Universiteit
Leiden
The Netherlands

Bachelor Computer Science

Yune: A Runtime-Free Programming Language
with Homogeneous, Hygienic Macros

Ivo te Sligte

Supervisors:
Tobias Kappé & Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

7/6/2026

Abstract

Metaprogramming and embedded domain specific languages are able to drastically reduce the amount of code required to express complex concepts. However, only a few languages without practical considerations expose their fullest potential. We analysed these languages and designed and implemented our own runtime-free language able to freely interoperate with C++. We compare our metaprogramming system with alternatives on hygiene, syntax, homogeneity, and quality of error messages. Results show that our language is competitive with or surpasses the state-of-the-art in the named categories.

Contents

1 Introduction

A metaprogram is a program that manipulates source code [1]. Most mainstream programming languages support metaprogramming because it can drastically decrease the amount of code that must be written manually. Some forms of metaprogramming can also increase runtime performance by moving computations to compile-time [2, 3].

Domain Specific Languages (DSLs) are another concept used to improve the ease of writing code. These allow for the concise and clear representation of the programmer's ideas by specialising syntax and behaviour for their domains [4]. Well-known examples are JSON, SQL, Bash, and HTML, each specialised for a different purpose.

DSLs can either be embedded in a normal program or used independently [4]. Embedded DSLs (EDSLs) are placed where they are used, clearly conveying intent, and can often utilise variables in their surrounding context. Independent DSLs are placed in separate files, making it possible to parse them as fully separate languages, which typically results in superior tooling (e.g. syntax highlighting and error messages).

Metaprograms can also have several degrees of integration with a host language. The least integrated is a preprocessor, such as the one in C, which modifies the text in a file according to user-specified directives, with little understanding of the underlying language, before passing it to the compiler. This makes them much simpler to implement and often language-agnostic. Fully integrated metaprograms have a superior understanding of the host language and can therefore apply more intricate transformations to the source code, but require either full support from the compiler, or an external program that reimplements parts of the compiler [1].

This thesis reports on the design and implementation of a programming language, focusing on metaprogramming, DSL embedding, and interoperation with other languages. The reader is assumed to have a basic understanding of lexing, parsing, semantic analysis, code generation, C macros, C++ templates, JSON, and SQL.

1.1 Background

This section explains additional knowledge that is required to understand the research question.

1.1.1 Terminology

We introduce a few terms used throughout the thesis: *metalanguage* is a shortened form of "metaprogramming language". The *host language* is the language containing a metalanguage or DSL embedding. For example, C++ is a host language, and its templates are a metalanguage. We use *static* as a shorter form of "at compile-time". We refer to functions and variables in the outermost scope, also known as the *top-level*, as globals. *Locals* are variables that are not *globals*. Lastly, *pure* code is code that always produces the same output for the same input.

1.1.2 EDSLs in Existing Languages

Many programming languages have support for EDSLs, with varying degrees of syntactic freedom and compile-time safety [2]. In its simplest form, an EDSL can be represented as a string literal parsed at runtime. SQL is often handled this way. This method barely restricts syntax, only requiring that delimiters are escaped. However, syntactic and semantic errors in the embedded

language are detected at runtime, and tooling rarely supports syntax highlighting in string literals. Additionally, regular string literals cannot capture variables from the environment.

Alternative methods with superior guarantees of static correctness exist in languages with less common features. Some languages, such as Rust, Nim, and Jai, natively support compile-time code execution [5, 6, 7]. Section 2.7.7 touches on using operator overloading to embed DSLs in languages without special support.

1.1.3 Macros

Macros are a kind of metaprogram that provide a middle ground between complete separation and complex composition. We define macros to be functions or pattern-matchers executed at compile-time that enrich the grammar of the host language and produce code in it. They may enable inline DSLs similar to string literal-based DSLs, with compile-time code generation, semantic verification, IDE integration, and the same syntactic freedom. In practice, the syntax is more limited, and semantic correctness is often only partially guaranteed (see Section 2). Metalanguages typically resolve grammar conflicts of DSLs in macros by either modifying DSLs' grammar (e.g. Rust [6], Nim [8], Racket [9]) or introducing more complex brackets (e.g. Nemerle [10]).

1.1.4 Hygiene

Macros are able to generate code that uses variables in the environment, however, naive macro implementations may generate unintuitive code that captures the wrong variables [2]. This is known as a *hygiene* violation. Concretely, hygiene is violated when the definition of a macro refers to a variable or function in the definition's context by its name, but the name is redefined at the macro's invocation site, causing the generated code to use the wrong symbol. A simple C example is shown in Figure 1. Note that the violation would not occur if `printTEXT` was a function instead. Hygienic macros ensure that names referred to in the macro's definition always refer to definitions in the macro definition's context, unless specified otherwise. This binding strategy is referred to as *lexical scoping*.

1.1.5 Homogeneity

Metaprogramming languages often differ significantly from their host languages [1]. For example, C++'s template language has distinct syntax and cannot call functions defined in the host language. There is little code reuse between the languages and the programmer has to learn unique syntax and semantics to be able to use the template language. Languages where the host- and metalanguage are identical are called *homogeneous*.

1.1.6 Semantic Annotations

Programming languages vary significantly in the amount of semantic annotations. More annotations mean that the compiler can have a better understanding of the source code, improving its ability to detect semantic errors at compile-time, while requiring an initial time investment from the programmer.

Type annotations are a common form of semantic annotations with itself a number of levels of explicitness. Dynamically typed languages report some or all of their type errors at execution

```c
const char* TEXT = "Hello, World!";

#define printTEXT { printf("%s\n", TEXT); }

int main(void) {
  printTEXT
  // prints "Hello, World!"

  // redeclares TEXT in a different context
  char* TEXT = "Goodbye, Hygiene!";
  printTEXT
  // prints "Goodbye, Hygiene!"
}
```

Figure 1: Hygiene violation in C.

time (e.g. Python and TypeScript). They typically do not require type annotations. Conversely, statically typed languages do require annotations. Gradual typing means that annotations are not required, but used by the compiler when supplied (e.g. Typed Racket). The most complex class is dependent typing, present in proof-oriented languages (e.g. Rocq), which expresses relations of runtime values using the type system.

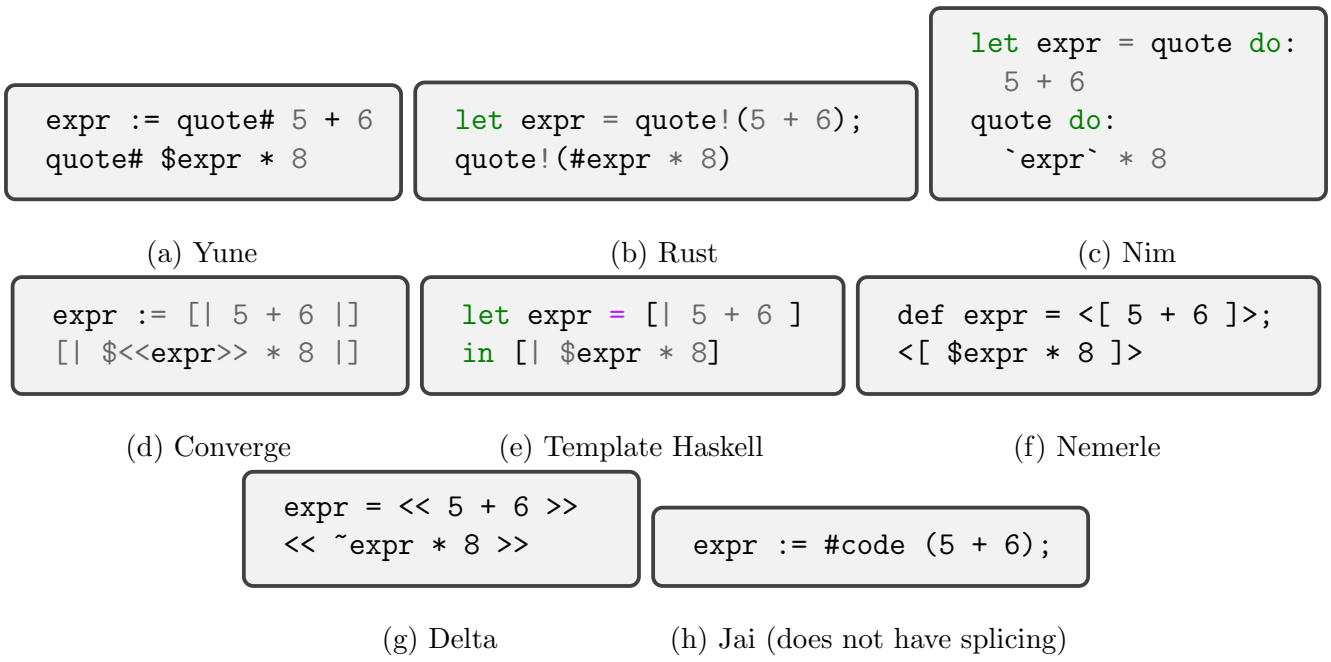
1.1.7 Runtime

Another trade-off in language design is the complexity of features in the execution environment. Fewer features make it easier to interoperate with other languages due to the simplified interface, but often more difficult to program purely in the language itself. Examples are garbage collectors and JIT interpreters. We define a *runtime* to be the full collection of language features that execute in the background for the duration of the main program. Data created by a runtime requires additional manual management when passed to another runtime as the original runtime is no longer in control of it. Code automatically executed before the main program starts is not considered to be part of the runtime as it does not complicate calling functions in the language from another language.

1.1.8 Quasi-Quotation

Quasi-quotation is a language feature that constructs an abstract syntax tree (AST) from text, allowing the concise inline expression of ASTs [4]. Figure 2 shows quasi-quotation as implemented in several languages. The first statement of each example shows a simple quotation that parses a syntactical expression from the text `5 + 6`, and the second statement shows *splicing*, which means injecting the AST stored in a variable into a quotation. Intuitively, quasi-quotation is similar to string interpolation. In Python, `f"Hello, {name}!"` creates a string using a literal and a variable, while in Template Haskell, `[| "Hello, " + $expr ]` creates an expression using the same.

Some languages provide built-in syntax for quasi-quotation, which typically uses a bracket



(a) Yune (b) Rust (c) Nim

(d) Converge (e) Template Haskell (f) Nemerle

(g) Delta (h) Jai (does not have splicing)

Figure 2: Quasi-Quotation Examples

notation, and others emulate it with macros (e.g. our language, Rust, and Nim). Having a form of quasi-quotation is expected of languages that require the construction of syntax trees, for example as macro output.

1.2 Research Question

The motivation to reduce runtime errors using static typing, trivially interoperate with other languages, and utilise powerful macros with flexible syntax that predictably interact with their environment, raises the question:

How can a runtime-free, statically typed programming language be implemented to support hygienic, homogeneous metaprogramming targeting the integration of domain specific languages with unrestricted syntax?

