A pred-LL(*) Parsable Typed Higher-Order Macro System for Architecture Description Languages

Christoph Hochrainer
TU Wien
Vienna, Austria
christoph.hochrainer@tuwien.ac.at

Andreas Krall
TU Wien
Vienna, Austria
andi@complang.tuwien.ac.at

Abstract

Macro systems are powerful language extension tools for Architecture Description Languages (ADLs). Their generative power in combination with the simplicity of specification languages allows for a substantial reduction of repetitive specification sections. This paper explores how the introduction of function- and record types in a template-based macro system impacts the specification of ADLs. We present design and implementation of a pattern-based syntax macro system for the Vienna Architecture Description Language (VADL). The macro system is directly integrated into the language and is analyzed at parse time using a context-sensitive pred-LL(*) parser. The usefulness of the macro system is illustrated by some typical macro application design patterns. The effectiveness is shown by a detailed evaluation of the Instruction Set Architecture (ISA) specification of five different processor architectures. The observed specification reduction can be up to 90 times, leading to improved maintainability, readability and runtime performance of the specifications.

CCS Concepts: • **Software and its engineering** → *Domain specific languages*; **Macro languages**; • **Hardware** → Hardware description languages and compilation.

Keywords: macro system, higher-order macros, pattern-based, generative programming, architecture description language

ACM Reference Format:
Christoph Hochrainer and Andreas Krall. 2023. A pred-LL(*) Parsable Typed Higher-Order Macro System for Architecture Description Languages. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23), October 22–23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3624007.3624052

GPCE '23, October 22–23, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0406-2/23/10.
https://doi.org/10.1145/3624007.3624052

1 Introduction

Macros and Domain-Specific Languages (DSLs) are two programming concepts that contribute to faster development of artifacts and code quality. Macros are used to simplify repetitive code patterns by providing a shorter, more concise expression. Domain-Specific Languages offer a higher level of abstraction than conventional General-Purpose Languages (GPLs) for a specific domain. DSLs allow for a wide variety of applications, implementation techniques and design choices [18]. Macros and DSLs are strongly coupled. Many DSLs are implemented as a collection of macro definitions, while on the other hand, macros can contribute to the language extensibility for existing DSLs. A special form of DSLs are Architecture Description Languages (ADLs). In this article we present our experience with the development of a macro system with special focus on ADLs. Through the development of our Vienna Architecture Description Language (VADL), see Section 2, we gathered valuable insights regarding language extensibility for ADLs.

1.1 Architecture Description Languages

ADLs are computer languages used to describe the architecture of hardware and software systems. Particularly interesting for this article is the ADL subgroup of Processor Description Languages (PDLs). PDLs allow hardware designers to describe instruction set, register set, memory hierarchy, and other aspects of a microprocessor. We identified a particular need for macros regarding PDLs, especially when it comes to the specification of an instruction set architecture (ISA). Section 2.2 will provide an overview of the fragment of VADL used to describe ISAs and explain in more depth where the repetitiveness comes from and how it influenced our macro system design. Of course these observations are not limited to us and can also be found in other description languages like LISA [21], ISDL [10] or ArchC [2]. Additionally, we want to clarify some key properties of VADL and PDLs in general. A PDL is not, and should not be, an executable program. It can be thought of as a complex configuration for artifacts like hardware, simulator or compiler. This is an important concept as an error is no longer a programming error, which can be debugged with the specification alone. Debugging a specification requires specially generated tools and techniques like co-simulation. Hence, it is most important to reduce any other sources of errors, e.g. semantic errors, to a minimum. In Section 1.2, we describe how specific macro designs contribute to this desired property.

1.2 Macro Systems

Macro systems are one of the oldest forms of language extensions. In general, macros are user-defined procedures, transforming one program sequence to another program sequence. This transformation is called *macro expansion*. Based on the technique used the macro system is categorized into a *lexical* or *syntactical*, and *procedural* or *pattern-based* macro system [16]. Lexical macro systems, such as the C preprocessor (CPP) [23] and Unix M4 [12], are language agnostic and work on a lexical level, for example a token stream. In contrast to lexical macro systems, syntax macros are aware of syntactic structures. They are integrated into the core language and usually perform AST (Abstract Syntax Tree) to AST transformations. Representatives for example are LISP [24], Scheme [1] or Racket [9]. If the macro system supports algorithmic computations on their inputs, they are classified as *procedural* macro systems. On the other hand, macro systems that rely on pattern matching and substitution are called *pattern-based*. The presented concepts are not mutually exclusive and may be present in all combinations. The Rust programming language [13] incorporate both, *procedural* and *pattern-based* techniques within its macro system. Another interesting example is the Java Syntactic Extender (JSE) [3], which supports full procedural macros and an extendable pattern-matching engine. Finally, a concept often considered when talking about macro system is *hygienic macros* [4, 8, 14]. The main idea of hygienic macros is to prevent accidental capture of identifiers during expansion. When we started the design of our macro system, we anticipated, that macro hygiene was of secondary importance for us as we either want to capture identifiers or we pass the identifiers as arguments providing us with more control over the used names. We will address hygienic macros again in Section 3.6 together with our *lexical macros*. For now, hygienic macros are part of our future work.

1.3 Macros for DSLs

When we were considering language extensibility for our DSL, *syntax type safety* and *termination* were the top priorities. We use the term *syntax type safety* in the sense that a *syntax type safe* macro system is able to detect *syntax type errors*. Hence, the system prevents the generation of syntactically incorrect code. Furthermore, many macro systems designed for DSLs have a feature-rich host-language or environment they can exploit [5].

VADL on the other hand is a standalone DSL/ADL with no meta- or host-language available. This decision helps us to develop the VADL syntax more freely and explore different design possibilities for PDLs without syntactical restrictions or superfluous features of a host language. The drive of keeping the specification simple led us to investigate a lightweight and language dependent implementation, i.e. a *syntactical pattern-based* macro system. We also considered a language agnostic approach, but decided against it due to the lack of safety, available debug information and IDE support.

While we were satisfied with the choice of syntactical type safety, the pattern-based templates felt very limiting in expressiveness. Switching to procedural macros is for us (and we believe also for many other DSLs with a non Turing-complete specification language as host language) not beneficial as it compromises the simplicity of the host language. This inspired us to develop the higher-order *models* for our macro system discussed in Section 3.

A final aspect worth considering is computation time for DSL macro systems. Macro expansion becomes a prerequisite for any DSL related analysis and task. Therefore, a main goal should be to make sure that the macro system's execution time is as short as possible. We incorporated our macro system directly into the language grammar without requiring any preprocessor. This helped us to reduce unnecessary precomputations. Additionally, our LL(k) parsable host language encouraged us to preserve the top-down parsing fashion. We designed the built-in macro system to be pred-LL(*) parsable.

