IDE for Regular Games

(IDE dla Regular Games)

Jakub Cieśluk

Praca inżynierska

Promotor: mgr inż. Radosław Miernik

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

19 stycznia 2024

Abstract

High-quality tooling is crucial for programming language adoption. Developers are used to conveniences such as auto-completion or syntax highlighting. In this work, we improve the tooling for the Regular Games language, a game description language for General Game Playing, developed at the University of Wrocław. Specifically, we present an implementation of an Integrated Development Environment that runs entirely in a browser. We also address the significance of the Language Server Protocol (LSP) in standardizing the development of language intelligence.

We extend the previous Regular Games parser with error recovery capability and describe a way to collect semantic data from the Abstract Syntax Tree. This work implements both the *language server*, which is responsible for providing LSP features, and the *language client*, which serves as an intermediary for communication with a code editor.

Solidne narzędzia wspierające programistów są bardzo ważne dla adopcji języka programowania. Programiści są przyzwyczajeni do udogodnień, takich jak automatyczne podpowiedzi lub kolorowanie składni. W tej pracy zajmujemy się ulepszeniem narzędzi dostępnych dla języka Regular Games. Język ten powstał na Uniwersytecie Wrocławskim i służy do opisu reguł gier w kontekście General Game Playing. W szczególności przedstawiamy implementację zintegrowanego środowiska programistycznego, które działa całkowicie w przeglądarce. Zajmujemy się również znaczeniem Language Server Protocol (LSP) w standaryzacji rozwoju inteligencji językowej.

Rozszerzamy poprzedni parser Regular Games o możliwość tolerowania błędów i opisujemy sposób zbierania danych semantycznych z abstrakcyjnego drzewa składni. Niniejsza praca implementuje zarówno *serwer języka*, który jest odpowiedzialny za dostarczanie funkcji LSP, jak i *klienta języka*, który służy jako pośrednik do komunikacji z edytorem kodu.

Contents

Chapter 1

Introduction

Integrated Development Environments (IDEs) are crucial tools in software development, significantly enhancing programmers' efficiency. An IDE typically combines various components and functionalities that programmers need, creating a unified environment for writing and debugging code.

The main part of an IDE is a code editor, where developers write their software. An editor should support features like syntax highlighting, auto-completion, and diagnostics (*error squiggles*).

IDEs often cooperate with a compiler or an interpreter to allow their users to run or test their code within the development environment[1]. They also provide tools to navigate the codebase, e.g., jumping to a function definition and its references, or a workspace-wide symbol search. Modern IDEs, like Visual Studio Code[1], come with an integrated terminal and version control tools.

The choice of an IDE depends on its support for a chosen programming language and the developer's personal preferences. Options vary from robust IDEs equipped with extensive toolkits to basic text editors enhanced with essential plugins.

1.1 Introducing Language Server Protocol

Having good IDE support is very important for solid language adoption. Poor tooling can outweigh the many advantages of the language and determine its popularity. For these reasons working on providing a good programmer experience is very important for teams involved in developing programming languages.

Previously, this work could be very tedious. Every editor had a different set of capabilities and the entire analysis had to be done almost from scratch for each editor since there was no common way to integrate with them.

[1]`https://code.visualstudio.com/`

Language Server Protocol[2] (LSP), designed by Microsoft, was created to overcome these issues and streamline the development of tooling[2, 3, 4]. It provides a unified way for communicating between an editor and a *language server*, which is a provider of all IDE features like syntax highlighting, navigation, and completions. These days almost every mainstream code editor supports LSP, including Atom, Eclipse, Emacs, Neovim, Sublime Text, and Visual Studio Code.

1.2 Language server

At first, a compiler, or an interpreter, may seem to be a good candidate for a language server base. However, they serve a different function in the ecosystem, and their design principles are not suited for other roles. While both a language server and a compiler perform analysis of code, they are optimized for different use cases.

Differences begin already at the parsing stage[5]. Compilers typically follow a single-pass or multi-pass approach, where they parse the whole file at once, optionally tokenizing its contents first. Language servers prefer incremental parsing since they must react to every change made in the code in real-time.

While the user writes their program, it is almost always incorrect, but the server should still be able to process it and provide as many helpful diagnostics and decorations as possible. For this reason, its parser should also be fault-tolerant and be able to handle erroneous code[6].

1.3 Regular Games Language

Regular Games (RG) language was created at the University of Wrocław by Marek Szykuła, Radosław Miernik, and Jakub Kowalski. It is a successor of another language, Regular Board Games[7], designed at the same university by Jakub Kowalski, Marek Szykuła, and their team.

Both languages were designed to describe games in General Game Playing[8] (GGP), a field of artificial intelligence, where programs can successfully play different kinds of games. Traditionally, agents are written directly for a single game and have no idea how to play any other. They depend on specialized algorithms and heuristics which had to be previously researched. On the other hand, in GGP, programs have to be able to play any sort of game, without knowing its rules beforehand. For that, we need a formal way of defining the game rules, called game description language.

Two remarkable GGP languages are GDL[9] and Ludii[10]. GDL encodes the game rules as a set of logical clauses. It is general enough to describe any deterministic n-person game with simultaneous moves and perfect information.

[2]`https://microsoft.github.io/language-server-protocol/`

Ludii builds upon the principles of GDL but goes beyond being just a language. It provides a whole platform for playing and analyzing games. However, neither of these two has an IDE.

RG language is characterized by its readability to the programmer and is easy to parse and analyze. It has a clear but expressive syntax that naturally resembles the game rules set. Like most programming languages, it contains types, constants, and mutable variables. The game is represented as a finite automaton in the form of a graph with parametrized edges.

