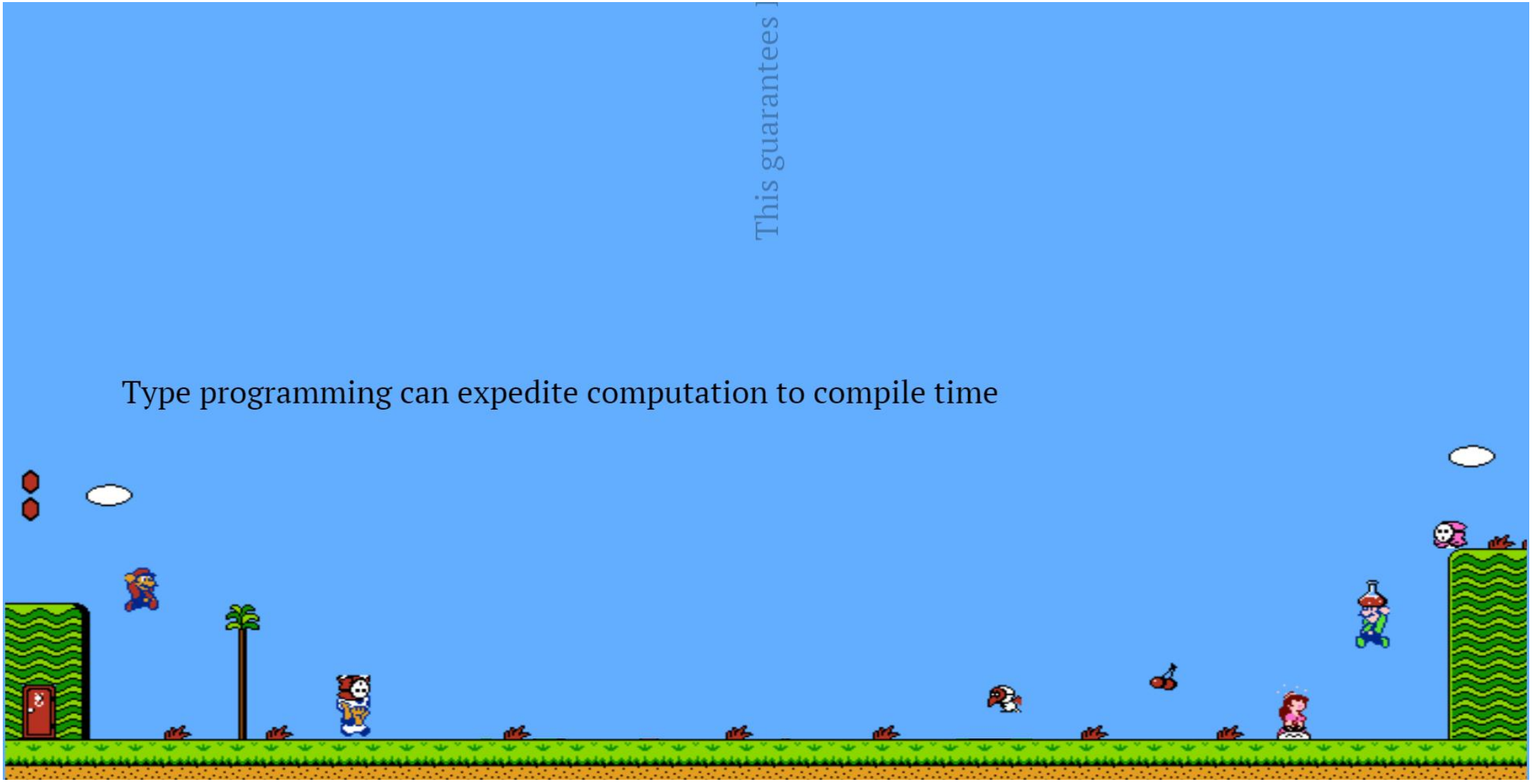
Retrosp

Types are not just the shape of my objects

