Execution vs. Parse-Based Language Servers

Tradeoffs and Opportunities for Language-Agnostic Tooling for Dynamic Languages

Stefan Marr
s.marr@kent.ac.uk
School of Computing
University of Kent
United Kingdom

Humphrey Burchell
h.burchell@kent.ac.uk
School of Computing
University of Kent
United Kingdom

Fabio Niephaus
fabio.niephaus@oracle.com
Oracle Labs
Potsdam, Germany

Abstract

With the wide adoption of the language server protocol, the desire to have IDE-style tooling even for niche and research languages has exploded. The Truffle language framework facilitates this desire by offering an almost zero-effort approach to language implementers to providing IDE features. However, this existing approach needs to execute the code being worked on to capture much of the information needed for an IDE, ideally with full unit-test coverage.

To capture information more reliably and avoid the need to execute the code being worked on, we propose a new parse-based design for language servers. Our solution provides a language-agnostic interface for structural information, with which we can support most common IDE features for dynamic languages.

Comparing the two approaches, we find that our new parse-based approach requires only a modest development effort for each language and has only minor tradeoffs for precision, for instance for code completion, compared to Truffle's execution-based approach.

Further, we show that less than 1,000 lines of code capture enough details to provide much of the typical IDE functionality, with an order of magnitude less code than ad hoc language servers. We tested our approach for the custom parsers of Newspeak and SOM, as well as SimpleLanguage's ANTLR grammar without any changes to it. Combining both parse and execution-based approaches has the potential to provide good and precise IDE tooling for a wide range of languages with only small development effort. By itself, our approach would be a good addition to the many libraries implementing the language server protocol to enable low-effort implementations of IDE features.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**; *Frameworks.*

Keywords: language server protocol, comparison, parse time, execution time

ACM Reference Format:
Stefan Marr, Humphrey Burchell, and Fabio Niephaus. 2022. Execution vs. Parse-Based Language Servers: Tradeoffs and Opportunities for Language-Agnostic Tooling for Dynamic Languages. In *Proceedings of the 18th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '22), December 07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3563834.3567537

1 Introduction

A bit more than half a decade ago, the language server protocol (LSP)[1] was proposed as a way to provide IDE features such as code navigation, completion, and error reporting independent of a specific IDE. In contrast to previous approaches, the LSP enables language developers to implement such IDE features once for a language, and use the resulting language server in various editors and IDEs, including Atom, Eclipse, Emacs, Visual Studio, and VIM. In the past, one would need to implement separate plugins for each IDE.

Soon after the LSP was announced, we started wondering[2] how one could utilize language implementation frameworks such as Truffle [Würthinger et al. 2012] to not just make it simpler to have IDE support for a language in many different editors and IDEs with a single implementation, but simplify the creation of such language servers by identifying the language-agnostic parts.

Stolpe et al. [2019] built a first version of such a system for the Truffle language framework, which was adapted and released as the GraalVM Language Server.[3] Their key idea was to utilize the instrumentation [Van de Vanter et al. 2018] and language interoperability [Grimmer et al. 2018] features of the Truffle framework to enable IDE features. This approach minimized the extra effort for language implementers while

[1] *Language Server Protocol*, Microsoft, https://microsoft.github.io/language-server-protocol/
[2] *Can we get the IDE for free, too?*, S. Marr, https://www.stefan-marr.de/2016/08/can-we-get-the-ide-for-free-too/
[3] *GraalVM Language Server Protocol*, Oracle, https://www.graalvm.org/22.1/tools/lsp/

still enabling useful tooling in IDEs. However, to minimize the implementation effort, the approach relies solely on what is available in a language's runtime, and it requires that the code that is being worked on is executable and ideally fully covered by unit tests to retrieve all available details. In some way, this moves some of the burden for tooling to the developer using the tools, who have to care more to produce well-formed and well-tested code to get a rich tooling experience. Another limiting factor is that language runtimes rarely keep information irrelevant for execution. Thus, declarative aspects such as where classes or fields are defined, code comments with documentation, or the semantics of tokens for highlighting, may not be preserved at run time and thus cannot be shown in an IDE using this approach.

In this paper, we explore how we can trade a little extra effort from the language implementers for more complete tooling for dynamic languages. Instead of relying solely on run-time information, we will demonstrate how to design a language server framework so that only a small amount of information needs to be extracted as part of the parsing and compilation step of a language implementation. We argue that we can realize most common IDE features for an order of magnitude less effort than ad hoc language servers.

With Stolpe et al.'s approach arguably achieving support for the most common and perhaps highest priority IDE features with minimal effort, we chose it as a baseline for comparison and will focus on the same set of IDE features. Thus, we will discuss how to support listing symbols for instance in an outline view, enable goto definition, referencing and highlighting elements related to a currently selected one, communicating syntax errors, providing help for method signatures, as well as general support for displaying hover information, semantic highlighting, and code completion. The paper's contributions are

- a design for a language-agnostic language server that provides IDE features common for dynamic languages based on an interface that can be used from custom parsers, compilers, as well as parsers generated e.g. by ANTLR, and independent of any language implementation framework;
- a detailed analysis of tradeoffs between parse and execution-based approaches and the opportunities of combining them;
- an implementation demonstrating our parse-based[4] approach for the languages SOM, Newspeak, and SimpleLanguage.

Furthermore, we find that we need less than 1,000 lines of code to implement support for a language. Thus reaching the goal of requiring only little more effort than the about 500 lines of code for the execution-based approach. Based on a survey of 20 open source language servers, this is an order of magnitude less code than needed for the median ad hoc language server (see section 7.3). A brief performance assessment confirms that parsers not designed for IDE use are practical, even for larger files (see section 8).

Overall, we believe our design would be a useful addition to many libraries implementing the language server protocol, since it enables low-effort language server implementations.

2 Background

This section reviews the language server protocol, discuss alternative approaches such as language workbenches, and summarizes the work by Stolpe et al. [2019].

2.1 Language Server Protocol

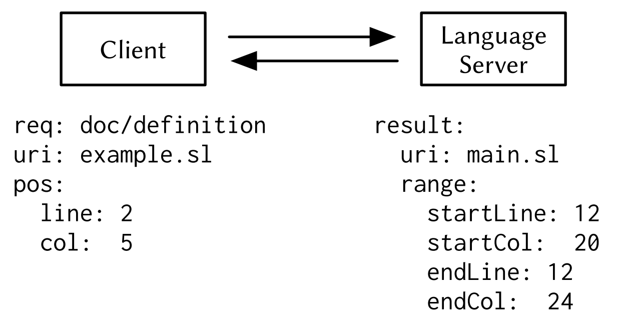
Announced in June 2016, the language server protocol (LSP) has gained wide adoption in editors, IDEs, and many language communities implemented language servers. In June 2022 at the time of writing, version 3.17 of the LSP was current. It supports document-related events, support for windowing features such as notifications, log messages, and progress updates, as well as a wide range of language and workspace-related features that are our focus.

The workspace features include events around file changes, creation, renaming, deletion, and similar, but also workspace-wide listing and resolving of language symbols.

The language-related features include classic ones: navigating to a declaration or definition, finding of references, requesting call and type hierarchies, highlighting related elements in an editor, tooltips or hover information, selection or folding of ranges, code completion, method signature help, diagnostic information, i.e, errors and warnings, and document renaming and formatting. It also covers more modern features such as code lenses, semantic highlighting, inline values as typically displayed during debugging, and inlay hints like parameter names of methods. Arguably, this covers most, if not all common, features IDEs provide these days.

