The Live Language

1 Overview

In this whitepaper, we will describe the Live language. The Live language is what powers Makepad's live coding environment, allowing the style of an application to be changed at runtime. The Live language is not a full programming language, but rather has a similar role as CSS has for HTML: that is, it describes the presentation of an application. In contrast, Makepad uses Rust to describe the structure and behavior of an application.

Section 2 begins by explaining the motivation behind the Live language. Section 3 gives a high-level description of the Live language by highlighting its most important features. The remaining sections are intended as a reference, and are written in semi-formal style. Section 4 describes the lexical structure of a Live code block. Section 5 describes the syntactic structure of a Live code block. Section 6 describes the different types that the expressions in a Live block can take. In section 7, we describe the internal representation used for Live code blocks. This representation takes the form of a flattened node list, which allows for extremely fast updates Section 8 describes how a Live code block is translated to a flattened node list. Section 9 describes the expansion process, during which objects that inherit from other objects are expanded into flat objects.

2 Motivation

In this section, we will give the motivation behind the Live language.

One of the primary goals of Makepad is to provide a live coding environment. A live coding environment is a coding environment in which application code can be changed at runtime, and changes to the code are immediately reflected in the application while it runs.

Makepad applications are developed in Rust. Rust is a multi-paradigm, general-purpose programming language, designed for performance and safety. Rust's focus on performance and safety makes it a good fit for applications with heavy demands on rendering in general, and UI rendering in particular, which are the kind of applications Makepad is designed for.

Unfortunately, Rust is a compiled language, which is at odds with Makepad's goal to provide a live coding environment: code written in a compiled language needs to be recompiled whenever a change to the code is made. Such a recompilation can easily take on the order of multiple seconds: hardly a live coding experience. To make matters worse, a recompilation requires the application to be restarted before the changes in the code are reflected.

The obvious alternative is to use a scripted language, such as Javascript, instead. Code written in a scripted language can be either interpreted or recompiled on the fly, allowing for much

faster turnaround times. Unfortunately, scripted languages lack the performance benefits of compiled languages, which is at odds with the kind of rendering heavy applications that Makepad seeks to enable. And using a scripted language together with Rust, creates a very uneasy balance where your code lives as the optimal solution is application specific.

Our solution out of this conundrum is to use a hybrid approach instead: Makepad applications are written in Rust, but the part of the code responsible for the look-and-feel of the application (i.e. the styling) is written in a DSL, which we call the Live language. The primary role of the Live language is analogous to that of CSS: it describes how the UI of an application should be presented. In particular, the Live language facilitates *overridable* styling in a similar vein as CSS.

Unlike Rust code, Live code is compiled and executed at runtime. Moreover, the visual design application, or visual designer, that we are building for Makepad is aware which parts of the code are Live code and which aren't. When you use the visual designer to change a piece of Live code, instead of recompiling the application, the visual designer sends the updated Live code to the running application. This allows the application to recompile and execute the new Live code, causing the changes in the code to be immediately reflected, without the need to recompile any Rust code, or restart the application.

3 High-level Description

In the previous section, we gave the motivation behind the Live language. In this section, we will give a high-level description of how the Live language is used. We do this by highlighting its most important features, and how these features are used. Note that this description is not intended to give a complete overview of the language, but rather to give a general feel for it. For a complete overview of the Live language, we refer to the reference sections instead.

The primary purpose of Live code is to initialize Rust structs that are responsible for rendering the UI with their initial values, and to automatically update these values at runtime when the Live code changes. This will be the subject of section 3.1.

An important requirement for Live code is that it provides an overridable styling mechanism. This is accomplished by allowing objects to inherit from other Live objects. This form of inheritance is explained in section 3.2. There is also a second form of inheritance that is supported by the Live language, which allows objects to inherit from Rust structs. This form of inheritance is explained in section 3.3. An important feature of Live code is that it allows the embedding of sub DSLs. This will be further explained in section 3.4. Finally, section 3.5 explains how the Live language supports dynamic components, for which the concrete type is not known until runtime.

3.1 Initializing and Updating Rust structs

In this section, we will illustrate how Live code can be used to initialize and update Rust structs.

Lets consider the following snippet of Rust code:

```
live_register! {
  Button: {{Button}} {
    bg: {
      color: #FFF
    }
  }
}

#[derive(Live)]
struct Button {
  bg: DrawQuad
}
```

This code defines a Rust struct `Button` that will be used to draw a button to the screen. For the sake of simplicity, we have left the code that actually draws the button out of the example. In addition to the struct definition, the code contains a call to the `live_register` macro. This macro is used to define a block of Live code. A Live code block looks similar to a JSON document, but there are some important differences.

Like a JSON document, a Live code block can contain primitive data types, arrays, and objects. However, unlike in JSON, objects in a Live code block can be 'inherited' from a Rust struct. This is what the line `Button: {{Button}}` means: it defines a top-level property called `Button` (there is an implicit root object on which top-level properties are defined), which is a Live object that inherits from the Rust struct `Button`.

When an Live object inherits from a Rust struct, we say that this object provides a definition for the Rust struct in the Live code. In other words, what we have done here is provide a definition for the Rust struct `Button` in the Live code block.