Types (Figure 1.1) in RG can take two forms. A type can be either a set of identifiers or an arrow type. Sets are used to represent all possible values of a given type like allowed positions on a game board or different kinds of pieces. Arrow types work similarly to function types in other languages – each of them has a parameter type and a result type. There are no tuple types in the Regular Games language, but arrow types can represent multi-parameter functions.

```
type Player = {X,O};
type Score = {0,50,100};
type Coord = {0,1,2};
type Piece = {e,X,O};
type ColumnOfBoard = Coord -> Piece;
type Board = Coord -> ColumnOfBoard;
type PlayerToPlayer = Player -> Player;
```

Figure 1.1: Type definitions

Constants (Figure 1.2) are immutable constructs that are mostly used for defining moves available at a given position on a game board. A constant consists of an identifier, a type annotation, and a value. A value can be a single element or a list of `value entries`. Each value entry serves as a mapping from an identifier to a value. This identifier is optional, which enables defining default mappings. Each map has to have exactly one default value. Values in the form of a list of value entries can be understood as functions, possibly of a higher order.

```
const opponent: PlayerToPlayer = {X:O, :X};
const otherInLine1: Coord -> Coord = {0:1, :0};
const otherInLine2: Coord -> Coord = {:2, 2:1};
const onLRDiagonal: Coord -> Coord -> Bool = {0:{0:1,:0}, 1:{1:1,:0}, 2:{2:1,:0}, :{:0}};
const onRLDiagonal: Coord -> Coord -> Bool = {2:{0:1,:0}, 1:{1:1,:0}, 0:{2:1,:0}, :{:0}};
```

Figure 1.2: Constants definitions

Variables (Figure 1.3) look similar to constants, they also have an identifier, a type annotation, and an initial value. However, variables are mutable, so they are used to store the current state of the game. Usually, they contain data about the current player or the positions of pieces on the game board.

```
var goals: Goals = {:50};
var playerTurn: Player = X;
var posX: Coord = 0;
var posY: Coord = 0;
var board: Board = {:{:e}};
```

Figure 1.3: Variable definitions

Edges (Figure 1.4) make up a majority of all games written in the Regular Games Language. Each edge consists of two vertices and a label. A vertex, in addition to its name, can also have associated values that it passes on to its neighbors.

Labels define the transitions between given vertices. It can be a comparison that dictates whether the transition is allowed, an assignment that changes the variable's value, a tag used to distinguish moves, and a reachability check between two different vertices in a graph.

```
move,chooseX: player = PlayerOrKeeper(playerTurn);
chooseX,chooseX(coordX:Coord): $ coordX;
chooseX(coordX:Coord),chooseY: posX = Coord(coordX);
chooseY,chooseY(coordY:Coord): $ coordY;
chooseY(coordY:Coord),check: posY = Coord(coordY);
check,set: board[posX][posY] == Piece(e);
set,endmove: board[posX][posY] = Piece(playerTurn);

endmove,checkwin: player = PlayerOrKeeper(keeper);
checkwin,win: ? checkline -> endcheckline;
checkwin,nextturn: ! checkline -> endcheckline;
nextturn,turn: playerTurn = opponent[playerTurn];
```

Figure 1.4: Edges

The Regular Games language also defines pragmas. They serve as hints for the compiler. Although they are analyzed and included in the abstract syntax tree (AST), they have no influence on the semantics of a game.

```
@unique begin;
@unique chooseX;
@unique chooseX(coordX:Coord);
@unique chooseY(coordY:Coord);
@unique checkwin;
```

Figure 1.5: Pragmas

Chapter 2

Related work

Almost every commercially used programming language has its implementation of a language server, and the vast majority of them are open source. Examples include `elixir-ls`[1] for Elixir and `rust-analyzer`[2] for Rust.

What separates them from the language server created in this work is their close connection with the compiler and build tool. Knowledge about files containing tests allows users to run them directly from the editor. Using these language servers, users can navigate to files that are not part of their projects, such as dependency sources. Since the RG language is based on single-file sources, these features have no counterpart in the presented IDE.

2.1 Code editors

There are many code editors available for developers, but only a few of them can run in a browser[11]. Previously the IDE for Regular Games was using CodeMirror[3], which is also embedded in Scastie[4], an online editor for Scala. We replaced it with Monaco as the latter provides better LSP support and documentation. Monaco is also used on the CodinGame[5] website, an educational platform for developing game-playing bots.

`lsp-web-demo`[6] is a great minimalistic demonstration of how to build a language server in Rust. Its repository contains also an example of building a server with WebAssembly[12] and connecting it to a client. The project implements a minimalistic language server for JavaScript. Instead of writing everything from scratch, it uses Tree-sitter for parsing and extracting semantic information.

[1]`https://github.com/elixir-lsp/elixir-ls`
[2]`https://rust-analyzer.github.io/`
[3]`https://codemirror.net/`
[4]`https://scastie.scala-lang.org/`
[5]`https://www.codingame.com/home`
[6]`https://github.com/silvanshade/tower-lsp-web-demo`

2.2 Tree-sitter

Tree-sitter[7], a groundbreaking parsing library, has emerged as a powerful tool in the field of language processing and code analysis. GitHub developed it for the Atom editor but is now also commonly used in Emacs, Neovim, and Helix.

Tree-sitter was designed directly for the needs of IDEs. One of its key features is incremental parsing, which allows it to not rebuild the whole abstract syntax tree on every keystroke, but only update affected nodes[13, 14, 15]. It is also able to provide useful error messages and produce a correct AST in their presence. Tree-sitter is language-agnostic, which makes it a reasonable choice for tools that aim to support multiple languages.

Editors like Neovim use trees generated by Tree-sitter not only for syntax highlighting but also for navigation and refactorings using `queries`. Parsers are written for plenty of languages and library bindings are available in JavaScript, Python, and Rust, among others. Tree-sitter also provides bindings for WebAssembly, which makes it a perfect tool for a code editor that runs entirely in a browser.

2.3 Advanced language server implementantations

The Metals project[8], implementation of Language Server Protocol for Scala, is arguably one of the more complex language servers. It supports multiple versions for both Scala 2 and Scala 3, together with multiple build tools including `sbt`, `mill`, `Maven`, and `scala-cli`. To cooperate fully with them, Metals uses Build Server Protocol (BSP). This channel of communication is used to get data about whole modules and their compilation state. BSP informs the language server about project dependency sources, test classes, and errors encountered during compilation.

For navigation and renaming, Metals uses SemanticDB[16], a data model storing semantic information about symbols defined and referenced in each file. SemanticDB files are generated by the compiler and saved on the disk to be later consumed by tools like language servers. Although they are very useful, finding all usages of a symbol using them is not fast enough. Similarly, a language server doesn't need all the information about dependency sources. Because of this, Metals implements its set of indexers, that collect only definition locations of top-level symbols.

Features that operate in the scope of a single file, like auto-completion or syntax highlighting use Presentation Compiler, which is a faster, asynchronous version of the Scala Compiler, but is not able to perform any phases of the latter that come after typechecking. It provides Metals with completions, symbols available at a given position, and typed abstract syntax trees.

