Fast & Easy ASTs for Flexible Embedded Interpreters*

Michael Homer
michael.homer@vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

James Noble
kjx@programming.ac.nz
Creative Research & Programming
Wellington, New Zealand

Abstract

Self-hosted software language systems need to bootstrap core components such as data structure libraries, parsers, type checkers, or even compilers. Bytecode interpreters can load byte code files, while image-based systems can load in images of entire systems — Emacs, for example, does both. Bootstrapping is more of a problem, however, for traditional AST-based systems, especially when they must be portable across multiple host systems and languages.

In this short paper, we demonstrate how abstract syntax trees can quickly and easily be incorporated into the source code of an embedded interpreter. Our key insight is that a carefully engineered format enables textually identical ASTs to be valid across a wide spectrum of contemporary programming languages. This means languages can be self-hosted with very little bootstrapping infrastructure: only the host interpreter or compiler and a minimal default library, while the rest of the system is imported as ASTs.

This paper outlines our technique, and analyses the engineering design tradeoffs required to make it work in practice. We validate our design by describing our experience supporting the on-going development of GraceKit, which shares a single Grace parser across host language implementations from Java and JavaScript to Haskell and Swift and to Grace itself, and even more eccentric languages like Excel.

CCS Concepts: • **Software and its engineering** → *Interpreters*; *Interoperability*; *Reusability*; *Software design tradeoffs*; *Source code generation*; *Retargetable compilers.*

Keywords: polyglot, abstract syntax trees, metacircularity, self-hosting, Grace, portability

*This work is supported in part by the Royal Society of New Zealand Te Apārangi Marsden Fund Te Pūtea Rangahau a Marsden grant CRP2101.

MPLR '25, Singapore, Singapore

ACM Reference Format:
Michael Homer and James Noble. 2025. Fast & Easy ASTs for Flexible Embedded Interpreters. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3759426.3760977

1 Introduction

Bringing a new language implementation to life is a complex task. This is especially so for partially or fully "self-hosted" implementations, that is, the language is to be implemented in *itself* [19]. Such a "meta-circular" [1] implementation may be an interpreter or a compiler, and offers several key advantages over building the system in a seperate "host" language: the implementation can be easier to understand and modify by people who understand only the new target language; portable components can be shared across different language implementations; programmers often prefer to work within the target language they are building, rather than in older legacy host languages. While Lisp-family languages (the "second-oldest language still in common use" [38]) are the paradigmatic examples of self hosting, languages as diverse as BCPL [30], PL/0 [40]), and Smalltalk [5, 16] are often self-hosted. Self-hosted implementations face an obvious problem: the conceptual circularity of implementing a language in itself, typically manifested as a pragmatic problem of how best to build a self-hosted system in practice, without writing large swathes of host language code.

Our contribution is an "obvious in retrospect" [33] solution to the problem of complexity standing up a new implementation of a language: we represent ASTs in a format that can be directly embedded in a wide range of host languages. For example, the AST for a simple declaration "`object { def x = 1 }`" can be embedded into a host program as "`objCons(one(defDec("x",nil,nil,numLit(1))),nil)`". Section 2 outlines the core of this representation, and then section 3 validates this design, using as a running example GraceKit's parser, which is embedded identically across Java, Haskell, and JavaScript implementations, and is portable to Swift and even Excel.

Like many solutions that are "obvious in retrospect", while the idea may be simple the practical realisation is not, so in section 4 we contribute a detailed analysis of the engineering and design required for a format to be flexible enough *actually* to be embedded within a wide range of host languages.

In this paper we describe our experience with GraceKit, a portable framework for implementing the Grace programming language [2]. Grace is an object-oriented language with suspiciously conventional curly-brace syntax, along with multi-part method names and object-oriented semantics closer to Smalltalk. Grace's relatively small syntax and straightforward semantics make it particularly suitable for this experiment — although we expect this technique will be applicable to different target languages as well as simultaneously portable across different host languages.

2 A Flexible, Embeddable AST

Figures 1(a) and 1(b) show an example Grace program (generating every GraceKit AST node) and the corresponding GraceKit AST format. In one sense, our AST embedding is straightforward, drawing on the traditional syntax of function names followed by parenthesised comma-separated argument lists. In another sense, this embedding is carefully engineered to be as simple as possible, to capture all the constructs of the Grace AST, to a limited extent to be human-readable and editable, but above all to be directly embeddable across many different host languages.