The novel programming language this thesis introduces - named Yune - improves the safety of macros using hygiene, and leverages the boost to writability of macros with unrestricted syntax. Yune compiles to C++, capitalising on its interoperability and lack of runtime.

1.3 Overview

The rest of this thesis is structured as follows: This section contains the introduction. Section 2 gives an overview of related work including other metaprogramming systems, tools for creating DSLs, emulation of metaprogramming in languages lacking native support, and the integration of DSLs with host languages. Section 3 details the methodology behind this work. Section 4 reports difficulties in the implementation and their solutions. Section 5 describes experiments comparing

Yune's macros with the state-of-the-art and their outcomes. Section 6 concludes, and Section 7 discusses the limitations of our work and directions for future research.

2 Related Work

This section discusses relevant existing work, focusing on the topics of metaprogramming and DSL embedding.

2.1 Metaprogramming Framework and Delta

Lilis and Savidis (2015) [1] introduce a framework for implementing a language with metaprogramming features that can be used with the same convenience as the host language itself, i.e. perfectly homogeneously. It emphasises the importance of state sharing and predictable evaluation order across macro invocations. It also declares that metaprograms should have equal tooling to regular programs, including IDE integration and a debugger. Contrary to Lilis and Savidis' advice, our macro evaluation order is mostly unpredictable due to a flaw in the implementation (further explained in Section 7) and we do not provide tooling besides an Emacs syntax highlighting extension, to limit the scope of this thesis.

The paper categorises programming languages based on the number of compilation stages they support. Languages without code generation facilities are single-stage, and languages with such facilities are at least two-stage. When code generators can generate more code generators, the number of stages is unbounded, as in our language.

While the insights discussed in the article can be applied to any language, Lilis and Savidis primarily design and implement an extension to the Delta programming language. As described by ISC-FORTH: "The Delta language is untyped object-based (classless), dynamically-typed (weakly, duck typing), lexically-scoped, garbage collected, compiled to platform-neutral byte code, run by a virtual machine". In short, Delta is dynamically typed and has a heavy runtime containing a garbage collector and virtual machine. Delta also lacks macros as we defined them. Compared to Yune, it is therefore more difficult to embed DSLs in Delta and use Delta code from another language.

2.2 Categorisation and Comparison

Lilis and Savidis (2019) [2] categorise 110 programming languages. It focuses on older languages and does not analyse more recent languages such as Go, Rust, and Jai, despite them already existing at the time of publication. However, due to the incredibly broad overview and clear categorisation, it is still relevant.

The first category is the "metaprogramming model", which broadly groups programming languages by their support for macros, reflection, multi-stage programming, generative programming, and two (less relevant) subcategories for runtime metaprogramming: meta object protocols and aspect-oriented programming. Yune only has support for macros and multi-stage programming. Note that "generative programming" in this case refers to generative code that is clearly separated from regular code, unlike macros.

The second is the "metaprogram evaluation phase", which is indicates when metaprograms

are evaluated: preprocessing-time, compilation-time, or execution-time. It also relates to whether the metaprogram is interpreted or compiled. Yune macros are evaluated at compile-time and all metaprogram code is compiled.

Third is the "metaprogram source location", either embedded or external, with relations to context-aware, context-free, and global generation. The latter referring to a metaprogram being able to inspect and modify the entire program during evaluation. Yune macros are embedded and typically seen as context-free, as they cannot freely inspect the code surrounding the invocation site. However, they can request type information of variables in the environment, making them partially context-aware.

Last is the homogeneity of the metalanguage. Yune's metalanguage is "indistinguishable" from its host language, because it uses "the same constructs through the same syntax" as the host language. Alternatives are that the metalanguage "extends" the host, introducing constructs such as quasi-quotation and annotations that mark code as executed at compile-time, and that the metalanguage is "different" from the host language, namely macros in C and templates in C++.

Khalass [11] supplements [2] by describing the metaprogramming systems of 8 modern languages: C, C++, D, Python, Go, Jai, Rust, and Sparrow. Only the first four of these were discussed by Lilis and Savidis.

It describes how D's lack of a preprocessor forces the compiler to natively support certain features. Specifically, it notes that conditional compilation has its own syntax and semantics, and that D's version of macros splice code in the form of string constants into the program. The author also shows the power of these features when combined with compile-time code execution. In D, a subset of the language can be executed at compile-time. Furthermore, it compares D's templates (which are semantically closer to generics) to C++'s and states that D's solution solves several issues with C++'s approach. Yune also does not have a preprocessor, but it can emulate conditional compilation using macros. A macro can, for example, omit specific code if an environment variable is set. Our language is capable of executing any code during compile-time, unlike D.

We analyse Sparrow in Section 2.3 because its authoritative source is an academic paper, and other relevant languages from the thesis that do not provide papers in the following sections: C++ in Section 2.8.1, Rust in Section 2.8.2, and Jai in Section 2.8.4.

2.3 Sparrow

Teodorescu [3] introduces the Sparrow programming language and describes its metaprogramming system, as well as its performance, empowered by compile-time code execution. It provides many features for compile-time metaprogramming, and there is some common ground in design and philosophy with Yune. Both languages are capable of achieving C++-levels of performance, place importance on embedding DSLs, and have DSLs convert strings to ASTs (instead of tokens to tokens, as in Rust, for example).

Sparrow names itself a descendant of C++ and Scala, which is evident in its design and implementation. It has the same emphasis on performance as C++, as it compiles to the LLVM intermediate representation and focuses on moving computations to compile-time. C++ templates are replaced by Sparrow generics, and `constexpr` is superseded by more permissive compile-time expressions.

The inspiration from Scala lies in the domain of operators, specifically user-defined infix operators.

However, the operators follow Haskell's style more closely than Scala's, since users can declare precedence, which is fixed based on the first character of the operator in Scala.

Sparrow's macros follow the Lisp style of functions that take the ASTs of their arguments and output another AST. This tree is created using the Sparrow compiler API, which various internal functions for metaprogramming use. The primary limitation of Sparrow macros, which Yune overcomes, is that they are not guaranteed to generate hygienic code.

While macros exist, due to their limited syntactic capabilities, Sparrow prefers embedding DSLs as string literals. Teodorescu also illustrates the applications of operator overloading for embedding DSLs, described in Section 2.7.7. Quasi-quotation is supported, simplifying code generation. DSLs can contain Sparrow expressions if the specific DSL parser supports it, which allows variables from outside the DSL to be used inside. The DSL compiler must parse these expressions itself, although the compiler API provides helper functions for it. This decision increases the number of DSLs that can be embedded without syntax changes, as the DSL can work around conflicts with the Sparrow grammar by using special symbols. For example, to reference a variable outside the DSL, `$variable` can be used instead of `variable` if `variable` can be the name of a variable in the DSL itself. We have adopted this DSL-specified syntax philosophy.

2.4 Converge

Tratt (2008) [4] discusses the state of metaprogramming-based EDSLs and describes a macro system for EDSLs in the Converge programming language similar to Yune's. It provides valuable insight into the required features and user-friendly behaviour of macro-based EDSLs.

It first introduces DSLs, emphasising their expressiveness in their respective domains compared to general purpose languages. Then it states that standalone DSLs gradually sacrifice quality in favour of general programming features, which is not an issue for EDSLs due to their ability to borrow features of the host language. This makes EDSLs easier to implement than standalone DSLs, particularly for homogeneous embeddings where the DSL implementation is written in the host language. Conversely, heterogeneous systems have increased flexibility.

Later sections introduce Converge and describe how *DSL blocks* can be used to embed arbitrary DSLs. Tratt describes Converge's support for quasi-quotation, compile-time code execution, and compile-time AST construction. It is also stated that Converge's standard library contains a parser-generator with Converge's grammar rules built-in, which can be used to parse DSLs in DSL blocks and simplifies embedding Converge expressions in DSLs.

Figure 3 shows an example of a DSL block and DSL compilation function. The function that translates the parse tree to a Converge AST is omitted as it uses a standard approach. Particularly of note are the similarities between Converge's and Racket's approaches for embedded DSLs. Both languages have DSLs that use the languages' parser-generators and compile to the high-level languages themselves.

Yune does not provide an API for using its parser because the parser and compiler are written in Go, and making it possible to call Go code from Yune would be non-trivial due to Go's runtime. Alternatively, the parser could be rewritten in C++, which would be easy to integrate with. However, both approaches were deemed to be beyond the scope of this thesis.

A notable feature of Converge's metaprogramming system is its error tracing. As can be seen in Figure 3, a DSL compilation function takes a `src_infos` argument, which contains information about the location of the specific function invocation (i.e. the DSL block), and can be extended

```
func macro(dsl_block, src_infos):
 // Calls Converge's parser with, dsl_block text, src_infos,
 // DSL keywords, and grammar rules.
 parse_tree := dsl_parse(dsl_block, src_infos, ["keyword"], [".."],
  GRAMMAR, "macro_rules")
 return parse_tree_to_Converge_AST(parse_tree)

// The "macro" function is called with the text in the indented block
// and the source location of the text as arguments.
$<<macro>>:
 This is text parsed by the macro.
 It can span multiple lines.
```

Figure 3: An example of a DSL in Converge.

by the DSL author with information about the current compilation step. The extra information is used by the Converge compiler to provide a meaningful backtrace in case of an error during compilation. For the end user, this makes it significantly easier to debug errors caused by macros and macro-generated code. Tratt argues that a backtrace reduces confusion when the error is caused by a bug in the macro.

Yune also has Converge-style backtraces for macros, though we do not allow the user to add information to the backtrace as they can simply print to the standard output when needed. We solve the accountability problem by declaring that a macro that does not report errors must generate correct code. The result is complete removal of ambiguity (unlike in Converge), but increased semantic analysis requirements for the macro author, which we deemed a justifiable tradeoff because DSLs typically require little semantic analysis.

Three important differences with Yune are that Converge is a dynamically typed language, has a runtime, and appears to have no interface for communication with other languages. The lack of static typing implies that it has fewer guarantees of compile-time correctness than Yune — a particularly notable issue for macro-generated code — and code written in Converge cannot be reused from another language, nor can Converge use libraries written in other languages, which significantly limits what can be expressed in Converge. Adding a foreign function interface to the language is possible, but complicated by its runtime.

2.5 Template Haskell

Template Haskell (TH), as introduced by Sheard and Peyton Jones [12], has a feature-set similar to Converge. It replaces DSL blocks with an extension to traditional quasi-quotation, with the user being able to specify a custom parser for a quasi-quotation in order to embed DSLs. The TH library provides parsers for Haskell expressions, types, and declarations. Quasi-quotations have the syntax `[parser| a + b or anything else! |]`, making them explicitly delimited with fixed syntax. No leading whitespace is removed from lines in multi-line quasi-quotations, unlike in Yune and Converge, where leading whitespace is removed according to the indentation level. Custom

parsers and other TH components are written in (Template) Haskell, therefore TH is a homogeneous metaprogramming system.