Note the use of the attribute `#[derive(Live)]` on the Rust struct `Button`. This attribute is used to generate the glue code that allows `Button` to interact with Live code, and vice versa. Among other things, it generates a constructor function that can be used to instantiate `Button` from its corresponding definition in the Live code block, using the following code:

```
let button = Button::new_from_module(
  cx,
  &module_path!(),
  id!(Button)
).unwrap();
```

This function first creates an instance of the Rust struct `Button`, with all its fields set to their default values, and then initializes these fields to their proper values by applying the Live object `Button`, defined as a top-level property in the current module (`module_path` is a Rust macro that expands to a string representing the current module path).

By default, applying a Live object to a Rust struct behaves as follows. We iterate over the properties of the Live object. For each field, we check if the Rust struct has a field with the same name. If so, we first check if the property and the field have compatible types. If the types are compatible, one of two things happens:

- If both the property and the field have a primitive or array type, the value of the property in the Live object overrides the value of the field in the Rust struct.
- Otherwise, the Live object has an object type and the field has a struct type, the application process is applied recursively.

Consequently, when we call `Button::new_from_module` to instantiate and initialize the Rust struct `Button` with the Live object `Button`, the end result will look like this:

```
Button {
  bg: DrawQuad {
    color: Vec4 { x: 1.0, y: 1.0, z: 1.0, w: 1.0 }
    // Additional fields on DrawQuad are not shown
  }
}
```

This is all well and good, but why go through such a convoluted way of initializing a Rust struct? The answer is that the visual designer we are developing for Makepad can tell when a change is made to a Live code block while the application is running. When the visual designer detects such a change, it does not trigger a recompilation and restart of the application. Instead, it sends a list of changes between the old and the new Live code block to the application.

When the running application receives such a diff from the visual designer, it responds by updating it's version of the Live code block, and then reapplying it to update the values of the same Rust struct. It then triggers a rerender of the application so that the new values immediately take effect. The reason this works is because Live code blocks are loaded at runtime, not compile time, so the application can keep a representation of each Live code block in memory. This internal representation of Live code blocks takes the form of a flattened node list, which makes updating the code at runtime extremely fast.

3.2 Inheriting from Live Objects

In the previous section, we illustrated how Live code can be used to initialize and update Rust structs. In this section, we will show how the Live language allows objects to inherit from other Live objects, and how this facilitates overridable styling.

Let's consider the following snippet of Rust code:

```
live_register! {
  Label: {{Label}} {
    text: {
      color: #FFF,
    },
    name: "Hello, world!",
  }
}

#[derive(Live)]
struct Label {
  text: DrawText,
  name: String,
}
```

This code defines a Rust struct `Label` that will be used to draw a label to the screen. This is similar to our previous example, except that the struct now contains two fields. This is because the `DrawText` component does not store the text to be drawn, so we have to pass it as an argument later on. We have again left the code that actually draws the label out of the example for the sake of simplicity.

If we call `Label::new_from_module` to instantiate and then initialize the Rust struct `Label` with the Live object `Label`, we get:

```
Label {
  text: DrawText {
    color: Vec4 { x: 1.0, y: 1.0, z: 1.0, w: 1.0 }
    // Additional fields on DrawText are not shown
  },
  name: "Hello, world!"
}
```

So far so good. Suppose now that we want to create a new kind of label that is always colored red. Let's call it `RedLabel`. We can do so by extending our Live code block with:

```
RedLabel: Label {
    text: {
      color: #F00,
    },
  }
}
```

The line RedLabel: Label defines a top-level property `RedLabel,` which is a Live object that inherits from the Live object `Label`. If we now call `Label::new_from_module` to instantiate and initialize the Rust struct `Label` with the Live object `RedLabel`, we get:

```
Label {
  text: DrawText {
    color: Vec4 { x: 1.0, y: 0.0, z: 0.0, w: 1.0 }
    // Additional fields on DrawText are not shown
  },
  name: "Hello, world!"
}
```

Note that the field `color` obtained its value from the Live object `RedLabel`, whereas the field `name` obtained its value from Label. This is because when the Live code block is loaded, `RedLabel` is expanded to:

```
RedLabel: {{Label}} {
  text: {
    color: #F00,
  },
  name: "Hello, world!",
}
```

That is, `RedLabel` inherited the value of the property `name` from `Label`, but overrode the value of the property `color` with its own `value`. It is by this mechanism that the Live achieves overridable styling.

3.3 Inheriting from Rust Structs

In the previous section, we showed how the Live language allows objects to inherit from other Live objects. In this section, we will show the other form of inheritance allowed by the Live language works, which is inheriting from Rust structs.

We begin by changing our definition of the `Live` object Label in the previous example to:

```
Label: {{Label}} {
  name: "Hello, world!",
}
```

If we now call `Label::new_from_module` to instantiate and initialize the Rust struct `Label` with the Live object `RedLabel`, we get:

```
Label {
```

```
  text: DrawText {
    color: Vec4 { x: 0.0, y: 1.0, z: 0.0, w: 1.0 }
    // Additional fields on DrawText are not shown
  },
  name: "Hello, world!"
}
```

Note that the field `color` obtained the value #0F0, even though we provided no definition for this field. So where does this value come from? The answer is that Makepad has a built-in Live definition for the struct `DrawText`, which looks something like this:

```
DrawText: {{DrawText}} {
  color: #0F0,
  // Additional properties on Drawtext are not shown
}
```

When the Live code block is loaded, because the Live object `Label` inherits from the Rust struct `Label`, and this struct contains a field `draw_text` with a struct type `DrawText`, and a Live definition for `DrawText` exists, the Live object `Label` is expanded to:

```
Label: {{Label}} {
  text: {
    color: #0F0,
  },
  name: "Hello, world!",
}
```

That is, the property `text` is set to the value of the top-level property `DrawText`, which provides a definition for the Rust struct `DrawText`, which is the type of the field `text` on the Rust struct `Label`, from which the Live object `Label` inherits.