[7]`https://tree-sitter.github.io/tree-sitter/`
[8]`https://scalameta.org/metals/`

Chapter 3

Implementation

3.1 Methodology

The existing RG interpreter is written in Rust, which is also the chosen language for developing the language server. The parser utilizes `nom`[1], a parser combinator library. We enhance it with the `nom-locate`[2] *crate* (*crates* in Rust are used for sharing code between projects), which equips us with an input type preserving the locations of tokens within the source file.

The language server is built on top of the `tower-lsp`[3] library. Then, we build both interpreter and language server WASM modules with `wasm-pack`[4] and `wasm-bindgen`[5].

The website is available at `https://radekmie.dev/rg/`. It is written using TypeScript and React. As a code editor, we use the aforementioned Monaco[6]. Thanks to `monaco-vscode-api`[7] we can use Visual Studio Code API and register LSP features easily. For running the project we use `parcel`[8] build tool. The project is published automatically with GitHub Actions[9].

[1]`https://github.com/rust-bakery/nom`
[2]`https://github.com/fflorent/nom_locate`
[3]`https://github.com/ebkalderon/tower-lsp`
[4]`https://github.com/rustwasm/wasm-pack`
[5]`https://github.com/rustwasm/wasm-bindgen`
[6]`https://microsoft.github.io/monaco-editor/`
[7]`https://github.com/CodinGame/monaco-vscode-api`
[8]`https://parceljs.org/`
[9]`https://github.com/features/actions`

3.2 Architecture

This project implements both ends of the Language Server Protocol i.e., the client and the server. When a user enters the website, an editor is loaded and all services are initialized. The client and the server are started and communication between them begins.

We take advantage of using multiple web workers for better responsiveness and overall user experience. Writing happens in the main thread, so it is not interrupted by calculations from the interpreter and the language server. Both of them also run in separate workers, this way LSP features are not blocked by expensive calculations connected with generating optimized code. This is a common approach, taken by most IDEs.

As a transport layer, LSP uses JSON-RPC (Remote Procedure Call). The client begins a connection with the server by sending `initialize` request. The request's parameters contain information about the client, including its capabilities. They define which LSP features are supported on the editor's end.

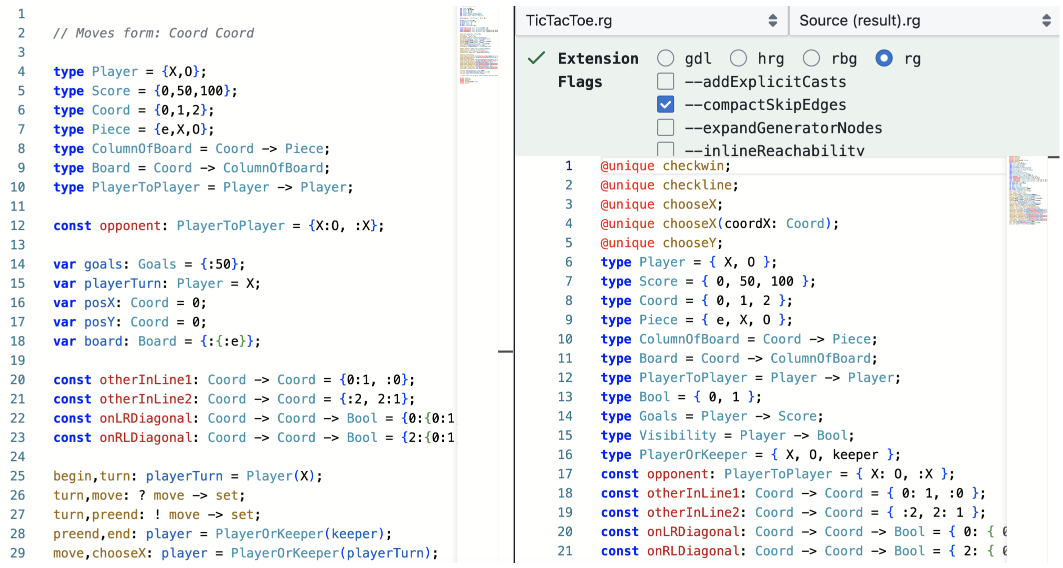
The server then responds with its own set of capabilities, which lists supported programming languages, describes implemented LSP endpoints, and informs how should synchronization proceed. In the case of the Regular Games language server, after opening a new file or changing its contents, a notification with the whole file's contents is sent.

The language server then parses the code and collects semantic data from the AST created in the process. Any errors detected during this phase are published to the client as notification `publishDiagnostics`. They can inform the user about syntax errors or unknown identifiers.

Most LSP methods require some interaction from the user. After they are invoked in the editor, the client sends the appropriate request and the server processes it and responds with a result. It is then translated to data structures understood by Visual Studio Code API. The whole communication process happens asynchronously.

3.3 Code editor

The website view is split into two main parts. The left side is a standard editor in which the user writes its code (Figure 3.1). It supports four languages: Regular Games, High-level Regular Games, Regular Board Games, and Game Description Language. The language server supports only the first one, but we provide basic features for all of them.



```

// Moves form: Coord Coord

type Player = {X,O};
type Score = {0,50,100};
type Coord = {0,1,2};
type Piece = {e,X,O};
type ColumnOfBoard = Coord -> Piece;
type Board = Coord -> ColumnOfBoard;
type PlayerToPlayer = Player -> Player;

const opponent: PlayerToPlayer = {X:O, :X};

var goals: Goals = {:50};
var playerTurn: Player = X;
var posX: Coord = 0;
var posY: Coord = 0;
var board: Board = {:{:e}};

const otherInLine1: Coord -> Coord = {0:1, :0};
const otherInLine2: Coord -> Coord = {:2, 2:1};
const onLRDiagonal: Coord -> Coord -> Bool = {0:{0:1
const onRLDiagonal: Coord -> Coord -> Bool = {2:{0:1

begin,turn: playerTurn = Player(X);
turn,move: ? move -> set;
turn,preend: ! move -> set;
preend,end: player = PlayerOrKeeper(keeper);
move,chooseX: player = PlayerOrKeeper(playerTurn);
```