Contribution.

- A simple pred-LL(*) parsable *syntactical pattern-based* macro system for specification languages
- Syntax type safe higher-order macro templates using *models*
- Composable syntax types using *records* and type aliases
- Demonstration of the presented macro system using the Vienna Architecture Description Language

Additionally, we present a variety of smaller macro features supporting a high configurability and usability in the context of specification languages. We found the following implemented features particularly useful for our exploratory language design of VADL.

- Inheritance of macro definitions across language definitions
- Lexical manipulation of identifiers and strings
- Configurable and conditional macro expansions using *match* and command line arguments

2 Overview

In this section we give an overview of the Vienna Architecture Description Language (VADL) with special focus on the instruction set architecture (ISA) section.

2.1 Vienna Architecture Description Language

VADL is a Domain-Specific Language in the domain of computer architecture and compiler construction. It permits the complete formal specification of a processor architecture.

Additionally, it is possible to specify the behavior of generators which produce different artifacts from a processor specification like a compiler or an instruction set simulator. VADL strictly separates the specification of the instruction set architecture (ISA), the micro architecture (MiA) and the application binary interface (ABI). To provide a proof of concept, we only implemented and evaluated our macro language for the instruction set architecture specification section. However, the ideas and techniques presented can be applied to the other sections as well as to any similarly structured DSL.

2.2 ISA Syntax Elements

Presenting the whole syntax and semantics of VADL used to describe ISAs, let alone the VADL language as a whole, is out of the scope of this article. Therefore, we will focus only on the relevant portion of the instruction set architecture definition. First, we have to establish how instructions are defined. VADL separates the abstract concept of an instruction into three parts. The instruction definition, the instruction encoding and the textual representation, i.e. assembly. The instruction definition is the core part of the three definitions, holding information on the name, the used encoding format and the instruction semantics. The instruction encoding specifies the values of the static encoded fields. The instruction assembly definition specifies a pattern on how the assembly string is computed. Figure 1 shows a specification of an *ADD* instruction, which adds two registers together and stores the result in a third. The format fields of the format definition *F*, which are not assigned to a static value inside the encoding definition, become dynamic fields or operands. Inside the instruction semantics, we can observe that *rd*, *rs1* and *rs2* are indeed used as operands. The call expressions to *X(.)* represent indexing of a register bank *X*, defined somewhere else in the ISA.

If we define a new instruction, e.g. *AND*, that differs from *ADD* in a single encoding bit and the binary operator, we would need to create a completely new instruction definition, encoding and assembly. Which brings us to the downside of such element or block based specification languages like VADL. We designed VADL to be descriptive and simple, which led us to a very small core language for the ISA section. While we support functions, our core type system is very simple and does not support these definitions as first class citizens. During our language development phase we also experimented with different language built-in features that could reduce code duplication, but we came to the conclusion that they only introduce a lot of complexity and obfuscate the original code. This led us to the idea of designing a template-based macro system specifically directed towards specification languages.

```
format F =
  { funct7  : Bits<7>
  , rs2     : Bits<5>
  , rs1     : Bits<5>
  , funct3  : Bits<3>
  , rd      : Bits<5>
  , opcode  : Bits<7>
  }

instruction ADD : F = {
  X(rd) := X(rs1) + X(rs2)
  }

encoding ADD =
  { opcode = 0b011'0011
  , funct3 = 0b000
  , funct7 = 0b000'0000
  }

assembly ADD =
  ( "ADD"
  , register( rd ), ","
  , register( rs1 ), ","
  , register( rs2 )
  )
```

Figure 1. ISA Example Specification for an ADD instruction

3 VADL's Macro System

In this section we give a detailed description about the syntax and techniques implemented for VADL's macro system.

3.1 Syntax Models

At the core of our macro system are the so-called *syntax models*. A syntax model can be seen as a parameterized and well typed template. Figure 2 shows how such a syntax model can be defined.

```
model InstModel (op: BinOp, name : Id)
  : IsaDefs = {
  instruction $name : F = {
    X(rd) := X(rs1) $op X(rs2)
  }
}
```

Figure 2. Syntax Model Definition

Every model has a name, a typed parameter list, a result type and a body. Note how the use of the parameters are indicated by a leading "$". This design decision has two advantages. First, it simplifies parsing as it explicitly marks the use of a macro element. Second, the "$" captures the model parameter names, preventing name collisions with

ISA definitions, which strengthens the hygiene of the macros. Similarly to the parameters, we use the "$" for the instantiation of defined syntax models. Figure 3 shows how the model from Figure 2 can be instantiated. To separate the syntax elements from each other we use ";" as separator inside an instantiation. Recall the dilemma of Section 2.2, the introduction of *models* provides us now with a mechanism to efficiently specify both instructions without code repetition.

```
$InstModel( ADD ; + )
$InstModel( AND ; & )
```

Figure 3. Syntax Model Instantiation

In the example of Figure 2 we only used the identifier (*Id*), binary operator (*BinOp*) and ISA element (*IsaDefs*) syntax types. However, the syntax model definition supports a variety of types discussed in the following sections.

3.2 Syntax Types

This section introduces all the available core syntax types. We designed our syntax types to have a one-to-one relation to parser rules. This already provides us with a partial order, where the relation is a partially ordered subtype relation. Table 1 gives an overview of all the available base types with a short description and examples. Additionally, it is important to note that the presented base types, function types (Section 3.4), record types and type aliases (Section 3.3) can be arbitrarily nested. The resulting types can be used everywhere a *syntax type* is expected with the only exception being result types of *models* and function types. Figure 4 displays the subtype relation between the presented core types. The macro type system provides an implicit upcasting of the value types. For example, if a model expects a value of type *Val*, any subtype, i.e. *Bool*, *Int* or *Bin* will be accepted as argument.