The protocol itself is based on JSON-RCP and consists of requests, responses, and notifications. To be as language-agnostic as possible, the protocol tries to communicate location information and the information to be displayed without encoding specific semantics. Thus, responses typically contain code ranges, strings to be shown, as well as meta information to enable an IDE to differentiate language concepts, e.g., methods and fields, with different icons. Code ranges are represented as pairs of line and column to indicate start and end of a section, for instance to indicate where an identifier is in a file. The protocol does not generally encode structural details, except for basic containment, e.g., methods belonging to a class, for an outline view. This also means, language-specific constructs such as abstract syntax trees are not exposed to the editor. Instead, for instance semantic highlighting is realized as list of tuples that characterize tokens based on offsets. Figure 1 illustrates this with

[4] When referring to parse time in this paper, we consider any type of semantic analysis to be part of it.



```
req: doc/definition           result:
uri: example.sl                 uri: main.sl
pos:                            range:
  line: 2                         startLine: 12
  col:  5                         startCol:  20
                                  endLine: 12
                                  endCol:  24
```

Figure 1. Example interaction in the Language Server Protocol. The client requests the definition for a symbol at a position in a document identified by line and column. The server responds with a result, identifying the file where the symbol is defined, and the range at which it is found. The example uses pseudo code for legibility and conciseness.

an example for an interaction when an IDE users would try to navigate to the definition of a symbol.

Very similar to the LSP is Monto [Keidel et al. 2016], which also solves the issue that each language needed to implement separate plugins for each IDE with a similar protocol.

2.2 Language Workbenches

Language Workbenches [Erdweg et al. 2015] aim to simplify the creation of programming languages. Often they provide a set of languages or tools to define new languages. This can include a way to specify syntax, scoping rules, typing systems, and even the full execution semantics. For example, Rascal [Klint et al. 2009], Spoofax [Kats and Visser 2010], and Xtext [Eysholdt and Behrens 2010] also derive IDE features from the languages implemented in them. The IDE feature set is similar to that of the LSP. Xtext for example directly supports the LSP to provide these IDE features. The Rascal-based Bacatá [Merino et al. 2020] uses the LSP to bring IDE features to computational notebooks.

When designing a new language, some of these workbenches can make it unnecessary to implement a language server. Though, for language implementers that can not use such workbenches, perhaps when building IDE support for an existing language, other approaches such as the one proposed in this paper will remain relevant.

2.3 A Language-Agnostic Design of a Execution-Based Language Server

Since we rely on Stolpe et al. [2019]'s work, this section briefly summarizes their approach and what a language implementer needs to do to benefit from it.

Their approach to provide IDE features utilizes Truffle's support for instrumenting language implementations [Van de Vanter et al. 2018] and its language interoperability facilities [Grimmer et al. 2018].

The basic idea is to execute the code shown in the IDE and obtain the necessary information about symbols, definitions, source locations, documentation, variables, or method signatures from the annotated and instrumented abstract syntax tree as well as the run-time objects of a language, taking inspiration from Haupt et al. [2011]. For this purpose, they execute the code either using unit tests, or if none are available, the well-known entry points, such as main methods. To avoid side effects, everything is executed in a sandbox. Their sandboxing approach also provides the necessary support to execute the edited files based on their in-memory versions. Since execution can take noticeable amounts of time, previously harvested information will be cached.

SimpleLanguage is a small language to document how the Truffle framework can be used. Stolpe et al. use it as a running example. It is dynamically typed and has first-class functions, local variables, various control structures, and objects with dynamically added fields. Though, it does not have any notion of classes or types.

The language server implementation uses the Truffle framework to parse a file whenever the IDE sends file-open or changed events. When parsing fails, the error is reported as a diagnostic to the client.

While parsing, functions are collected by the framework and tools get access to the abstract syntax tree (AST). "Functions" here includes also methods, and things such as static initializers depending on a language. The AST nodes belonging to a function are language-specific. To have language-agnostic information about their semantics, the nodes can be tagged, which is then used for tooling. A node also encodes the code range it originates from.

To identify for example variable declarations, AST nodes are tagged with the `DeclarationTag` and the corresponding node can implement the `getNodeObject()` method which can provide additional details such as the kind of language element declared, and any statically known types.

Function calls and similar are identified by the `CallTag` on the corresponding AST nodes. The corresponding function definitions are either identified by name, based on the concrete functions called at run time, or using `findTopScopes()`, Truffle's API to look for the name in global scopes. Once candidate objects representing for instance the functions are found, the source position where they are defined can be requested with Truffle's `findSourceLocation()` method.

Variables are tagged with `ReadVariableTag` or `WriteVariableTag` to identify all references to a specific variable.

Truffle keeps details on nested scopes to support debuggers. It can be queried with `findLocalScopes()` to correctly distinguish symbols based on the language's scoping rules.

To answer an IDE's request for documentation and signature help, the run-time objects need to implement the `getDocumentation()` and `getSignature()` methods. The information is enriched with run-time type information that can be queried using Truffle's `findMetaObject()` function.

For code completion, completion candidates are determined using the scoping-related functions at the point in the source where there request originates. To distinguish different kinds of entities in the IDE, the language interop mechanism is used to identify the entity as for instance *callable* or *instantiable*. A language can further advice an IDE on when code completion should be triggered by implementing the `getCompletionTriggerCharacters()` method.

Code completion for object property accesses needs run-time information. Since a file typically has a parse error when requesting the completion, for instance after typing a dot, a previous version is used to try to obtain a run-time object and enumerate its members via the language interop API.

Overall, a language implementer needs to add support for four different tags (`DeclarationTag`, `CallTag`, `ReadVariableTag`, `WriteVariableTag`) on AST nodes, and implement four methods (`getNodeObject`, `getDocumentation`, `getSignature`, `getCompletionTriggerCharacters`) that are specific to the IDE support. The other APIs have other uses for tools such as Truffle's debugger or language interop support. While these APIs may require substantial implementation effort, we exclude their implementation effort from further consideration, since they are not only used for IDE support.[5] In their absence, one may want to consider our parse-based approach to support IDE features more directly, as discussed in the remainder of this paper.

Overall, we find that the effort needed to implement IDE-specific support is minimal when using the approach proposed by Stolpe et al. [2019].

3 A Language-Agnostic Design for a Parse-Based Language Server

Our goal is to devise a design for a language server that keeps the effort to gain IDE support as small as possible, while maximizing the utility of the IDE features. Thus, we aim to keep the increase in the implementation effort compared to Stolpe et al.'s execution-based approach as small as possible.

3.1 Architectural Overview

Figure 2 gives a high-level overview of language servers in the context of the language server protocol. The LSP connects a language server with IDEs (see section 2.1) and is used for instance to communicate the file content of the file currently being edit. In the simplest form, an IDE will send the complete content of a file. It will also send requests for instance for code completion for which it then will receive a result. A language server may also send notifications, for instance to report the results of a lint tool.

While the execution-based approach requires a language's interpreter, tagged AST nodes, as well as the implementation of Truffle's debugging and interop APIs, for our parse-based approach, one needs only a parser that creates a *collection of structures*, which encodes the relevant information for an IDE. This collection of structures is then queried by the language-agnostic part of the language server, whenever the IDE requests information. Thus, a key insight here is that the needed information can be represented with language-agnostic structures to enable IDE services.

Though, our main focus is on the design of the API that enables us to populate this collection of structures with minimal effort during parsing.

