Tree Sitting will continue until Scala highlighting improves

Anton Sviridov, Scala Days 2025

Tree Sitting will continue until Scala highlighting improves

NOW WITH AI

Anton Sviridov, Scala Days 2025

About me

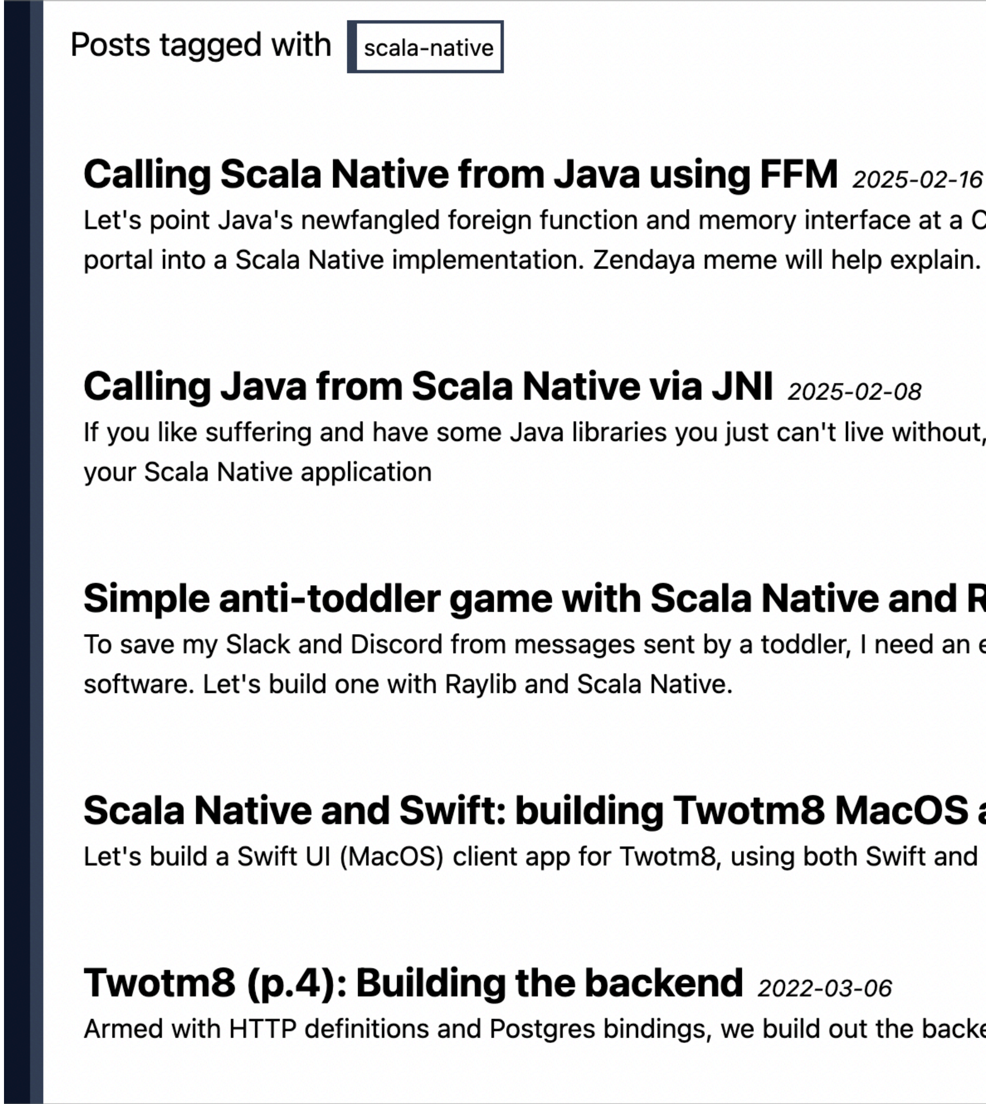
https://blog.indoorvivants.com

- Worked with Scala since 2014
- Advocated for Scala *everywhere* for years
- Maintainer and contributor to Scala Native and Scala.js ecosystem, tools, and libraries
- Run a Scala-centric blog

Github: keynmol

Twitter: velvetbaldmime

Bluesky: @indoorvivants.com



Slides and code

Code

https://github.com/keynmol/scala-treesitter-highlighting

Slides

https://slides.indoorvivants.com/scala-days-2025

Motivation

There are only 4 big issues our industry should be concerned with.

Motivation

There are only 4 big issues our industry should be concerned with.

Security

Motivation

There are only 4 big issues our industry should be concerned with.