Table 1. Core Syntax Types

Type	Description	Examples
Ex	Generic VADL Expression	X(rs1) + X(rs2)
Lit	Generic VADL Literal	1, "ADD"
Val	Generic VADL Value Literal	1, 0b001
Bool	Boolean Literal	true, false
Int	Integer Literal	1, 2, 3
Bin	Binary or Hexadecimal Literal	0b1, 0xff
Str	String Literal	"ADD"
CallEx	Arbitrary Call Expression	MEM<2>(rs1)
SymEx	Symbol Call Expression	rs1, MEM<2>
Id	Identifier Symbol	rs1, ADD, X
BinOp	Binary Operator	+, -, *
UnOp	Unary Operator	-
Stat	Generic VADL Statement	X(rd) := X(rs)
Stats	List of VADL Statements	X(rd) := X(rs) …
IsaDefs	List of VADL ISA Definition	instruction ADD : F = {…} …
Encs	Element(s) of an Encoding Definition	func = 0b000, …

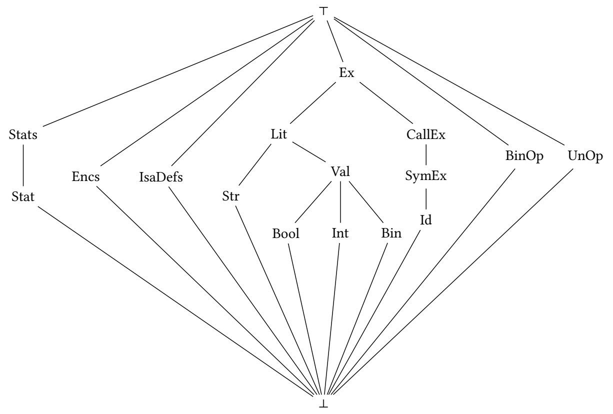

Figure 4. Syntax Type Relation

3.3 Type Alias and Composition

The VADL macro system provides a feature rich type interface. Besides the basic types mentioned in Section 3.2, the macro system also supports type aliasing and a form of type composition to make the typed templates more readable. Figure 5 shows a type alias definition *BinExprType*, which from now on can be used instead of the function type *(Ex, Ex) -> Ex*. An application of *BinExprType* can be seen in Figure 8.

Figure 6 shows a *record* definition used for type compositions. In this particular case the record definition composes an *Id* and *BinOp* type to the new type *BinInstRec*. The body of a record consists of a parameter list providing typed fields. Figure 7 shows how the record is initialized and the fields *name* and *op* are accessed. Passing a record type argument can be either done by reference or by creating a syntax tuple. A syntax tuple is specified the same way a model argument list is provided, i.e. syntax elements are separated by ";" and enclosed inside brackets. Accessing the passed elements is done using the record's name followed by a "." and the desired field. Accesses of sub-records can be arbitrary chained together. The whole access may be wrapped inside brackets, i.e. "$(...)", to better indicate what is part of the access and what belongs to the VADL specification. Furthermore, it is important to note that records are treated as type tuples. Their field names do not affect the type and are only used to access the internal elements.

```
model-type BinExType = ( Ex , Ex ) -> Ex
```

Figure 5. Syntax Type Alias Example

```
record BinInstRec ( name: Id , op: BinOp )
```

Figure 6. Record Example

```
model InstModel (info: BinInstRec)
  : IsaDefs = {
  instruction $info.name : F = {
    X(rd) := X(rs1) $info.op X(rs2)
    }
}

$InstModel( ( SUB ; - ) )
$InstModel( ( ADD ; + ) )
```

Figure 7. Record Application

3.4 Higher-Order Macros

To the best of our knowledge, we have not seen typed higher-order macros in a *pattern-based syntax* macro system as presented in this paper. In this section we will shortly describe how they are used in the context of our macro system. In Section 3.8, we will further discuss why they are important for ADLs and how we use them in VADL. A higher-order macro is a statically evaluated function, mapping a list of syntax types to a result syntax type. We chose the term higher-order, to underline the capability of providing model references as argument. In Figure 5 we have already defined the type signature of a model in form of a function type. We will reuse this type for the higher-order model *BinExStat* in Figure 8. The instantiation of *BinExStat* in the presented figure, produces an assignment statement of *X(rd)* taking the addition of *X(rs1)* and *X(rs2)* as argument.

A valid argument for a parameter with a function type is either a model reference, as seen in the example, or a parameter of function type from an outer model. In both cases the types are evaluated and checked during parse time.

3.5 Conditional Expansion

A minor difference to some pattern-based approaches is our conditional expansion. VADL macros provide an explicitly typed *match*-statement shown in Figure 9. The entries are processed from top-to-bottom and it uses the right-hand-side of the first satisfied left-hand-side for expansion. The *match*-statement has the requirement of providing a default case at the last position, indicated by the "_". Beside the default case, each entry contains a condition that is either matching equality ("=") or inequality ("!=") of a parameter and a syntax element matching the type of the parameter. The comparison is done on a lexical, i.e. token-based, level and performed

```
model BinExStat
  //         ( Ex, Ex ) -> Ex
  ( binEx : BinExType ) : Stat = {
    X(rd) := $binEx( X(rs1), X(rs2) )
  }

model AddExp ( rhs: Ex, lhs : Ex ) : Ex = {
  $rhs + $lhs
  }

$BinExStat( AddExp )
```

Figure 8. Higher-Order Macro Example

on the expanded representation of the parameter. A *match*-statement is only allowed inside a model. Therefore, it is only evaluated if the parent model is instantiated, which is why the argument-parameter pairs are always available for the *match*-statement expansion. Figure 9 shows how it could be used in configuration management. More on configuration can be found in Section 3.7.

```
// $BitSize( Arch32 ) -> 32
// $BitSize( Arch64 ) -> 64
model BitSize ( arch : Id ) : Int = {
  match : Int
    ( $arch = Arch32 => 32
    ; $arch = Arch64 => 64
    ; _              => 0
    )
  }
```

Figure 9. Match Statement Example

3.6 Lexical Macro Functions

While most of the needs are covered by *syntactical* macros, we came to the conclusion that string and identifier manipulation is best done using *lexical macros*. A lexical macro acts on the abstraction level of token streams in contrast to an already parsed AST. Through language exploration, we narrowed the *lexical* macros down to two use-cases. These use-cases are safely implemented using special macro functions. Firstly, templates generating instruction behavior and assembly used to require the instruction name once in form of an identifier (*Id*) and again in form of a string (*Str*). We solved this issue by introducing the *IdToStr* function. This function takes an *Id* typed syntax element and converts it to a *Str* typed syntax element. Secondly, we encountered the problem of not being able to efficiently manipulate our identifiers. This is especially tedious when dealing with different configurations. To provide a type safe identifier manipulation, we introduced the *ExtendId* function. This function takes an *Id* typed identifier and an arbitrary number of *Str* typed syntax elements, concatenates them together and returns a single *Id* typed syntax element. Figure 10 shows a small example of both functions with their typed result as comment. It is important to note that the context of the *lexical macros* generated identifiers is strictly separated from the context of the *syntactical macros*. Therefore, it is not possible to define or refer to a model name or parameter using a generated identifier.