TicTacToe.rg | Source (result).rg

✓ Extension ( ) gdl ( ) hrg ( ) rbg [✓ rg]
Flags [ ] --addExplicitCasts
[✓] --compactSkipEdges
[ ] --expandGeneratorNodes
[ ] --inlineReachability

```
@unique checkwin;
@unique checkline;
@unique chooseX;
@unique chooseX(coordX: Coord);
@unique chooseY;
type Player = { X, O };
type Score = { 0, 50, 100 };
type Coord = { 0, 1, 2 };
type Piece = { e, X, O };
type ColumnOfBoard = Coord -> Piece;
type Board = Coord -> ColumnOfBoard;
type PlayerToPlayer = Player -> Player;
type Bool = { 0, 1 };
type Goals = Player -> Score;
type Visibility = Player -> Bool;
type PlayerOrKeeper = { X, O, keeper };
const opponent: PlayerToPlayer = { X: O, :X };
const otherInLine1: Coord -> Coord = { 0: 1, :0 };
const otherInLine2: Coord -> Coord = { :2, 2: 1 };
const onLRDiagonal: Coord -> Coord -> Bool = { 0: { 0
const onRLDiagonal: Coord -> Coord -> Bool = { 2: { 0
```

Figure 3.1: Website view

For syntax highlighting we use Monarch[10], which allows us to define syntax coloring in declarative style. It is not as precise as semantic highlighting but still improves the readability of the code, without requiring any interaction with the language server.

Initially, the editor field contains a code from one of the prepared samples, which can be switched at any point. Changing the code sample is adequate for closing a file and opening a new one.

The right part of the website presents the view of the results of transforming the code on the left by the interpreter. Depending on the chosen option, users can see an automaton representing the game, AST in JSON format, or run benchmarks testing their game.

It is also possible to see the results of translating code between the three supported languages. The right view then contains an editor working in read-only mode, connected to the same language server as the main editor. This part was done outside of the scope of this work.

[10]https://microsoft.github.io/monaco-editor/monarch.html

Chapter 4

Language server

Implementing the language server was the most important part of the work, but to do it, we had to extend parts of the Regular Games interpreter for LSP needs. Developers' code is seldom free of errors.

Previous RG parser would fail on the missing semicolon and not provide any information besides a single syntax error. To make it suitable for the language server, we had to rewrite it with the ability for partial parsing and error recovery[17, 18]. This would allow it to continue parsing after encountering an issue and report multiple diagnostics at once, which streamlines the development and debugging process.

Another characteristic of a desired parser is keeping track of token positions and creating AST enriched by this information. All navigation features of LSP must be able to find the definition position of every symbol easily. Error recovery requires one more essential ability – creating syntax trees that include erroneous nodes[19]. Those trees will not be used for compiling but should be as precise as possible to provide accurate semantic information about the code.

4.1 Parsing

The parser is written with `nom`, a parser combinator library. This type of parser is characterized by concise code and a direct representation of the grammar of the language, as seen in Listing 4.1. It makes it easy to write and reason about[20, 21].

```
fn typedef(input: Input) -> Result<Option<Typedef<Identifier>>> {
  with_semicolon(tag("type"),
    expect(separated_pair(preceded_opt_id("typedef"), expect_preceded_tag("="), type_),
      "type <identifier> = <type>;",
    ),
  )(input)
}
```

Listing 4.1: Example parser written with `nom`

On the other hand, combinatorial parsers usually do not provide as good error recovery as handwritten ones. We decided to use a mixed approach: we enriched the previous combinatorial parser with error recovery methods, written in a more by-hand manner (Listing 4.2).

```
fn edge_name(input: &str) -> Result<EdgeName<&str>> {
  context("edge_name", into(many1(separated(edge_name_part))))(input)
}

fn edge_name(input: Input) -> Result<EdgeName<Identifier>> {
  let (input, first ) = expect(edge_name_part, "edge name")(input)?;
  if let Some(name) = first {
    let (input, rest) = many0(preceded_whitespace(edge_name_part))(input)?;
    let mut parts = vec![name];
    parts.extend(rest);
    Ok((input, parts.into()))
  } else {
    let identifier = Identifier::none(Span::at(&input));
    Ok((input, vec![EdgeNamePart::Literal { identifier }].into()))
  }
}
```

Listing 4.2: Comparison of combinatorial parser and parser with error recovery

Most of the error handling takes place in the `expect` (Listing 4.3) method. It takes a `parser` function, applies it to an input, and in case of a failure, reports the encountered error. Inner `parser` could expect a single character but also a whole expression or statement. Commonly used is also `expect_id` method (Listing 4.4), which parses an identifier (with optional preceded whitespace). If the identifier is missing, it creates an `Identifier::none` node in its place.

```
fn expect<T>(
  parser: Input -> Result<T>,
  error_msg: &str,
) -> Input -> Result<Option<T>> {
  move |input| {
    let error_pos = Span::at(&input);
    match parser(input) {
      Ok((remaining, out)) => Ok((remaining, Some(out))),
      Err(input) => {
        let err = Error(error_pos, "expected: " + error_msg);
        input.extra.report_error(err);
        Ok((input, None))
      }
    }
  }
}
```

Listing 4.3: `expect` method

```
fn expected_id(context: &str) -> Input -> Result<Identifier> {
  move |input| {
    let start = Position(&input);
    expect(preceded_whitespace(identifier), context + ": identifier ")(input)
      .map(|(input, res)| {
        if let Some(res) = res {
          (input, res)
        } else {
          let span = start.with_end(&input);
          (input, Identifier::none(span))
        }
      })
  }
}
```

Listing 4.4: `expected_id` method

The input type seen in examples uses `LocatedSpan` from `nom-locate` crate. It extends the basic string by keeping track of the position in the source code. It is also enhanced further to store encountered errors.

4.2 Collecting semantic information

Given an abstract syntax tree, the task is to gather all instances of each symbol within the source file and divide them into definitions and references. A symbol can be a `variable`, a `type`, a `typemember`, an `edge` identifier, a `label`, or a `constant`. We store this information about every symbol as `Flag` (Listing 4.5). First, we traverse the AST and collect all definitions. This process is mostly simple, except for `labels`.

```
struct Symbol {
  flag: Flag,
  id: String,
  owners: Option<Vec<usize>>,
  position: Span,
}
```

Listing 4.5: `Symbol` struct

In RG, edges form a graph, and `label`s are shared between neighboring nodes. For example in Listing 4.6, every occurrence of `p` refers to the same symbol. We solve this by recreating the graph described by the source code and from it calculate which edges share given `label`. To differentiate identical `label`s at a later stage, we maintain within the `owners` field a collection of edges to which they are linked.

```
move, selectPos: player = PlayerOrKeeper(turnPlayer);
selectPos, selectedPos(p:Position): $ p;
selectedPos(p:Position), setPos(p:Position): p != Position(null);
setPos(p:Position), setFinished: position = Position(p);
```

Listing 4.6: Passing `label` between edges

Regular Games language also has several built-in types and variables. Users can not navigate to their definitions or see their types because we do not have access to the syntax trees containing their declarations. We still see them in completions and can list all their usages. Users can override any of these symbols, then the corresponding built-in is replaced and all LSP features are enabled for it.