3.2 API Overview

The language-agnostic API to record the structural information for a file is illustrated in fig. 3. It defines only two new concepts: an element and an element id. However, it borrows concepts such as tokens and diagnostics from the LSP, which are indicated with the LSP prefix for type names. The LSP's enum types identify for instance what kind of element it is, as defined in the protocol. The others merely group descriptive information that is used to either locate an element in a file or display information to the user. The LSP types do not generally encode structure or semantics directly (see section 2.1).

With our API, most IDE features can be directly derived from the notion of the language element, or as the LSP calls it, a symbol. Elements can be contained in other elements, and have an id, name, what kind of element they are, a detail string, a signature descriptor, an overall range of code where its definition can be found, and the range of code that should be selected when navigating to it, typically the name.

When parsing and analyzing an input file, the `Structures` class is used to record the information. It provides methods that record elements either in one or in two steps. This is useful to capture the nesting of elements. Thus, at one point the definition is started, and it is completed only later, for instance once its closing parenthesis is parsed. The nesting structure can then for instance be displayed in an outline view. The single-step definition also takes a bool to indicate whether the definition is *after navigation*. This enables us to model for instance object fields.[6]

In addition to recording their definition, we also record when elements are referenced. They can be referenced either indirectly with an element id, or directly, for instance when a class is named and the parser can directly resolve it.

Element ids encode equality classes for language concepts. By subclassing `ElementId`, we distinguish different types of elements, e.g. classes, methods, and local variables, without introducing a notion of lookup, lexical scopes, or similar into the API, keeping the effort of implementing the recording for a new language low. While this does not provide sufficient structure to implement concepts such as types, in

[5]We had them already implemented, because debugging and profiling support were more urgent for our uses.

[6]While this approach is somewhat naive, it works well for SimpleLanguage, Newspeak, and SOM.

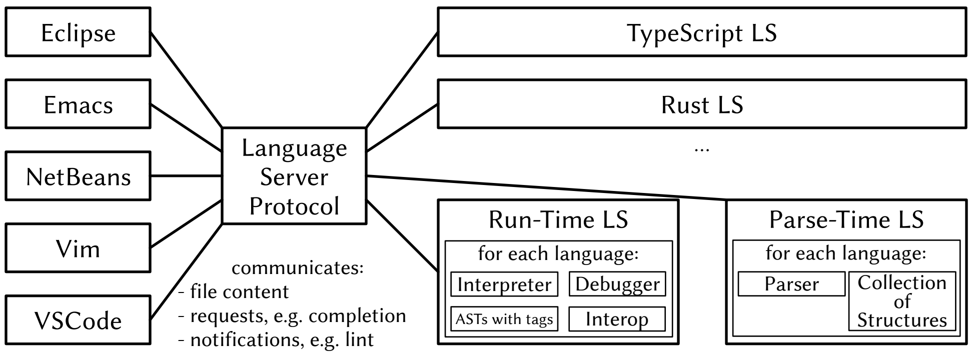
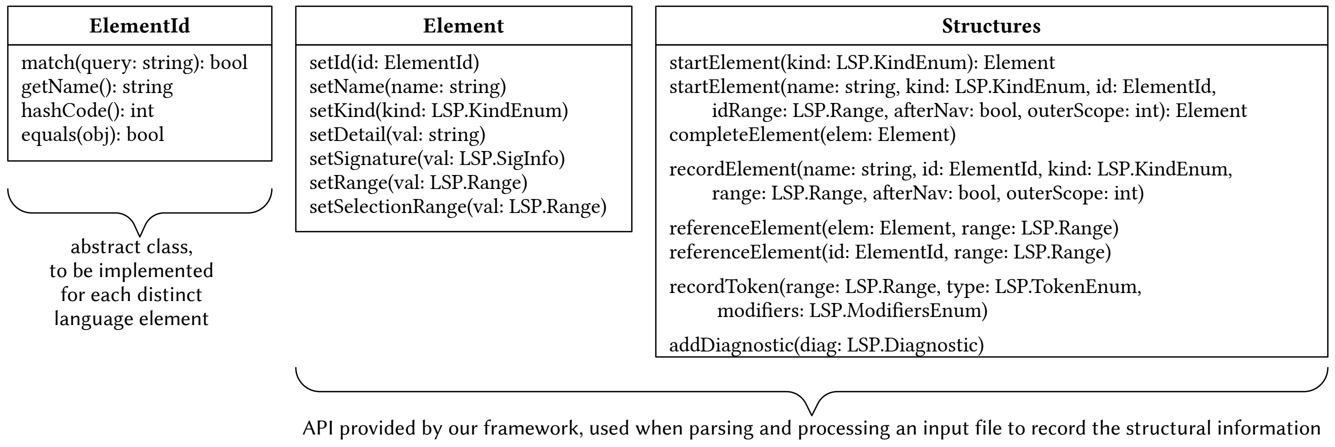


Figure 2. High-level overview of the Execution and Parse-based Language Servers. The language server protocol connects IDEs with language servers communicating file content, requests (e.g. for code completion), and notifications (e.g., for lint information). The language servers are commonly specific to a language such as TypeScript or Rust. The language-agnostic approaches differ in their key components.



```
ElementId
match(query: string): bool
getName(): string
hashCode(): int
equals(obj): bool
```

```
Element
setId(id: ElementId)
setName(name: string)
setKind(kind: LSP.KindEnum)
setDetail(val: string)
setSignature(val: LSP.SigInfo)
setRange(val: LSP.Range)
setSelectionRange(val: LSP.Range)
```

```
Structures
startElement(kind: LSP.KindEnum): Element
startElement(name: string, kind: LSP.KindEnum, id: ElementId,
      idRange: LSP.Range, afterNav: bool, outerScope: int): Element
completeElement(elem: Element)
recordElement(name: string, id: ElementId, kind: LSP.KindEnum,
      range: LSP.Range, afterNav: bool, outerScope: int)
referenceElement(elem: Element, range: LSP.Range)
referenceElement(id: ElementId, range: LSP.Range)
recordToken(range: LSP.Range, type: LSP.TokenEnum,
      modifiers: LSP.ModifiersEnum)
addDiagnostic(diag: LSP.Diagnostic)
```

abstract class, to be implemented for each distinct language element
API provided by our framework, used when parsing and processing an input file to record the structural information

Figure 3. Overview of the API of our parse-based Language Server. These concepts are sufficient to support all common IDE features with minimal development effort for a language implementer. Subclasses of `ElementId` need to be implemented for each distinct language element. `Element` and `Structures` are provided by our framework, and are used to capture all needed information for the language server. Types with the LSP-prefix correspond to the structures in the language server protocol, and typically consist of source locations and string labels to be shown to the user in the IDE.

practice, it results in a highly useful developer experience for dynamic languages. Furthermore, we assume that a parser distinguishes different variables in some way. When it represents them as objects or a unique identifier, an element id only needs to compare these representations to distinguish variables correctly, too. See section 4 lst. 2 for an example.

Tokens are recorded to enable semantic highlighting. Tokens distinguish different kinds, e.g., for a local access, a field access, or a keyword, as well as possible modifier, such as static, abstract, or documentation. The IDE will use this token information for syntax highlighting. Since we can use the semantic analysis from the language's parser/compiler to disambiguate for instance accesses to fields and local variables, the resulting highlighting can be more precise than custom grammar approximations used in many editors, or even the actual grammar, which may for instance conflated fields and locals to a single type of `Identifier`.