For VADL, the lexical macro functions are the "alternative" to *hygienic macros*. Consider the ISA elements described in Section 2.2. When defining a new instruction we either want to capture identifiers, e.g. registers or memories, or we want full control over the identifier name. From our experience, VADL's *IdToStr* and *ExtendId* are sufficient for this use-case.

```
ExtendId( I, "Am", "An", "Identifier" )
    // --> IAmAnIdentifier : Id
IdToStr( IAmAString )
    // --> "IAmAString"     : Str
```

Figure 10. Lexical Macros Example

3.7 Configuration and Inheritance

VADL provides the possibility of passing configuring information to the macro system using the command line. Currently, this mechanism is kept very simple and is based on *Id* elements. To prepare a configurable macro variable, one has to create a simple model of type *Id* containing a default value. Figure 11 shows such a variable of name *Arch*, with the default setting *Aarch32*. Without any passed configurations the instantiation of *Arch* results in the identifier *Aarch32*. However, if VADL receives the command-line option *-m* or *–model*, followed by the string *"Arch=Aarch64"* the value of *Arch* is overridden. If *Arch* is instantiated given the previous command-line option, it would result in *Aarch64*. In combination with conditional expansion, see Section 3.5 and Figure 9, this simple mechanism already provides powerful configuration capabilities.

```
model Arch() : Id = { Aarch32 }
```

Figure 11. Macro Configuration Variable

Additionally, we want to shortly mention VADL's inheritance in this section. Every ISA component in VADL can inherit from an arbitrary other ISA component using the keyword *extending*. Since we tightly coupled the macro system into our language, the inheritance and visibility does also affect the macro system. Figure 12 shows this mechanism in action and provides additional information in the comments.

```
instruction set architecture A = {
  model ModelA() : IsaDefs = // ...
  }

instruction set architecture B
    extending A = {
  $ModelA() // OK, defined in ISA A
  }

instruction set architecture C
    extending B = {
  $ModelA() // OK, defined in ISA A
}

instruction set architecture D = {
  $ModelA() // ERROR, not defined
  }
```

Figure 12. ISA Inheritance Example

3.8 Macro Application in Processor Specifications

The following macro application design patterns demonstrate with simplified examples the usage of macros for instruction set architecture specifications. The simplified examples are based on a real specification of the AArch32 instruction set architecture from ARM. The most common case is a simple argument substitution pattern shown in Figure 13.

AArch32 has a register file called R consisting of 16 registers which are 32 bits wide. Conditions are specified by boolean expressions on flags of the status register APSR, e.g. the zero flag Z. Every instruction can be executed conditionally. There are 15 different conditions which are described by an enumeration in the specification and encoded by the cc field in an instruction word which is 32 bits wide. Arithmetic/logic instructions which have an immediate value as second source operand share a common instruction encoding specified in the ArLoImm instruction format. The ALImmInstr instructions themselves differentiate each other only by the unique instruction identifier, the assembly instruction name, the binary operation to be executed and the instruction encoding. Therefore, a model with these four parameters is defined which substitutes these four parts in the instruction specification. Then with a single line macro call an arithmetic/logic immediate instruction can be specified. This leads to concise instruction set architecture specifications,

```
register file R: Bits<4> -> Bits<32>

enumeration cond: Bits<4> =
  { EQ  // equal                  Z == 1
  , NE  // not equal              Z == 0
  // ...
  , AL  // always
  }

// arithmetic/logic immediate format
format ArLoImm: Bits<32> =
  { cc     [31..28] // condition
  , op     [27..21] // opcode
  , flags  [20]     // set status register
  , rn     [19..16] // source register
  , rd     [15..12] // destination register
  , imm12  [11..0]  // 12 bit immediate
  }

model ALImmInstr (id:Id, ass:Str, op:BinOp,
                  opcode:Bin) : IsaDefs = {
  instruction $id : ArLoImm = {
    R(rd) := R(rn) $op imm12
    }
  encoding $id =
    {cc = cond::AL, op = $opcode, flags = 0}
  assembly $id = ($ass, ' ', register(rd),
    ',', register(rn), ',', decimal(imm12))
  }

$ALImmInstr ( ADD ; "add" ; + ; 0b000'0100 )
$ALImmInstr ( SUB ; "sub" ; - ; 0b000'0010 )
$ALImmInstr ( AND ; "and" ; & ; 0b000'0000 )
$ALImmInstr ( ORR ; "orr" ; | ; 0b000'1100 )
```

Figure 13. Instruction Specification applying simple Macros

improves the maintainability and reduces the probability of errors.

Most instruction set architectures are too complex to get by with the substitution pattern. As in the AArch32 architecture every instruction can be executed conditionally, a basic instruction exists in 15 variants for 15 different conditions. This problem can be solved smartly by an extension macro pattern using higher-order macros as demonstrated in Figure 14.

To reduce the number of macro arguments record types are defined for an instruction and a condition. The Inst record type definition groups the four arguments describing an instruction from Figure 13 together. The Cond record type definition consists of a string representing the extension of the assembly name, the identifier of the enumeration of the condition encoding and a boolean expression for condition evaluation.

In contrast to the previous example in Figure 14 now 15 different instructions with a unique identifier have to be created. This can be handled with the lexical macro function ExtendId by appending the extension string of the condition to the identifier.

The final problem is that there is a set of models which describe different kinds of conditional instructions and all these models should be called 15 times for the 15 different conditions. This can be solved by the higher-order model CondInstr, which takes the instruction model as first argument. The instruction model is then called 15 times with an argument list, which has been extended by the conditions. In the above example the 4 macro calls expand to 60 different instructions. The AArch32 architecture has instructions with a lot of additional variants like setting the status register, shifted operands or complex addressing modes. This leads to a specification with multiple higher-order macro arguments.

4 Implementation

In this section we give an overview of our macro system implementation for VADL. VADL manages the macros in two separated phases: parsing and expansion. It is important to note that the parsing phase does only analyze the macros. It guides the parser through the different kind of macro actions while asserting their syntactical and partly semantical correctness. Applications of the macro actions are done in the expansion phase.

4.1 Parsing

Parser. The VADL frontend uses a modified version of the Xtext framework [6]. The Xtext framework is a Java based DSL development tool. It takes a Xtext grammar file as input and generates a variety of useful artifacts, e.g. IDE integration, metamodel classes for the syntax-tree or a parser. We have refrained from using any non LL(k) Xtext grammar functionalities and disabled the backtracking feature of the generated parsers to start our implementation from a true LL(k) parser. A LL(k) parser is a top-down parser processing the language from left to right. The k indicates a constant lookahead, which may be performed by the parser. Additionally, we extended the implementation to allow semantic predicates [20] and code actions. This grammar extension lifts the parser to pred-LL(*). The *pred* prefix indicates the use of predicates in combination with grammar rules and the star (*) lifts the lookahead requirement of a fixed constant k to an arbitrary constant. The Xtext framework targets ANTLR [19], which already supports code actions and semantic predicates. Therefore, extending the grammar was a quite straight forward task for our simple purposes. In Sections 4.1 and 5.3 we will further comment on the context-sensitivity.