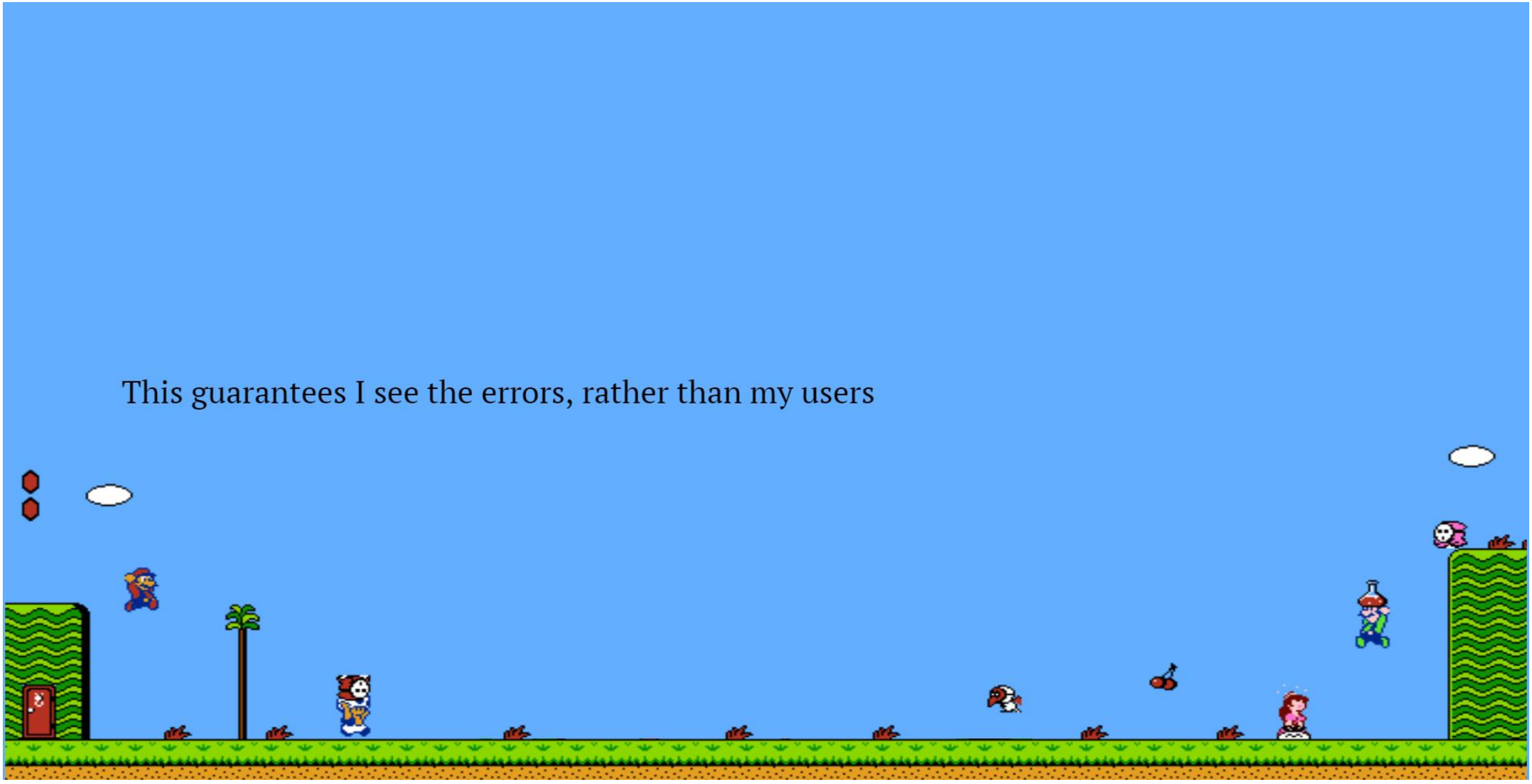
Type programming

This guarantees