Security

Scalability

Motivation

There are only 4 big issues our industry should be concerned with.

Security
Scalability
Maintainbility

Motivation

There are only 4 big issues our industry should be concerned with.

Security

Scalability

Maintainbility

Making sure code snippets are pretty and colourful

Motivation

- This didn’t start as a talk
- An in-browser webapp with Scala.js first
- It worked well and gave me inspiration to explore other platforms
- Native first, then JVM
- It was a journey, one I want to take you on

What’s in it for you, the Scala developer?

- Learn a bit about Tree Sitter
- See Scala on all three platforms
- Learn a useful pattern for building interfaces in Scala 3
- Nourish or uncover a deeply destructive obsession with syntax highlighting

Syntax highlighting

- Structure of the program through the usage of colours and text decorators
- Performed without typechecking or involving the compiler
- Speeds up comprehension by involving the subconscious

Syntax highlighting

Even in a particularly egregious case, compare

```
if text.trim.nonEmpty then
  positionedTokens += PositionedToken(
    text,
    x = lineWidth,
    y = height,
    color = color
  )
  lineWidth += ((!extents).width).max((!baseExtents).width * text.length)
else lineWidth += text.count(_.isWhitespace) * (!baseExtents).width
```

with

```
if text.trim.nonEmpty then
  positionedTokens += PositionedToken(
    text,
    x = lineWidth,
    y = height,
    color = color
  )
  lineWidth += ((!extents).width).max((!baseExtents).width * text.length)
else lineWidth += text.count(_.isWhitespace) * (!baseExtents).width
```

Highlighting with regular expressions

- Most common approach
- Most commonly using Highlight.js
- Regexes are used
- Fast, easy to write, small code in the browser
- Struggles with ambiguity, limited context, "soft" keywords, etc.

It served (and will continue to serve) us well enough – but in some cases we can do better, much better.

Tree Sitter

https://tree-sitter.github.io/tree-sitter/

- A comprehensive parsing system
  - Authoring grammars in JS
  - Queries for syntax tree tagging and processing
  - CLI tools and templates
  - Testing infrastructure
- Lots of languages have high fidelity grammars
- Used on Github for syntax highlighting
- Used by companies doing static analysis

Tree Sitter: in editors

Some popular editors are either built with Tree Sitter at the core, or support it natively

- Neovim
- Zed
- Helix
- Emacs

Tree Sitter: in editors

Some popular editors are either built with Tree Sitter at the core, or support it natively.

- Neovim
- Zed
- Helix
- ~~Emacs~~ Sorry, I meant **popular**

Tree Sitter: grammar

Grammars are defined using a JavaScript DSL.

```
enum_definition: $ =>
  seq(
    repeat($.annotation),
    "enum",
    $._class_constructor,
    field("extend", optional($.extends_clause)),
    field("derive", optional($.derives_clause)),
    field("body", $.enum_body),
  ),
```

Tree Sitter: queries

Queries use a Scheme-like language to deeply match and label particular syntax nodes:

```
(call_expression
  function: (operator_identifier) @function.call)
```

Tree Sitter API allows extracting all the matched nodes along with their labels.