Then the AST is traversed again, collecting all occurrences of every symbol. To save memory, only the position and unique symbol identifier are stored for each occurrence. If an unrecognized symbol is encountered during this process, a diagnostic about it is forwarded to the user together with parsing errors.

Occurrences together with symbols create a `SymbolTable` (Listing 4.7) data structure, that is later continually used by the language server.

```
struct Occurrence {
  position: Span,
  symbol: Option<usize>,
}

struct SymbolTable {
  occurrences: Vec<Occurrence>,
  symbols: Vec<Symbol>,
}
```

Listing 4.7: Structures for storing semantic data

4.3 Language server

The Language server is built using `tower-lsp`, which allows us to focus on defining LSP endpoints without worrying about low-level implementation details. It works completely asynchronously and can resolve multiple requests at the same time. It uses a map of `Document` structs, that store all the required data about source files: AST, `SymbolTable`, and the code itself.

For the `initialize` request that begins the connection with the client, it responds with the set of capabilities it provides. They define which endpoints are implemented, what kind of result to expect from them, but also how should be carried out the synchronization on change in the document. Upon opening a new file or changing anything in the code, respectively `didOpen` and `didChange` notifications are received. They result in parsing the code, building a `SymbolTable`, and publishing diagnostics back to the client.

Most implementations of LSP also trigger compilation here, to inform the user about mismatched types or unresolved imports, but doing it on every keystroke can be very computation-heavy. Because of this, language servers like Metals took a different approach, and even though they parse the file on every change, a compilation process is run only after saving the file.

4.3.1 Goto definition

The `goto definition` feature allows users to navigate through their codebase by jumping from symbol reference to its definition. Its implementation is fairly simple, as shown in Listing 4.8. First, the server searches for symbol occurrence at the cursor position then tries to get its definition and maps the result to the LSP data type. Although simple, it is one of the most commonly used functions of LSP.

```
fn definitions(
  uri: &Url,
  position: &Position,
  symbol_table: &SymbolTable,
) -> Option<GotoDefinitionResponse> {
  // We map a LSP position to an AST position.
  let rg_position = position.to_rg();
  // Gets symbol at a given position.
  let enclosing_symbol = symbol_table.get_symbol_at(&rg_position)?;
  // We make sure that the symbol has definition position.
  let symbol_position = enclosing_symbol.safe_pos()?;
  // Maps the result position to an LSP location.
  let location = symbol_position.to_location(uri);
  Some(GotoDefinitionResponse::Scalar(location))
}
```

Listing 4.8: `Goto definition` function

4.3.2 Show references

Connected to the previous method, `show references` allows developers to navigate from a variable definition to all its usages. Together they provide a way to effortlessly explore the structure of the code. Both these features work workspace-wide, which means they can navigate to other files, but since Regular Games does not support multi-file projects, this possibility is not utilized.

Implementation of this function is also simple – we need to find the symbol under the current cursor position, then we collect all its occurrences from the `SymbolTable` and filter out the occurrence in the definition.

For more complex languages that support importing functions and classes from other modules, implementation of this method can be much more complicated. It's almost impossible to traverse the whole workspace of every request in search of connected occurrences. That's why other language servers implement indexes, using data structures like trie that allow one to easily find in which files appear references to a given symbol.

4.3.3 Document highlight

This feature works alike `show references`, but only in the scope of a single file. Its implementation is also similar, but it doesn't exclude definition occurrence. Because this function doesn't look into other files, it is much faster and be used to quickly identify all usages of a variable.

4.3.4 Document symbols

`Document symbols` request awaits the list of all symbols defined in the source file. It allows developers to quickly navigate to them, without needing to find its occurrence first, like in `goto definition`. Based on `Flag` field, we enriched returned information with `SymbolKind`, which helps users resolve between identifiers of `variables`s, `type`s, and `edge`s.

4.3.5 Rename

The `rename` command is significantly more complex than those mentioned previously. It consists of two parts. The first, `prepare rename`, checks if an identifier on the current position can be renamed. If it succeeds, a text field with a placeholder value is displayed and awaits entering the new name. The function `prepare rename` can fail if the underlying identifier refers to a built-in symbol.

Afterwards, a `rename` request is sent from the client. It contains the span of an old identifier and a new text to be inserted. It is worth noting, that this command can be run on both definition and reference spans. Then, just like in `document highlight`, all occurrences of the symbols are collected, and each of them is used to create a `TextEdit` that replaces text under its position (Listing 4.9).

```
fn rename(
  uri: &Url,
  position: &Position,
  symbol_table: &SymbolTable,
  new_name: String,
) -> Option<WorkspaceEdit> {
  let symbol = symbol_table.get_symbol_at(&position.to_rg())?;
  let sym_idx = symbol_table.sym_idx(symbol)?;
  symbol.safe_pos().map(|_| {
    let changes = symbol_table.all_symbol_occurences(sym_idx).iter()
      .map(|occ| TextEdit {range: occ.pos.to_lsp(), new_text: new_name.clone()}).collect();
    WorkspaceEdit([(uri.clone(), changes)])
  })
}
```

Listing 4.9: Rename command

Similarly to `goto definition` and `show references`, this method works on the whole codebase and can apply edits in every file. In object-oriented languages like Java, if `rename` is called on a class identifier it also changes the name of the file containing its definition, because those two are closely connected.

4.3.6 Hover

The `hover` feature allows users to get instant information about a symbol. It is especially useful when trying to comprehend already written code. With it, the developer does not have to navigate to a variable definition to see its type.