3.4 Embedding Sub-DSLs

In this section, we will show how the Live language allows for other sub DSLs to be embedded in Live code. We currently use this feature in Makepad to embed shader programs as part of the styling of an application.

Consider the following snippet of Live code:

```
A: {
  color = fn(self) -> vec4 {
    return #0f0;
  },
  pixel: fn(self) -> vec4 {
```

```
    return mix(#F00, self.color(), 0.5)
  }
}

B: A {
  color = fn(self) -> vec4 {
    return #00f;
  }
}
```

The first thing to notice is the use of the separator (`=`) instead of (`:`) for the property color. (=) defines a so-called instance property, whereas (:) defines a so-called field property. Field properties are properties for which a corresponding field on the Rust struct exists, whereas instance properties are properties for which a corresponding field on the Rust struct does not exist. The reason we distinguish between these different kinds of properties is because we don't want their namespaces to overlap: you can define a field property and an instance property with the same name on the same object.

The second thing to notice here are the blocks starting with `fn`. These indicate the start of a function expression. Function expressions fulfill a similar role as quote does in Lisp. Instead of being fully parsed, function expressions are embedded as a list of tokens, which can then be interpreted differently, depending on what type they are applied to.

In the case of Makepad, when a function expression is applied to a shader struct, we interpret it as a function written in the Makepad Shading Language. The Makepad Shading Language is its own DSL, separate from the Live language. It largely follows GLSL syntax and semantics, and allows us to transpile the same shader code to either Metal, HLSL, or GLSL.

Embedding shader code as functions in live code blocks allows shaders to be edited at runtime. Because all UI in Makepad is ultimately rendered through shaders, this idea is extremely powerful. Moreover, using inheritance, function properties can be overridden just as any other values, allowing for derived UI components to reach in the shader code itself. In the example above, the property `pixel` is a function defining a pixel shader that calls the function `color`. Because the object B inherits from A, it has the same value for the property `pixel`, but a different value for the function `color`. Effectively, by overriding the function color, one can parametrise the behavior of the function `pixel`.

3.5 Dynamic Components

In the previous section, we have seen how Live code can be used to provide definitions for the fields of Rust structs, and to override their values. However, in a UI system, there are many cases where the concrete type of a component is not known until runtime. In this section we will explain how the Live language facilitates the use of such dynamic components.

Consider the following snippet of Rust code:

```
#[derive(Live)]
struct Document {
      header: FrameComponentRef
      body: FrameComponentRef
}
```

This defines a Rust struct `Document` that can be used to render a document to the screen. However, unlike in our previous examples, the fields `header` and `body` do not have a concrete type. Instead, they have the type `FrameComponentRef`, which is a newtype that wraps a trait object that implements the trait `FrameComponent`.

Given this Rust struct, we can now write the following in Live code:

```
Document {
  header: Fill {
    color: #F00
  },
  body: Button {}
}
```

Both `Fill` and `Button` are dynamic components. This means they have registered a factory function to create an instance of themselves, and expose a generic way to apply a Live value (by using the `FrameComponent` trait). The Live language is not tied to a particular component system, but supports registering any component system by name.

4 Lexical Structure

In this section, we describe the lexical structure of a Live code block.

A Live code block is a sequence of Unicode code points, encoded in UTF-8. The lexical structure of a Live code block groups these code points into sequences with an assigned meaning, known as tokens. A token is either a comment, an identifier, a keyword, a literal, a punctuation symbol, or a delimiter. In the remainder of this section, we will describe these different kinds of tokens in more detail. Since the lexical structure of Live code is intended to match that of Rust as closely as possible, we will point out where it differs from that of Rust.

4.1 Comments

Comments are used to annotate code with human-readable text, with the purpose of making the code easier to read.

A *comment* has one of two forms:

- A *line comment* starts with the character sequence `//`, and lasts until the end of the line.
- A *block comment* starts with the character sequence `/*`, and lasts until the character sequence `*/`. Block comments can be nested.

Unlike in Rust, doc comments are not supported.

4.2 Identifiers

Identifiers are used to identify entities, such as constants, properties, and variables.

An *identifier* starts with a letter or underscore, followed by zero or more digits, letters, or underscores. Identifiers may not be keywords.

Unlike in Rust, identifiers may not contain non-ASCII digits or letters.

4.3 Keywords

A keyword is an identifier that has special meaning, and therefore cannot (always) be used as a name. All keywords in Live code are weak, meaning they only have special meaning in certain contexts.

A *keyword* is one of the following:

- `crate`
- `fn`
- `use`
- `vec2`
- `vec3`
- `vec4`

4.4 Literals

A *literal* is a single token, and is used to directly represent a value. The different kinds of literals supported by Live code are listed here below.

4.4.1 Boolean Literals

A *boolean literal* is either `true` or `false`.