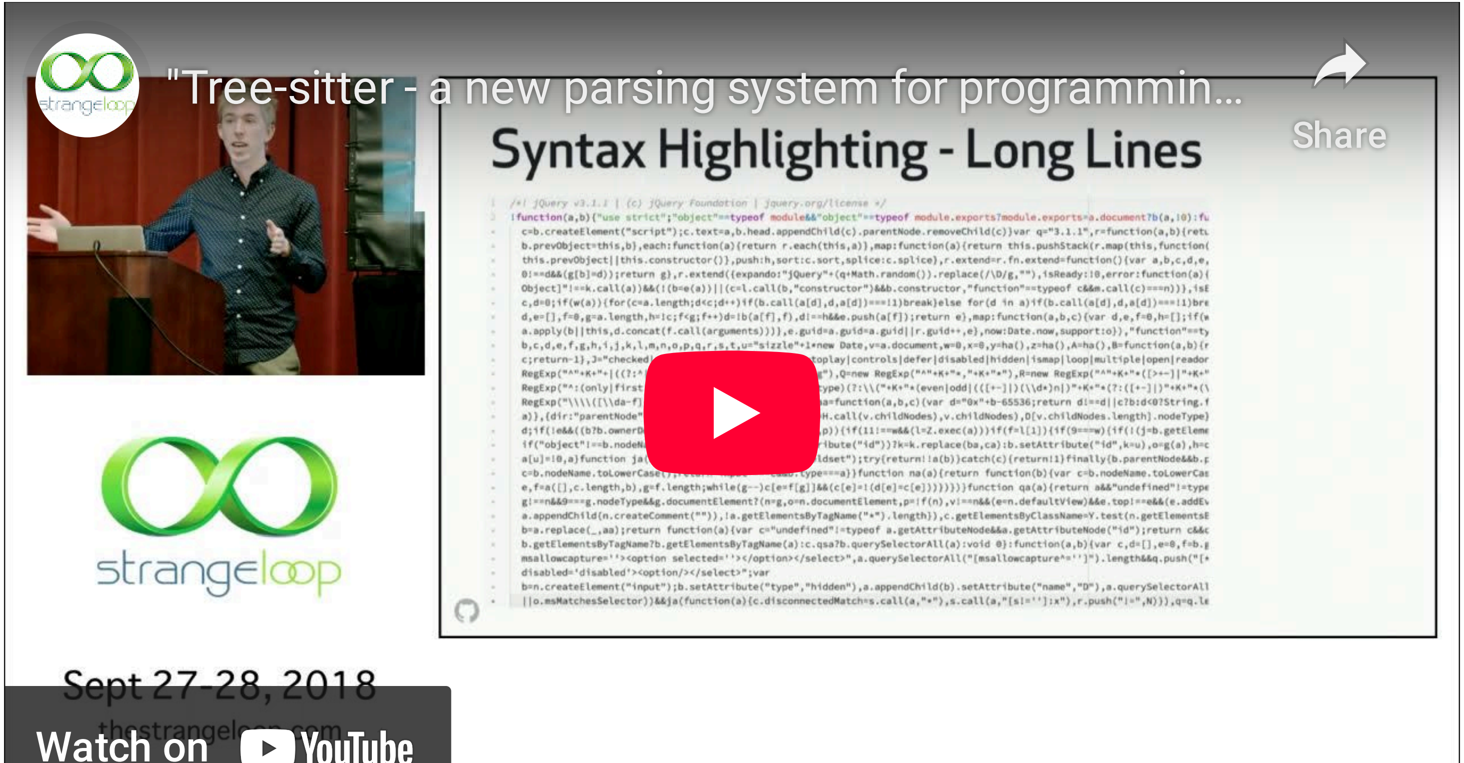
Labels have no meaning in Tree Sitter itself – different applications use different label sets and interpret them as they wish.

Tree Sitter: interface

- Grammar definition is verified and then a **gigantic** C file can be generated from it. It uses C runtime to parse text.
- Generated C parser can be compiled to WASM, usable on the Web
- There are some first-class bindings for popular languages

Tree sitter: learn more

Tree Sitter has a lot more to offer. Watch this video from Max Brunsfield.



The Project

Here's the state of it as of this presentation:

- ✅ Scala.js frontend application
- ✅ Scala Native CLI
- 🚧 Scala JVM backend

Scala code highlighter

This highlighter uses the Tree Sitter parser for Scala, compiled to WASM. It is much more accurate than any regex-based engines such as Highlight.js or Textmate grammars

Github | Author | Tree Sitter Scala grammar

Scala code:

```scala
  basicRequest
    .get(uri"https://wttr.in/London?format=4")
    .send(backend)
    .body
    .right
    .get

@main def hello =
  val mcp = MCPBuilder
    .create()
    .handleRequest(initialize): req =>
      InitializeResult(
        capabilities =
          ServerCapabilities(tools = Some(ServerCapabilities.Tools())),
        protocolVersion = req.params.protocolVersion,
        serverInfo = Implementation("scala-mcp", "0.0.1")
```

Theme: VS Code (light)

Tree Sitter highlighting:

```scala
package sample

import mcp.*
import upickle.default.*

import sttp.client4.*
import sttp.client4.upicklejson.default.*

val backend = DefaultSyncBackend()

def weather(city: String) =
  basicRequest
    .get(uri"https://wttr.in/London?format=4")
    .send(backend)
    .body
    .right
    .get
```

The Project: Scala.js frontend

High level overview:

- Using Vite as a bundler and build tool
- Tree Sitter has special JavaScript bindings specifically designed to work with WASM parsers
- Scala.js and Laminar for the frontend interactivity

The Project: loading WASM parser

```
import TreeSitter from "web-tree-sitter";
import init from "web-tree-sitter/tree-sitter.wasm?init&url";
import initScala from "/tree-sitter-scala.wasm?init&url";

let parser = await (async () => {
  await TreeSitter.init({
    locateFile(scriptName, scriptDirectory) {
      return init;
    },
  });
  const parser = new TreeSitter();
  const Lang = await TreeSitter.Language.load(initScala);
  parser.setLanguage(Lang);
  return parser;
})();

export default parser;
```