To provide that, the language server first tries to get the symbol the user hovers over. `SymbolTable` does not store information about the variable type, so we need to find the statements enclosing this symbol definition in the `AST` and extract it from there (Listing 4.10).

```
fn hover(
  position: &Position,
  symbol_table: &SymbolTable,
  game: &Game<Identifier>,
) -> Option<Hover> {
  let symbol = symbol_table.get_symbol_at(&position.to_rg())?;
  let span = symbol.span();
  let type_ = game.stat_enclosing_span(span).and_then(|stat| stat.symbol_type(symbol));
  let contents = hover_signature(symbol, type_);
  Some(Hover {contents, range: Some(span.to_lsp())})
}
```

Listing 4.10: Hover function

4.3.7 Semantic highlighting

One of the most interesting LSP features is `semantic highlighting`. Most grammar-based syntax highlighters are not able to recognize if a given identifier is a constant, a mutable variable, or a parameter. If the user doesn't follow naming conventions, syntax coloring might be incorrect and misleading.

`Semantic highlighting` extends the capabilities of other coloring tools by taking into account semantic information. Thanks to this it can distinguish identifiers based on symbols `Flag`, which results in more precise decorations. For every word that should be colorized, it creates a `Token`, which consists of five integers: its type and modifier that are later resolved by the client, length, and distance in columns and lines from the previous token.

To provide them, it uses both the `AST` and `SymbolTable`. Using the former, it creates a `Token` for every keyword and pragma in the file. The latter helps to collect `Token`s for all symbol occurrences. It assigns each of them a token type referring to the `SemanticTokensLegend`, that is passed to the client among capabilities during the initialization process. After collecting all the `Token`s, they are sorted based on position and for every two of them, the distance between them is calculated.

Although in Listing 4.11 they are represented as a vector of `SemanticToken` structs, the low-level API of Language Server Protocol flattens the result to an array of integers.

```
fn semantic_tokens_full(document: &Document) -> Vec<SemanticToken> {
  let keywords = ast_tokens(&document.game);
  let symbols = symbol_table_tokens(&document.symbol_table);
  let mut tokens = [&keywords[..], &symbols[..]].concat();
  tokens.sort_by_key(|t| t.pos);
  let mut delta = Delta::default();
  tokens.into_iter()
    .map(|token| {
      delta.step(&token.pos);
      SemanticToken {
        delta_line: delta.line,
        delta_start: delta.column,
        length: token.len,
        token_type: token.token_type,
        token_modifiers_bitset: token.token_modifier,
      }
    }).collect()
}
```

Listing 4.11: `semantic_tokens_full` method

4.3.8 Completion

Completions are by far the most important and useful feature of an IDE. They accelerate coding speed, prevent typos, and reduce the burden on developers' memory associated with remembering all the function and variable names. Providing context-aware completion makes writing the code much easier since they suggest only those symbols, that can appear at a current position. For these reasons, we were especially attentive to them in the project.

First, based on cursor position, the `CompletionKind` has to be recognized (Listing 4.12). This was a motivation for writing a precise parser with exact spans and accurate error handling. Then, document symbols are filtered based on their `Flag`, which decides if they are suitable for the calculated `CompletionKind`. Every valid completion that encodes a constant or variable is annotated with its type from the `AST`. This helps developers decide which suggestion to apply.

```
fn completion_items(
  pos: Position,
  game: &Game<Identifier>,
  symbol_table: &SymbolTable,
) -> Vec<CompletionItem> {
  let completion_kind = game.stat_enclosing_position(&pos)
    .map(|stat| stat.completion_kind(&pos)).unwrap_or(CompletionKind::Toplevel);
  let mut items = get_symbols(symbol_table, &completion_kind.predicate())
    .map(|sym| completion_item(game, sym));
  if CompletionKind::Toplevel = completion_kind {
    items.extend(keyword_completion());
  }
  items
}
```

Listing 4.12: `completions` method

A related `completion_item_resolve` endpoint is called after changing an item in the suggestions list, to provide additional information about it. The reason for resolving this lazily is that getting documentation for a symbol is often computation-heavy. However, it was not implemented in this project.

4.3.9 Code actions

Code actions can be split into multiple categories, including `refactor`, `extract`, `inline`, and `quick-fix`. Which functions are implemented depends on the language needs. Every `codeaction` category is displayed differently in the IDE's UI

The response for `codeaction` request contains all actions that can be performed on the current position. They can be implemented as commands, this way only after one of them is applied its effect is calculated.

For the Regular Games language, we have implemented one `refactor` code action, namely `Split edge`. Running it when the cursor is over an edge will result in splitting the edge into two and adding an intermediate node between them. The exact edit depends on which part of the edge the cursor is on.

4.4 Testing

For testing LSP features, we need to simulate the behavior of the editor. Each test is constructed as a fragment of code which includes a cursor at some position. Before parsing the test, we extract the cursor position from it and use it as a request parameter. Example in Listing 4.13 tests which completions are allowed at a given position. Here `^` character indicates the cursor.

```
fn const_def() {
  completion_kind("const ^",CompletionKind::None);
  completion_kind("const foo: ^",CompletionKind::Type);
  completion_kind("const foo: ^ = 1;",CompletionKind::Type);
  completion_kind("const foo: Bar = ^ ",CompletionKind::Value);
  completion_kind("const foo: Bar = {:null, ^}",CompletionKind::Value);
  completion_kind("const foo: Bar = {:null, e1:^}", CompletionKind::Value);
}
```

Listing 4.13: Testing completions

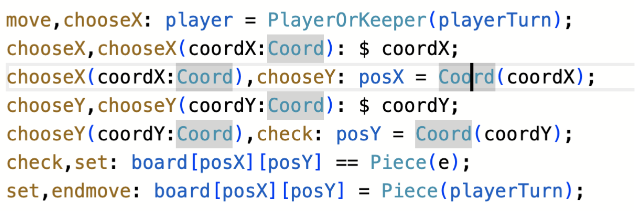
Chapter 5

User Manual

In this chapter, we show how to use the editor and trigger LSP features. The project is available at `https://radekmie.dev/rg/`, although the source code is not yet public.

- **Document Highlight**

  After clicking on a variable, or an edge name, all of its occurrences become highlighted. Users can then navigate between the highlights using `F7` and `Shift + F7`.