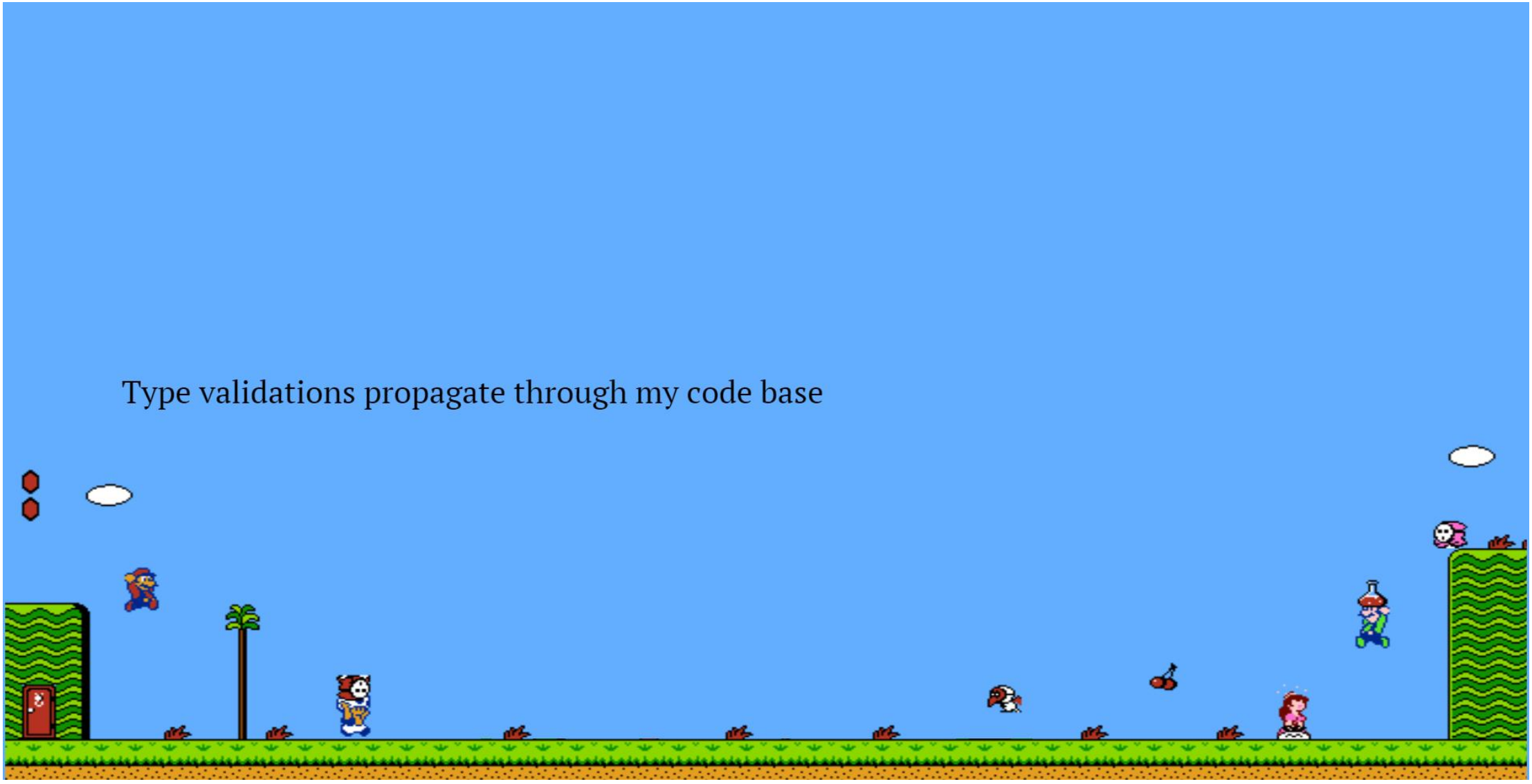
Type programming can expedite computation to compile time