The Project: loading WASM parser

```
import TreeSitter from "web-tree-sitter";
import init from "web-tree-sitter/tree-sitter.wasm?init&url";
import initScala from "/tree-sitter-scala.wasm?init&url";

let parser = await (async () => {
  await TreeSitter.init({
    locateFile(scriptName, scriptDirectory) {
      return init;
    },
  });
  const parser = new TreeSitter();
  const Lang = await TreeSitter.Language.load(initScala);
  parser.setLanguage(Lang);
  return parser;
})();

export default parser;
```

The Project: loading WASM parser

```
import TreeSitter from "web-tree-sitter";
import init from "web-tree-sitter/tree-sitter.wasm?init&url";
import initScala from "/tree-sitter-scala.wasm?init&url";

let parser = await (async () => {
  await TreeSitter.init({
    locateFile(scriptName, scriptDirectory) {
      return init;
    },
  });
  const parser = new TreeSitter();
  const Lang = await TreeSitter.Language.load(initScala);
  parser.setLanguage(Lang);
  return parser;
})();

export default parser;
```

The Project: loading WASM parser

```
import TreeSitter from "web-tree-sitter";
import init from "web-tree-sitter/tree-sitter.wasm?init&url";
import initScala from "/tree-sitter-scala.wasm?init&url";

let parser = await (async () => {
  await TreeSitter.init({
    locateFile(scriptName, scriptDirectory) {
      return init;
    },
  });
  const parser = new TreeSitter();
  const Lang = await TreeSitter.Language.load(initScala);
  parser.setLanguage(Lang);
  return parser;
})();

export default parser;
```

The Project: loading WASM parser

```
import TreeSitter from "web-tree-sitter";
import init from "web-tree-sitter/tree-sitter.wasm?init&url";
import initScala from "/tree-sitter-scala.wasm?init&url";

let parser = await (async () => {
  await TreeSitter.init({
    locateFile(scriptName, scriptDirectory) {
      return init;
    },
  });
  const parser = new TreeSitter();
  const Lang = await TreeSitter.Language.load(initScala);
  parser.setLanguage(Lang);
  return parser;
})();

export default parser;
```

The Project: binding to web Tree Sitter

Now that we have the parser on hand, how do we work with it in Scala?

We take a look at the exposed API, and come up with this:

```scala
@js.native
@JSImport("/tree-sitter.js", JSImport.Default)
private object Parser extends js.Any:
  def parse(path: String): Tree = js.native

  def getLanguage(): Language = js.native

  @js.native
  trait Language extends js.Any:
    def query(source: String): Query = js.native

  @js.native
  trait Query extends js.Any:
    def matches(node: Node): Arr[Match] = js.native

  // ...
```

I can already see that Native and JVM bindings won’t look anything like that! **Time to step back.**

Tree Sitter interface

Our goal: **the highlighting logic will be implemented in a syntactically identical way across platforms**

But platforms are different! For example, a Tree Sitter match:

Scala.js

```scala
@js.native
trait Match extends js.Any:
  val name: String = js.native
  val captures: Arr[Capture] = js.native
```

Scala Native

```scala
opaque type TSQueryMatch = CStruct4[uint32_t, uint16_t, uint16_t, Ptr[TSQueryCapture]]
// and a bunch of static methods
```

Scala JVM

```java
public record QueryMatch(@Unsigned int patternIndex, List<QueryCapture> captures) {}
```

No shared traits, no simple shared representation, different memory semantics.

Tree Sitter interface

If we squint, we can uncover the intrinsic Tree Sitter model as such:

```
trait TreeSitterInterface:
  type Tree
  extension (t: Tree) def rootNode: Node
  def parse(source: String): Tree

  def getLanguage: Language

  // ...

  type Capture
  extension (t: Capture)
    @targetName("capture_name")
    def name(q: Query): String
    def node: Node

  type Match
  extension (t: Match)
    def captures: Iterable[Capture]
```

Using abstract extension methods dramatically simplifies the usage.

Tree Sitter interface

If we squint, we can uncover the intrinsic Tree Sitter model as such:

```scala
trait TreeSitterInterface:
  type Tree
  extension (t: Tree) def rootNode: Node
  def parse(source: String): Tree

  def getLanguage: Language

  // ...

  type Capture
  extension (t: Capture)
    @targetName("capture_name")
    def name(q: Query): String
    def node: Node

  type Match
  extension (t: Match)
    def captures: Iterable[Capture]
```

Using abstract extension methods dramatically simplifies the usage.