Concrete Syntax Tree. The Xtext framework automatically creates classes for each non-terminal grammar rule that has at least one labeled rule field. While parsing a source file, it uses these classes to build a concrete syntax tree (CST). To keep the implementation effort manageable we kept this

```
record Instr (id: Id, ass: Str, op: BinOp, opcode: Bin)
record Cond  (str: Str, code: Id, ex: Ex)

model ALImmCondInstr (cond: Cond, instr: Instr) : IsaDefs = {
  instruction ExtendId ($instr.id, $cond.str) : ArLoImm = {
    if ($cond.ex) then
      R(rd) := R(rn) $instr.op imm12
    }
  encoding ExtendId ($instr.id, $cond.str) =
    {cc = cond::$cond.code, op = $instr.opcode, flags = 0}
  assembly ExtendId ($instr.id, $cond.str) =
    ($instr.ass, $cond.str, ' ', register(rd), ',', register(rn), ',', decimal(imm12))
  }

model-type CondInstrModel = (Cond, Instr) -> IsaDefs

model CondInstr (modelid: CondInstrModel, instr: Instr) : IsaDefs = {
  $modelid (( "eq" ; EQ ;  APSR.Z = 0b1 ) ; $instr)
  $modelid (( "ne" ; NE ;  APSR.Z = 0b0 ) ; $instr)
  // ...
  }

$CondInstr(ALImmCondInstr ; ( ADD ; "add" ; + ; 0b000'0100 ))
$CondInstr(ALImmCondInstr ; ( SUB ; "sub" ; - ; 0b000'0010 ))
$CondInstr(ALImmCondInstr ; ( AND ; "and" ; & ; 0b000'0000 ))
$CondInstr(ALImmCondInstr ; ( ORR ; "orr" ; | ; 0b000'1100 ))
```

Figure 14. Instruction Specification applying Higher Order Macros

default behavior. VADL does not have a separate preprocessor, therefore performs the macro expansion directly on the generated CST.

Wrapper Rules. Since the CST consists of Java classes based on grammar rules, we decided to introduce additional wrapper rules. Each rule, which we would like to use as macro type, is enclosed in an additional rule to create a clear location for replacement later on. In most cases, such a wrapper rule contains two alternatives: a generic macro replacement rule guarded by a semantic predicate and the concrete value rule. The semantic predicate is used to perform a minor lookahead to see if the next tokens are part of a concrete value or a macro action. Unfortunately, this introduces a slight overhead as we have to create an additional wrapper rule for each syntax type. Figure 15 shows a wrapper rule *StringRule* and a concrete rule *ConcreteStringRule* to express strings. Retrieving the current context with reflections or the parser itself was quite tedious, so we decided to simply parameterize our *isMacroAction* predicate with the current syntax type context (*Str*). The *StringRule* rule can now be used anywhere as if it was a normal grammar rule for strings. Inside the *StringRule* the *value* field is used as location for replacement. We implemented all non-terminal rules used as syntax types (see Section 3.2) in the same fashion.

```
StringRule:
  $$ isMacroAction( Str ) $$?=> // sem-pred
  value=MacroActionRule
| value=ConcreteStringRule
;

ConcreteStringRule: // ...
```

Figure 15. Wrapper Rule Example

Context Sensitivity. By using a context-sensitive parse approach, we guide the parser in such a way that only syntactically and semantically correct macro occurrences are parsable. To manage the context sensitivity, we implemented a parser state specifically for macros. It provides an API used by semantic predicates and code actions to compare and update symbols and macro information. The core of the parse state itself consists of a symbol table containing information on macro related definitions in the current scope. Moreover, it holds a variety of type information on actively parsed macro constructs. In the initial parse state only the core syntax types listed in section 3.2 are registered. During parsing, ISA namespaces, models, their parameters, records and syntax type aliases are added. Figure 16 shows a simplified version of our grammar rules handling a model definition. The start of the *ModelRule* is straight forward by expecting the model keyword, a name, a typed parameter list and a syntax type. Before the parser enters the model body, the parse state has to be updated. In the example this is done by the code actions, indicated with an opening and closing "$$". We feed the parse state the name, the syntax type and the parameters. Note that the passed values are CST nodes and therefore already Java classes, making it possible to access type and name of the parameters. Inside the parse state, the symbol table is extended by the model name and the information on the parameters. The passed type is used to select the correct rule to parse the model body. This is done using syntax predicates, which can be seen in Figure 16 inside the *ModelBodyRule*. They are similar to code actions but with an additional "? =>" after the enclosing "$$". The predicates are tested in-order from top to bottom. If a predicate is satisfied, the parser tries to apply the rule(s) on the right-hand side of the current alternative. The *ModelBodyRule* reveals another slight overhead as we have to manually implement the relation between syntax type and desired rule. The presented example was of course just a simplified version of the actual implementation and should help to understand the main idea. A similar approach was applied for all the other context-sensitive tasks, e.g. managing ISA namespaces or checking the correctness of syntax types of passed arguments.

```
ModelRule:
   "model" name=IdentifierRule
   "(" parameters+=ModelParameterRule* ")"
   ":" type=SyntaxTypeRule "="
   "{"
    $$ state.enterModel
        ( name, type, parameters ); $$
    body=ModelBodyRule
    $$ state.leaveModel
        ( name, type, parameters ); $$
   "}"
;

ModelBodyRule:
    $$ state.isModelBodyOfType( String  ) $$
        ?=> StringRule
    $$ state.isModelBodyOfType( Integer ) $$
        ?=> IntegerRule
    // ...
;

ModelParameterRule:
    name=IdentifierRule
    ":" type=SyntaxTypeRule
;
```

Figure 16. Model Grammar Rules

4.2 Expansion

The macro expansion is done in a separate pass after parsing and works solely on the CST representation. This pass is responsible to perform all macro related CST manipulations. Note that at this stage the correctness of the macros were already checked by the parser. The examples in Figure 17 and Figure 18 show a textual representation of the CST before and after the expansion pass. The expansion is executed in two steps.

Setup. The first step is to remove all the macro nodes of the CST that do not produce any new nodes. This includes model, syntax type alias and record definitions. During the removal, each definition is stored in a symbol table to preserve its information. Additional to the parsed definitions, the command line configurations are added to the symbol table.