Unlike Converge, existing code can be inspected and modified with TH, a process called reification. Yune supports only first-class types (which can be considered a reification feature) as full support was deemed out-of-scope.

As TH is an extension of Haskell, it can interoperate with other languages via Haskell's C interface, though it also inherits the Haskell runtime. Unlike Yune, TH is a functional language, which significantly complicates operations involving state, including state transfer between metaprogram invocations.

2.6 Language Workbenches

Language-oriented programming helps with programmer productivity by allowing users to write syntax suitable for their domain. It is empowered by tools for each phase of the language compilation pipeline. A *language workbench* contains a complete set of tools for the entire development process. We discuss three workbenches with varying approaches: Spoofax, Jetbrains MPS, and Racket. Traditional workbenches (e.g. Spoofax and MPS) significantly speed up prototyping, but tend to be limited in their expressivity. However, most DSLs can still be expressed easily using a workbench due to their simplicity. Macro-based workbenches (e.g. Racket) can theoretically emulate all functionality provided by a traditional workbench while mitigating its flaws, though code generation is limited to the host language.

2.6.1 Spoofax and Stratego

Spoofax [13] provides DSLs SDF, Statix, NaBL, and Stratego for specifying the various components of a compiler, significantly reducing the amount of code required and consequently speeding up the development process. It uses SDF to generate a parser, Statix for static semantic analysis, NaBL for scoping and namespaces, and Stratego for AST transformation and code generation. The workbench also contains tools for adding IDE integration.

SugarJ [14] and SugarHaskell [15] use Stratego (and SDF) to add "syntax extensions as libraries" to Java and Haskell, respectively. This indicates its relative importance for the workbench, and shows that Stratego can perform both semantic analysis and code generation.

2.6.2 Jetbrains MPS

The Jetbrains MPS language workbench [16, 17] follows similar strategies for semantic analysis and code generation as Spoofax, but the parser-generator is vastly different. MPS does not define textual grammars as most languages do, instead declaring the rules of a *structured editor* for the DSL. Structured editors, also known as projectional editors, have users edit the AST directly instead of the underlying text. This completely eliminates syntactical errors, and allows graphical elements such as tables and diagrams, to be natively implemented without the textual abstraction required by languages such as Markdown and HTML.

2.6.3 Racket

Racket is a programming language in the Lisp family with powerful metaprogramming features [9]. While Racket belongs to the category of language workbenches as it provides all the tools for developing a DSL, it does not describe itself as one. This is likely because the method of developing DSLs in Racket differs a lot from the method used by traditional language workbenches. Instead of producing a fully-featured, standalone language, Racket DSLs compile to high-level Racket code. The new languages are therefore reliant on the Racket runtime, but have the benefit of flawlessly interoperating with existing Racket code.

Interestingly, because Racket DSL compilers generate Racket code, they can also be considered macros. In fact, Racket's compilation model ensures they are compile-time macros, unlike other languages in the Lisp family. This has the previously mentioned benefits of static semantic analysis. However, as Racket is a dynamically typed language, there are still fewer static guarantees of error-free code than in our language. Racket's sister language *Typed Racket* is gradually typed [18], meaning that it performs type checking when the programmer has added type annotations, but does not require them.

2.6.4 Yune

Our language does not provide tools for DSL creation, which means that it is not a language workbench. We only provide the foundation for creating a DSL embedding in our language (macros), not the convenience of a workbench. However, it is possible for a library to implement utilities for DSL development to create a macro-based language workbench similar to Racket. Such a library may contain macros for embedding Spoofax's DSLs in order to concisely define DSLs, combining the advantages of both Racket and Spoofax.

2.7 Macro Alternatives

Few languages natively support powerful macros due to the complexity of their implementation. In such cases, it is still possible to generate code or embed DSLs through other means, which we discuss in this section.

2.7.1 External Preprocessors

An external preprocessor does not increase compiler complexity as it is not part of the compiler [2]. This inherently means that they have a limited degree of integration with the compiler itself, so code generated by the preprocessor is often not analysed syntactically or semantically. The extreme case is lexical preprocessors, such as the one powering C, which does little more than manipulating text. It has no awareness of the underlying language. This also means that it can be used to generate code in a variety of target languages. Syntactic preprocessors are aware of the syntax of the target language and do not violate it, reducing errors.

2.7.2 Build Scripts

Unlike preprocessors, build scripts can execute code to generate code in the target language instead of merely performing transformations on existing code. A build script can be written in any language,

allowing the use of tools written in different languages from the target. Some build tools, such as Rust's Cargo and Jai's compiler, natively support homogeneous build scripts [6, 7].

Build scripts can be used when a language's native metaprogramming capabilities are not powerful enough, and the metaprogramming task can be sufficiently separated from the host language. Some well-known tools used for this purpose are Gradle, Maven, Makefiles, and shell scripting languages such as Bash, Batch, and Powershell.

2.7.3 Transpilers

Transpilers, similarly to build scripts, are independent programs that generate source code. They translate code written in one language to another at a similar level of abstraction, such as from Rust to C++ or JSON to XML. Transpilers provide a unique opportunity for cross-language code reuse.

2.7.4 Manual Parsing

Adding the ability to inspect code as data to a language involves allowing the user to request information about each code element in a language, including how it is understood by the compiler and its relations to other code. This can be non-trivial if the compiler does not have this information readily available. Odin, as an example [19], therefore does not expose this functionality directly. However, it provides access to its internal parser as part of its core library, allowing users to parse Odin files at runtime without making the compiler more complex. Moving the inspection logic to a build script allows for emulating compile-time reflection. Note that this is trivial for Odin because the compiler is also written in Odin (i.e. it is bootstrapped). If the compiler is written in a different language, exposing a parser API can be much more difficult.

2.7.5 Generics

Generics are a method of writing code that can handle several data types. While much less flexible than true metaprogramming, generics are easier to implement and provide safety beyond their more general counterparts' abilities. They can be seen as a specialised variant of *templates*, which perform textual substitution with few restrictions [20]. Generics instead operate on constants (primarily types), of which it is ensured that they are valid in the context where they are used. They cannot perform complex computations, unlike (C++) templates, which means that it can be proved that a generic function passes type checking for all possible inputs. Rust uses *traits* to describe what methods a generic type has [6].

2.7.6 Reflection

Reflection enables code in a programming language to inspect and modify itself [2]. This can be performed either at runtime or compile-time. The latter has better runtime performance as the computations are performed at compile-time, but is less common despite this because it requires some form of compile-time code execution. Runtime reflection requires the presence of information about the program and its components at runtime for inspection, which interpreters already have, making runtime reflection significantly easier to implement for interpreters than statically compiled languages.

```
std::string name = "World";
std::cout << "Hello, " << name << "!" << std::endl;
```

Figure 4: Operator overloading used by `std::cout` in C++.

2.7.7 Operator Overloading

A common trick for embedding DSLs is operator overloading. Existing operators can be used to represent syntactical features in the DSL. In statically typed languages, the compiler can verify type-correctness of the DSL at compile-time. The restriction in syntax of using this method can be reduced if the language supports user-defined operators.

Notable programming languages featuring operator overloading are Python, C++, Scala, and Haskell. Python and C++ only support overloading existing operators, while Scala and Haskell even support user-defined operators. Haskell, in particular, makes extensive use of operators when representing complex concepts in the language, such as monads and functors. Almost unique to Haskell, operator precedence and fixity (prefix/infix/postfix) can be specified. Figure 4 shows how C++ overloads the left-shift operator `<<` for writing to a stream, in this case the standard output.

In addition to operator overloading, C++ supports implicit conversions between types, including the implicit construction of objects, such as from initializer lists. These lists can hold arbitrary types and have simple `{ a, b, c }` set notation, making it suitable for representing DSLs that use this syntax, of which JSON is a prime example. This is further discussed in Section 5.

2.8 Modern Programming Languages

This subsection discusses C++, Rust, Nim, Jai, Nemerle, and Rocq, which each have a unique set of features relating to macro-based metaprogramming and EDSLs. For Nim, Jai, Nemerle, and Rocq, academic sources do not accurately describe the current state of the language, so official sources are used to supplement.

2.8.1 C++

C++ is the most commonly used language we discuss, which makes it a great reference point for other languages [2, 20]. C++ has templates for generic metaprogramming, limited compile-time code execution with `constexpr` and `consteval`, and since C++26, compile-time reflection [20].

Templates enable code generation in a more general way than generics, as they can be used for complex computations, static analysis, and more intricate substitution than C macros. The significant drawbacks of templates are its cascading error messages and that it is an entirely different language from C++ with its own syntax and semantics that the user has to learn.

The compile-time code execution facilities in C++ prior to C++26 were limited to code without side-effects, which means that it was not possible to parse a DSL embedded in a string literal at compile-time, or manipulate code in other files. While C++26 still restricts compile-time code execution, it now has functions for file reading and memory allocation, meaning that DSLs can be parsed at compile-time.

Variants of C++ have emerged that resolve a subset of C++'s flaws and limitations. Most

notably, the Circle programming language [22] extends C++17 using arbitrary compile-time code generation and compile-time code execution using a minimal interpreter. It also adds other convenience features and ensures memory safety using Rust's memory model.

2.8.2 Rust

Rust is a modern language with the core philosophy of preventing memory-related errors without relying on a runtime [6]. It has a macro system that manipulates source-code tokens and limited inline compile-time code execution that does not rely on macros, similar to C++'s `consteval`.

The macro system consists of two variants: `macro_rules!` (the `!` is part of the name) and *procedural* macros. The former is uses a regular expression-like syntax to match and generate code, and ensures partial hygiene by differentiating between captured and fresh declarations' names based on the context. Note that hygiene guarantees do not apply to special constructs such as lifetimes and generics.

Procedural macros instead use *procedures* (i.e. normal Rust functions) to manipulate token streams. These must be placed in a separate "procedural macro library" to be used as they must be compiled independently of the main project, unlike `macro_rules!` macros that can be declared in the same file as they are used. They also do not have the same hygiene guarantees.

Both macro variants can produce syntactically invalid code as they output token streams instead of ASTs. Rust's macro syntax is delimited by a fixed set of characters: `()`, `[]`, or `{}`. To reduce the number of possible conflicts with DSL syntax, Rust requires that the delimiters are *balanced*, meaning that every opening delimiter is matched with a closing delimiter in a macro invocation, unless the opening delimiter is part of a string literal.