4.4.2 Integer Literals

An *integer literal* has one of four forms:

- A *binary integer literal* starts with the character sequence `0b`, followed by one or more binary digits or underscores, with at least one digit.
- An *octal integer literal* starts with the character sequence `0o`, followed by one or more octal digits or underscores, with at least one digit.
- A *hexadecimal integer literal* starts with the character sequence `0x`, followed by one or more hexadecimal digits or underscores, with at least one digit.
- A *decimal integer literal* starts with a decimal digit, followed by one or more decimal digits or underscores, with at least one digit.

Unlike in Rust, integer literals may not be followed by a suffix.

4.4.3 Floating-point Literals

A *floating-point literal* has one of two forms:

- A decimal integer, followed by a period, maybe followed by another decimal integer, maybe followed by an exponent.
- A decimal integer, followed by an exponent.

If a floating-point literal is a decimal integer, followed by a period, it may not be followed by an underscore or identifier.

Unlike in Rust, floating-point literals may not be followed by a suffix.

4.4.4 Vec Literals

A *vec literal* has one of three forms:

- A *vec2* literal starts with the keyword `vec2`, followed by a sequence of 2 floating-point literals, separated by commas, and delimited by parentheses.
- A *vec3* literal starts with the keyword `vec3`, followed by a sequence of 3 floating-point literals, separated by commas, and delimited by parentheses.
- A *vec4* literal starts with The keyword `vec4`, followed by a sequence of 4 floating-point literals, separated by commas, and delimited by parentheses.

4.4.5 Color Literals

A *color literal* starts with a hash, followed by one of the following forms:

- 4 pairs of hexadecimal digits, denoting an RGBA value.
- 3 pairs of hexadecimal digits, denoting an RGB value.
- A pair of hexadecimal digits, denoting a luminance value.
- 4 hexadecimal digits. This is a shorthand notation for the form with 4 pairs of hexadecimal digits, where both digits of each pair are the same.
- 3 hexadecimal digits. This is a shorthand notation for the form with 3 pairs of hexadecimal digits, where both digits of each pair are the same.
- 1 or 2 hexadecimal digits. This is a shorthand notation for the form where all color channels are filled with this digit making a grayscale value

4.4.6 Strings

A *string* is a sequence of zero or more unicode characters, delimited by double-quotes.

Special characters can be encoded using an *escape*. An escape starts with a backslash, followed by one of the following forms:

- A *hexadecimal escape* starts with the character `x`, followed by two hexadecimal digits. It denotes the ASCII character with the given value.
- A *unicode escape* starts with the character `u`, followed by at most six hexadecimal digits, enclosed in braces. It denotes the unicode code point with the given value.
- A *whitespace escape* is the character `n`, `r`, or `t`. It denotes a line feed (LF), carriage return (CR), or horizontal tab (HT), respectively.
- A *null escape* is the character `0`. It encodes the null character (NUL).
- A *backslash escape* is the backslash. It denotes the backslash itself.

4.4.7 Raw Strings

A raw string starts with the character `r`, followed by one or more hashes and a double-quote. The body of the raw string is a sequence of zero or more characters, and is terminated by a second double-quote and the same number of hashes that preceded the opening double-quote.

All characters in a raw string represent themselves: the double-quote (except when followed by the same number of hashes that preceded the opening double-quote) and backslash character do not have any special meaning.

4.5 Punctuation

The following table lists the different kinds of punctuation symbols used in the DSL, and their usage:

Symbol	**Name**	**Usage**
`+`	Plus	Addition
`-`	Minus	Subtraction, negation
`*`	Star	Multiplication
`/`	Slash	Division
`::`	DoubleColon	Path separator
`:`	Colon	Field name/value separator
`=`	Equals	Instance name/value separator
`=?`	EqualsQuestion	Template name/value separator

`,`	Comma	Various separators
`->`	Arrow	Function return type

4.6 Delimiters

The following table lists the different kinds of delimiters used in Live code:

Symbol	**Name**
`{ }`	Braces
`[ ]`	Brackets
`( )`	Parentheses

5 Syntactic Structure

In this section, we describe the syntactic structure of a Live code block.

The lexical structure of a Live code block is that of a sequence of tokens. The syntactic structure of a Live code block further organizes these tokens into a parse tree. The top of the parse tree always consists of a sequence of items. Items in turn can contain expressions. In the rest of this section we will describe items and expressions in more detail.

5.1 Items

An item is anything that can appear at the top-level of a Live code block. An item is either a use declaration or a top-level property definition. We will describe these items here below.

5.1.1 Use Declaration

Syntax
use_declaration:
`use` *initial_path_segment* (`::` *additional_path_segment*) *final_path_segment*

initial_path_segment:
identifier
| `crate`

additional_path_segment:
identifier

final_path_segment:
identifier

| *

Use declarations are used to import top-level properties from live code blocks defined in other modules.

A use declaration starts with the keyword `use`, followed by an initial path segment, followed by a sequence of zero or more path separators (`::`) and additional path segments, followed by another path separator (`::`) and a final path segment.

An *initial path segment* is either an identifier, or the keyword `crate`.

An *additional path segment* is an identifier.

A *final path segment* is either an identifier, or a wildcard.

5.1.2 Top-level property Definitions

Syntax
top_level_property_definition:
identifier? *identifier* = *object_expression*

Top-level property definitions are used to define properties on the implicit root object.