Finally, diagnostics are additional information, warnings, or errors to be reported to the developer. They can be syntax errors preventing parsing, or even linter warnings that identify stylistic issues or code smells.

4 Utilizing the API for SimpleLanguage

We will illustrate how our language-agnostic API can be used to provide IDE features for SimpleLanguage. SimpleLanguage is a language used to document the Truffle framework (see section 2.3). Its parsers is generated by ANTLR [Parr 2013] and its semantics are implemented using Truffle.

To utilize our design with SimpleLanguage, we forego the existing SimpleLanguage implementation and use only its

ANTLR grammar file. Thus, we demonstrate how to use our approach independent of Truffle and that it is general enough to be used with the widely used ANTLR parser generator.

Structural Adaptation. The SimpleLanguage implementation and its grammar remain unchanged. This is possible since ANTLR generates a `ParseTreeListener`, which provides enter and exit events for all grammar rules. This allows us to retrieve all relevant information from the parser. To avoid changes to any of the other generated classes and utilize the grammar actions without change, we provide our own version of the `SLNodeFactory` class. This class is normally used to construct Truffle nodes that represent the AST. In our case, we replace it by a factory that captures the high-level information already produced by the grammar actions.

We subclassed the generated `SimpleLanguageParser` to adapt the parser initialization and use our custom node factory, our parse tree listener, as well as a custom error listener. The error listener merely extracts the necessary position information and adds it as a diagnostic object to the `Structures` data structure.

```
1  class LSFactory extends SLNodeFactory {
2    @Override
3    void startFunction(Token id, Token s) {
4      currentFn = structures.startElement(
5        id.text(), new VarId(id.text()),
6        FUNCTION, id.getRange());
7
8      paramNames = new ArrayList<>();
9      structures.recordToken(id, FUNCTION, NONE);
10   }
11
12   @Override
13   SLStatementNode createBreak(Token b) {
14     structures.recordToken(b, KEYWORD, modifier);
15     return super.createBreak(b);
16 } }
```

Listing 1. Capturing token and structural information in the node factory, which is otherwise used to construct the nodes of the SimpleLanguage AST.

```
class VarId extends ElementId {
  public final String name;
  public final Element scope;

  public int hashCode() {
    return Objects.hash(name, scope); }
  public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass()
        || !name.equals(((VarId) o).name)) {
      return false;
    }
    return scope == ((VarId) o).scope;
} }
```

Listing 2. Sketch of `VarId` to identify variable and function names distinguishing them from object properties in SimpleLanguage.

Capturing the Language's Structures. Our node factory overrides most methods to use the information from the parser to record the elements and token semantics as illustrated in lst. 1. For instance, when parsing a function definition, first we call `startElement(name, id, kind, range)` on line 4 with the lexical details extracted from the parser. This creates a new `Element` to store these details, The actions contained in SimpleLanguage's grammar already manage some details like adding parameters to a function object by using the node factory. Thus, here we only need to override its `addFormalParameter()` method to record an element for each parameter. Since these elements are implicitly contained between the `startElement()` and `endElement()` calls for the function, we do not need to handle containment or scope explicitly.

The other methods overridden in the node factory use `recordToken(range, type, modifiers)` to capture token semantics for the semantic highlighting (see lines 9 and 14).

In SimpleLanguage, both function definition and assigning to a name are parsed as variable definitions. Object fields are created by assigning to an object property. To distinguish between variable definitions and object properties, we define two subclasses of `ElementId`, i.e., two different equality classes: `VarId` and `PropertyId`. Lst. 2 sketches `VarId`, and `PropertyId` is similar, but would omit the scope. Since SimpleLanguage does not have any notion of classes or types, and because object properties can be added arbitrarily, there is not enough structure in the language, which would benefit from including a scope.

The node factory's methods for creating read and assignment nodes use `referenceElement(id, range)` to record the referencing of variables. Note that we rely here solely on the `ElementId` mechanism (see section 3.2), where our `VarId` class encodes the scope of `Elements`, which avoids the need to manage lexical scopes more explicitly.

The factory method for writing to a property will record a new element, since in SimpleLanguage writing to an object property may create it dynamically. The reading of a property will simply record a reference to an element.

The custom parse tree listener is needed since SimpleLanguage's grammar actions do not pass all relevant lexical details to the node factory. For instance, we use it to record the tokens for various keywords such as `function`, `if`, `while`, and `return` for highlighting. Furthermore, when leaving the grammar rule for a function, we add signature details, based on the seen parameter names, and can trigger the completion of the function's element definition by calling `completeElement(elem)`.

This concludes a brief but representative description of what a language implementer roughly needs to implement to gain IDE support based on our approach. Our implementations for Newspeak and SOM are fairly similar, the main difference is that they subclass custom recursive descent parsers instead of an ANTLR grammar.

5 Supporting Existing Parsers for Use in Language Servers

One of the challenges for a language-agnostic language server is the reliance on existing parsers and/or compilers. For responsiveness, IDEs may prefer incremental parsers. They may also prefer parsers that can recover from a wide range of errors to provide feedback while editing. A naive heuristic in a language like Java might for instance ignore all tokens after a parse error until a statement terminating semicolon or a closing brace is found, and then restart parsing from there. For common code edits, such an approach would mean that most of a file can be parsed and most information can be obtained reliably, avoiding undesirable "flicker" effects, for instance when elements constantly appear and disappear in an outline view.

By striving for a low-effort approach to tooling, we cannot expect a language implementation to provide an incremental or even an error-recovering parser. Instead, we have to assume that it aborts after the first error. This means our language-agnostic language server has to compensate for it and still provide a predictable user experience. We use two slightly different approaches for the handling of structural data and the handling of semantic highlighting. Though, in either case, we attempt to reparse a file on every change.

5.1 Caching Structural Information

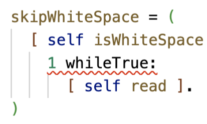
Whenever a file is changed, we try to parse it with the parser subclass augmented to collect structural information. Once we have a successful parse, we will cache the result on the file level, and use the found classes, methods, etc, for future requests, for instance code completion or navigation. Though, we do not use any information obtained from erroneous parse attempts, to minimize UI flicker, i.e., elements appearing and disappearing.

This approach is based on the assumption that most edits will be made on existing files and that files are typically checked into a project's repository without parse errors. Thus, when opening a file, the parser will initially succeed, and we cache a valid representation of the file's structure. The cache will only be updated after another successful parse.

The worst-case scenario for this approach is when a new file is created. Depending on the programmer, this file may take a long time to parse successfully for the first time, and until then, no structural information will be available.

The other major weak point is that structural information can become outdated when major changes are made to a file without the parser succeeding. This means that position information may be outdated, for instance when large chunks are inserted into the file, or deleted from it. In the future, we may use the `didChange` notification of the LSP to correct location information.

In our own use, neither of these two scenarios caused usability issues, and our simple caching approach gives surprisingly usable results. This is because we usually work on existing files that we change step-by-step with small edits.


```
skipWhiteSpace = (
  [ self isWhiteSpace ]
    1 whileTrue:
      [ self read ].
)
```
Figure 4. For the semantic highlighting, we combine the parse result from before the error, with the ones of the last successful parse, which is used for the lines after the error.

5.2 Caching Semantic Highlighting

For the semantic highlighting, the situation is a little different. While the structural information is only observable either in an outline view, where only very coarse details are given, or when interacting with the code, the semantic highlighting is more directly observable as token coloring.