AST nodes are represented as calls to methods in an abstract factory interface [14] (a.k.a. an object algebra [9]). Nested calls to the node functions represent the tree structure of the AST, with the arguments to each function call representing the children of that node. Once an interpreter or compiler has implemented the abstract interface, invoking the root AST node will recursively call the factory methods to build an AST to suit the host language implementation. This is why the interface is an *abstract* factory: the actual nodes of the reified AST may be embodied by lists, records, objects, (generalised) algebraic data types – whatever is most convenient in each host language. Table 1 in the Appendix outlines the entire API for the GraceKit AST.

3 Realisations

To validate our design, and to demonstrate its portability and flexibility, we have constructed bindings for the GraceKit abstract factory interface across eight different host languages (at time of writing). Java, Haskell, JavaScript, and Grace (self–hosted) support essentially interpreters for Grace; Swift, ML/F♯, and Perl are currently proof-of-concept demonstrators that can e.g. load ASTs and print them out, while Excel is more a proof-of-bad-concept.

Recall that the contribution of this work is to enable portability of ASTs embedded across different host programing languages — each host language still needs its own core interpreter for the internal AST structure it constructs. Given a core interpreter and some basic I/O primitives, a much larger Grace system can be bootstrapped, notably including a lexer and a parser, with little extra effort.

Although Grace is not a particularly complex language, nor is it particularly simple. Our proofs-of-concept demonstrate that identical textual renderings of an AST can be embedded and manipulated in each host language. The code of these implementations is available at https://github.com/mwh/wg.

3.1 Java

The **Java** implementation contains a standard tree-walking evaluator capable of executing embedded GraceKit ASTs. In particular, it is capable of executing the AST for the GraceKit parser itself, demonstrating that an interactive ("REPL" loop) interpreter can be bootstrapped without any native parser written in the host language. Alternatively, a fixed program AST can be hard-coded into a templated Java file with all of the run-time support code, and this file can be compiled or executed anywhere to run the original Grace program (see Section 3.3).

The GraceKit parser is written in Grace and has been structured to impose minimal library and runtime requirements, in order to ease this bootstrapping process. For example, it does not make use of the language's pattern-matching [22] or advanced string-processing features, although these are useful for a parser, because those would then need to be included in the host's initial implementation. It also does not rely on any annotations, visibility, or other modifiers that Grace includes having any impact at run time, so the host is free to ignore them temporarily, or permanently. Originally developed running on the preëxisting Kernan reference implementation of Grace [20] (an interpreter fully in C♯), this parser has been self-hosting in our Java implementation from a very early stage to validate the effectiveness of our approach.

3.2 Haskell

The **Haskell** implementation can also load the full parser or any other serialised AST. In Haskell (or other ML-like languages), the "argument lists" are in fact tuples, but precisely the same serialised syntax is used unchanged. Unlike the Java implementation, this implementation builds a continuation-passing style (CPS) evaluator rather than tree-walking the AST. The structure of the serialisation format does not constrain the nature of the evaluator, and two very contrasting languages are capable of taking in exactly the same encoded source code within their own syntax. With the parser compiled in, this implementation can also load and evaluate Grace source code in-process, constructing the necessary CPS functions from the parser's AST dynamically. Figure 2 shows a brief extract from the Haskell implementation.