A top-level property definition starts with an optional identifier, followed by another identifier, followed by an instance name/value separator, followed by an object expression.

5.2 Expressions

An *expression* is a sequence of tokens that evaluates to a value.

5.2.1 Primary Expressions

5.2.1.1 Literal Expressions

A *literal expression* is one of the literals described earlier. It evaluates to the value of the literal.

5.2.1.2 Array Expressions

Syntax
array_expression:
`[` *primary_expression* (`,` *primary_expression*)* `,`? `]`

An *array expression* is a sequence of one or more primary expressions, separated by commas,, and delimited by brackets. It evaluates to a value of type array.

5.2.1.3 Object Expressions

Syntax
object_expression:
base_object_specifier? `{` (*property_definition* (`,` *property_definition*)*)? `}`

base_object_specifier:
identifier
| `{ {` *identifier* `} }`

property_definition:
identifier? *identifier* `:` *value*
identifier? *identifier* `=` *value*
identifier? *identifier* `=?` *value*

An *object expression* starts with an optional base object specifier, followed by a sequence of one or more key/value pairs, separated by commas, and delimited by braces. It evaluates to a value of type object.

A *base object specifier* has one of two forms:

- A *Live base object specifie*r is an identifier.
- A *Rust* base *object specifier* is an identifier, enclosed in double brackets.

A *property definition* has one of three forms:

- A *field property definition* starts with an optional identifier, followed by another identifier, followed by a field name/value separator (`:`), followed by a value.
- An *instance property definition* starts with an optional identifier, followed by another identifier, followed by an instance name/value separator (`=`), followed by a value.
- A template *property definition* starts with an optional identifier, followed by another identifier, followed by a template name/value separator (`=?`), followed by a value.

5.2.1.4 Function Expression

Syntax
function_expression:
`fn (` *token_list* `) (->` *identifier)*? `{` *token_list* `}`

A *function expression* starts with the keyword `fn`, followed by a token list delimited by parentheses, followed by an optional arrow and identifier, followed by another token list delimited by braces.

Delimiters can nest inside token lists. That is, a token list is not terminated by a closing delimiter unless that delimiter is at the same level as its opening delimiter.

5.2.1.5 Identifier Expressions

Syntax
identifier_expression:
identifier

An *identifier expression* is an identifier.

5.2.1.6 Grouped Expressions

Syntax
grouped_expression:
（ *expression* ）

A *grouped expression* is an expression, delimited by parentheses.

5.2.2 Compound Expressions

5.2.2.1 Call Expressions

Syntax
call_expression:
identifier （ (*expression* (, *expression*)*)? ）

A *call expression* starts with an identifier, followed by a sequence of expressions, separated by commas, and delimiter by parentheses.

5.2.2.2 Operator Expressions

Syntax
operator_expression:
unary_operator expression
expression binary_operator expression

An *operator expression* has one of two forms:

- A unary operator, followed by an expression.
- An expression, followed by a binary operator, followed by another expression.

The unary operator (−) operates on integer and floating-point scalars, and vectors. If the operand is a scalar, the negation is applied, resulting in another scalar. If the operand is a vector, the scalar negation is applied independently to each component of the vector, resulting in another vector of the same size.

The binary operators (`+`), (`-`), (`*`), (`/`) and operate on integer and floating-point scalars, and vectors. If one operand is an integer and the other is not, the integer operand is converted to a floating-point operand. After conversion, the following cases are valid:

- Both operands are scalars. In this case, the operation is applied, resulting in another scalar.
- One operand is a scalar, and the other is a vector. In this case, the scalar operation is applied independently to each component of the vector, resulting in another vector of the same size.
- Both operands are vectors of the same size. In this case, the operation is done component-wise, resulting in another vector of the same size.

6. Types

In this section we will describe the different types that each expression in a Live block can take.

Every expression in a Live code block has a type. The type of an expression determines the values that it can take, and the operations that can be performed on it. Types are organized into primitive types, which cannot be composed from other types, and composite types, which can. In the rest of this section we will describe both primitive and composite types in more detail.

6.1 Primitive Types

6.1.1 Boolean Type

The *boolean type* or *bool* can take one of two values, called true and false.

There are currently no operations that can be performed on values of type bool.

6.1.2 Integer Type

The *integer type* or *int* can take any integer value between -2^{63} and $2^{63}-1$.

Values of type int can be negated, or added/subtracted/multiplied with other values of type int, resulting in another value of type int.

6.1.3 Floating-point Type

The *floating-point type* or *float* corresponds to the IEEE754 "binary64" type.

Values of type float can be negated, or added/subtracted/multiplied with other values of type float, resulting in another value of type float. Alternatively, values of type float can be added/subtracted/multiplied/divided with other values of type vec2, vec3, or vec4. In this case,

the scalar operation is applied independently to each component of the vector, resulting in another value of type vec2, vec3, or vec4, respectively.

6.1.4 Vec2 Type

The *vec2 type* or *vec2* comprises all 2-tuples of values of type floating-point.

Values of type vec2 can be negated, or added/subtracted/multiplied/divided with other values of type vec2. In this case, the operation is done component-wise, resulting in another value of type vec2. Alternatively, values of type vec2 can be added/subtracted/multiplied/divided with other values of type float. In this case, the scalar operation is applied independently to each component of the vector, resulting in another value of type vec2.