Here we will also only cache results from a successful parse, but if parsing fails, we combine token information from the last successful parse with token information from the one that failed. Specifically, from the failing parse, we take the token information up to the position of the error. Then we add the token information from the last successful parse for the following line and beyond. This means, when there is an error, only the bit after the error, on the same line, will not be highlighted, as can be seen in fig. 4.

The main assumption here is that edits would not have changed the file drastically, and reusing the older highlighting for the rest of the file is likely fine. Thus, the worst-case scenario is once again inserting or deleting large chunks of the file. But, since we use the beginning of the failing parse, starting a new file still gets useful highlighting information.

Despite the approach being very simplistic and not yet adapting highlighting information based on the `didChange` notification, the result feels very usable and does not cause "flickering" highlighting in the editor.

6 Implementation

Our language-agnostic language server is implemented in Java 17 and uses the LSP4J[7] library as an implementation of the LSP. The implementation is available as open source[8] and has been tested with Visual Studio Code as the client.

[7]*A Java implementation of the language server protocol*, Eclipse Foundation, https://github.com/eclipse/lsp4j
[8]*Effortless Language Servers*, Stefan Marr et al., https://github.com/smarr/effortless-language-servers

The server keeps all information for parsed files directly in memory. The data is kept separately per file and the data format is close to the one required for the LSP.

The main classes were already illustrated in fig. 3. The `Structures` class keeps the data for a specific file. It keeps a list of scopes, diagnostics, elements for after-navigation lookups, and maps from element id to element or references.

The nesting of elements is implicitly constructed by the `startElement` and `completeElement` methods. Each element itself can contain a list of child elements, as well as a list of contained references. We use this structure for lookups based on source locations in the form of line and column numbers, by identifying the element most specific to the location. This enables us for instance to identify the current element for hover information, goto definition, and code completion requests. For code completion, we also keep a list of elements that can only appear after navigation, e.g., after the dot to access fields or methods in Java.

The list of diagnostics collected during parsing and possibly any compilation steps, are also kept here.

To enable lookups based on names, we keep one map from an `ElementId` to a set of `Element` objects, and one to map from an `ElementId` to a list of references.

The subclasses of `ElementId` need to implement `equals()` and `hashCode()` methods to define equivalence classes, for instance to distinguish local variables from class names, in languages that distinguish them. The subclasses can also override the `matches(str)` method to adapt the search semantics based on language-specific requirements. For instance, a language with Pascal case or camel case may want to adapt the matching so that identifiers match based on the capitalized letters, when someone searches using them as an abbreviation. Our current standard implementation will only match the case-insensitive beginning of an element's name, which works well for a variety of different languages.

Future work could combine our approach with something like SemanticDB [Burmako 2018] to avoid keeping all data in memory. Keeping everything in memory makes our language server likely impractical for code bases with millions of lines of code. Though, since our goal is to lower the entry barrier for the implementation of language servers, the corresponding young or researchy languages are unlikely to have large enough code bases to require such optimizations.

7 Tradeoffs between Execution and Parsing

To assess the tradeoffs between parse and execution-based language servers, we contrast the requirements for both approaches, and then compare the precision and completeness, as well as the effort needed to realize either.

7.1 Architectural Requirements

For both parse and execution-based language servers, we assume as previously mentioned that the standard parser of the language implementation is used. Thus, we expect a batch parser/compiler, which aborts on the first error. While incremental parsers would be desirable, they are not required, but can be supported for both types of language servers.

For our parse-based approach, we need to be able to adapt the parser. Though, we do not have any other requirements. While we implemented this approach for languages in Java, we do not see any obstacles to apply it to languages or LSP libraries implemented in other languages.

As demonstrated with SimpleLanguage's ANTLR grammar, it is even possible to support ANTLR-based languages without any change to their grammar. Instead, all details can be collected based on ANTLR's `ParseTreeListeners`, or in the case of SimpleLanguage, in tandem with the language-specific node factory.

For SOM we used TruffleSOM, and for Newspeak we used SOMns' implementation. In both cases, we rely on their hand-written recursive descent parsers. We were able to subclass the parser and lexer classes to access the desired information. For convenience, we changed the parsers and lexers slightly, to avoid having to track the grammar context in the subclass explicitly. Thus, we made `private` methods `protected` so that we can override them, and in some cases, introduced methods in the parsers to disambiguate the grammar context, for instance to distinguish local variables from fields.

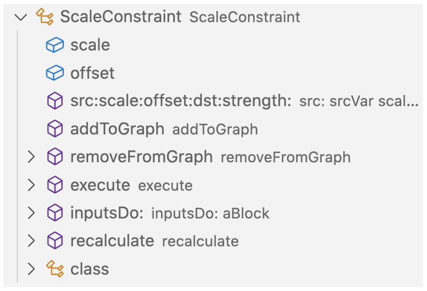
Since we cache information only from a successful parse, as discussed in section 5.1, information for the language server is only available after parsing indeed succeeded.

For the execution-base approach of Stolpe et al., the major requirement is that languages are implemented based on the Truffle language framework. As detailed in section 2.3, a language implementer can augment the AST with tags to identify declarations, function calls, or variable accesses, and needs to implement a few methods to provide details such as the kind of element, documentation, and signatures. By relying on existing Truffle APIs for debugging and language interoperability, the investment into those also gives the benefit of these other features.

However, the main requirement to collect much of the desired data is that the code does not only parse successfully, but can also be executed. This is needed for instance to collect structural information, identify concrete elements referenced, or run-time types, as we detail in the following section. The biggest drawback of this approach is that one first needs well-defined entry points into the code and ideally fully cover the code with unit tests. Furthermore, the runtime may not have all desired information, because the language designers may not have needed, e.g., access to class comments at run time.

7.2 Precision and Completeness of IDE Features

For each of the IDE features, the parse and execution-based approaches have different benefits and drawbacks, which we discuss in the following and summarize in table 1 on page 12.



```
ScaleConstraint ScaleConstraint
  scale
  offset
  src:scale:offset:dst:strength: src: srcVar scal...
  addToGraph addToGraph
  removeFromGraph removeFromGraph
  execute execute
  inputsDo: inputsDo: aBlock
  recalculate recalculate
  class
```

Figure 5. The outline view shows all elements defined in a file and their nesting relation.

Navigational Features.

Symbols. To display a list of elements contained in a file or workspace, or to display a tree-like outline view as in fig. 5, we need to report all defined symbols to the client.

For the parse-based approach, this is straightforward. The language-specific part supplies all details via our API. This includes what kind of element it is, documentation, any available type information, and source locations.

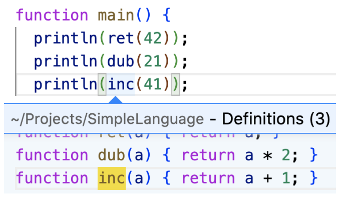
For the execution-based approach, the situation is slightly more complex. Before execution, one only has the list of functions or methods in a file, and can traverse their ASTs to identify local variable definitions using the `DeclarationTag` (see section 3). Only after execution, information on classes or other containing structures can be obtained. This information may then still be only partial depending on the code structure. For instance, in the case of Newspeak, which supports nested classes, executing the top-level file would not provide information for nested classes. A full suite of test cases or other well-defined entry points for execution would be needed to be able to extract all structural information.

Thus, locals and functions can be obtained in both approaches, but for globals, including classes, as well as properties the execution-based approach may require execution.