3.3 JavaScript

The **JavaScript** implementation also contains an evaluator capable of evaluating the parser itself. The reflexive potential

```
import "ast" as ast

// This file makes use of all AST nodes

def x = object {
  var y : Number := 1

  method foo(arg : Action) bar(n) → String {
    self.y := arg.apply + n
    return "y @ {y}!"
  }
}

print(x.foo { 2 } bar 3)
```

```
objCons(cons(importStmt("ast", identifierDeclaration("ast", nil)), cons(
  ↪ comment(" This file makes use of all AST nodes"), cons(defDec
  ↪ ("x", nil, nil, objCons(cons(varDec("y", one(lexReq(one(part("
  ↪ Number", nil)))), nil, one(numLit(1))), one(methDec(cons(part("
  ↪ foo", one(identifierDeclaration("arg", one(lexReq(one(part("
  ↪ Action", nil)))))))), one(part("bar", one(identifierDeclaration("n",
  ↪ nil))))), one(lexReq(one(part("String", nil)))), nil, cons(assn(
  ↪ dotReq(lexReq(one(part("self", nil))), one(part("y", nil))), dotReq(
  ↪ dotReq(lexReq(one(part("arg", nil))), one(part("apply", nil))), one
  ↪ (part("+", one(lexReq(one(part("n", nil)))))))), one(returnStmt(
  ↪ interpStr(safeStr("y ", charAt, " "), lexReq(one(part("y", nil))),
  ↪ strLit(safeStr("", charExclam, "")))))))), nil)), one(lexReq(one(
  ↪ part("print", one(dotReq(lexReq(one(part("x", nil))), cons(part("
  ↪ foo", one(block(nil, one(numLit(2))))), one(part("bar", one(
  ↪ numLit(3))))))))))))), nil)
```

Figure 1. (a) A simple Grace program that covers every AST node, corresponding to the AST in (b). **(b)** An AST containing every GraceKit node type, corresponding to the Grace program in (a).

```
1  data ASTNode = ObjectConstructor [ASTNode] [String]
2  | VarDecl String [ASTNode] [String] [ASTNode]
3  | NumberNode Double
4  ...
5  cons = uncurry (:)
6  one hd = [hd]
7  nil = []
8  objCons (body, anns) = ObjectConstructor body anns
9  varDec (name, dtype, anns, val) =
10   VarDecl name dtype anns val
11 ...
12 toFunc :: ASTNode → (Context → IO ())
13 toFunc (NumberNode v) =
14   \ctx → (continuation ctx) (GraceNumber v)
15 ...
16 program = objCons(cons(
17   varDec("x", nil, nil, one(numLit(1))),
18   one(lexReq(one(part("print",
19     one(lexReq(one(part("x", nil))))))))), nil)
20 main = do
21     let func = toFunc program
22     func dropContext
```

Figure 2. An extract from the Haskell implementation of the evaluator for this serialisation.

of embedded ASTs raises an interesting opportunity in a language like JavaScript: because our ASTs are valid JavaScript, Grace code can be fed to the parser and the resulting output AST `eval`ed directly, *as JavaScript*, in the scope of the abstract factory interface, but without any further coordination to construct the in-memory AST structure. This AST can then be executed itself, also resulting in a complete interpreter for the language. This is a powerful demonstration of the flexibility of our approach. Any other language with an `eval` function could also use this strategy, and most of the well-established drawbacks of `eval` [29] will not apply, as the input ASTs are known to be well-formed and safe.

This implementation is accessible online as a web page, accessible at https://mwh.nz/demos/mplr2025/. This implementation demonstrates our approach in multiple ways. It can:

- Execute a Grace program directly, running the parser in-browser to create the AST and then evaluating it.
- Produce and display the textual format of a Grace program, which can then be copied and pasted into another host language.
- Produce a single-file self-contained Java, Haskell, or JavaScript program encapsulating both the serialisation of the entered Grace program and the run-time support code, which can then be compiled or run directly to execute that program using the host language's infrastructure (for example, by running `java GraceProgram.java` to execute the given Grace program on any system with Java).

While this last point is a convenient means of distributing and proving the functioning of our implementations, the Java and Haskell implementations on their own are more general and not confined to a single file or fixed program. It is possible to inspect the generated code to see that the *only* difference between the generated host language code

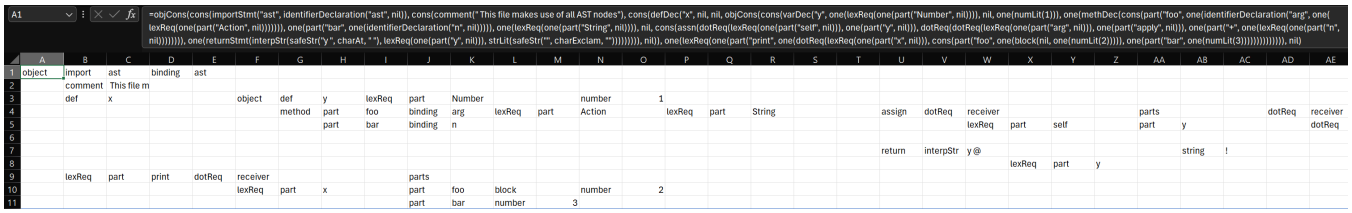

Figure 3. The AST from Figure 1(b) loaded into Excel. Cell A1 contains the full AST, with the displayed values in all other cells produced by the Excel implementation.

for different input programs is the single line of embedded AST at the bottom of the generated host language file — the rest of the file, being the AST interpreter and associated primitives, will be identical in all cases.