This guarantees I see the errors, rather than my users

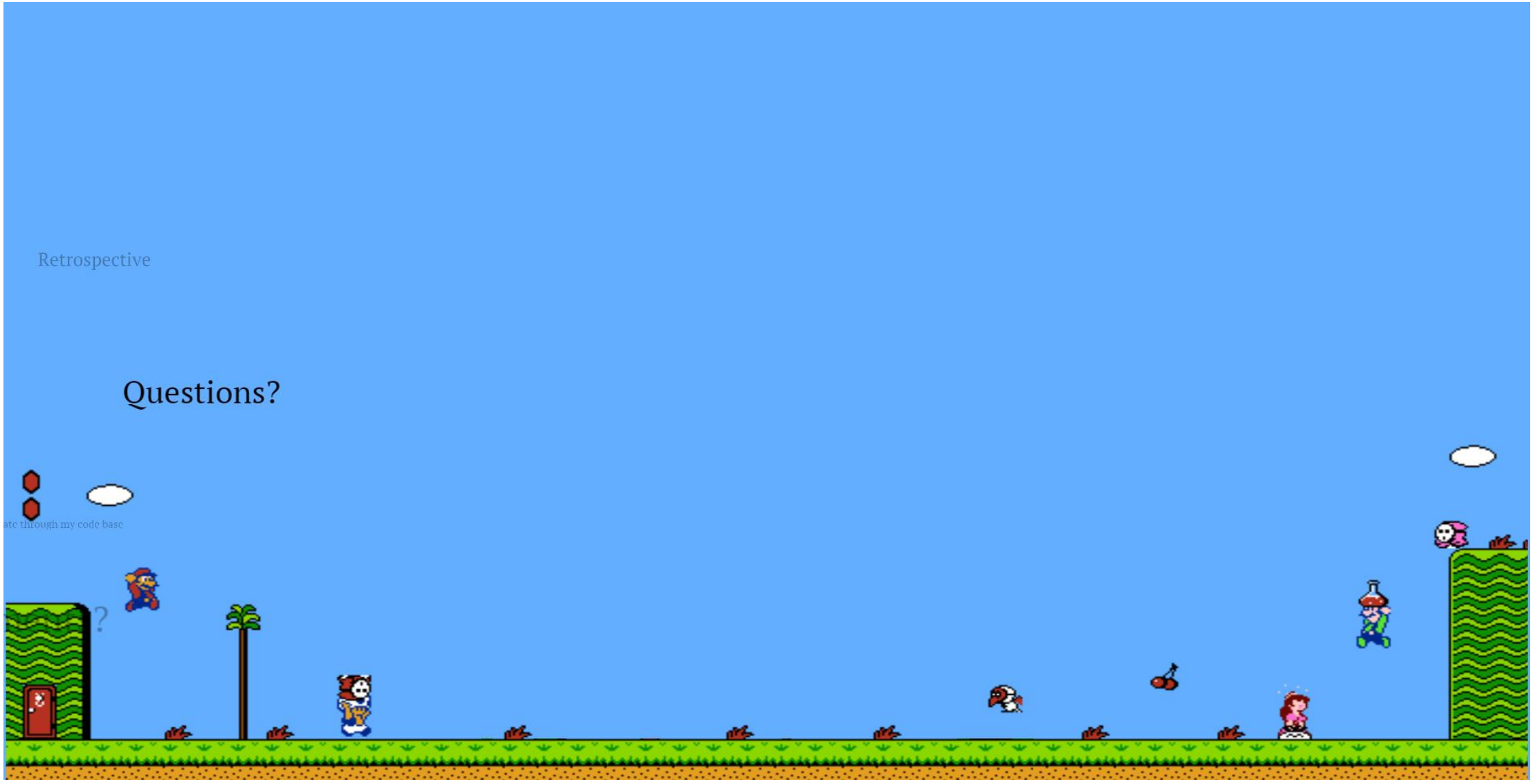


Type validations propagate through my code base

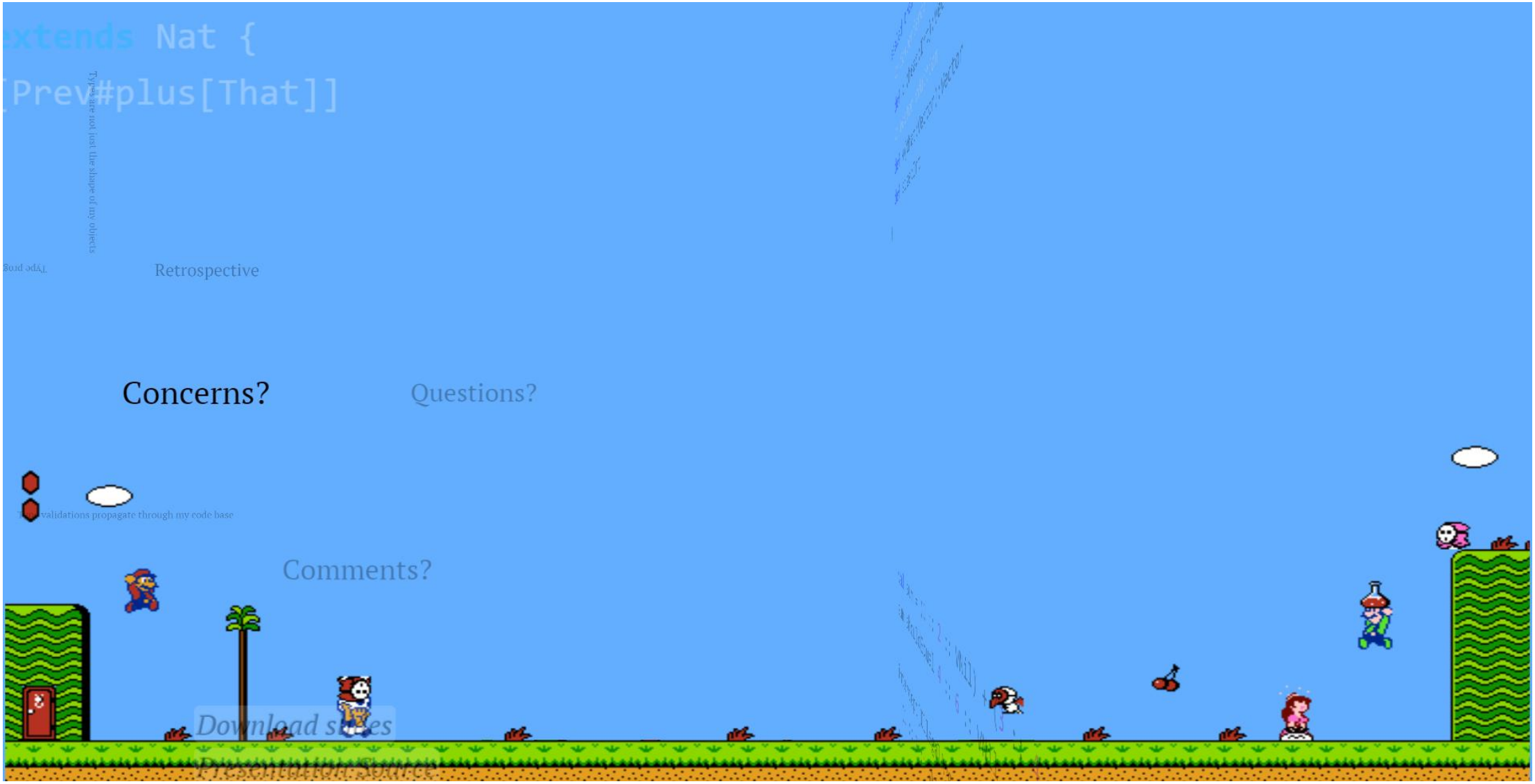


Retrospective

Questions?




erns?
Retrospective
Questions?
Comments?
ations propagate through my code base
Download slides
Presentation Source


extends Nat {
[Prev#plus[That]]
Retrospective
Concerns?
Questions?
Comments?
Download slides
Presentation Source

Questions?

Retrospective

Comments?

Download slides

Presentation Source