6.1.5 Vec3 Type

The *vec3 type* or *vec3* comprises all 3-tuples of values of type floating-point.

Values of type vec3 can be negated, or added/subtracted/multiplied/divided with other values of type vec3. In this case, the operation is done component-wise, resulting in another value of type vec3. Alternatively, values of type vec3 can be added/subtracted/multiplied/divided with other values of type float. In this case, the scalar operation is applied independently to each component of the vector, resulting in another value of type vec3.

6.1.6 Vec4 Type

The *vec4 type* or *vec4* comprises all 4-tuples of values of type float.

Values of type vec4 can be negated, or added/subtracted/multiplied/divided with other values of type vec4. In this case, the operation is done component-wise, resulting in another value of type vec4. Alternatively, values of type vec4 can be added/subtracted/multiplied/divided with other values of type float. In this case, the scalar operation is applied independently to each component of the vector, resulting in another value of type vec4.

6.1.7 Color Type

The *color type* or *color* comprises all 4-tuples of values of type float, where the values of the components are clamped to the interval [0.0, 1.0].

Values of type color support the same operations as values of type vec4. The only difference is that after each operation the values of the components are reclamped to the interval [0.0, 1.0].

6.1.8 String Type

The *string type* or *string* comprises all sequences of Unicode characters.

There are currently no operations that can be performed on values of type string.

6.1.9 Function Type

The function type or function comprises all sequences of tokens that make up a function.

There are currently no operations that can be performed on values of type function.

6.2 Composite Types

6.2.1 Array Type

The *array type* or *array* comprises all sequences of `N` values of type `T`.

There are currently no operations that can be performed on values of type array.

6.2.2 Object Type

The *object type* or *object* comprises all collections of named properties.

There are currently no operations that can be performed on values of type object.

7. Internal Representation

In this section, we describe the internal representation of a Live code block.

A Live code block defines a data structure that consists of primitive values, arrays, objects, or expressions. This data structure is represented internally as a flattened list of nodes. Traversing this list yields the nodes of the data structure in depth-first order. The reason we use a flattened node list as our representation is that it allows for extremely fast updates of the code at runtime.

A *node* represents either an object property, or an array element. Nodes representing a top level item or object property contain both the name and value of the item/property they represent. Nodes representing an array element only contain the value of the element they represent.

In addition, every node has an origin, which defines the position in the token stream where the node was defined, and a set of flags, which are used for various purposes, such as indicating the kind of property (field, instance, or template) the node represents, or whether the name has a prefix. Note that the prefix is not contained in the node but can be obtained from the token stream using the origin.

A *value* is an enumerated data type, and can represent a value in different ways, depending on how that value was defined.

When the value was defined using a literal, array, or object expression, the value of the expression is stored directly. The primitive values defined by literal expressions, such as bool,

int, and float, are represented as themselves. The composite values defined by array or object expressions, i.e. arrays and objects, are represented with a value that indicates the start of an array/object, and are followed by a sequence zero or more nodes, representing the elements/properties of the array/object. The sequence is terminated by a node with the special value **close**.

When the value was defined using a function expression, the sequence of tokens making up the function

When the value was defined using an identifier expression, instead of the value of the identifier, the identifier itself is stored, and needs to be resolved later.

When the value was defined using a composite expression, instead of the value of the expression, the expression itself is stored, and needs to be evaluated later. Operators and built-in function calls are each represented with their own value type, and are followed by a fixed number of nodes representing the operands/arguments of the operator/call.

The following table lists the different values that a node can contain, and their meaning:

Value	**Meaning**
bool(x)	A value of type bool.
int(x)	A value of type int.
float(x)	A value of type float.
vec2(x)	A value of type vec2.
vec3(x)	A value of type vec3.
vec4(x)	A value of type vec4.
color(x)	A value of type color.
string(x)	A value of type string.
array	A value of type array. A node with this value is followed by a sequence of zero or more nodes, representing the elements of the array. This sequence is terminated by a node with the special value **close**.
object	A value of type object. A node with this value is followed by a sequence of zero or more nodes, representing the properties of the object. This sequence is terminated by a node with the special value **close**.

clone(id)	A value of type object. This is similar to **object**, except that it inherits from a DSL object named id, and then extends it with additional properties. A node with this value is followed by a sequence of zero or more nodes, representing the additional properties of the object. This sequence is terminated by a node with the special value **close**. Nodes with this value are replaced during expansion, and are further explained there (see below).
class(typeid)	A value of type object. This is similar to object, except that it inherits from a Rust struct with the given typeid, and then extends it with additional properties. A node with this value is followed by a sequence of zero or more nodes, representing the additional properties of the object. This sequence is terminated by a node with the special value **close**. Nodes with this value are replaced during expansion, and are further explained there (see below).
close	A special value that denotes the end of an array or object.
fn(tokens)	A value of type function. Values of this type represent a pointer to a list of tokens, and are interpreted differently depending on what Rust struct they are applied on.
ident(id)	The value of the item named id. Values of this type need to be resolved before the final value can be obtained.
unop(op)	A value that is the result of applying the unary operator op to an operand. A node with this value is followed by another node, representing the operand. Values of this type need to be evaluated before the final value can be obtained.
binop(op)	A value that is the result of applying the binary operator op to two operands. A node with this value is followed by two other nodes, representing the two operands. Values of this type need to be evaluated before the final value can be obtained.
call(id, n)	A value that is the result of applying the function id to n arguments. A node with this value is followed by n other nodes, representing the arguments. Values of this type need to be evaluated before the final value can be obtained.

use(path)	A special value that is used during name resolution. Nodes with this value are used during expansion, and are further explained there (see below).