3.4 Excel

The **Excel** implementation is a more unusual case. While not a general-purpose programming language, the recent lambda and dynamic array [26, 39] features of Excel permit defining the functions necessary for an AST to be loaded into a spreadsheet cell. Like the other implementations, individual named functions are defined for each of the nodes. The AST itself is then produced spilling into adjacent cells, indented to show the structure of the tree. Figure 3 shows the AST from Figure 1(b) loaded into Excel, with all displayed cell values produced by the Excel implementation. The AST formula in cell A1 is visible in the formula bar. This implementation does not evaluate the AST, and it is not clear that it (practically) could do, but it is a demonstration of the portability of this approach that the resulting AST can be loaded into a spreadsheet at all! It is possible to perform some analyses on the loaded AST using the ordinary spreadsheet features of Excel, and this may be useful in some circumstances.

3.5 Swift

Swift is an interesting target, as the restrictions on its principal platform may make embedding an AST from another language desirable, but Swift also has some unusual syntactic and type features that mean it is not obvious that it is compatible. In particular, the use of Swift's Smalltalk-style parameter labels would be incompatible with the AST format, but these are not mandatory. `nil` is also a built-in polymorphic constant in Swift, however its uses are compatible with how `nil` is used in our ASTs. Our Swift implementation does not yet include an evaluator, but is capable of loading an AST and regenerating (i.e. pretty-printing) the *Grace source* that produced it.

3.6 Grace

Finally, our AST is valid in **Grace** itself. Our implementation of Grace-in-Grace uses the language's "dialects" [21] feature to bring the necessary functions and helpers into scope, allowing a three-line program to load and evaluate the translated code:

```
1 dialect "grast"
2 def program = objCons(one(lexReq(one(part("print", one(
      ↪ strLit("Hello, world")))))), nil)
3 run(program)
```

This is a toy example, but any Grace code, including the parser or even this Grace-in-Grace implementation itself, could be loaded in this way, opening up interesting self-hosting or sandboxing possibilities.

We expect many other "curly-bracket" languages (as Ward Cunningham once put it) will exhibit similar patterns to the languages listed above, allowing for a wide variety of languages to load a syntax tree in this format.

4 Design Considerations

The core goal of this format is to be simultaneously valid in multiple languages from different traditions, a "polyglot" program. Polyglot programs are not common in practice, but are sometimes used in recreational programming; for example, the Code Golf & Coding Challenges Stack Exchange site [31] has over 1,800 answers to polyglot challenges, including one that is valid in over 450 languages at once. These recreational polyglot exercises do not have the same constraints as a practical AST format! Instead, they typically rely on hiding some of the code from some of the languages, and embody a *single* well-controlled program, while our ASTs must be able to handle arbitrary user-defined input programs. This makes a portable encoding a more challenging problem than it might first appear. In this section, we discuss some of the complications that arise in the design of such a format.

4.1 String Literals

Grace programs may contain string literals, and various other constructs have textual names. These strings must be encoded in the AST, so we must be able to encode arbitrary user-defined content into host language string literals, which runs into a variety of complications across languages. In particular, string escape characters, escape sequences, and interpolation all raise distinct issues.

String interpolation is a feature of many languages where an expression or variable may be substituted into a string

literal, by writing some special syntax within the string. For example, Perl will interpolate variables prefixed by `$`, `@`, or `%` into double-quoted strings, while Swift takes whole expressions bounded by \(...); other languages use various combinations of braces, parentheses, and symbols. String literals in Grace source programs may include these characters or sequences, but they should not be interpolated when loaded from the AST in a target language. To avoid these problems, our encoding must prefix potential interpolations with an escape character, such as a backslash — but different host languages also have different rules for what can be escaped in this way! For example, in Perl "`\$`" is needed to represent a literal dollar sign, but in Java this is a fatal lexical error, as "`\$`" is not a Java escape sequence.