Tree Sitter interface

If we squint, we can uncover the intrinsic Tree Sitter model as such:

```
trait TreeSitterInterface:
  type Tree
  extension (t: Tree) def rootNode: Node
  def parse(source: String): Tree

  def getLanguage: Language

  // ...

  type Capture
  extension (t: Capture)
    @targetName("capture_name")
    def name(q: Query): String
    def node: Node

  type Match
  extension (t: Match)
    def captures: Iterable[Capture]
```

Using abstract extension methods dramatically simplifies the usage.

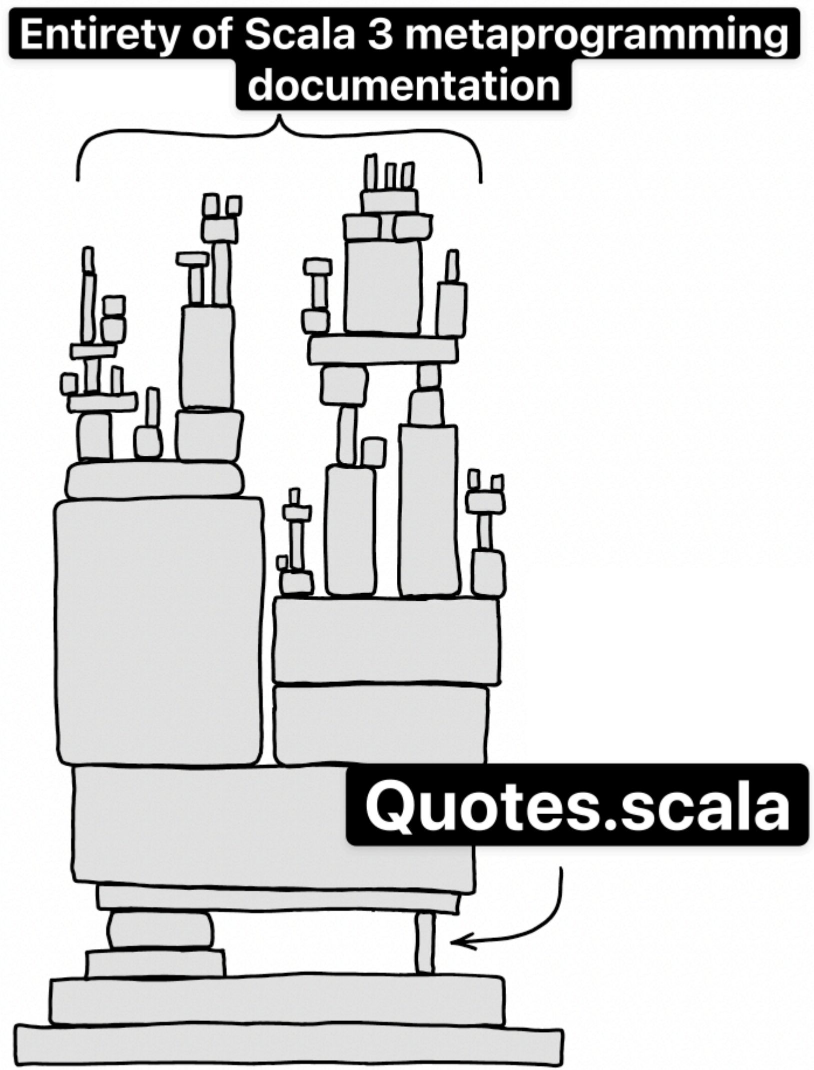
Tree Sitter interface

If we squint, we can uncover the intrinsic Tree Sitter model as such:

```
trait TreeSitterInterface:
  type Tree
  extension (t: Tree) def rootNode: Node
  def parse(source: String): Tree

  def getLanguage: Language

  // ...

  type Capture
  extension (t: Capture)
    @targetName("capture_name")
    def name(q: Query): String
    def node: Node

  type Match
  extension (t: Match)
    def captures: Iterable[Capture]
```

Using abstract extension methods dramatically simplifies the usage.

Does this look familiar to anyone?


Entirety of Scala 3 metaprogramming documentation
Quotes.scala

Tree Sitter interface: Scala.js

Here's a sample of concrete implementation:

```scala
class TreeSitter(p: Parser.type) extends TreeSitterInterface:
  // ...
  override opaque type Capture = p.Capture

  extension (t: Capture)
    @annotation.targetName("capture_name")
    override inline def name(q: Query): String = t.name
    override inline def node = t.node
  // ...
```

We use opaque types and inline extension methods to hide the representation semantics of the underlying Tree Sitter API.