2.8.3 Nim

Nim is a feature-rich, modern language with many metaprogramming features [8]. Unlike C++ and Rust, Nim relies on the garbage collectors in its runtime for memory management. Nim's hygiene requirements roughly follow that of Rust. It has templates, which are comparable to C++ templates, but with the same hygiene guarantees as Rust's `macro_rules!`. The language also has macros comparable to Rust's procedural macros, although these are not compiled and can therefore only execute a subset of Nim code. The advantage of these interpreted macros is that they can be declared in the same file as they are used, while Rust requires them to be in a separate library so that they can be compiled independently. Additionally, they can access type information of their arguments. The most significant difference with other languages is that Nim templates and macros cannot be customised syntactally. This is not as limiting as it appears, because single-argument macros can omit the delimiters and a block of statements can be passed as final argument.

2.8.4 Jai

Jai is a systems programming language inaccessible to the public, including us. We discuss it due to the relevancy of its metaprogramming system, but note that the most credible source available is a community-maintained wiki [7].

The language supports interpreted compile-time code execution similarly to Nim using `#run` code blocks, quasi-quotation of expressions with `#code`, and textual code insertion using `#insert`. These can be combined for arbitrary code generation. The code to be executed can depend on

```
macro(macro())
            ^ detected as end of
              first macro
```

(a) Nemerle

```
// The macro named 'macro' is
// invoked with the text "macro#".
macro#macro#
```

(b) Yune

Figure 5: Examples of a recursively invoked macros.

compile-time variables, including generic type parameters. The introduction of side-effects makes it impossible to determine if a generic function is error-free for all possible type parameters[1], which forces deferred semantic analysis, meaning that analysis happens when the compiler encounters a function invocation, rather than a definition, resulting in worse error detection and reporting. The `#code` macro can also only be used to create Jai expressions, splicing is not supported.

Compared to Yune, Jai is more permissive with regards to where code can be inserted, as Yune only supports code generation in expressions, for reasons discussed in Section 4.2, and Yune does not have generic functions. Both languages can dynamically construct types using regular code at compile-time.

2.8.5 Nemerle

Nemerle is a programming language heavily inspired by C#, even promising that all C# code is valid Nemerle code. Nemerle supports for quasi-quotation and syntax extensions [10] on top of this. The extensions are implemented as scoped modifications to the Nemerle parser. This mixes compilation phases even more than regular macros, complicating the compiler's internal structure. Yune and other languages previously discussed restrict custom syntax to the body of a macro invocation, which means that the macro needs to be explicitly invoked, while in Nemerle the syntax extension only needs to be in scope to be applied. The positive consequence is that extensions can often be trivially combined, while the negative consequence is that they can clash, which is difficult to resolve. The latter is why the XML macro shown in Figure 6 still uses the prefix `xml` to indicate the start of a macro.

Syntax extensions can be used to emulate Rust-style macros, including performing variable capture. Nemerle conventionally uses a different pair of delimiters for each macro to minimise the restriction on syntax. The downside of this approach is that macros following this convention (see Figure 5a) cannot contain their own invocation, which Yune macros do support (see Figure 5b). Nemerle's approach reports an unexpected symbol '`)`'. As discussed in Section 2.8.2, Rust resolves this by requiring that delimiters are balanced.

Writing a parser extension, such as the XML macro in Figure 6, requires a lot of code. Nemerle therefore provides a convenient shorthand (confusingly called a `macro` in Nemerle) similar to Rust's `macro_rules!`. An example is shown in Figure 7.

Besides the metaprogramming system itself, Nemerle differs from Yune in that it has a heavy runtime (the .NET virtual machine), which makes it more difficult to call Nemerle code from a language outside of the .NET ecosystem. However, unlike Converge, .NET does provide a foreign

[1]The generic function can, for example, report an error only if the time is exactly `15:42`, which the compiler cannot make any assumptions about.

```
def title = "Programming language authors";
def authors = ["Anders Hejlsberg", "Simon Peyton-Jones"];

def html = xml <#
  <html>
    <head>
      <title>$title</title>
    </head>
    <body>
      <ul $when(authors.Any())>
        <li $foreach(author in authors)>$author</li>
      </ul>
    </body>
  </html>
#>
Trace.Assert(html.GetType().Equals(typeof(XElement)));
WriteLine(html.GetType());
```

Figure 6: Official example of the usage of Nemerle's XML macro [10].

```
macro while_macro (cond, body)
syntax ("while", "(", cond, ")", body) {
  <[
    def loop () {
      when ($cond) {
        $body;
        loop ()
      }
    }
    loop ()
  ]>
}
```

Figure 7: Official example of a while-loop macro definition in Nemerle [23].

function interface that enables calling C functions.

2.8.6 Rocq

Rocq (formerly known as Coq) is a mathematical proof-oriented programming language. As such proofs are complex and contain many repetitive components, metaprogramming is a suitable means of reducing verbosity. Rocq features "notations": syntax extensions that behave the same as Nemerle "macros".

In practical use, Rocq differs significantly from Yune, as Rocq is not intended to be used as a regular programming language, indicated by its inability to execute code with side-effects. While executable Rocq code can be converted to other languages such as OCaml, Haskell, and Scheme, most Rocq code is only verified, not executed.

MetaRocq (formerly MetaCoq) is a Rocq plugin that adds metaprogramming features such as quasi-quotation and AST transformations [24].

2.9 Comparison

Table 1 presents an overview of the metaprogramming features discussed, partially based on the categories discussed by Lilis and Savidis [2]. As shown in the table, Yune's metaprogramming system is similar to Converge's, with the primary differences between the languages being that Yune emulates quasi-quotation using macros instead of having built-in syntax for it, analyses the generated code more strictly using static typing, and does not rely on a runtime, which makes it easier to use Yune code from another language. Compared to Sparrow's, Yune's macros are less error-prone due to hygiene guarantees, and more flexible syntactically due to indentation-based delimiters. Both languages have static typing and no runtime. Racket, Template Haskell, and Nemerle also have macros similar to Yune's, but calling code in these languages from another can be difficult due to their runtimes. Rust, Jai, Nim, Delta, and Rocq differ from Yune in various ways. None of their macro systems perform the same text-to-AST conversion that Yune macros do.

3 Methodology

To shape the language and determine how the macro system should be implemented, we researched the relevant literature. For this, we focused academic research and the metaprogramming systems of C++, Rust, Jai, Racket, Nim, Nemerle, and Rocq.

Initially, we set only a few short-term goals in order to create an extendable starting point. Primarily, we needed to implement a starting grammar able to describe a simple program with a macro definition and invocation, as well as arithmetic operations, variable manipulation, and function calls. We added a simple test file for validation.

After creating a basic general-purpose language, we started working on the more complex task of implementing the macro system, as well as writing the thesis. For the duration of this task, we iteratively designed, implemented, and tested components of the language, only adding those required to support the current iteration of the macro system. The macro system itself therefore gradually grew in expressiveness until its current state. Additionally, once single-file testing became cumbersome, we added a basic C-style import system, allowing us to factor out shared logic from individual tests, which formed Yune's standard library.

Language	Compile-Time Code Execution	Hygienic Macros	Quasi-Quotation	Macro Input	Macro Output	Macro Delimiters	Runtime-Free	Typing
Yune	Y	Y	E	Text	AST	Ind	Y	S
Converge	Y	Y	Y	Text	AST	Ind	N	Dyn
Sparrow	P	N	Y	AST	AST	F	N	S
Racket	Y	Y	Y	AST	AST	F	N	Dyn
TH	Y	Y	Y	Text	AST	F	N	S
Nemerle	Y	Y	Y	Text	AST	DSL	N	S
Rust	Y	P	E	Tok	Tok	B	Y	S
Jai	P	Y	P	Text	Text	F	Y	S
Nim	P	N	E	TAST	AST	Imp	N	S
Delta	Y	Y	Y	AST	AST	DSL	N	Dyn
Rocq	Pure	N	Y	AST	AST	F	N/A	Dep

Abbreviation	Meaning
Y	Yes
N	No
P	Partial
E	Emulated
N	Native
Ind	Indent
Imp	Implicit
B	Balanced

Abbreviation	Meaning
Tok	Tokens
TAST	Typed AST
F	Fixed
DSL	DSL-specific
S	Static
Dyn	Dynamic
Dep	Dependent

Table 1: A comparison table of modern metaprogramming technologies with a focus on DSLs.

```
Complex: Type = (Float, Float)
i: Complex = (0.0, 1.0)
```

The code used to generate the dependency tree.

Nodes are either declarations or code to be executed, and edges are dependencies between nodes. A declaration depends on its type and value expressions. An expression depends on declarations it names. Each layer in the diagram indicates a compilation stage.

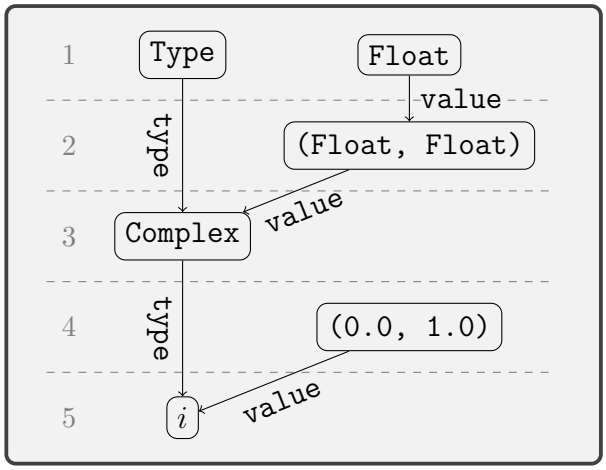


Figure 8: A 5-stage dependency tree visualisation of Yune code.

Finally, we implemented and wrote about DSLs for JSON, a subset of SQL, and string interpolation. As these were more complex than the simple test cases, they exposed some shortcomings of the macro system, which we resolved or documented. In this final stage, we also improved C++ interoperation.

4 Yune

Yune is the name of our novel programming language. It features a hygienic, homogeneous macro system that can embed domain specific languages without restricting their syntax. Macros, which are written as regular Yune functions, have clear location-based error messages. Types can be generated by arbitrary code. Yune compiles to C++, which allows for trivial C++ interoperability as shown in Section 4.8. This section discusses important features and solutions to problems encountered during development.

4.1 Stages

The first problem encountered was to determine an analysis (and consequently macro-evaluation) order which ensures that all types are computed correctly, macros are evaluated after their dependencies, and the entire program is analysed.