Strings may similarly contain double quote characters. Different languages provide different means for including double quotes into string literals, such as escaping with a backslash (e.g. C) or another escape character (e.g. PowerShell), doubling it (e.g. BASIC), or a generic character escape (e.g. FORTRAN). A portable AST encoding cannot use *any* of these constructs. Other escape sequences, such as `\n` for a newline, are similarly problematic and represented differently in different targets. To be able to transport ASTs across host languages, we must be able to represent these strings faithfully with only constructs shared by all target languages.

To permit encoding arbitrary strings, including those containing metacharacters, interpolations, or escape sequences that may cause issues in some target languages, we use a "safe string" format for any string that may contain such characters. As well as functions, several constants `charDQuote`, `charDollar`, ... are defined in the host, and the string is encoded as `safeStr("pre", charDQuote, "post")`. Our typical host implementation constructs the fixed string directly rather than representing this in the AST, so this construction only exists in the serialisation to ensure portability.

4.2 Node Sequences

Many AST nodes are containers for arbitrary-length sequences of nodes, such as the list of statements in the body of a method. With a single host language, there will often be an obvious approach available: either use variadic parameter lists, or use a list or array literal. Neither of these is usable for a portable serialisation format, however, because the syntax and availability of both constructions varies between languages. Instead, our AST must use a structure that is portable across as many host languages as possible.

For this reason, the abstract factory interface represents sequences using cons lists. A singleton list is represented as `one(node-expression)` and an empty list as `nil`. As with the other methods in the API, this does not constrain implementations necessarily to operate on linked lists. Rather, they can function as the interface to a Builder [14] constructing a sequence representation that suits the host language and API (e.g. using mutable lists, an immutable hash-cons'ed DAG [11], etc). In this way we trade concision for portability, not relying on specific host-language features but with a more verbose serialisation.

4.3 Language Keywords and Naming

The names of functions and terms in the serialisation must not conflict with any keywords or reserved names in any of the target languages. While this seems like a simple requirement, it is not entirely straightforward, especially with a goal of each name being relatively short. Obvious names for AST nodes like `string`, `method`, and `var` are reserved or built-in types in many languages, while more unusual alternatives that might be chosen for some term sometimes emerge as reserved in a single language. For example, one draft of the format used `no` as a marker for absent values, but this is a keyword in Perl. Ensuring that all names used are permissible in all target languages requires manual checking. Similarly, some languages impose other restrictions on names, such as case-sensitive semantics, or disallowing certain characters. These also impose constraints on the names that can be used in the serialisation format. Our design adopts mixed-case names for API elements, with the first letter lower-case, and all are at least three characters long (see Table 1 in the Appendix). More common nodes are given shorter names within these constraints, with the aim of being human-readable, with effort, while being concise to save space.

4.4 Type Correctness

To be portable to statically typechecked languages, the AST must be acceptable to the type checker in the host language. The catch of course is that different host languages have different type checkers: C's type checker is rudimentary compared with Haskell's expansiveness. Similarly, there are encodings (JSON) that are viable in dynamically-typed languages but that should be rejected by any self-respecting statically-typed language: we cannot use these.

We are well-served in that the host languages we hope to address not only can tolerate data structure "literals" in an Algol-like suffix-parenthesized syntax, but also that that syntax does not require type declarations at point of use. So while more advanced type systems like Family Polymorphism [10] or Generalized Algebraic Data Types [24] can better capture the finer details of the recursive and structural type relationships within syntax trees, we can get by with coarser-grained types, or indeed none at all. Host implementations may need to make notionally-unsafe casts, double dispatches, or similar to resolve types with a finer granularity. The ASTs we generate should always be valid in practice, so these casts should never fail, even in expansively typed languages [28].

4.5 Line Length

By default, we produce one long line representing the complete AST. As languages have different ideas about internal

line breaks and indentation, generating just one line lets us sidestep all these issues.

When embedded in actual source code text files for a target language, however, this can cause problems for some tools, such as text editors or language servers. Long lines may breach the POSIX definition of a text file [36], which only mandates supporting lines of at least 2048 characters. Similarly, a language may define thresholds for line length or nesting parentheses, and allow conforming implementations to reject programs exceeding those limits (for example, the C specification requires only 4,095 bytes in a line and 63 levels [23] of nesting).