Execution. The second step is the actual execution and expansion of macro code. The expansion is done in a top-down fashion and performs iterative replacements on wrapper nodes. The first top level construct, or root node, the macro expander encounters, is by design a model instantiation node. The instantiation contains information on the model and the passed parameters. Similar to function inlining, the expander creates a copy of the referenced model body and replaces each parameter occurrence with the respective argument. The new body is now passed again to the expander to resolve nested macro actions. Afterwards, the macro node is replaced by the newly created subtree of the updated copy of the model body. Conditional macro expansions are also done during the model instantiation. The implementation is very simple and currently only uses a unified symbolic equality comparison. The *match* statement is symbolically executed and replaced by the right-hand side of the first satisfied left hand side condition. If no condition evaluates to true, the mandatory last wild card statement is used. Similar to the model instantiation, the resulting node is again iteratively expanded. The implementation of the lexical macro actions (*IdToStr*, *ExtendId*) are also straight forward. We simply create a new identifier or string CST node and replace the old macro node occurrence with the newly created node. Recall that the newly created identifier is only part of the expanded program and not available during expansion. This means that it cannot be used to refer to macro models or macro parameters.

Termination. Finally, we want to emphasize again the fact that the termination of our expansion is always guaranteed. Although we allow for multiple nested model instantiations and invocation of higher-order model parameters, the top-down parsing and our conscious opposition to use-before-define, make recursive model calls impossible. Note that a model is only considered to be defined *after* it is fully parsed, which prevents a recursive call to itself or passing itself as argument inside its body. Therefore, it should be clear that our iterative expansion is bounded by a finite number of steps.

```
model Inc( val : Ex ) : Ex = { $val + 1 }
model Dec( val : Ex ) : Ex = { $val - 1 }
model InstrBody
    ( func : (Ex) -> Ex ) : Stat = {
  X(rd) := $func( X(rs) )
  }

instruction A : F = { $InstrBody( Inc ) }
instruction B : F = { $InstrBody( Dec ) }
```

Figure 17. Model Instantiation Example (before)

```
instruction A : F = { X(rd) := (X(rs) + 1) }
instruction B : F = { X(rd) := (X(rs) - 1) }
```

Figure 18. Model Instantiation Example (after)

5 Evaluation

For the evaluation of the macro system's efficiency and expressiveness we used VADL ISA specifications of *AArch64*, *Aarch32*, *MIPS IV*, *RISCV* and our RISCV-like toy architecture *TriLen*. The following section is separated into a qualitative evaluation and a runtime evaluation section. The qualitative evaluation investigates a variety of source code properties like amount of *models*, *records* or type-alias. The runtime section provides an overview of the execution time.

5.1 Qualitative Evaluation

This section should provide an overview on the macro system's expressiveness with a particular focus on the presented macro concepts, e.g. higher-order *models* or type compositions. For evaluation, we implemented data collection passes before and after the macro expansion pass. Lines of code and lines of comments were collected manually. Lines of code exclude any trailing empty lines in a file. To determine the comments only lines, we used following regex:

$$(\hat{}[\text{" "}]^*//.^*\$) \mid (\hat{}[\text{" "}]^*/\backslash^*[\backslash s\backslash S\backslash n]+?\ \backslash^*/)$$

Firstly, we present the overall expressiveness by providing a comparison between the original specifications and their expanded, pretty-printed results. Table 2 contains data on the lines of code, lines of comments, CST nodes and instruction definitions before and after expansion for each architecture respectively. The instruction definitions value includes all *instruction* language elements (see Section 2.2) no matter if it contains placeholders or if it is located inside a model template. The concept of CST nodes are described in Section 4.1. We believe that instruction definitions and CST nodes are a more accurate metric to demonstrate the actual generative capabilities of our system. Lines of code (including comment lines) is a more tangible metric, which is why we also provided them.

It is important to note that our pretty printer does not insert new lines in-between an expression, which in case of nested if-else-expressions could result in even more output lines. Additionally, all the comments are not preserved during the parse step and are therefore missing in the output.

Table 2. Expansion Statistics

		AArch32	AArch64	MIPS IV	RISCV	TriLen
Original	Lines of Code	1273	2334	1131	635	521
	Lines of Comments	101	177	108	99	113
	CST Nodes	17158	36699	10156	6264	8557
	Instr. Def.	53	52	45	17	20
Expanded	Lines of Code	110369	10227	1432	612	1301
	Lines of Comments	-	-	-	-	-
	CST Nodes	1520796	134601	14123	7698	16066
	Instr. Def.	8865	799	106	37	123

Our prime examples are the ARM specifications *AArch64* and especially *AArch32*. Their specifications heavily use the macro system to model the different instruction variations. This can be seen by their 52 and 53 initial instruction definitions, which expand to 799 and a tremendous 8865 instruction definitions. These big numbers can be explained by the fact that the final specification explicitly models each hardware mode and conditional execution combination as a separate instruction definition. Although the other examples are not that impressive, *MIPS IV*, *RISCV* and *TriLen* still show improvements regarding code size and especially instruction definition abstraction. *RISCV*'s lines of code is a perfect example why we chose to provide additional metrics to compare the expanded specifications to the originals. The lines of codes decrease after the expansion, indicating that the macro system introduced more verbosity. However, when we look at the CST nodes and the instruction definitions, one can see that there actually was an increase in syntax elements after the expansion. Additionally, the instruction definition amount doubled, which is an indication that the system provides a good abstraction. Overall, we are satisfied with the expressiveness of the macro system as the code reduction measured by the CST nodes of our examples are between the factors 1.2 and 90.

Furthermore, we were interested in the absolute frequency of each macro element for each architecture specification respectively. Table 3 displays the overall amount of *model* definitions, their placeholders inside the template, model instantiations (macro invocations), *record* definitions, type-alias definitions and the uses of our macro conditional (*match*).

Table 3. Macro Element Statistics

	AArch32	AArch64	MIPS IV	RISCV	TriLen
Models	90	142	64	4	21
Placeholders	1168	1270	311	30	178
Instantiations	225	322	163	24	56
Records	4	9	0	0	0
Type-Alias	4	8	0	0	0
Match	0	12	2	0	0

An interesting discovery when looking at Table 2 and Table 3 is that more model definitions and model instantiations do not necessary mean a higher increase in CST nodes. This can be seen by comparing the properties for *AArch32* and *AArch64*. In general, complex architectures like *AArch32*, *AArch64* and even *MIPS IV* seem to use the set of features more than the simpler ones like *RISCV* and *TriLen*. Especially, when it comes to the use of type aliases and records, the complex architectures benefit more from it. The reason is that their model templates are more complex. For the given examples, the ratios of placeholders to models seem to give a good indicator on this complexity. Furthermore, it is interesting to see that records, type aliases and *match*-statements have been used sparsely.