Initially, before the introduction of macros, running any code at compile-time required creating a C++ file, compiling it using `clang++`, and running it. To maximise compilation speed, it was important to batch pieces of code that could be compiled together. For this purpose, we constructed a directed acyclic graph of all top-level definitions and type expressions, separated into stages based on compilation order. An example is shown in Figure 8. Types of declarations are always evaluated before their values. For recursive functions, this is a necessity, as the function's type must be known before the function's body can finish type checking. This restriction on evaluation order was generalised to all declarations to simplify the implementation, at the cost of decreased complication speed. The number of stages in Figure 8 could be reduced to 3 without this restriction, as the type expressions of `Complex` and `i` could be evaluated at the same time as their value expressions. While

```
defineAlpha(text: String, getType: Fn(String, Type)): Code =
    name := "Alpha"
    type := getType("Beta") // waits until the type of Beta is known
    body := tupleExpression(0, [])
    variableDeclaration(name, type, body)

defineBeta(text: String, getType: Fn(String, Type)): Code =
    name := "Beta"
    type := getType("Alpha") // waits until the type of Alpha is known
    body := tupleExpression(0, [])
    variableDeclaration(name, type, body)

defineAlpha#()
defineBeta#()
```

Figure 9: A hypothetical cyclic macro dependency caused by top-level macros.

this restriction is still part of the latest version of the compiler, its impact on performance is now non-existent.

A second version constructed the stages as-needed instead of all in advance, which was required for the unpredictable code generation capabilities of macros, including macros generating other macros.

The final implementation greedily evaluates compile-time expressions one-at-a-time using the Clang-REPL C++ JIT.

4.2 Cycles

Adding top-level macros to a language creates the possibility for cyclic dependencies between macros, which can only be detected when they are evaluated (if at all) due their ability to execute arbitrary code. Figure 9 illustrates how two macros that dynamically retrieve the types of declarations defined by each other (using `getType`) cause a dependency cycle. Error handling is omitted for simplicity. Yune prevents such cycles by disallowing top-level macros, requiring that macros directly generate expressions, not (top-level) declarations. The generated code can still contain variable declarations, though only within closures as these create a new scope, preventing variable access from outside. Cycles in other forms, such as a declaration `A: A = A`, are detected by checking if a declaration is already being analysed when it is encountered.

4.3 Algebraic Data Types

Macros output either an error or a expression. Yune represents this using unions tagged by their type. In functional languages, this is referred to as a *sum type*. Unions are implicitly created from their variants, unlike explicitly tagged unions in Rust and similar languages. This introduces some issues, shown in Figure 10, which are resolved by flattening nested unions and eliminating

```
// Without deduplication, this triggers an error in C++, because
// the backing std::variant implementation does not allow duplicate types.
duplicate: Union[Int, Int] = 17

a: Union[Int, Union[Bool, Float]] = true
b: Union[Union[Int, Bool], Float] = false
// Without flattening, this triggers an error about type inequality.
a == b
```

Figure 10: Union issues in Yune.

```
// First version
macro(text: String): (List(String), Fn(List(Type), Expression))

// Second version
macro(text: String, getType: Fn(String, Type)): Union[Expression, String]

// Third version
GetType: Type = Fn(String, Union[Type, ()])
macro(text: String, getType: GetType): Union[Expression, (Int, String)]
```

Figure 11: The three major versions of Yune's macro function signature.

duplicate types.

Besides sum types, all mainstream programming languages also have types for modelling values that store multiple smaller values of different types (i.e. *product types*), such as *structs*, *records*, and *tuples*. We chose to implement tuples, which typically have unnamed fields, though there is no strong argumentation for this decision.

4.4 Macro Function Signature

In Yune, for the purpose of homogeneity, macro declarations are regular functions with a specific signature (i.e. type). This signature has evolved alongside the compiler, through three major versions, as shown in Figure 11.

The first version was created when the compilation stages were clearly separated, which we discussed in Section 4.1. The signature should be read as "define a macro named `macro` that takes the text it is invoked with and outputs a list of variable names, as well as a function that generates an expression given the types of the named variables". The macro requests the types of all relevant variables at once instead of one-by-one in this version, because it allows the compiler to compute the types of the relevant variables in a single compilation stage (unless prevented by dependencies between them), which would otherwise require a number of stages equal to the number of relevant variables, significantly slowing down compilation. This version's signature requires that the macro

```
Number : Type   = Int
TAU    : Number = 628318 // Analysis depends on the value of Number.
Natural: Type   = Number // Analysis depends on the type of Number.
```

Figure 12: Example of a type and value dependency in Yune.

declaration function is able to return another function, possibly with some state attached to the returned function. A closure is precisely a function with state, which is why we originally added them.

The second and third versions differ from the first in that the user must now request the types of relevant variables one-by-one using the `getType` function, which is provided as a parameter. The second version was created soon after the transition to using a JIT compiler for compile-time code execution, when the performance drop of having many stages became negligible. If the type of the variable requested using `getType` has not yet been computed, then the compiler stops executing the macro and compiles the relevant variable first. Besides `getType`, macros gained the ability to return error messages, which is the `String` variant of the return type.

The final version improves the error handling in macros by making `getType` return either a `Type` or nothing "`()`" if the requested variable does not exist, and by allowing the user to indicate the location of an error in terms of the number of characters since the start of the text.

4.5 Out-of-Order Top-Level

Yune's top-level is out-of-order, which means that global functions and variables can be accessed before they are declared. Initially, this was intended to simplify the definition of recursive types, though these are not supported in the current version. However, in the more general case, this still ensures that mutual recursion between top-level declarations is trivial, since forward-declarations as in C/C++ are not required. Rust implements an out-of-order top-level by collecting all globals' names and types first, and only afterwards performing semantic analysis in a second pass [6]. Our compiler must execute code to compute types in the same pass as analysis of the code to be executed, making such a structure infeasible.

Each global declaration first has its declared type semantically analysed and evaluated. Other top-level declarations that require only the type of this declaration can now be analysed and executed. Globals are required to have an explicitly declared type to allow for this. Following this, the global's body is analysed. This two-step algorithm enables the definition of (mutually) recursive functions.

Figure 12 shows an example of the different kinds of dependencies in Yune. `TAU` uses `Number` as its type, so in order to perform type checking on `TAU`, the value of `Number` must be known. Diversely, `Natural` is an alias of `Number` and only uses `Number` in its body, so the type of `Natural` can be computed independently of `Number` (because its type is declared, not inferred). However, to evaluate `Natural`, the value of `Number` must be known.

Typically, the evaluation order of top-level declarations is undefined, though macros requesting type information using `getType` can force their analysis and evaluation.

```
There is a bug in macro `invalid`.
Error-less invocation at 8:4 produced invalid code. Error:
Binary operator + cannot be applied to expressions of types 'String' and 'Int'.
---> file.un line 8 column 13
8 |      invalid#[ignored]
  |              ~
Macro trace:
  `invalid` defined in file `file.un` at 4:0
```

Figure 13: An example of an error message shown when a macro generates invalid code in Yune.

4.6 Macro Semantics

Code generated by macros must be semantically analysed, which is done immediately after substitution. A stack tracks the macros that generated an expression in order to display a macro "stack trace" when a macro-generated expression causes an error. An example is shown in Figure 13. The compiler indicates that the macro is at fault when it produces invalid code without reporting an error.

Hygiene is an important characteristic of Yune macros, enforced through language design rather than semantic analysis. The `inject` function directly injects a value into an expression, which is unambiguous regardless of the macro invocation site. A function's value has the `::` namespace qualifier (which indicates a global) added in the generated C++ code, preventing ambiguity with locals of the same name. Ambiguity with globals is avoided by ensuring that each global has a unique name.

4.7 Compile-Time Value Embedding

Globals are all computed at compile-time to improve performance, following the same argumentation as Sparrow's [3]. To embed the values of globals in the final compiled binary, they must be converted from how they are stored in memory by C++ to a representation that can be understood by the compiler, which is non-trivial for closures. Named functions can be uniquely identified by their names and have no state, so they are simply represented by their names. Closures have no name, so a unique identifier is generated based on their location in the source-code. Their state consists of captured variables, which are serialised alongside the closure identifier. A C++ `class` is created for each closure that provides access to its captured variables (as C++'s built-in closures are black boxes), which in turn requires specialised code generation and tracking of variable captures.

4.8 Runtime-Free and Interoperation

Yune uses C++23 as compilation target and does not include a garbage collector or interpreter at runtime. In order to link against the compiler-generated C++ code, either a C++ compiler can be used, which automatically adds the necessary libraries and linker flags, or the arguments can be specified manually for use with a generic linker. This follows the same procedure as linking regular C++ code and will therefore not be described in detail. Most critically, the C and C++ standard

```
`
#import <string>
std::string str{"Hello, World!"};
`

STRING: String = `str`

main(): () =
    printlnString(STRING)
```

Figure 14: C++ code injection in Yune.

libraries need to be linked with the arguments `-lstdc++ -lc` if not using a C++ compiler.

Yune projects without a `main` function are compiled into C++ header files, which can be trivially integrated with C++ projects or compiled into a dynamic library file for use with a different language. Users can also directly inject C++ at the start of a Yune file and as expressions inside a backticked quotation. This enables arbitrary usage of C++ libraries from Yune code, enabling bidirectional interoperation with C++. An example is shown in Figure 14.

4.9 Unrestricted DSL Syntax

To support unrestricted DSL syntax, indentation is used to indicate what the body of a macro is. This requires an indentation-sensitive parser, meaning that our grammar cannot be context-free. As a result, our language must use a custom ANTLR lexer to track indentation. The grammar itself is also more complex as newlines must be handled explicitly instead of being treated as whitespace.

An example is shown in Figure 15a. The start of the macro is indicated with `#` and `macro` is the name of the macro called. An alternative syntax we considered is presented in Figure 15b, which only requires that the parser is aware of newlines, similar to the Go parser. The difference is that indentation is explicitly indicated. Both Yune and the proposed alternative can trivially strip leading indentation from the macro body, which is not the case for the C-style macro in Figure 15c.

4.10 Type Inference

The compiler implements type inference to the same degree as Go, meaning that the types of local variable declarations can be inferred, but not of functions (including closures) and global variables. It is theoretically possible to infer the type of any value except of recursive functions, as discussed in Section 4.1. However, we choose not to do so because we believe that explicit types on functions and global variables improve readability.

In most instances, the type of an expression can be inferred using its components and the variables in the environment, with a few exceptions. Yune does not differentiate syntactically between a tuple containing types and `Type` itself, which means that in order to differentiate them semantically, the context must be taken into account. This is illustrated by Figure 16, which assigns two different types to the same value. Similarly, unions also cause ambiguity. A list of integers `[0, 1, 2]` may be typed as `List(Int)` or a list of unions; `List(Union[Int, String])`,

```
macro#
    This section
    is parsed as
    inside.
This is outside the
macro, as there is
no indentation.
```

(a) An example of Yune's macro syntax.

```
macro#
    | This section
    | is parsed as
    | inside.
This is outside the
macro, as there is
no "|" at the start
of the line.
```

(b) An alternative syntax for unrestricted macros.

```
#define macro \
    This section \
    is parsed as \
    inside.
This is outside the
macro, as there was
no backslash on the
previous line.
```

(c) A C-style syntax for unrestricted macros.