In these scenarios, it is possible to format the code in any manner accepted by the host language, such as by breaking subexpressions across lines or indentation adjustments. In practice, no implementation we have made has *required* any of this massaging, even when embedding the GraceKit parser (whose AST is over 100KB) into a source file.

5 Discussion, Related Work, and Conclusion

As we mentioned in the introduction, simultaneously portable polymorphic ASTs are "obvious in retrospect" — so obvious, perhaps, that there is little analysis in the literature. Lisps (especially Scheme) have the strongest tradition of building self-interpreters [35], with the complementarity between data constructors and function calls being well known [32]. Scala incorporates the TASTY format for their ASTs, also built using an abstract factory, however this is a binary format rather than a portable multilingual program [34].

Based originally on the self-interpreter at the end of the Smalltalk language book [16], Smalltalk systems often use special-purpose domain-specific languages to implement primitive operations or to extend virtual machines [27]. In some sense this is the exact reverse of the approach we propose here: we propose a single common serialisation that works *as is* across many implementation languages — while DSLs are single unique representations that are not natively supported by any implementation languages, but must be transcoded or transpiled [3, 4, 13]. Similar DSLs have been used to support or extend Python [12] and Lua [17, 18], while GOOL takes this to the extreme, with a "generic" syntax embedded in Haskell that generates code in a range of object-oriented languages [7]. Selfie [25] implements a self-hosted subset of C, along with a parser and bytecode VM, in a single C source file. Many other systems, including especially Ensō and POPLog [6], have focussed on self-hosted support for languages with more syntax than just S-expressions or FORTH postfix operator streams. Simultaneously portable polyglot ASTs could also be useful in attempts to "debootstrap" language implementations [8].

We have constructed this system for the Grace language, but the techniques used should be applicable to any language that can be represented as an AST. A similar embeddable format for a different language would use a different set of node-constructing functions, but the general principles, and particularly the engineering considerations addressed in Section 4, would be the same. While in this work we have focused on direct representations of abstract syntax trees, the same techniques could also be used for lower-level intermediate representations.

This format offers some interesting possibilities for self-hosting systems. Because the parser is written in the language it parses, it can be bootstrapped via an embedded AST with only a straightforward AST interpreter, plus I/O primitives, on the host side. After this, though, a more interesting avenue emerges: because the parser will produce ASTs as valid Grace objects, those ASTs can be fed into *other* Grace programs — such as a type checker or macro rewriter — also written in Grace, and also embedded as an AST. This allows for a very simple bootstrapping process, where only the core interpreter is needed in the host language to get up and running, while other phases are written in the source language and shared across diverse host languages. These phases do not need to be aware of each other, so long as they use the same AST. Furthermore, as an AST can be fed back into a different implementation of the abstract factory interface, these phases do not even need to share the same internal structures (object algebras [9], metamorphisms [15]).

We expect the general scheme of embedding ASTs should work across almost all programming languages, or at least all textual programming languages. While our AST design covers a wide class of "Algol-like" host languages – and simple textual substitutions should enable support of even more (Smalltalk and Pascal, for example, delineate comments with double quotes "" and strings with single quotes ''), there are families of language syntax with which our syntax is less easily compatible, such as Lisp, TeX, Forth, and most command languages. These languages have fundamentally different syntax regarding parentheses, commas, or ordering of terms. Some of these languages' advanced features (such as Lisp's reader macros [37], Forth's parsing words, or TeX's catcodes) should even permit embedding our Algol-like ASTs directly — we leave this for future work, since we lack the necessary lifetimes of hermetic background. Other languages, such as Unix shells or Tcl, would likely require the design to be reworked to fit that context — although the principles from section 4 will still apply.

To conclude: a carefully engineered polyglot AST serialisation format can be a powerful tool for aiding the implementation and embedding of programming languages, particularly in self-hosting systems. This short paper is a demonstration of this vision, and we hope that it will inspire further work in this area.