Finally, we decided to provide an overview of a variety of parameter related statistics for *model* definitions. Table 4 shows the minimum, maximum, absolute and average amount of arguments for:

- **Normal Parameters**: This includes all model parameters.
- **Flattened Parameters:** This includes all model parameters with a slight manipulation. All parameters, which resolve to the type *record* or tuple are flattened. This means that each element inside a record or tuple is iteratively unpacked and moved to the outer parameter list. The original type container is removed.
- **Higher-Order Parameters:** This includes all parameters that are of higher-order, i.e. are of type, or contain a subtype, of the function type.

Table 4. Model Parameter Statistics

		AArch32	AArch64	MIPS IV	RISCV	TriLen
Normal	Min	0	0	1	4	2
	Max	9	8	5	6	8
	Abs	343	417	187	21	96
	Avg	3.81	2.93	2.92	5.25	4.57
Flattened	Min	0	0	1	4	2
	Max	14	15	5	6	8
	Abs	738	886	187	21	96
	Avg	8.2	6.23	2.92	5.25	4.57
Higher-Order	Min	0	0	0	0	0
	Max	3	2	0	0	0
	Abs	37	36	0	0	0
	Avg	0.41	0.25	0	0	0

Table 4 presents statistics regarding the number of macro arguments. It shows that for the two complex architectures the usage of record types halves the maximum and average number of arguments. Higher-order and record arguments are only used by the complex architectures.

5.2 Runtime Evaluation

The runtime evaluation was done on an Apple Mac mini M2 Pro with 32 GB memory under macOS Ventura 13.4 using OpenJDK 64-Bit Server VM Temurin-17.0.6 and the newest version of the VADL tool. Figure 5 shows the runtime of the source parsing and macro expansion pass in milliseconds on the original source specification and on the expanded source code which is the result of the expansion of all macros.

Table 5. Runtime of the Macro System in milliseconds

	Original Source		Expanded Source
	Parse	Expand	Parse
AArch32	542	1144	2415
AArch64	579	213	782
MIPS IV	520	28	514
RISCV	449	10	454
TriLen	453	24	462

Each specification was executed 3 times and the minimal time was selected. The variance between the 3 runs was very low. The parse pass scans the text and generates the CST. The expansion pass traverses the CST, does macro expansion and generates an expanded CST. Additionally, we applied the same approach to the already expanded specifications to have a direct comparison between parsing with and without macros. This provides an idea for how much time is actually taken up by the macro system. Table 5 shows that the macro expansion adds almost no additional runtime for specifications with sparse use of macros. More surprisingly is the savings in runtime for *AArch32*. This can be explained by the increased I/O and parsing effort, compared to much faster in-memory expansion operations.

5.3 Reflection

In this section we would like to give a brief overview on interesting experiences we gained while investigating a macro system design for VADL.

To use VADL's existing IDE integration feature, the syntax errors must be detected by the parser. While most definitions, e.g. *models*, did not impose a real problem, we soon realized that *model* instantiations, and parameter uses are not safely parsable in LL(k) without some form of syntax type hints. These hints were essentially the syntax types of parameters or models written before an identifier, e.g. "$\$(Id\ a)$" or "$\$(Stmt\ b)$". While this helped in the context of grammar ambiguities, the identifiers may still fail in the latter applied expansion pass as they were never checked. Furthermore, we ran into similar issues when it came to parsing instantiation arguments, as they potentially allow a huge set of syntax rules. With the small extension to *pred-LL(*)* we were able to not only remove the unwanted syntax, but also guide the parser in a much cleaner manner. Now the parser guarantees syntactic correctness even before the macro expansion, all while preserving the IDE functionalities.

Another interesting realization was that a macro system for PDLs does not necessarily require hygienic macros. Most of the time, capturing identifiers is a desired behavior. For the remaining cases we preferred the control over the names provided by our *lexical* macro functions.

In contrast to GPLs, VADL macros are mainly used for code generation instead of language extension. *Lexical* macros have proven to be prone to errors, especially for larger specifications. *Procedural* macros are very powerful, but we have not yet discovered a use case where the resulting increase in complexity would be profitable. This is why we found the *pattern-based* macro approach extended with our *higher-order macros* a sensible compromise between complexity and implementation effort for PDL specification.

Finally, we would like to make a few comments about the uses of our macro elements. The *record* element improves the readability of our specification greatly as it enables us to group related parameters together. The *models* were mostly used to express variance of instruction semantic. The *match* statement together with the command line configurability were mainly used to model the variance of processors.

6 Related Work

Most of the related work we found that fits our class of macro system, i.e. *syntactical* and *pattern-based*, were focusing on syntactically rich GPLs and not simple DSLs. Our work took inspiration from *<bigwig>* [7] and programmable syntax macros [26]. We tried to capture some core ideas and incorporate them in our higher-order, composable syntax types in form of the presented VADL macro system.

One of the first approaches to bring syntax macros to syntactically rich languages or even languages outside the LISP family are programmable syntax macros for C [26]. The pattern-based macro system by Weise and Crew uses an extended version of the C language as macro language. Similar to our approach, a macro is defined by a meta construct providing the resulting syntax type, typed parameters and a template body with *placeholders*. Weise and Crew's macro headers enable parameter parsing in form of patterns, providing more syntactic freedom than our approach. However, this freedom may lead to unparsable macros if it cannot be deterministically parsed, which is always guaranteed by our approach. Both approaches rely on a context-sensitive parser to manage macros. Weise and Crew's macro system does not support such a rich type system as ours (e.g. higher-order, records, type-alias). Furthermore, they do not support alternatives in their templates.

The *<bigwig>* [7] is an extensible system for interactive web service. Similarities to our approach can be found in the usage of non-terminal grammar rules as syntax types and the way their macros and our *models* are defined. However, our approach differ in fundamental design decisions concerning the *metamorphisms*. Morphing new keywords into the host language's grammar would require costly rebuilds of our parser. As a result, we decided to increase the initial complexity by making our parser context-sensitive, but save ourselves the execution of a preprocessor. The *<bigwig>* macro system has a similar approach to our higher-order *models* by allowing *metamorph* rules as arguments. However, these rules are translated into grammar rules. They are not instantiated with syntax typed arguments, but guide the way on parsing the arguments. This allows for more syntactic freedom, for example arbitrary arity.