Goto Definition. In our parse-based approach, we utilize the equality of `ElementId`s (see section 3.2) to find candidate definitions when receiving a request for *goto definition*. For dynamic languages, this means we may have low precision, i.e., we may propose candidates that cannot be referenced, but we can identify candidates in all successfully parsed files.

To distinguish local variables, our approach differs slightly between the languages. For SOM and Newspeak, the parser provides us with an object representing the definition, which allows us to reject variables with the same name in other functions as candidates. For SimpleLanguage, we include the current function as part of the element id for variables, which models the lexical scope accurately (see lst. 2). An example can be seen in fig. 6.

The execution-based approach has initially only the structural information on functions and variable declarations



```
function main() {
  println(ret(42));
  println(dub(21));
  println(inc(41));
~/Projects/SimpleLanguage - Definitions (3)
function dub(a) { return a * 2; }
function inc(a) { return a + 1; }
```

Figure 6. The goto definition action shows all candidates for the `inc` function, directly showing one of the three here.

identified by the `DeclarationTag`. But at run time, it can observe concrete functions/methods being used at call sites and obtain globals and information about properties from the debugging and interop APIs. With this, it can either eliminate candidates with the same name, or rank the ones observed at execution time higher.

Thus, the execution-based approach can use this extra information to improve precision, but requires execution to obtain the globals and properties.

References and Highlights. IDEs may highlight all occurrences of a selected symbol, e.g., a variable or function, and offer a search for all references to a symbol.

With the parse-based approach, we detect all candidates and can distinguish the reading and writing accesses, as supported by the protocol. Eliminating candidates, for instance based on a language's scoping rules is done based on the equality semantics of `ElementId`.

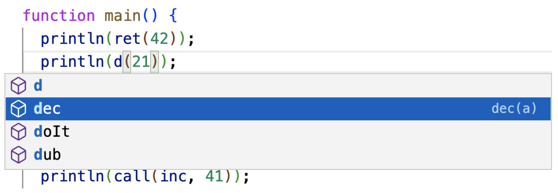
With the execution approach, using the `ReadVariableTag` and `WriteVariableTag`, local variable identity can be determined without execution. For functions, all candidates can be determined by name. Based on run-time information, these can be ranked and possibly even eliminated. References to globals, e.g., classes, require run-time information.

Informational Features.

Syntax Errors. For reporting syntax errors, or any other errors reporting during parsing, both approaches have similar benefits. Since errors are determined already during parsing, execution is not generally required.

Signature Help. Our parse-based approach can harvest the details about function signatures, argument names, and similar during parsing. It can also capture comments, though, here a change to the parser or lexer may be necessary. In ANTLR grammars, comments are often *skipped* at the lexer level, but can also be redirected to a separate token channel. Similarly, hand-written parsers may simply discard comments. In either case, the changes to capture comments are normally fairly localized.

For the execution-based approach, if the language keeps this information at run time, one may be able to query objects. For SimpleLanguage, the parser neither keeps comments nor separates arguments from local variables. Thus, signature



```
function main() {
  println(ret(42));
  println(d(21));
d
dec                                   dec(a)
doIt
dub
  println(call(inc, 41));
```

Figure 7. Code completion for functions yields all possible candidates, including signature information.

help and documentation will be incomplete. On the other hand, execution can be used to augment signatures for instance with run-time type information.

Hover Information. For hover information, the situation is similar to signature help. For the parse-based approach, we can capture for instance class comments and display them. The execution-based approach will only get access to such structural information via execution, and only if it is preserved. On the other hand, it is possible to show concrete run-time types, or even values, and extract concrete information from those.

Code Completion. Supporting code completion brings its own set of challenges. Stolpe et al. already identified the most important issue: in many cases code completion is requested at a point where the code may not currently parse correctly, which is a challenge both approaches need to overcome.

Completing Globals and Locals. In the simplest case for code completion, we may try to complete some kind of basic expression. Depending on the language, many of these requests will parse successfully. In our parse-based approach, we either propose all symbols, i.e., elements at this point in the scope, or when the cursor is at the end of an identifier, use this identifier to select candidates with partial name matching. Generally, we walk the nested `Element` objects, and propose all elements found in the lexical order, inside out, which seems to provide good results.

With the execution-based approach, results from the last successful parse and/or run are used. Information for locals can be used from the last parse and will be complete. Though, globals require run-time information. For improved results, the run-time information can be used to eliminate and rank the completion candidates.

Completing Functions. For functions and methods, both approaches rely on the last successful parse of the file, which yields all lexically defined functions and methods as illustrated in fig. 7. Though, run-time information can be used to eliminate or rank completion candidates.

Completing Properties. A complication for offering code completion of property names is that in many languages the parser will reject the program at the point when the character to access a property, e.g., the dot as in `obj.member`, is typed into the editor. The program will only parse when at least the first letter of the property name is entered, too.

For our parse-based approach, this did not present a major challenge. We keep the structural information from the last successful parse, which in most cases will be after `obj` was typed completely. When the parsing fails, we see that the last inserted character was a dot and can use this to query for completions that are *after navigation* as we call it. We include this concept directly in the API to be able to more easily identify completion candidates. In most languages, members in the form of fields and methods will follow lookup rules separate from lexical scoping. The after-navigation flag is part of the definition of `Elements`, and considered when searching for candidates. When there are already characters of a property name, we will use it to narrow down the list of candidates. Otherwise, we propose all candidates of the current file first, and then all the other ones.

As mentioned earlier, we do not have any type information or other structural details that could be used to further narrow down the candidates, which is a classic challenge for tooling for dynamic languages.

For the execution-based approach, an earlier successful result will be used. Here the concrete type, e.g., of the object before the dot can be queried for its members, which are then proposed as completion candidates. Assuming that the type observed in the test run is representative, this gives highly accurate completion proposals.

Convenience Features. Features such as linting and semantic highlighting are popular in IDEs, and add to their convenience.

Linting. Linters often try to identify superficial or easy-to-spot issues in code. They can comment on code formatting, style, and sometimes even bugs.

In our parse-based approach, we added a few linters that address basic issues. The simplest is a linter that detects whether a file ends with a newline, which is something of a convention in some environments. Perhaps more useful, we also have a linter that checks all references created during parsing for a corresponding candidate definition. If no definition is found, depending on the language, it may indicate a bug. These linters are opt-in per language and either work on the whole workspace, or just on a single file.

For the execution-based approach, it is conceivable to implement some basic linting, too. For instance, since all method and function calls are tagged with `CallTag`, one can identify whether there are any calls for which no candidates exist. The errors of unit tests executed in the background may also be used to add lint-like hints.

Semantic Highlighting. Syntax highlighting is popular in many editors and IDEs. Usually, this requires replicating a language's parser in some framework to express the highlighting information. These days TextMate grammars[9] and Tree-sitter[10] seem popular. Though, in the author's experience, for researchy languages, these grammars are often minimal, incomplete approximations of the actual language.

With our parse-based approach, we capture the semantics of all relevant tokens using the semantic analysis of the existing parser/compiler. Thus the highlighting can distinguish, e.g., fields and methods, or locals and arguments. In our experience, capturing this information requires typically a single line of code per token type.

Marr et al. [2017] demonstrated the use of AST tagging to a similar effect. Unfortunately, this requires many different AST tags, and will only highlight the body of functions and methods. Thus, it would be possible also in a execution-based approach, but likely feel rather incomplete.

Code Lenses for Unit Tests. The LSP supports the notion of *code lenses*, which enables us to annotate specific code ranges for instance with additional information and actions.