Figure 15: Several options for unrestricted macro syntax.

```
tuple: (Type, Type) = (Bool, Float)
type : Type         = (Bool, Float)
```

Figure 16: Type-tuple ambiguity in Yune.

for example.

4.11 Closures

Yune supports closures, which are unnamed functions that capture variables from their environment. Figure 17 shows an example. Functional languages often support them as a shorthand, and imperative languages (e.g. Rust, Java, C#) typically use them for callbacks and operations on streams of data. Closures have various applications in macro-generated code, as they are the only expression that can create a new scope, making it possible to declare variables. We prevent macros

```
list := [3, 4, 5]
y := 10
addY := |x: Int|: Int = x + y
added := map(list, addY)
map(squared, println)
// Output:
// 9
// 16
// 25
```

Figure 17: An example of a closure in Yune.

```
variable := 10
doStuff#
// Error: duplicate declaration
// 'variable'
```

(a) The code the user sees.

```
variable := 10
variable := 20
// Error: duplicate declaration
// 'variable'
```

(b) After macro expansion.

Figure 18: Error due to a macro-declared variable in the same scope.

```
N: Int = 90

// Create an expression that adds two numbers,
// injecting N from the environment.
addNumbers: Expression = quote#8000 + {N}

// Create a more complex expression by splicing
// an existing expression stored in a variable.
expression: Expression = quote#pow(2, 5 * {addNumbers})
```

Figure 19: An example showing how quasi-quotation can be emulated using Yune macros.

from directly declaring variables in the scope they are invoked, as the declared variables may have the same name as variables in scope, resulting in errors that are difficult to understand. This is illustrated by Figure 18. A language such as Rust that allows redefining variables in the same scope does not report an error in this scenario, which may introduce subtle bugs because the value of a variable unexpectedly changes. Closures also enable the expression of function definitions in macro-generated code, since macros cannot generate code at the top-level where regular functions are defined.

4.12 Quasi-Quotation

While Yune does not have built-in syntax for quasi-quotation, it can theoretically be perfectly emulated with macros. An example of a quasi-quotation macro (named `quote`)'s usage is shown in Figure 19. Rust uses the same approach, proving its effectiveness. Note that while it is possible to create, there is currently no implementation of the `quote` macro due to time constraints. The macro requires the presence of a Yune parser implemented in either C++ or Yune, which we do not have as our parser is implemented in Go. Due to Go's runtime, it is non-trivial to use the parser from C++ and, by extension, Yune.

4.13 Generics

Macros can be used to generate functions for arbitrary types, though not with the same convenience as regular generics. Figure 20 shows a hypothetical, simplified implementation of a

```
generateAdd(type: String): Expression =
    quote#add_$type(left: $type, right: $type): $type =
        left + right
```

Figure 20: A simplified example of a generic Yune function-generation macro.

macro that generates a function for adding two numbers, given the type of its arguments. This can theoretically be generalised to a macro that translates a superset of Yune with generics to regular Yune.

5 Experiments

Several experiments have been performed to compare the ease-of-use of DSLs among the languages previously discussed. Except for Converge, Rocq, and Delta, which do not implement a (macro-based) JSON, SQL, or string interpolation DSL, each language discussed so far will be analysed based on its implementation of at least one these DSLs. JSON in C++ will also be shown despite not being macro-based, as its implementation is interesting.

5.1 Syntax

5.1.1 Yune

The macros in Figure 21 embed their respective DSLs without syntax compromises and perform variable capture. Figure 22 shows the string interpolation example with the macro expanded. The `fmt` macro outputs an `Expression` in the form of a tree `binaryExpression`s, with as leaves either a `variableExpression`, which captures a variable given a string name, or a `stringExpression`, which is a string literal.

5.1.2 C++

Figure 23 shows an example from the Boost.JSON documentation [25]. The DSL embedding relies on *initializer lists* for the `{ "key", value }` syntax. The syntax resembles JSON, but is not exactly same, so existing JSON needs to be modified to be embedded in this way.

5.1.3 Nim

Figure 24a shows that Nim achieves a near-perfect JSON embedding. In this example, the macro converts the captured syntax tree to a valid Nim expression. It takes JSON objects made up of the combined syntax of sets (`{ element, element2 }`) and pairs (`key: value`), and JSON lists using Nim array syntax. It should be noted that embeddings of different languages, such as XML and SQL, tend to use string literals or method-call-expressions instead, as their syntaxes cannot be represented in Nim as easily, though supported syntaxes are trivial to specify. An alternative, Nim-style syntax for the JSON example is shown in Figure 24b.

```
pi := 3.141 // captured
value := json#
  {
    "pi": pi,
    "happy": true,
    "name": "Yune",
    "nothing": null,
    "answer": { "everything": 42 },
    "list": [1, 0, 2],
    "object": {
      "currency": "EUR",
      "value": 42.99
    }
  }
```

(a) JSON

```
class_name := "3B" // captured
query      := sql#
  SELECT first_name, last_name
  FROM students
  WHERE class_id = (
    SELECT id
    FROM classes
    WHERE class_name = $class_name
  )
```

(b) SQL

```
animal   := "chicken"
obstacle := "road"
result   := fmt#Why did the {animal} cross the {obstacle}?
```

(c) String interpolation

Figure 21: Yune macro examples.

```
animal   := "chicken"
obstacle := "road"
result   := "Why did the " + animal + " cross the " + obstacle + "?"
```

Figure 22: `fmt` macro generated code.

```cpp
float pi = 3.141; // captured
json::value value = {
    { "pi", pi },
    { "happy", true },
    { "name", "Boost" },
    { "nothing", nullptr },
    { "answer", {
        { "everything", 42 } } },
    {"list", {1, 0, 2}},
    {"object", {
        { "currency", "USD" },
        { "value", 42.99 }
    }}
};
```

Figure 23: Boost.JSON value construction using initializer lists.

```nim
let pi = 3.141 # captured
let value = %*
  {
    "pi": pi,
    "happy": true,
    "name": "Nim",
    "nothing": nil,
    "answer": { "everything": 42 },
    "list": [1, 0, 2],
    "object": {
      "currency": "EUR",
      "value": 42.99
    }
  }
```

(a) JSON example using the official macro.

```nim
let pi = 3.141 # captured
let value =
  macro:
    pi: pi
    happy: true
    name: "Nim"
    nothing: nil
    answer:
      everything: 42
    list: [1, 0, 2]
    object:
      currency: "EUR"
      value: 42.99
```

(b) General Nim indentation-based representation.

Figure 24: Nim JSON representations.

```
def pi = 3.141; // captured
def value = json({
  "pi": pi,
  "happy": true,
  "name": "Nemerle",
  "nothing": null,
  "answer": { "everything": 42 },
  "list": [1, 0, 2],
  "object": {
    "currency": "EUR",
    "value": 42.99
  }
});
```

(a) JSON

```
var className = "3B"; // captured
var query =
  from student in students
  where student.ClassId ==
    (
      from c in classes
      where c.Name == className
      select c.Id
    ).Single()
  select new
  {
    student.FirstName,
    student.LastName
  };
```

(b) SQL/LINQ

Figure 25: Nemerle macro examples.

5.1.4 Nemerle

As displayed in Figure 25a, it is possible to embed JSON in Nemerle without syntactic conflicts. This example uses the `Nemerle.Json` library that uses the `Nemerle.Peg` library internally to concisely create a parser. It should be noted that `Nemerle.Peg` only performs lexing and parsing, not parse-tree creation as ANTLR does, or semantic analysis and code generation as workbenches do.

Figure 25b shows an expression in C#'s built-in LINQ syntax [26], which Nemerle also supports due to its guarantee that Nemerle is a superset of C# [27]. LINQ is used for operations on "enumerable objects", which includes datastructures and databases. While it has a syntax similar to SQL, adapting an SQL expression does require modifications. It provides similar functionality to Rust and C++ iterators. For C#, LINQ is a special case, as the language does not have syntax extensions or built-in support for other DSLs, such as JSON.

5.1.5 Rust

The JSON example in Figure 26 uses the unofficial — but widely adopted — `serde_json` library. The macro represents a JSON object using its natural syntax, which results in a macro that is identical to Nemerle's and has the benefit of not modifying the compiler's parser. The string interpolation example in Figure 27 uses the standard library `format!` macro.

5.1.6 Template Haskell

Template Haskell's Aeson JSON library was used for this experiment. The macro (i.e. quasi-quotation) has a different syntax for variable captures than Yune and Nemerle, but is otherwise the same.

```rust
let pi = 3.141; // captured
let value = json!({
  "pi": pi,
  "happy": true,
  "name": "Rust",
  "nothing": null,
  "answer": { "everything": 42 },
  "list": [1, 0, 2],
  "object": {
    "currency": "EUR",
    "value": 42.99
  }
});
```

Figure 26: Rust Serde JSON macro [28].

```rust
let animal   = "chicken";
let obstacle = "road";
let result   = format!("Why did the {animal} cross the {obstacle}?");
```

Figure 27: Rust string interpolation example.

```haskell
value :: Value
value = [aesonQQ|
    {
      "pi": #{pi},
      "happy": true,
      "name": "Template Haskell",
      "nothing": null,
      "answer": { "everything": 42 },
      "list": [1, 0, 2],
      "object": {
        "currency": "EUR",
        "value": 42.99
      }
    }
  |]
  where
    pi = 3.141 :: Float
```

Figure 28: Template Haskell JSON Quasi-Quotation example using the Aeson library.

```
import WSI_Asm

test := $<<WSI_Asm::ab>>:
    R0 := 0
    L0:
    SWI printi
    R0 := R0 + 2
    IF R0 < 8 JMP L0
    SWI exit

test.new().exec()
```

Figure 29: Converge assembly example.

```
animal   := "chicken";
obstacle := "road";
sprint("Why did the % cross the %?", animal, obstacle);
```

Figure 30: Jai string interpolation example.

5.1.7 Converge

Converge lacks libraries for JSON and SQL, and string interpolation is a language built-in feature rather than a macro or syntax extension, so these cannot be used as an example of macro syntax, though the repository does provide DSL examples [29], one of which is shown in Figure 29. As is the case for Yune macros, the indented block clearly separates the body of the macro from the surrounding code for the both the programmer and compiler.