A AST API

Table 1. GraceKit AST Abstract Factory Interface

ObjectConstructor [ASTNode] [String]	An object constructor.
VarDecl String [ASTNode] [String] [ASTNode]	Variable declaration: name, type, annotation, initialiser.
DefDecl String [ASTNode] [String] ASTNode	Constant definition: name, type, annotation, initialiser.
ExplicitRequest ASTNode [Part]	method request with an explicit receiver ("dotted call")
LexicalRequest [Part]	method request with an implicit receiver ("undotted call")
NumberNode Float	Number literal
Block [ASTNode] [ASTNode]	Closure with parameters and body
MethodDecl [Part] [ASTNode] [String] [ASTNode]	A method declaration: names, types, annotation, body
Assign ASTNode ASTNode	Assignment
ReturnStmt ASTNode	Return
IdentifierDeclaration String [ASTNode]	Declaration of String as an identifier
StringNode String	Plain String literal
InterpString String ASTNode ASTNode	Interpolated String Literal
Comment String	Comment
ImportStmt String ASTNode	Import statment to load a module
DialectStmt String	Dialect statement to declare a sublanguage
TypeDecl String ASTNode	Type declaration
InterfaceConstructor [MethodSignature]	Interface declaration
MethodSignature [Part] (Maybe ASTNode)	Grace multi-part method signature
Part String [ASTNode]	An individual part of a method signature.

Table 2. Variable Element List Interface

Nil	Empty List or missing optional item.
One Item	Singleton List.
Cons Head List	Cons a new Head onto a List.

Table 3. String Escapes

charDollar	$
charBackslash	\
charDQuote	"
charLF	Line Feed
charCR	Carriage Return
charLBrace	{
charStar	*
charTilde	~
charBacktick	`
charCaret	^
charAt	@
charPercent	%
charAmp	&
charHash	#
charExclam	!

References

[1] Harold Abelson and Gerald Jay Sussman. 1985. *Structure and Interpretation of Computer Programs.* MIT Press and McGraw-Hill.

[2] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. 2012. Grace: The Absence of (Inessential) Difficulty. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Tucson, Arizona, USA) *(Onward! '12)*. ACM, New York, NY, USA, 85–98. https://doi.org/10.1145/2384592.2384601

[3] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. 2008. Back to the Future in One Week — Implementing a Smalltalk VM in PyPy. In *Self-Sustaining Systems*.

[4] Tom Braun, Marcel Taeumel, Eliot Miranda, and Robert Hirschfeld. 2023. Transpiling Slang Methods to C Functions: An Example of Static Polymorphism for Smalltalk VM Objects. In *VMIL* (Cascais, Portugal) *(VMIL 2023)*. 88–93. https://doi.org/10.1145/3623507.3623548

[5] Tim Budd. 1987. *A Little Smalltalk.* Addison-Wesley.

[6] R.M. Burstall, J.S. Collins, and R.J. Popplestone. 1968. *POP2 Papers.* OLIVER & BOYD.

[7] Jacques Carette, Brooks MacLachlan, and Spencer Smith. 2020. GOOL: a generic object-oriented language. In *PEPM@POPL*. 45–51.

[8] Nathanaëlle Courant, Julien Lepiller, and Gabriel Scherer. 2022. Debootstrapping without Archeology - Stacked Implementations in Camlboot. *Art Sci. Eng. Program.* 6, 3 (2022), 13.

[9] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *ECOOP*.

[10] Erik Ernst. 2001. Family polymorphism. In *European Conference on Object-Oriented Programming*. Springer, 303–326.

[11] A.P. Ershov. 1958. On programming of arithmetic operations. *CACM* 1, 8 (1958), 3–6.

[12] Greg Ewing. 2010. Pyrex - a Language for Writing Python Extension Modules. www.csse.canterbury.ac.nz/greg.ewing/python/Pyrex/, accessed July 2024.

[13] Bert Freudenberg, Dan H.H. Ingalls, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. 2014. SqueakJS: a modern and practical Smalltalk that runs in any browser. In *ACM Symposium on Dynamic Languages (DLS)a*. https://doi.org/10.1145/2661088.2661100

[14] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1994. *Design Patterns.* Addison-Wesley.

[15] Jeremy Gibbons. 2007. Metamorphisms: Streaming representation-changers. *Sci. Comp. Prog.* (2007).

[16] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

[17] Hugo Musso Gualandi and Roberto Ierusalimschy. 2020. Pallene: A companion language for Lua. *Sci. Comput. Program.* 189 (2020).