In our parse-based approach, languages can opt into such lenses and configure them. For instance, we provide a code lense to detect methods that start with `test` as common for the xUnit frameworks.

While not mentioned in Stolpe et al. [2019], providing code lenses for such scenarios should be trivial, since it already relies on unit tests to harvest information for the IDE.

7.3 Implementation Effort

To compare implementation efforts, we have only a limited amount of data available. Though, for the execution-based approach, in the ideal case, one needs to merely add tags to AST nodes and implement a few small methods. Adding tags is as simple as implementing a `hasTag(tag)` method, returning `true` for the four tags to indicate declarations, calls, reads, and writes on the relevant nodes. One may argue that the APIs for debugging and language interop are already implemented for the other desirable functionality. Thus, the effort is minimal.

To assess the implementation size, we use `cloc` version 1.92, and report the number of lines of code (LOC).

For SimpleLanguage, we estimate that the implementation of the interop and debugging APIs is about 454 LOC. This is an estimate, since there is in some cases not a clear separation, and we simply counted all lines of code of the classes `SLObject`, `SLFunction`, `SLLanguageView`, and `SLType`, which are dominated by the debugging and interop APIs.

In comparison, the language-specific code for our parse-based approach is 491 LOC for SimpleLanguage, 920 LOC for SOM, and 711 LOC for Newspeak.

Much of it is however boilerplate. For SimpleLanguage, considering only the parser subclass, our custom node factory, and the token listener, all information is collected with merely 320 LOC. In the SOM implementation, the main code is in the parser and lexer subclasses, which add up to 472 LOC. For Newspeak, we only needed to subclass the parser, which amounts to 473 LOC. The other code is boiler plate, `equal()` and `hashCode()` methods for element ids, as well as conversion functions from one representation of a lexical position to another.

To provide a rough comparison, we sampled 20 language server implementations from the LSP wiki,[11] for languages we had heard of before. The goal here is to gain an intuition of how our numbers compare to ad hoc language server implementations. Note, these numbers are not well-defined baseline for comparison, since these servers can spent arbitrary amounts of effort on better analysis and more precise results and support different sets of LSP features, which we did not access here. Comparing against the execution-based approach provides are more well-defined baseline.

Taking the latest version of the git repositories, removing tests and configuration files, and measuring the LOC, we find the smallest language server to be 1,577 LOC and the largest one 95,878 LOC, Microsoft's deprecated open source Python language server.[12] The mean is 18,985 LOC and the median 12,055 LOC. Some repositories contain however also the language server protocol implementation itself, making the numbers appear somewhat larger.

Overall, we observe that with our approach, a language server is about an order of magnitude less code, while providing the most widely used IDE features.

8 Discussion

The focus so far was on the specific IDE features, though there are a number of other concerns worth discussing, including language built-ins, polyglot code bases, various limitations, as well as how practical batch parsers are from the perspective of performance.

Recognizing Built-Ins. The execution-based approach is implicitly able to capture all functions that are built into a language implementation, since they are announced via Truffle's standard instrumentation mechanisms.

However, with our approach, we require the information to be given via our proposed API. This means built-ins are often not directly represented. In the case of Newspeak and SOM, we already collected the built-ins for other tooling, and they were readily available. Though, for SimpleLanguage we only used the ANTLR grammar, which means, this information is not directly available, but can be provided manually.

[9] *TextMate: Language Grammars*, MacroMates Ltd., https://macromates.com/manual/en/language_grammars

[10] *Tree-sitter*, GitHub Inc., https://tree-sitter.github.io/

[11] *Implementations: Language Servers*, Microsoft, https://microsoft.github.io/language-server-protocol/implementors/servers/

[12] Microsoft's current PyLance Python Language Server is closed source, and not available for comparison.

Table 1. Tradeoffs between Parse and Execution-based Language Servers

LSP Feature	Parse Time	Run Time	
Symbols: locals, functions	✓	✓	
Symbols: globals, properties	✓	★	
Goto definition: locals	✓	✓	
Goto definition: functions	✓	✓	+ call targets
Goto definition: globals, properties	✓	★	+ types and properties
References and Highlights: locals	✓	✓	
References and Highlights: functions	✓	✓	+ call targets
References and Highlights: globals, properties	✓	★	+ types and properties
Syntax Errors	✓	✓	
Signature Help	✓	★	+ documentation (if preserved), types
Hover Information	✓	★	+ documentation (if preserved), types
Completion: globals	✓	★	
Completion: locals	✓	✓	
Completion: functions	✓	✓	+ call targets
Completion: properties	✓	★	+ types
Linting	✓	★	+ unit test errors
Semantic Highlighting	✓	○	+ run-time tags [Marr et al. 2017]
Code Lens for Unit Tests	✓	★	

✓ Available directly
○ Limited static information
★ Reaches similar level after collecting run-time information
+ Enhanced details as more information can be observed at run time

Support for Polyglot Code Bases. One of the notable features of the Truffle and GraalVM ecosystem is its support for polyglot programming. As already noted by Stolpe et al. [2019], the language server has immediate support for it, simply be virtue of using the language interop API [Grimmer et al. 2018] to obtain the needed information.

In our parse-based approach, this is not considered directly. Even with the three languages supported, it does not work out of the box to get for instance code completion proposals from the other languages.

To support it, we could take inspiration from Truffle's interop approach and add abstract methods on `ElementId` such as `isExecutable()` and `isMemberReadable()` for which languages can then indicate whether an element is suitable, for instance as candidate for code completion given the current lexical context.

Supporting Non-Local Syntax Features. When implementing our approach for Newspeak, we stumbled over Newspeak's primary factory method syntax, which is essentially a constructor. Lst. 3 sketches the `Error` class, where the constructor takes `msg` as an argument. Inside of what one could consider the lexical body, we define in line 2 the public field `message`.

For our API (see fig. 3), this meant we needed to add an `outerScope` parameter, in which we pass an integer to indicate the number of levels to go up the nesting chain when recording an element. Though, Newspeak is not the only language that benefits from this approach. Python and Ruby create fields during the constructor execution, as part of the initial assignment. With this feature, these *definitions* can be attributed to the surrounding class.

```
1  class Error signal: msg = (
2  | public message = msg. |
3    self signal.
4  )()
```

Listing 3. Newspeak class syntax defines fields inside the constructor.

For other discontinuous language features, for instance extensions to the same class in different Ruby files, one can rely on the equivalence of `ElementId` objects and have the different extension parts be equivalent, which ensures that e.g., references and code completion work as expected.

Limitations and Practical Issues. From the user perspective, one observable limitation will be that code completion will offer to complete variables that have not been defined yet. In some scenarios, this might even be useful, e.g., when reordering code. Only on completion, the linter or parser will produce a warning when the code is invalid.

Another aspect that can cause practical limitations is the parser design. For instance in the case of SOM, it is difficult to get the correct details for code completion of a superclass's name. The problem is that the parser tries to load the superclass, and then triggers Smalltalk code to handle the load error, which does not preserve the error location. This makes it difficult to support completion at this particular point. We can imagine that other languages have other oddities, where changes to the parser might be desirable.

For some languages, parsers may even minimize the work performed for an initial parse. V8 does for instance delay the full parsing of a function body to its first activation. For tooling such as ours, this is of course undesirable. In these cases, one must hope that the parser has a flag like V8's that allows one to disable this optimization.