Similar to the previous mentioned techniques, the *ExJS* [25], a macro system for JavaScript, also splits its macro into a pattern and template part. Instead of *metamorph* non-terminals, it uses so called *phantom patterns* to establish arbitrary length repetition. Compared to us the template syntax types are rather poor, as it only supports *expressions* or *statements*. The biggest difference to us and the other approaches is the implementation. *ExJS* uses a first stage parser to build a second stage macro-aware parser to retrieve the AST. While similar implementations exist [4, 7], they further convert the AST to S-expression, feed them to a scheme macro expander and convert the resulting S-expression back to macro-free JavaScript.

The *Honu* [22] macro system follows a simpler approach when it comes to macro patterns and templates. Although the patterns are still syntactically more expressive than our simple parameter list, they follow a much simpler more LISP-like convention. The main idea presented is the *enforestation* parsing step, which converts a flat stream of tokens into an S-expression-like tree. It allows for LISP-style extensibility while still providing enough syntactic freedom to supporting macro infix operators.

One of the newer macro systems that is used in various fields ranging from DSLs to symbolic verification language extensions is the one from the Rust programming language [11, 13, 15, 17]. Specifically, the pattern-based *macro_rule!* is of interest compared to our work. It shares the technique of specifying a pattern with typed parameters that is rewritten based on a template. What stands out is that it supports more general and meta types, e.g. token-tree, pattern or item. Furthermore, the macros are invoked using their identifier and a trailing "!". The defined pattern is then provided inside parenthesis. This is closer related to our LISP-like macro invocation, in contrast to the previous mentioned approaches. As with the previous macro systems, Rust does not support higher-order macros.

7 Conclusion and Future Work

We have designed and implemented a type-safe higher-order macro system, specifically designed for (computationally weak) architecture description languages. Our presented pattern-based syntax macros are pred-LL(*) parsable and require no preprocessor or complex host language. The implementation was based on a context-sensitive top-down parser and an iterative expansion algorithm with termination guarantee. We demonstrated our approach using our work-in-progress specification language VADL. The evaluation of our macro system was conducted with the help of VADL based ISA specifications for the AArch64, AArch32, MIPS IV, RISCV and TriLen, our RISCV-like toy architecture.

Our macro system has still shortcomings we want to address in future work. First, the VADL module management and import system is currently put after the macro expansion, which limits the scope of a macro definition to a single file. Second, we would like to increase the syntactical freedom, for example variable arguments or arbitrary patterns, to see if they work well with our type system. Third, we want to extend our work by some form of hygienic macros. Finally, we want to continue our exploration of design possibilities for ADLs using macros.

Acknowledgment

We want to thank the reviewers for their constructive feedback and valuable suggestions. Part of this work was supported by a grant from Huawei.

References

[1] Norman I Adams IV, David H Bartley, Gary Brooks, R Kent Dybvig, Daniel Paul Friedman, Robert Halstead, Chris Hanson, Christopher Thomas Haynes, Eugene Kohlbecker, Don Oxley, et al. 1998. Revised report on the algorithmic language Scheme. *ACM Sigplan Notices* 33, 9 (1998), 26–76.

[2] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. 2005. The ArchC architecture description language and tools. *International Journal of Parallel Programming* 33 (2005), 453–484.

[3] Jonthan Bachrach and Keith Playford. 2001. The Java Syntactic Extender (JSE). In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Tampa Bay, FL, USA) *(OOPSLA '01)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/504282.504285

[4] Jonathan Bachrach, Keith Playford, and C Street. 1999. D-expressions: Lisp power, Dylan style. *Style DeKalb IL* (1999).

[5] Michael Ballantyne, Alexis King, and Matthias Felleisen. 2020. Macros for Domain-Specific Languages. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 229 (nov 2020), 29 pages. https://doi.org/10.1145/3428297

[6] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd.

[7] Claus Brabrand and Michael I. Schwartzbach. 2002. Growing Languages with Metamorphic Syntax Macros. *SIGPLAN Not.* 37, 3 (jan 2002), 31–40. https://doi.org/10.1145/509799.503035

[8] R Kent Dybvig. 1992. *Writing hygienic macros in Scheme with syntax-case.* Technical Report. Citeseer.

[9] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[10] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. 1997. ISDL: An Instruction Set Description Language for Retargetability. In *Proceedings of the 34th Annual Design Automation Conference* (Anaheim, California, USA) *(DAC '97)*. Association for Computing Machinery, New York, NY, USA, 299–302. https://doi.org/10.1145/266021.266108

[11] Kyle Headley. 2018. A DSL embedded in Rust. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages.* 119–126.

[12] Brian W Kernighan and Dennis M Ritchie. 1977. *The M4 macro processor.* Bell Laboratories Murray Hill, NJ.

[13] Steve Klabnik and Carol Nichols. 2023. *The Rust programming language.* No Starch Press.

[14] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming.* 151–161.

[15] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic sandboxing of unsafe components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems.* 51–57.

[16] Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* 52, 6, Article 113 (oct 2019), 39 pages. https://doi.org/10.1145/3354584

[17] Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No panic! Verification of Rust programs by symbolic execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 108–114.

[18] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[19] Terence Parr and Kathleen Fisher. 2011. LL (*) the foundation of the ANTLR parser generator. *ACM Sigplan Notices* 46, 6 (2011), 425–436.

[20] Terence J Parr and Russell W Quong. 1994. Adding semantic and syntactic predicates to LL (k): pred-LL (k). In *Compiler Construction: 5th International Conference, CC'94 Edinburgh, UK, April 7–9, 1994 Proceedings 5*. Springer, 263–277.

[21] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. 1999. LISA — machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference.* 933–938.

[22] Jon Rafkind and Matthew Flatt. 2012. Honu: Syntactic Extension for Algebraic Notation through Enforestation. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering* (Dresden, Germany) *(GPCE '12)*. Association for Computing Machinery, New York, NY, USA, 122–131. https://doi.org/10.1145/2371401.2371420

[23] Richard M Stallman and Zachary Weinberg. 1987. The C preprocessor. *Free Software Foundation* (1987), 16.

[24] Guy Steele. 1990. *Common LISP: the language.* Elsevier.

[25] Ken Wakita, Kanako Homizu, and Akira Sasaki. 2014. Hygienic Macro System for JavaScript and Its Light-Weight Implementation Framework. In *Proceedings of ILC 2014 on 8th International Lisp Conference* (Montreal, QC, Canada) *(ILC '14)*. Association for Computing Machinery, New York, NY, USA, 12–21. https://doi.org/10.1145/2635648.2635653

[26] Daniel Weise and Roger Crew. 1993. Programmable Syntax Macros. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 156–165. https://doi.org/10.1145/155090.155105

Received 2023-07-14; accepted 2023-09-03