Tree Sitter interface: generic usage

With this, we can write generic cross-platform algorithms:

```scala
class HighlightTokenizer[TS <: TreeSitterInterface & Singleton](
    source: String,
    highlightQueries: String,
    treesitter: TS
):
  private lazy val tree = treesitter.parse(source)
  private lazy val lang: treesitter.Language = treesitter.getLanguage
  private lazy val query = lang.query(highlightQueries)
  private lazy val matches: Iterable[treesitter.Match] =
    query.matches(tree.rootNode)
  // ...
```

The values are dependently typed based on the instance of `TreeSitterInterface` – you need to have that in scope for types to align.

Tree Sitter interface: spiralling out of control

- With this generic Tree Sitter interface we can implement highlighting logic in a platform-agnostic way
- Now that we have this outrageous power, what can we do with it?
- How about a Scala Native CLI that generates PNG images?

Tree Sitter interface: Scala Native

On Native, memory management is getting in the way. Thankfully we've adjusted our generic interface to account for that.

Native interface was the trickiest to get right.

```scala
class TreeSitter(parser: Ptr[TSParser], language: Ptr[TSLanguage])(using
    z: Zone
) extends TreeSitterInterface:
  // ...
  override opaque type Capture = Ptr[TSQueryCapture]
  // ...
  extension (t: Capture)
  @annotation.targetName("capture_name")
  override def name(q: Query) =
    val length = stackalloc[UInt]()
    val str = ts_query_capture_name_for_id(q, t.DEREF.index, length)
    val strZero = stackalloc[CChar](length.DEREF.toInt + 1)
    scalanative.libc.string.memcpy(strZero, str, !length)
    strZero(!length) = 0.toByte
    assert(str != null, "ts_query_capture_name_for_id returned null")
    fromCString(strZero)
  // ...
```

Tree Sitter interface: Scala Native

On Native, memory management is getting in the way. Thankfully we've adjusted our generic interface to account for that.

Native interface was the trickiest to get right.

```
class TreeSitter(parser: Ptr[TSParser], language: Ptr[TSLanguage])(using
    z: Zone
) extends TreeSitterInterface:
  // ...
  override opaque type Capture = Ptr[TSQueryCapture]
  // ...
  extension (t: Capture)
  @annotation.targetName("capture_name")
  override def name(q: Query) =
    val length = stackalloc[UInt]()
    val str = ts_query_capture_name_for_id(q, t.DEREF.index, length)
    val strZero = stackalloc[CChar](length.DEREF.toInt + 1)
    scalanative.libc.string.memcpy(strZero, str, !length)
    strZero(!length) = 0.toByte
    assert(str ≠ null, "ts_query_capture_name_for_id returned null")
    fromCString(strZero)
  // ...
```

Tree Sitter interface: Scala Native

On Native, memory management is getting in the way. Thankfully we've adjusted our generic interface to account for that.

Native interface was the trickiest to get right.

```
class TreeSitter(parser: Ptr[TSParser], language: Ptr[TSLanguage])(using
    z: Zone
) extends TreeSitterInterface:
  // ...
  override opaque type Capture = Ptr[TSQueryCapture]
  // ...
  extension (t: Capture)
  @annotation.targetName("capture_name")
  override def name(q: Query) =
    val length = stackalloc[UInt]()
    val str = ts_query_capture_name_for_id(q, t.DEREF.index, length)
    val strZero = stackalloc[CChar](length.DEREF.toInt + 1)
    scalanative.libc.string.memcpy(strZero, str, !length)
    strZero(!length) = 0.toByte
    assert(str ≠ null, "ts_query_capture_name_for_id returned null")
    fromCString(strZero)
  // ...
```

Tree Sitter interface: Scala Native

On Native, memory management is getting in the way. Thankfully we've adjusted our generic interface to account for that.