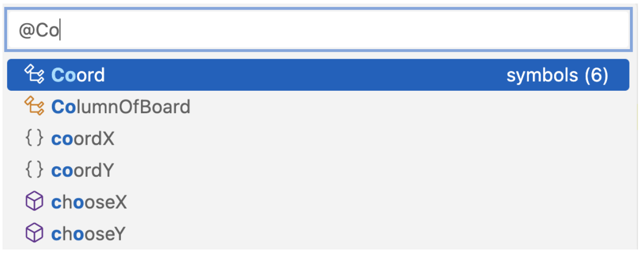


```
move,chooseX: player = PlayerOrKeeper(playerTurn);
chooseX,chooseX(coordX:Coord): $ coordX;
chooseX(coordX:Coord),chooseY: posX = Coord(coordX);
chooseY,chooseY(coordY:Coord): $ coordY;
chooseY(coordY:Coord),check: posY = Coord(coordY);
check,set: board[posX][posY] == Piece(e);
set,endmove: board[posX][posY] = Piece(playerTurn);
```

Figure 5.1: Document highlight

- **Document Symbols**

  To trigger document symbols, users need to press `Cmd + Shift + O`. They can then filter and navigate a list of symbols defined in the source.

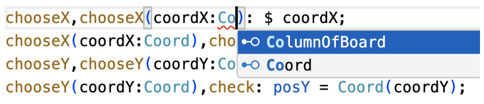


```
@Co
Coord                    symbols (6)
ColumnOfBoard
{} coordX
{} coordY
chooseX
chooseY
```

Figure 5.2: Document symbols

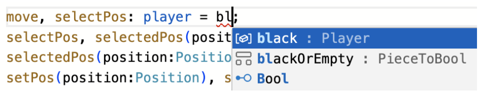
- **Completions**

  Completion requests are sent automatically while typing, but can also be forced with `Cmd + Space`.



```
chooseX,chooseX(coordX:Co): $ coordX;
chooseX(coordX:Coord),cho
chooseY,chooseY(coordY:Co
chooseY(coordY:Coord),check: posY = Coord(coordY);
```

(a) Type completions



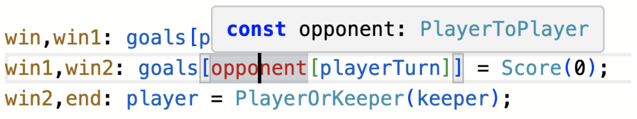
```
move, selectPos: player = bl;
selectPos, selectedPos(posit
selectedPos(position:Positio
setPos(position:Position), s
```

(b) Expression completions

Figure 5.3: Completions

- **Hover**

  Hovering over a variable displays a tooltip with information about its type.

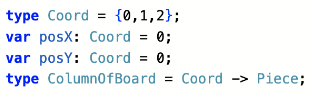


```
win,win1: goals[p
win1,win2: goals[opponent[playerTurn]] = Score(0);
win2,end: player = PlayerOrKeeper(keeper);
```

Figure 5.4: Hover

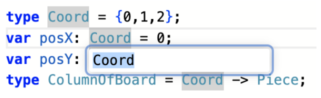
- **Rename**

  To rename a symbol, press the right mouse button on it and select `Rename Symbol` from the menu. Alternatively, users can press `F2` while having the cursor over a symbol. Then, a text field will appear and await inserting a new name.



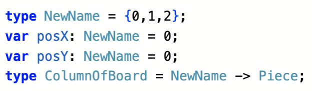
```
type Coord = {0,1,2};
var posX: Coord = 0;
var posY: Coord = 0;
type ColumnOfBoard = Coord -> Piece;
```

(a) Before rename



```
type Coord = {0,1,2};
var posX: Coord = 0;
var posY: Coord
type ColumnOfBoard = Coord -> Piece;
```

(b) Inserting new name



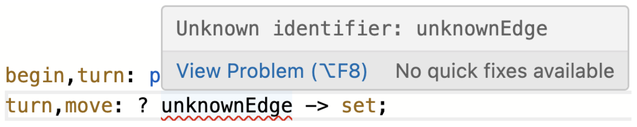
```
type NewName = {0,1,2};
var posX: NewName = 0;
var posY: NewName = 0;
type ColumnOfBoard = NewName -> Piece;
```

(c) Renaming result

Figure 5.5: Rename process

- **Diagnostics**

  Diagnostics are published automatically and displayed as squiggles. Users can hover over them to read related error messages.



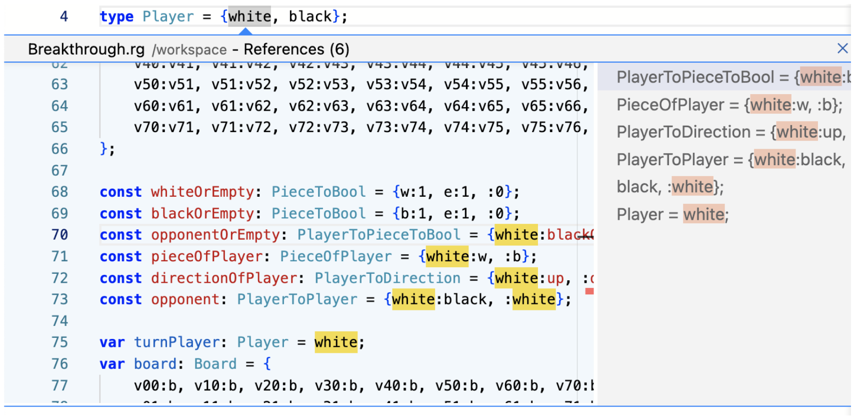
```
begin,turn: p
turn,move: ? unknownEdge -> set;
```

Unknown identifier: unknownEdge

Figure 5.6: Diagnostics

- **Navigation**

  To go to a symbol definition, use `Cmd` and click on a variable occurrence. Doing the same on the definition position will show all references to a symbol. Both commands can also be run from the menu after pressing the right mouse button on a symbol.



```
type Player = {white, black};
```

Breakthrough.rg /workspace - References (6)