5.1.8 Jai

Jai does not have macro-based JSON and SQL libraries, so a string interpolation example is provided in Figure 30. Jai's syntax for this resembles C's `sprintf` syntax, though it likely does not have the same security vulnerabilities. A flaw of this syntax is that the variable to capture is passed as a separate parameter, meaning that the user must count arguments to determine which `%` corresponds to an argument, which becomes cumbersome for a large number of arguments. It is uncertain if `sprint` is a macro or regular function as the syntax is identical.

5.1.9 Racket

Following a similar philosophy to Nim's, Racket can represent most DSLs in a readable way using its Lisp syntax. SQL is used as an example because Racket's standard library does not contain a macro for embedding JSON. The `sql` macro uses named arguments for `from` and `where` to mimic SQL, which can also be used in other languages. For example, in Python:

```
(define class_name "3B") ;; captured
(define result
  (sql
    (select first_name last_name
      #:from students
      #:where (= class_id
                 (subquery
                   (select id
                     #:from classes
                     #:where (= class_name ,class_name)))))))
```

Figure 31: Racket SQL example.

```
Macro 'json' at 240:13 returned an error.
---> json.un line 246 column 24
246 |              "answer": { 5 "everything": 42 },
    |                          ~ Expected string or '}'
```

Figure 32: First Yune syntactic error

```
select("first_name", from="students").
```

5.2 Error Messages

In this subsection, we discuss several types of errors created by slightly perturbing the code in the previously-discussed examples. We will not discuss Nemerle and Converge because we could not use their compilers due to incompatibilities with the modern systems available to us. Jai is also omitted because the compiler is not accessible to the general public, including us. Long error messages are moved to the appendices, starting at Appendix A.

When discussing JSON, the error types include a simple syntax error, for which we change `"everything"` to `5 "everything"`, a syntax error intended to ambiguate a macro's end, for which we add an extra closing bracket, and a variable capture-based semantic error, for which we replace the captured `pi` variable with `unknown`.

Racket's example does not use JSON, so we discuss SQL, and replace `#:from` with `:from` to cause a syntax error, as well as replacing the captured variable `class_name` with `undefined` to cause a semantic error.

5.2.1 Yune

Our error messages (in Figure 32, Figure 33, and Figure 34) follow Rust's style, indicating cause and location illustrated with the relevant source-code line, though lacking a suggestion for how to resolve the issue as Rust and C++ do. Differing from Template Haskell, errors reported by macros

```
Macro 'json' at 240:13 returned an error.
---> json.un line 246 column 22
246 |              "answer": }{ "everything": 42 },
    |                        ~ Expected expression
```

Figure 33: Second Yune syntactic error

```
Macro 'json' at 240:13 returned an error.
---> json.un line 242 column 18
242 |              "pi": unknown,
    |                    ~ Variable 'unknown' is not defined
```

Figure 34: Yune semantic error

follow the same style as regular errors, making them more familiar to users. Our messages are more concise than those of Rust and C++, ensuring the user is shown only important information. We most clearly indicate the cause of the error in each example. This is particularly evident for syntactic errors as no other language underlines the `5` as the issue in the first example and only TH does in the second (see Appendix E). When applicable, our messages ensure that the user is aware of the macro(s) that caused an error, as previously discussed in Section 4.6.

5.2.2 C++

C++'s JSON DSL has incredibly verbose error messages due to the heavy reliance on templates, and particularly for syntactic issues the errors seem to cascade, as in Appendix C, making it difficult to determine what the actual issue is. Conversely, it provides clarity superior to Nim in Figure 35. It seems that reducing the number of reported syntax error messages increases readability. In Appendix F, the compiler (`g++`) also reports a semantic error caused by the original mistake, which distracts from the original problem.

5.2.3 Nim

Nim's error messages are by far the shortest and do not show the relevant code line, sometimes making it more difficult for the user to grasp the context. In the first and last examples (presented

```
file.cpp:11:9: error: expected ‘}’ before string constant
   11 |     { 5 "everything", 42 } } },
      |     ~   ^~~~~~~~~~~~
file.cpp:11:9: note: probably missing a comma or an operator before
```

Figure 35: C++

```
/file.nim(4, 13) template/generic instantiation of `%*` from here
/file.nim(10, 17) Error: in expression '5 "everything"': identifier
 ↪  expected, but found '5'
```

Figure 36: First Nim syntactic error

```
/file.nim(10, 16) Error: expression expected, but found '}'
```

Figure 37: Second Nim syntactic error

in Figure 36 and Figure 38 respectively), Nim indicates that the error is related to a macro, but it fails to do so in the second example (in Figure 37) because the unexpected closing delimiter causes the parser to mistakenly assume the macro has finished parsing, despite indentation indicating otherwise.

5.2.4 Rust

Rust most clearly indicates what the true problem is, but it seems to have issues with verbosity similar to C++. For the first syntactic example (see Appendix A), Rust adds a large amount of text unrelated to the original problem about a semantic error caused by the problem. In the second example (shown in Appendix D) Rust reports two errors related to unclosed delimiters, one of which correctly diagnoses the syntax mistake using indentation as a guide. This implies that Rust's mechanism for error reporting tracks indentation, despite the language itself being indentation-insensitive.

5.2.5 Template Haskell

As can be seen in Appendix E, TH's choice of macro delimiter results in fewer syntactic conflicts than Rust's. The syntactic error messages in Appendix B and Appendix E contain (in order) the location within the macro invocation that caused the error, the error message reported by the macro, the macro definition that reported an error, the macro invocation, and finally the line in our source code. The necessary information for debugging is present, although rather verbose. As visible in Figure 40, the line of code displayed is the start of the macro, rather than the line containing the syntax error. In TH, the user must manually count lines to determine the location of an error in the source-file, while Yune automatically resolves locations reported by a macro relative

```
/file.nim(4, 13) template/generic instantiation of `%*` from here
/file.nim(6, 11) Error: undeclared identifier: 'unknown'
```

Figure 38: Nim semantic error

```
error[E0425]: cannot find value `unknown` in this scope
 --> src/main.rs:6:13
  |
6 |       "pi": unknown, // variable capture
  |             ^^^^^^^ not found in this scope

For more information about this error, try `rustc --explain E0425`.
```

Figure 39: Rust

```
app/Main.hs:9:18: error: [GHC-88464] Variable not in scope: unknown
  |
9 | value = [aesonQQ|
  |                  ^...
```

Figure 40: Template Haskell

to its invocation site. It is possible that this can be resolved using a trivial modification to the TH implementation, rather than being a fundamental limitation of the system.

5.2.6 Racket

Racket's syntactic error message in Appendix G is confusing due to its cascade of errors, though it does provide the relevant line number and multi-line expression. Conversely, the semantic error message in Figure 41 is short and lacks information about the surrounding code.

5.2.7 C

We briefly discuss C due to its relevance for string interpolation. Figure 42 shows a vulnerability that can be exploited by the program's user to maliciously read memory and crash the program. Because the other languages discussed have stronger static checks, this vulnerability is impossible to reproduce in those languages. Rust in particular provides a helpful error message, displayed in Figure 43.

```
file.rkt:15:44: undefined: unbound identifier
  in: undefined
  location...:
   file.rkt:15:44
```

Figure 41: Racket semantic error.

```c
#include <stdio.h>

void main(int argc, char** argv) {
    // Prints the user-provided argument.
    printf("%s\n", argv[1]);
    // Uses the user-provided argument as format string,
    // creating a security vulnerability.
    printf(argv[1]);
}
```

Figure 42: C format string vulnerability example.

```
error: format argument must be a string literal
 --> src/main.rs:4:14
  |
4 |     println!(s);
  |              ^
  |
help: you might be missing a string literal to format with
  |
4 |     println!("", s);
  |              +++++
```

Figure 43: Rust formatting vulnerability error message.

6 Conclusion

In this thesis, we introduced a novel programming language called Yune, motivated by the potential of a macro system that can concisely express complex domain-specific ideas and is able to interoperate with C++. We showed that while several other languages exist with a subset of the relevant characteristics, there are no other runtime-free languages that can arbitrarily embed DSLs. Yune trivialises interoperation with C++, which similar languages such as Converge and Sparrow do not. In general, it appears that academic languages focus on the metaprogramming system without considering interoperability. Among the runtime-free languages, Rust and Yune are unique in their ability to execute all valid code at compile-time as they do not rely on interpreters. Our work emphasises the importance of powerful metaprogramming systems and interoperability.

7 Discussion

Despite our language succeeding at achieving the predetermined goals, we have discovered several shortcomings of the current system, which we discuss in this section.

Our compiler is written primarily in Go, as we determined that it would be the most convenient ANTLR4 code generation target to work with. In hindsight, C++ may have been a better choice despite its inherent complexity because it would have reduced the amount of code required to interface with the C++ compiler. Additionally, C macros could have been used to reduce the amount of boilerplate code in our compiler.

Perhaps the greatest limiting factor of our compiler is its compilation speed. Compilation is single-core and processes 16 lines per second on our mid-range desktop computer. A fourth of this time is consumed by ANTLR's parsing process, and the remaining time by C++ JIT compilation. Other parts of the compilation process consume negligible amounts of time. It is therefore recommended for future work that requires the compilation of large amounts of code to interpret code at compile-time and use either a parser-generator other than ANTLR or a hand-written parser.

An undesirable trait of our implementation is its non-deterministic compilation. Due to usage of Go's built-in `map` datastructure, top-level declarations are analysed in a random order. This results in unpredictable compilation output for macros that rely on the evaluation order.

Lilis and Savidis (see Section 2.1) suggest adding tools that understand the metaprogramming system. Our language features only a syntax highlighting extension for Emacs. Adding proper tools can also be explored in future research.

Due to the difficulty of generating nested named functions and recursive types in C++, these are not supported by Yune. While nested named functions can theoretically be emulated with macros that perform lambda lifting, recursive types must be implemented in the compiler. Future work may seek to do so. The most powerful metaprogramming system can arbitrarily inspect, generate, and modify code, but the practical use of such a system relies on the existence of language features and tools to support it.

References

[1] Yannis Lilis and Anthony Savidis. An integrated implementation framework for compile-time metaprogramming. *Softw. Pract. Exper.*, 45(6):727–763, June 2015.

[2] Yannis Lilis and Anthony Savidis. A survey of metaprogramming languages. *ACM Comput. Surv.*, 52(6), October 2019.

[3] Lucian Radu Teodorescu. *Improving Flexibility and Efficiency in Programming Languages: A Natural Approach*. PhD thesis, 2015.

[4] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6), October 2008.

[5] Zig language reference. https://ziglang.org/documentation/master/, 2026. Accessed: 2026-02-28.

[6] Rust Project Developers. The Rust reference. https://doc.rust-lang.org/reference/, 2026. Accessed: 2026-02-17.

[7] Jai-Community. Jai-community wiki. https://github.com/Jai-Community/Jai-Community-Library/wiki, 2026. Accessed: 2026-02-28.