Native interface was the trickiest to get right.

```scala
class TreeSitter(parser: Ptr[TSParser], language: Ptr[TSLanguage])(using
    z: Zone
) extends TreeSitterInterface:
  // ...
  override opaque type Capture = Ptr[TSQueryCapture]
  // ...
  extension (t: Capture)
  @annotation.targetName("capture_name")
  override def name(q: Query) =
    val length = stackalloc[UInt]()
    val str = ts_query_capture_name_for_id(q, t.DEREF.index, length)
    val strZero = stackalloc[CChar](length.DEREF.toInt + 1)
    scalanative.libc.string.memcpy(strZero, str, !length)
    strZero(!length) = 0.toByte
    assert(str != null, "ts_query_capture_name_for_id returned null")
    fromCString(strZero)
  // ...
```

Tree Sitter: using from Scala Native

```
@extern
def tree_sitter_scala(): Ptr[TSLanguage] = extern

val parser = tree_sitter.all.ts_parser_new()
val lang = tree_sitter_scala()

val ts: TreeSitterInterface = TreeSitter(parser, lang)
```

Tree Sitter: using from Scala Native

```scala
@extern
def tree_sitter_scala(): Ptr[TSLanguage] = extern

val parser = tree_sitter.all.ts_parser_new()
val lang = tree_sitter_scala()

val ts: TreeSitterInterface = TreeSitter(parser, lang)
```

Tree Sitter: using from Scala Native

```scala
@extern
def tree_sitter_scala(): Ptr[TSLanguage] = extern

val parser = tree_sitter.all.ts_parser_new()
val lang = tree_sitter_scala()

val ts: TreeSitterInterface = TreeSitter(parser, lang)
```

Tree Sitter: using from Scala Native

```scala
@extern
def tree_sitter_scala(): Ptr[TSLanguage] = extern

val parser = tree_sitter.all.ts_parser_new()
val lang = tree_sitter_scala()

val ts: TreeSitterInterface = TreeSitter(parser, lang)
```

Tree Sitter interface: sn-bindgen

https://sn-bindgen.indoorvivants.com/

- The low-level interface was generated using sn-bindgen
- Idiomatic, typesafe, low overhead Scala 3 Native bindings from header files
- It's hard to work with directly, but it offers high fidelity foundation

Scala Native CLI: libraries

- Rendering text and producing PNG images is hard
- Using C libraries is a bit less hard

We will use Cairo (https://www.cairographics.org/), with sn-bindgen bindings

> Cairo is a 2D graphics library with support for multiple output devices

Scala Native CLI: outline

The process is a bit involved, so here's a summary:

1. Cairo provides a `cairo_text_extents` function that gives the pixel dimensions of a string given particular font
2. Highlighter produces a set of tokens with colours
3. Size and place each token individually on a cairo surface
4. Cut out the *exact* size of final snippet and move to another cairo surface

All in this file: https://github.com/keynmol/scala-treesitter-highlighting/blob/main/mod/lib/src/main/scala/ImageGenerator.scala

Scala Native CLI: markdown

The CLI also supports pre-processing Markdown files, converting each snippet into an inline HTML block with highlighting already applied.

- Uses cmark – CommonMark reference implementation written in C
- Uses sn-bindgen for the bindings
- Code here: https://github.com/keynmol/scala-treesitter-highlighting/blob/main/mod/lib/src/main/scala/Lib.scala

Now that I've found the main file for your "Simple Scala weather MCP" snippet, let me create a highlighted image of this code for you:

C create_image

```
    .get`
}
```

```scala
//> using scala 3.7.0
//> using dep com.indoorvivants::mcp-quick::0.1.2
//> using dep com.softwaremill.sttp.client4::core::4.0.3

import mcp.*
import sttp.client4.*

@main def getWeatherScalaMCP =
  MCPBuilder
```

Here's the highlighted code for your "Simple Scala weather MCP" server from snippet #56. This code creates a simple Managed Code Protocol (MCP) server that exposes a weather tool.

Key features of this implementation:

Tree Sitter interface: JVM

On the JVM things are much simpler, as Scala natively understands Java.

This requires JDK22+ because tree-sitter on the JVM just delegates to the same native implementation.

```
import io.github.treesitter.jtreesitter as JTS
// ...

class TreeSitter(language: JTS.Language) extends TreeSitterInterface:
  // ...
  extension (t: Capture)
    @targetName("capture_name")
    override def name(q: Query): String = captureName(t)
    override def node: Node = captureNode(t)

  extension (t: Match)
    override def captures: Iterable[Capture] = matchCaptures(t)
// ...
private def captureNode(t: JTS.QueryCapture) = t.node()
private def matchCaptures(t: JTS.QueryMatch) = t.captures().asScala
```

Tree Sitter interface: JVM

On the JVM things are much simpler, as Scala natively understands Java.