8. Translation Process

In this section, we describe the process of translating the parse tree of a Live code block into a flattened node list.

To aid the description of the translation process, we will introduce some notation. Consider the translation of the Live syntactic form `2 + 3` into the flattened list of nodes **binop**(+) 2 3. This translation process may be regarded as a function T, that takes a syntactic form as input and returns a list of nodes as output. We can write this translation as an equation:

T[[2 + 3]] = **binop**(+) 2 3.

If the syntactic form contains any non-terminals, T needs to be recursively applied to those non-terminals. For instance, the general translation scheme for binary operator expressions is:

T[[*expression binary_operator expression*]]
= **binop**(op) T[[*expression*]] T[[*expression*]]

where op is the operand.

EBNF operators, such as ? and *, which are applied to the syntactic forms on the left side of the equation, can also be applied to the corresponding result of T on the right side of the equation. This should be interpreted as that the result of T is repeated as often as the corresponding syntactic form appears on the left of the equation.

For instance, in the translation scheme,
T[[*expression*$^?$]] = T[[*expression*]]$^?$

The translated expression T[[*expression*]] on the right side only appears if the *expression* on the left side does.

Similarly, in the translation scheme:
T[[*expression**]] = T[[*expression*]]*

The translated expression T[[*expression*]] on the right side appears as many times as the *expression* on the left side does.

In the remainder of this section, we will describe the general translation scheme for the different syntactic forms that make up a Live code block.

8.1 Items

8.1.1 Use Declarations

The general translation scheme for use declarations is:
T[[`use` *initial_path_segment* (`::` *additional_path_segment*) *final_path_segment*]] = **use(**path)

where path is the concatenation of all the path segments, joined with path separators.

8.1.2 Top-level property Definitions

The general translation scheme for top-level property definitions is:
T[[*identifier*? *identifier* `:` *object_expression*]] = *identifier*? *identifier* = T[[*object_expression*]]

8.2 Expressions

8.2.1 Primary Expressions

8.2.1.1 Literal Expressions

The translation scheme for literals is straightforward. We simply translate the literal to a node with the value of the literal. We have:
T[[*boolean_literal*]] = **bool**(x)
T[[*integer_literal*]] = **int**(x)
T[[*floating_point_literal*]] = **float**(x)
T[[*vec2_literal*]] = **vec2**(x)
T[[*vec3_literal*]] = **vec3**(x)
T[[*vec4_literal*]] = **vec4**(x)
T[[*color_literal*]] = **color**(x)
T[[*string_literal*]] = **string**(x)
T[[*raw_string_literal*]] = **string**(x)

where x is the value of the literal.

8.2.1.2 Array Expressions

An array expression such as:
```
[2, 3]
```

is translated to:
array int(2) **int**(3) **close**

In general, array expressions are translated into a node indicating the start of the array, a node indicating the end of the array, and zero or more nodes in between that are the result of recursively translating the primary expressions defining the elements of the array.

The general translation scheme for array expressions is thus:
T[[*primary_expression* (`,` *primary_expression*)* `,`$^?$ `]`]]
= **array** T[[*primary_expression*]] T[[*primary_expression*]]* **close**

7.2.1.3 Object Expressions

An object expression such as:
`{ x: 2, y: 3 }`

is translated to:
object x: **int**(2) y: **int**(3) **close**

In general, object expressions are translated into a node indicating the start of the object, a node indicating the end of the object, and zero or more nodes in between that are the result of recursively translating the property definitions defining the properties of the object.

The general translation scheme for object expressions is thus:
T[[`{` (*property_definition* (`,` *property_definition*)*)$^?$ `}`]]
= **object** (T[[*property_definition*]] T[[*property_definition*]]*)$^?$ **close**

T[[*identifier* `{` (*property_definition* (`,` *property_definition*)*)$^?$ `}`]]
= **clone**(id) (T[[*property_definition*]] T[[*property_definition*]]*)$^?$ **close**

T[[`{ {` *identifier* `} }` `{` (*property_definition* (`,` *property_definition*)*)$^?$ `}`]]
= **class**(id) (T[[*property_definition*]] T[[*property_definition*]]*)$^?$ **close**

where id is the name of the identifier.

Property definitions are translated into property nodes. We have:
T[[*identifier*$^?$ *identifier* `:` *object_expression*]]
= *identifier*$^?$ *identifier* `:` T[[*object_expression*]]

T[[*identifier*$^?$ *identifier* = *object_expression*]]
= *identifier*$^?$ *identifier* = T[[*object_expression*]]

T[[*identifier*$^?$ *identifier* `=?` *object_expression*]]
= *identifier*$^?$ *identifier* `=?` T[[*object_expression*]]

8.2.1.4 Function Expressions

The translation scheme for function expressions is straightforward. We simply translate the list of tokens that make up the expression to a node with a pointer to that list. We have:
T[[`fn (` *token_list* `)` (`->` *identifier*)$^?$ `{` *token_list* `}`]] = **fn**(tokens)

where tokens is a pointer to the list of tokens that make up the expression.