[8] Andreas Rumpf and Zahary Karadjov. Nim manual. https://nim-lang.org/docs/manual.html, 2026. Version 2.2.6.

[9] Racket programming language. https://racket-lang.org/, 2026. Accessed: 2026-02-17.

[10] Kamil Skalski, Michał Moskal, Leszek Pacholski, and Paweł Olszta. Nemerle programming language. http://nemerle.org, 2003. Accessed: 2026-02-17.

[11] Nouri Khalass. Metaprogramming in modern programming languages, 2017.

[12] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

[13] Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and ides. *SIGPLAN Not.*, 45(10):444–463, October 2010.

[14] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: library-based syntactic language extensibility. *SIGPLAN Not.*, 46(10):391–406, October 2011.

[15] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. *SIGPLAN Not.*, 49(3):3–12, October 2013.

[16] JetBrains. *MPS User's Guide*. JetBrains, 2025. Accessed: 2026-04-30.

[17] Markus Völter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. 01 2010.

[18] Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen. A transient semantics for typed racket. *The Art, Science, and Engineering of Programming*, 6(2), November 2021.

[19] Ginger Bill. Odin programming language. https://github.com/odin-lang/Odin, 2026. Accessed: 2026-04-19.

[20] C++ ISO/IEC JTC1/SC22/WG21. Programming Languages — C++ (Working Draft). Working Draft N5032, ISO/IEC JTC1/SC22/WG21, dec 2025. Feature-complete committee draft for C++26; not an official ISO publication.

[21] Joel de Guzman, Hartmut Kaiser, and Dan Marsden. Boost.Spirit.Qi. Part of Boost C++ Libraries, Version 1.91.0, https://www.boost.org/doc/libs/1_91_0/libs/spirit/doc/html/spirit/qi.html, 2025. Accessed: 2026-04-30.

[22] Sean Baxter. Circle: The C++ automation language. https://www.circle-lang.org/, 2024. Accessed: 2026-03-05.

[23] RSDN Nemerle Project. Nemerle syntax extensions. https://github.com/rsdn/nemerle/wiki/Syntax-extensions, 2003. Accessed 2026-05-30.

[24] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020.

[25] Vinnie Falco, Krystian Stasiowski, and Dmitry Arkhipov. Boost.JSON. Part of Boost C++ Libraries, Version 1.91.0, https://www.boost.org/doc/libs/1_91_0/libs/json/, 2025. Accessed: 2026-03-02.

[26] ECMA International. ECMA-334: C# Language Specification. Technical Report ECMA-334, ECMA International, 2023.

[27] Nemerle Contributors. Nemerle wiki. https://github.com/rsdn/nemerle/wiki/, 2026. Accessed: 2026-05-10.

[28] Serde Developers. Rust Serde JSON. https://github.com/serde-rs/json, 2026. Accessed: 2026-05-03.

[29] Laurence Tratt. Converge. https://github.com/ltratt/converge, 2025. Accessed: 2026-05-16.

Appendices

A First Rust Syntactic Error

```
error: expected one of `)`, `,`, `.`, `?`, or an operator, found
↪  `"everything"`
  --> src/main.rs:10:21
   |
10 |       "answer": { 5 "everything": 42 },
   |                     -^^^^^^^^^^^^ expected one of `)`, `,`, `.`, `?`, or
   ↪  an operator
   |                     |
   |                     help: missing `,`

error[E0277]: the trait bound `std::string::String: From<({integer},
↪  &str)>` is not satisfied
  --> src/main.rs:5:17
   |
 5 |       let value = json!({
   |  _________________^
 6 | |       "pi": pi, // variable capture
 7 | |       "happy": true,
 8 | |       "name": "Rust",
...  |
16 | |     });
   | |______^ the trait `From<({integer}, &str)>` is not implemented for
   ↪  `std::string::String`
   |
   = help: the following other types implement trait `From<T>`:
             `std::string::String` implements `From<&mut str>`
             `std::string::String` implements `From<&std::string::String>`
             `std::string::String` implements `From<&str>`
             `std::string::String` implements `From<Box<str>>`
             `std::string::String` implements `From<Cow<'_, str>>`
             `std::string::String` implements `From<char>`
   = note: required for `({integer}, &str)` to implement
   ↪  `Into<std::string::String>`
   = note: this error originates in the macro `£crate::json_internal` which
   ↪  comes from the expansion of the macro `json` (in Nightly builds, run
   ↪  with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0277`.
```

B First Template Haskell Syntactic Error

```
app/Main.hs:9:9: error: [GHC-87897]
    • Exception when trying to run compile-time code:
        Error in aesonExp: "txt" (line 7, column 21):
unexpected "\""
expecting space or ":"
CallStack (from HasCallStack):
  error, called at src/Data/Aeson/QQ.hs:31:17 in
  ↪ aeson-qq-0.8.4-<hash>:Data.Aeson.QQ
      Code: template-haskell-2.21.0.0:Language.Haskell.TH.Quote.quoteExp
              aesonQQ
              "\n\
              \    {\n\
              \      \"pi\": #{pi}, \n\
              \      \"happy\": true,\n\
              \      \"name\": \"Template Haskell\",\n\
              \      \"nothing\": null,\n\
              \      \"answer\": { 5 \"everything\": 42 },\n\
              \      \"list\": [1, 0, 2],\n\
              \      \"object\": {\n\
              \        \"currency\": \"EUR\",\n\
              \        \"value\": 42.99\n\
              \      }\n\
              \    }\n\
              \  "
    • In the quasi-quotation:
        [aesonQQ|
    {
      "pi": #{pi},
      "happy": true,
      "name": "Template Haskell",
      "nothing": null,
      "answer": { 5 "everything": 42 },
      "list": [1, 0, 2],
      "object": {
        "currency": "EUR",
        "value": 42.99
      }
    }
  |]
  |
9 | value = [aesonQQ|
  |          ^^^^^^^^...
```

C Second C++ Syntactic Error

```
boost-json.cpp:11:6: error: expected ‘}’ before ‘{’ token
   11 |     }{ "everything", 42 } } },
      |      ^
boost-json.cpp:10:5: note: to match this ‘{’
   10 |     { "answer", {
      |     ^
boost-json.cpp:11:6: note: probably missing a comma or an operator before
   11 |     }{ "everything", 42 } } },
      |      ^
boost-json.cpp:12:5: error: expected unqualified-id before ‘{’ token
   12 |     { "list", {1, 0, 2}},
      |     ^
boost-json.cpp:12:25: error: expected unqualified-id before ‘,’ token
   12 |     { "list", {1, 0, 2}},
      |                         ^
boost-json.cpp:13:5: error: expected unqualified-id before ‘{’ token
   13 |     { "object", {
      |     ^
boost-json.cpp:17:1: error: expected declaration before ‘}’ token
   17 | };
      | ^
```

D Second Rust Syntactic Error

```
error: mismatched closing delimiter: `}`
  --> src/file.rs:5:22
   |
 5 |     let value = json!({
   |                       ^ unclosed delimiter
...
16 |     });
   |     ^ mismatched closing delimiter

error: unexpected closing delimiter: `)`
  --> src/file.rs:16:6
   |
 5 |     let value = json!({
   |                       - this delimiter might not be properly closed...
...
10 |       "answer": }{ "everything": 42 },
   |                 - ...as it matches this but it has different indentation
...
16 |     });
   |      ^ unexpected closing delimiter
```

E Template Haskell Semantic Error

```
app/Main.hs:9:9: error: [GHC-87897]
    • Exception when trying to run compile-time code:
        Error in aesonExp: "txt" (line 7, column 17):
unexpected "}"
expecting space, white space, "true", "false", "null", "\"", "{", "-",
 ↪  digit, "[" or "#{"
CallStack (from HasCallStack):
  error, called at src/Data/Aeson/QQ.hs:31:17 in
  ↪  aeson-qq-0.8.4-<hash>:Data.Aeson.QQ
     Code: template-haskell-2.21.0.0:Language.Haskell.TH.Quote.quoteExp
             aesonQQ
             "\n\
             \    {\n\
             \      \"pi\": #{pi}, \n\
             \      \"happy\": true,\n\
             \      \"name\": \"Template Haskell\",\n\
             \      \"nothing\": null,\n\
             \      \"answer\": }{ \"everything\": 42 },\n\
             \      \"list\": [1, 0, 2],\n\
             \      \"object\": {\n\
             \        \"currency\": \"EUR\",\n\
             \        \"value\": 42.99\n\
             \      }\n\
             \    }\n\
             \  "
    • In the quasi-quotation:
        [aesonQQ|
    {
      "pi": #{pi},
      "happy": true,
      "name": "Template Haskell",
      "nothing": null,
      "answer": }{ "everything": 42 },
      "list": [1, 0, 2],
      "object": {
        "currency": "EUR",
        "value": 42.99
      }
    }
  |]
  |
9 | value = [aesonQQ|
  |          ^^^^^^^^...
```

F C++ Semantic Error

```
file.cpp:6:13: error: ‘unknown’ was not declared in this scope; did you
 ↪  mean ‘union’?
    6 |     { "pi", unknown }, // variable capture
      |             ^~~~~~~
      |             union
file.cpp:17:1: error: could not convert ‘{{"pi", <expression error>},
 ↪  {"happy", true}, {"name", "Boost"}, {"nothing", nullptr}, {"answer",
 ↪  {{"everything", 42}}}, {"list", {1, 0, 2}}, {"object", {{"currency",
 ↪  "USD"}, {"value", 4.2990000000000002e+1}}}}’ from ‘<brace-enclosed
 ↪  initializer list>’ to
‘boost::json::value’
   17 | };
      | ^
      | |
      | <brace-enclosed initializer list>
```

G Racket Syntactic Error

```
file.rkt:8:2: sql: illegal character in untagged identifier
  at: :from
  in: (sql (select first_name last_name :from students #:where (= class_id
   ↪  (subquery (select id #:from classes #:where (= class_name (unquote
   ↪  class_name)))))))
  parsing context:
   while parsing Name
    term: :from
    location: file.rkt:10:6
   while parsing scalar expression
    term: :from
    location: file.rkt:10:6
   while parsing SelectItem
    term: :from
    location: file.rkt:10:6
   while parsing Select
    term: (select first_name last_name :from students #:w...
    location: file.rkt:9:4
   while parsing Statement
    term: (select first_name last_name :from students #:w...
    location: file.rkt:9:4
  location...:
   sql.rkt:10:6
  context...:
   /usr/share/racket/collects/syntax/parse/private/runtime-report.rkt:739:0:
    ↪  error/report
   /usr/share/racket/collects/syntax/parse/private/runtime-report.rkt:25:0:
    ↪  call-current-failure-handler
```

Figure 44: Racket syntax error: replaced `#:from` with `:from`