[18] Hugo Musso Gualandi and Roberto Ierusalimschy. 2022. A surprisingly simple Lua compiler - Extended version. *J. Comput. Lang.* 72 (2022).

[19] Tim Hart and Mike Levin. 1962. *The New Compiler.* Technical Report AIM-39. MIT Artificial Intelligence Laboratory. ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-039.pdf

[20] Michael Homer, Timothy Jones, and James Noble. 2015. From APIs to Languages: Generalising Method Names. In *Dynamic Language Symposium*. https://doi.org/10.1145/2816707.2816708

[21] Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. 2014. Graceful Dialects. In *ECOOP 2014 — Object-Oriented Programming*, Richard Jones (Ed.). Lecture Notes in Computer Science, Vol. 8586. Springer Berlin Heidelberg, 131–156. https://doi.org/10.1007/978-3-662-44202-9_6

[22] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. 2012. Patterns As Objects in Grace. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. ACM, New York, NY, USA, 17–28. https://doi.org/10.1145/2384577.2384581

[23] ISO/IEC. 2023. ISO/IEC 9899:2023, Programming languages, environments and system software interfaces – C. Section 5.2.5.2 "Translation limits".

[24] Andrew Kennedy and Claudio V. Russo. 2005. Generalized algebraic data types and object-oriented programming. In *OOPSLA*. 21–40. https://doi.org/10.1145/1103845.1094814

[25] Christoph M. Kirsch. 2017. Selfie and the basics. In *Onward!* 198–213. https://doi.org/10.1145/3133850.3133857

[26] Matt McCutchen, Judith Borghouts, Andy Gordon, Simon Peyton Jones, and Advait Sarkar. 2020. Elastic Sheet-Defined Functions: Generalising Spreadsheet Functions to Variable-Size Input Arrays. *Journal of Functional Programming* 30, e26 (August 2020). https://www.microsoft.com/en-us/research/publication/elastic-sheet-defined-functions-generalising-spreadsheet-functions-to-variable-size-input-arrays/

[27] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. 2018. Two decades of Smalltalk VM development: live VM development through simulation tools. In *VMIL* (Boston, MA, USA) *(VMIL 2018)*. https://doi.org/10.1145/3281287.3281295

[28] James Noble. 2023. All Languages Are Dynamic (Invited Talk). In *ACM Symposium on Dynamic Languages (DLS)*. 1.

[29] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer-Verlag, Berlin, Heidelberg, 52–78.

[30] Martin Richards and Colin Whitby-Stevens. 1980. *BCPL: the language and its compiler.* Cambridge University Press.

[31] Stack Exchange Inc. 2025. Answers matching '[polyglot] is:answer' - Code Golf Stack Exchange. https://codegolf.stackexchange.com/search?q=%5Bpolyglot%5D+is%3Aanswer; accessed 2025-08-20.

[32] Guy L. Steele. 1976. Lambda: The Ultimate Declarative. research.scheme.org/lambda-papers/, accessed July 2024..

[33] G. Steinem and E. Watson. 2019. *Outrageous Acts and Everyday Rebellions: Third Edition.* Picador.

[34] Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. Virtual ADTs for portable metaprogramming. In *MPLR*. 36–44.

[35] Gerald Sussman and Guy Steele. 1975. *SCHEME: An Interpreter for Extended Lambda Calculus.* Technical Report AI Memo 349. MIT Artificial Intelligence Laboratory.

[36] The IEEE and The Open Group. 2024. IEEE Std 1003.1-2024 / The Open Group Base Specifications Issue 8. Section 3.387 "Text File".

[37] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 132–141. https://doi.org/10.1145/1993498.1993514

[38] Wikipedia contributors. 2025. Lisp (programming language). https://en.wikipedia.org/w/index.php?title=Lisp_(programming_language)&oldid=1297605667. [Online; accessed 22-August-2025].

[39] Jack Williams, Nima Joharizadeh, Andy Gordon, and Advait Sarkar. 2020. Higher-Order Spreadsheets with Spilled Arrays. In *European Symposium on Programming*. https://www.microsoft.com/en-us/research/publication/higher-order-spreadsheets-with-spilled-arrays/

[40] Niklaus Wirth. 1976. *Algorithms + Data Structures = Programs.* Prentice-Hall.

Received 2025-06-24; accepted 2025-07-28