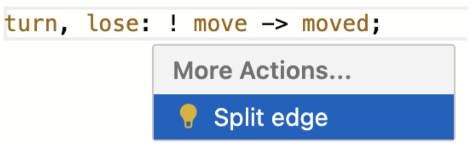
```
    v40:v41, v41:v42, v42:v43, v43:v44, v44:v45, v45:v46,
    v50:v51, v51:v52, v52:v53, v53:v54, v54:v55, v55:v56,
    v60:v61, v61:v62, v62:v63, v63:v64, v64:v65, v65:v66,
    v70:v71, v71:v72, v72:v73, v73:v74, v74:v75, v75:v76,
};

const whiteOrEmpty: PieceToBool = {w:1, e:1, :0};
const blackOrEmpty: PieceToBool = {b:1, e:1, :0};
const opponentOrEmpty: PlayerToPieceToBool = {white:black
const pieceOfPlayer: PieceOfPlayer = {white:w, :b};
const directionOfPlayer: PlayerToDirection = {white:up, :
const opponent: PlayerToPlayer = {white:black, :white};

var turnPlayer: Player = white;
var board: Board = {
    v00:b, v10:b, v20:b, v30:b, v40:b, v50:b, v60:b, v70:
```

```
PlayerToPieceToBool = {white:
PieceOfPlayer = {white:w, :b};
PlayerToDirection = {white:up,
PlayerToPlayer = {white:black,
black, :white};
Player = white;
```

Figure 5.7: Show references

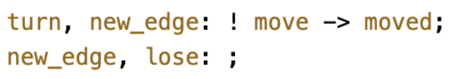
- **Code actions**

  To run **Split edge** code action, click on the edge and press `Cmd + .` (dot). Depending on the cursor position inside the edge, the result of the code action will differ.



```
turn, lose: ! move -> moved;
```

(a) Selection menu



```
turn, new_edge: ! move -> moved;
new_edge, lose: ;
```

(b) Split edge result

Figure 5.8: Code actions

Chapter 6

Summary and Further Development

6.1 Conclusion

This work presents an implementation of a custom IDE that runs entirely in a browser. First, we have introduced the Language Server Protocol and its associated features, showcasing its significance in modern software development. We have explained the requirements for writing a language server, specifically a parser with an error recovery mechanism, and a tool for extracting semantic data from code. We have presented solutions applied in a language server for the Regular Games language and compared them to approaches chosen in other projects.

Instead of using the Tree-sitter library for writing an incremental, fault-tolerant parser, we opted to refactor the existing RG parser to align it with the requirements of a language server. We have shown how to collect semantic information about symbols in a source file and use this data for various language server features.

This work also offers insight into how to create a fully functional Integrated Development Environment within a web platform, incorporating both a code editor and a language server. It shows how to set up a language client and describes the process of communicating with a language server.

Finally, we have presented the implementation of supported LSP functions. Additionally, a user manual has been provided, offering instructions on how to utilize these features.

6.2 Further work

This project can be extended in multiple ways. We could improve the support for Regular Games by adding more interaction between the language server and the interpreter. Compiling code on change in the editor could provide us with more error reports. Thanks to adding spans to AST nodes, we could enrich type-checker errors with positions.

The interpreter could also inform the language server about unused variables and redundant edges. These diagnostics would then be transformed into code actions and displayed in the editor as a warning with a way to fix them automatically.

Another improvement would be to add support for the three other languages: High-level Regular Games, Regular Boardgames, and GDL. Since parsers of two of them are not written in Rust, the biggest part of the work would be to rewrite them. Then, we would need to create a way to extract semantic data from their abstract syntax trees, preferably matching the `Symbol` and `SymbolTable` structs. Having that, adding LSP features for them should be easy, since the implementation would be shared between all languages.

Bibliography

[1] Stefan Marr, Humphrey Burchell, and Fabio Niephaus. 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*, page 1–14. Association for Computing Machinery, 2022.

[2] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, page 1–17. Association for Computing Machinery, 2020.

[3] Djonathan Barros, Sven Peldszus, Wesley K. G. Assunção, and Thorsten Berger. Editing Support for Software Languages: Implementation Practices in Language Server Protocols. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, page 232–243. Association for Computing Machinery, 2022.

[4] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a Language Server Protocol Infrastructure for Graphical Modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, page 370–380. Association for Computing Machinery, 2018.

[5] Frédéric Bour, Thomas Refis, and Gabriel Scherer. Merlin: a language server for OCaml (experience report). *Proceedings of the ACM on Programming Languages*, page 1–15, 2018.

[6] Tom Beckmann, Patrick Rein, Toni Mattis, and Robert Hirschfeld. Partial Parsing for Structured Editors. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, page 110–120. Association for Computing Machinery, 2022.

[7] Jakub Kowalski, Jakub Sutowicz, and Marek Szykuła. Regular Boardgames. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017.

[8] M. Genesereth and M. Thielscher. *General Game Playing*. Morgan & Claypool, 2014.

[9] Michael Genesereth and Nathaniel Love. General Game Playing: Game Description Language Specification. *Technical report, Stanford Logic Group*, 2006.

[10] Éric Piette, Dennis J. N. J. Soemers, Matthew Stephenson, Chiara F. Sironi, Mark H. M. Winands, and Cameron Browne. Ludii - The Ludemic General Game System. *CoRR*, 2019.

[11] Fabien Coulon, Alex Auvolat, Benoit Combemale, Yérom-David Bromberg, François Taïani, Olivier Barais, and Noël Plouzeau. Modular and Distributed IDE. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, page 270–282. Association for Computing Machinery, 2020.

[12] Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. Bringing the Web up to Speed with WebAssembly. *Commun. ACM*, page 107–115, 2018.

[13] Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. PhD thesis, EECS Department, University of California, Berkeley, 1998.

[14] Tim A. Wagner and Susan L. Graham. Efficient and Flexible Incremental Parsing. *ACM Trans. Program. Lang. Syst.*, page 980–1013, 1998.

[15] Eric R. Van Wyk and August C. Schwerdfeger. Context-Aware Scanning for Parsing Extensible Languages. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, page 63–72. Association for Computing Machinery, 2007.

[16] Eugene Burmako. SemanticDB: A Common Data Model for Scala Developer Tools (Invited Talk). In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*, page 2. Association for Computing Machinery, 2018.

[17] François Pottier. Reachability and Error Diagnosis in LR(1) Parsers. In *Proceedings of the 25th International Conference on Compiler Construction*, page 88–98. Association for Computing Machinery, 2016.

[18] Michael G. Burke and Gerald A. Fisher. A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery. *ACM Trans. Program. Lang. Syst.*, page 164–197, 1987.

[19] Sérgio Medeiros and Fabio Mascarenhas. Syntax Error Recovery in Parsing Expression Grammars. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, page 1195–1202. Association for Computing Machinery, 2018.

[20] Nils Anders Danielsson. Total Parser Combinators. *SIGPLAN Not.*, page 285–296, 2010.

[21] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. Practical, General Parser Combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, page 1–12. Association for Computing Machinery, 2016.