Performance of Batch Parsers. Since our approach relies on parsers that already exist, and are not designed for use in IDEs, we wondered what their performance would be. There are of course conceptual limitations, such as not having error recovery, and thus, only partial information for files with errors. Though, this aside, we wondered whether the parsers would be fast enough for usual usage scenarios.

To investigate the question, we build code generators for Newspeak, SimpleLanguage, and SOM, which, based on a library of methods, would generate files of arbitrary length.

Then we built benchmarks that measure the parsing speed, as well as the overhead of answering the requests for all symbols and the semantic tokens, which an IDE would query relatively often after file changes to update the outline view and semantic highlighting information.

Running these benchmarks on a decade-old Intel Xeon E5-2620 with 16GB RAM, Ubuntu 20.04 and Java 17, makes us believe that there is no practical issue preventing our approach from succeeding. The results are depicted in fig. 8. They indicate that batch parsers are fast enough, even for large files. The SimpleLanguage parser, being based on ANTLR is the slowest, but only takes 215ms for 10,000 LOC. The Newspeak and SOM parsers stay well below it with 86ms and 78ms respectively.

Johnson [2014] suggests that humans expect feedback in under one second, but ideally expect a visible effect in about 100ms. For the combination of slower parsers and large files, one thus may need to utilize the LSP's support for partial results and intermediate progress updates.

Current Status of the GraalVM Language Server. In this paper, we focused our comparison on the system described by Stolpe et al. [2019]. Though, as mentioned in the beginning, it was the foundation for the GraalVM language server. One important change in the product is that the `DeclarationTag` was removed. For static information, the system now delegates to existing language servers.

For widely used languages, this is a strategy that allows the users to benefit from the combination of a fully-featured custom-built language server, and the enhanced precision and polyglot capabilities based on the run-time information. Though, for niche and researchy languages this means that our parse-based approach is needed to fill in the gap.

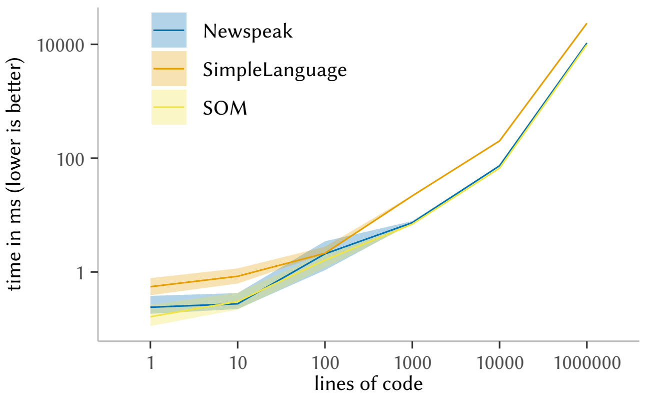


Figure 8. Performance results of SimpleLanguage's ANTLR-based parser, as well as the custom recursive descent parsers for Newspeak and SOM for files with different numbers of lines of code. They take at most 215ms for processing a file with 10,000 LOC. The area around the lines indicates the 5th and 95th percentile of 1,000 measurements.

9 Conclusion and Future Work

We propose a novel API that enables implementers of dynamic languages to provide information to a language server that covers many of the common features desired in an IDE. The API design makes it independent of any implementation framework and enables the use of preexisting parsers.

We compare our approach to Stolpe et al. [2019]'s execution-based approach. In the best case, they require only a few lines of code to tag AST nodes, and a small number of methods to extract structural details. However, the drawback is that files need to be executable and have either standard entry points or be executed by unit tests to provide much of the desirable information to the language server.

Our approach captures all information at parse time, and sacrifices only some precision when selecting candidates, e.g. for highlighting references or code completion.

From our three implemented languages, SOM required with 920 lines the most code overall. Much of it is Java boilerplate and largely straightforward code turning data from one form into another. Furthermore, it is an order of magnitude less code than the median 12,055 LOC for the 20 language servers we sampled.

Since we use existing parsers, which are not designed for use in IDEs, we also confirmed that their performance for reasonably sized files is practical for continuous use, perhaps to reparse after a keystroke.

Combined with the execution-based approach, our approach has the potential to give highly precise and complete information for an IDE with significantly less code than normal language servers. Thus, we believe our approach is not just practical, but even a desirable design for building tooling for existing niche and research languages.

Furthermore, by itself, our design enables low-effort language servers and would be an ideal complement to libraries implementing the language server protocol to give developers the option to support many IDE features rapidly.

Future Work. In this work, we focused on dynamic languages. In a sense, they are more forgiving and user expectations around the precision of code completion may also be different than perhaps for statically-typed languages. Thus, one of the next steps would be to explore how to benefit from type information in a language-agnostic way to expand the proposed API to support such languages better.

With our support for basic language-agnostic linting, we demonstrated that a useful correctness check can be done with minimal effort. Since static analysis frameworks such as Infer first map to an abstract intermediate form, one could investigate what a language-agnostic language server would need to capture to enable more advanced analyses.

Acknowledgments

The authors would like to thank Owen Oclee for the initial implementation of the SOM language server, paving the way for this language-agnostic language server. We also thank the DLS'22 reviewers for their detailed and constructive suggestions.

This work was supported by the Engineering and Physical Sciences Research Council (EP/V007165/1) and a Royal Society Industry Fellowship (INF\R1\211001).

References

Eugene Burmako. 2018. SemanticDB: A Common Data Model for Scala Developer Tools. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection* (Boston, MA, USA) *(META'18)*. ACM, 2. https://doi.org/10.1145/3281074.3281076

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007 Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement your Language Faster than the Quick and Dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (SPLASH '10)*. ACM, 307–309. https://doi.org/10.1145/1869542.1869625

Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. 2018. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Transactions on Programming Languages and Systems* 40, 2 (June 2018), 1–43. https://doi.org/10.1145/3201898

Michael Haupt, Michael Perscheid, and Robert Hirschfeld. 2011. Type Harvesting: A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages. In *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11*. ACM, 1282–1289. https://doi.org/10.1145/1982185.1982464

Jeff Johnson. 2014. *Designing with the Mind in Mind, Second Edition: Simple Guide to Understanding User Interface Design Guidelines* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. Association for Computing Machinery (ACM), 444–463. https://doi.org/10.1145/1932682.1869497

Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. 2016. The IDE Portability Problem and Its Solution in Monto. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (Amsterdam, Netherlands) *(SLE '16)*. ACM, 152–162. https://doi.org/10.1145/2997364.2997368

Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*. IEEE, 168–177. https://doi.org/10.1109/SCAM.2009.28

Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2017. Kómpos: A Platform for Debugging Complex Concurrent Applications. , 2 pages. https://doi.org/10.1145/3079368.3079378

Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. 2020. Bacatá: Notebooks for DSLs, Almost for Free. *The Art, Science, and Engineering of Programming* 4, 3 (Feb. 2020), 11:1–11:38. https://doi.org/10.22152/programming-journal.org/2020/4/11

Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf. 328 pages.

Daniel Stolpe, Tim Felgentreff, Christian Humer, Fabio Niephaus, and Robert Hirschfeld. 2019. Language-Independent Development Environment Support for Dynamic Runtimes. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'19)*. ACM, 80–90. https://doi.org/10.1145/3359619.3359746

Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *Programming Journal* 2, 3 (March 2018), 30. https://doi.org/10.22152/programming-journal.org/2018/2/14

Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium* (Tucson, Arizona, USA) *(DLS'12)*. ACM, 73–82. https://doi.org/10.1145/2384577.2384587