8.2.1.5 Identifier Expressions

The translation scheme for identifier expressions is straightforward. We simply translate the identifier to a node with the name of the identifier. We have:
T[[*identifier*]] = **ident**(id)

where id is the name of the identifier.

8.2.1.6 Grouped Expressions

The general translation scheme for grouped expressions is:
T[[(*expression*)]] = T[[*expression*]]

8.2.2 Compound Expressions

8.2.2.1 Call Expressions

A call expression such as:
```
f(2, 3)
```

is translated to:
call(f, 2) 2 3

The general translation scheme for call expressions is:
T[[*identifier* ((*expression* (, *expression*)*)$^?$)]]
= **call**(*identifier*, n) (T[[*expression*]] (, T[[*expression*]])*)$^?$

where n is the number of arguments to the call.

8.2.2.2 Operator Expressions

A unary operator expression such as:
```
-1
```

is translated to:
unop(-) 1

The general translation scheme for unary operator expressions is:
T[[*unary_operator expression*]]
= **unop**(op) T[[*expression*]]

where op is the operator.

A binary operator expression such as:

```
2 + 3
```

is translated to:
binop(+) 2 3

The general translation scheme for binary operator expressions is:
T[[*expression binary_operator expression*]]
= binop(op) T[[expression]] T[[expression]]

where op is the operator.

9. Expansion

During expansion, the original node list, representing the Live code block is converted into an expanded node list, in which objects that inherit from either other Live objects or Rust structs have been expanded into flat objects.

To illustrate the expansion process, consider the following Live code block:

```
A = {
  x: 2.0
}

B = A {
  y: 3.0
}
```

The original node list for this Live code block looks like this:
A: **object**
x: **float**(2.0)
close
B: **clone**(A)
y: **float**(3.0)
close

After expansion, the expanded node list looks like this:
A: **object**
x: **float**(2.0)
close
B: **object**
x: **float**(2.0)
y: **float**(3.0)
close

In general, the expansion process works as follows: given an original node list, we first create an empty expanded node list. We then iterate over the nodes in the original node list from left to right. For each step of the iteration, we add the current node to the expanded node list, using the following rules:

- If the node represents an object property, and a property with the same name already exists on the object currently being expanded:
    - If the property being overridden represents a value of type object:
        - Recursively merge the existing object with the new object.
    - Otherwise:
        - Replace the existing value with the new value.
- Otherwise:
    - Append the node to the end of the list.

There are two exceptions to the above rules: if the value of the current node is either **clone**(id) or **class**(typeid), the node is handled specially. We will now explain these two cases in the sections here below. The final section will explain how the name resolution process during expansion works.

9.1 Clone Nodes

If the value of the current node is **clone**(id)**,** it represents the start of an object that inherits from another Live object. We will refer to this object as the *child object*, and the object it inherits from as the *parent object*. Because the expansion process iterates from left to right, we can assume that the parent object has already been added to the expanded node list, and has been fully expanded. Our goal is therefore to find the nodes that make up the parent object, and copy them to the end of the expanded node list.

To accomplish this, we first need to find the start of the parent object in the expanded node list. We do this by searching backwards in the list, until we find the node denoting the start of the parent object. This will be a node representing an object property, with a name that matches the id of the parent, and a value that is either **object**(id) or **class**(id). Note that since the parent object has already been fully expanded, it can never start with a node with the value **clone**(id).

Once the node denoting the start of the parent object has been found, we copy the nodes making up the parent object to the end of the expanded node list. This includes the node denoting the start of the parent object and the nodes representing the properties of the object, but not the node denoting the end of the parent object, since we still want to add additional properties to the child object.

9.2 Class Nodes

If the value of the current node is **class**(typeid), it represents the start of an object that inherits from a Rust struct. At most one object can inherit from the same Rust struct at a time. We say that this object provides a definition for the Rust struct in the Live code block. For the sake of

this discussion, we assume that a mapping exists from typeids to type descriptors. A *type descriptor* describes a Rust struct, by listing the names and type-ids of all the fields on the struct with a type for which a definition exists in a Live code block.

To expand nodes with this value, we first append the node to the expanded node list. We then use the typeid of the node to look up the corresponding type descriptor. For each field in that type descriptor, we search backwards in the expanded node list until we find the node denoting the start of the object providing a definition for the type of the field. This will be a node with value **class**(typeid), where typeid matches the typeid of the field.

Once the node denoting the start of the object has been found, we copy the nodes making up the object to the end of the expanded node list. This includes the node denoting the start of the parent object, the nodes representing the properties of the object, and the node denoting the end of the parent object.

9.3 Name Resolution

The name resolution process used during expansion is very simple: we simply search backwards in the expanded node list, until we find the node we are looking for. Note that this gives the desired behavior with regards to shadowing: if two properties with the same name are defined on the same object, the property defined later shadows the one defined earlier. Moreover, if two properties on a path from the current node to the root have the same name, the one that is closest to the current node is preferred.

To facilitate cross module name resolution during expansion, use declarations are incorporated into the node list as nodes with the special value **use**(path). When we encounters such a node during name resolution, we look up the expanded node list for the live code block corresponding to the given path, and then search *forward* in that node list until we find the node we are looking for.