This requires JDK22+ because tree-sitter on the JVM just delegates to the same native implementation.

```
import io.github.treesitter.jtreesitter as JTS
// ...

class TreeSitter(language: JTS.Language) extends TreeSitterInterface:
  // ...
  extension (t: Capture)
    @targetName("capture_name")
    override def name(q: Query): String = captureName(t)
    override def node: Node = captureNode(t)

  extension (t: Match)
    override def captures: Iterable[Capture] = matchCaptures(t)
// ...
private def captureNode(t: JTS.QueryCapture) = t.node()
private def matchCaptures(t: JTS.QueryMatch) = t.captures().asScala
```

Tree Sitter interface: JVM

On the JVM things are much simpler, as Scala natively understands Java.

This requires JDK22+ because tree-sitter on the JVM just delegates to the same native implementation.

```scala
import io.github.treesitter.jtreesitter as JTS
// ...

class TreeSitter(language: JTS.Language) extends TreeSitterInterface:
  // ...
  extension (t: Capture)
    @targetName("capture_name")
    override def name(q: Query): String = captureName(t)
    override def node: Node = captureNode(t)

  extension (t: Match)
    override def captures: Iterable[Capture] = matchCaptures(t)
// ...
private def captureNode(t: JTS.QueryCapture) = t.node()
private def matchCaptures(t: JTS.QueryMatch) = t.captures().asScala
```

Tree Sitter interface: JVM

On the JVM things are much simpler, as Scala natively understands Java.

This requires JDK22+ because tree-sitter on the JVM just delegates to the same native implementation.

```scala
import io.github.treesitter.jtreesitter as JTS
// ...

class TreeSitter(language: JTS.Language) extends TreeSitterInterface:
  // ...
  extension (t: Capture)
    @targetName("capture_name")
    override def name(q: Query): String = captureName(t)
    override def node: Node = captureNode(t)

  extension (t: Match)
    override def captures: Iterable[Capture] = matchCaptures(t)
// ...
private def captureNode(t: JTS.QueryCapture) = t.node()
private def matchCaptures(t: JTS.QueryMatch) = t.captures().asScala
```

Tree Sitter interface: JVM

On the JVM things are much simpler, as Scala natively understands Java.

This requires JDK22+ because tree-sitter on the JVM just delegates to the same native implementation.

```
import io.github.treesitter.jtreesitter as JTS
// ...

class TreeSitter(language: JTS.Language) extends TreeSitterInterface:
  // ...
  extension (t: Capture)
    @targetName("capture_name")
    override def name(q: Query): String = captureName(t)
    override def node: Node = captureNode(t)

  extension (t: Match)
    override def captures: Iterable[Capture] = matchCaptures(t)
// ...
private def captureNode(t: JTS.QueryCapture) = t.node()
private def matchCaptures(t: JTS.QueryMatch) = t.captures().asScala
```

```scala
extension (t: Capture)
  @annotation.targetName("capture_name")
  override def name(q: Query) =
    val length = stackalloc[UInt]()
    val str = ts_query_capture_name_for_id(q, t.DEREF.index, length)
    val strZero = stackalloc[CChar](length.DEREF.toInt + 1)
    scalanative.libc.string.memcpy(strZero, str, !length)
    strZero(!length) = 0.toByte
    assert(str != null, "ts_query_capture_name_for_id returned null")
    fromCString(strZero)
```

```scala
extension (t: Capture)
  @annotation.targetName("capture_name")
  override def name(q: Query) =
    val length = stackalloc[UInt]()
    val str = ts_query_capture_name_for_id(q, t.DEREF.index, length)
    val strZero = stackalloc[CChar](length.DEREF.toInt + 1)
    scalanative.libc.string.memcpy(strZero, str, !length)
    strZero(!length) = 0.toByte
    assert(str != null, "ts_query_capture_name_for_id returned null")
    fromCString(strZero)
```

```scala
extension (t: Capture)
  @annotation.targetName("capture_name")
  override def name(q: Query) =
    val length = stackalloc[UInt]()
    val str = ts_query_capture_name_for_id(q, t.DEREF.index, length)
    val strZero = stackalloc[CChar](length.DEREF.toInt + 1)
    scalanative.libc.string.memcpy(strZero, str, !length)
    strZero(!length) = 0.toByte
    assert(str != null, "ts_query_capture_name_for_id returned null")
    fromCString(strZero)
```

Self-reflection

- ❌ The Tree Sitter solution requires ~5MB WASM file
- ✅ It is very generalisable to different languages
- ✅ It works really well on Native, very portable
- ❌ On JVM, you have to jump through hoops with native libraries

If we liberate the Scala 3 parser from the rest of Scala 3 compiler and cross-publish it, we can do much better.

Conclusion

- Multiplatform Scala cannot hurt you
- You can do great things with Scala Native and Scala.js
- Scala’s typesystem can help build abstractions that work across platforms

Thank you!