

A Comparison of Parsing and Editing Tools for Multi-Language Parsing

Pyl, Mitchel[†]

`mitchel.pyl@student.uantwerpen.be`

Vangheluwe, Hans[†]

`hans.vangheluwe@uantwerpen.be`

[†] University of Antwerp

August 2021

Abstract

Creating a textual Domain Specific Language requires both writing a parser and editing assistance services. Both of these are no trivial task and require careful selection of the tools used for designing the language. Existing tools for creating these languages offer good support for these, but are limited to supporting a single language only in a source document. We define a variability model to classify languages and evaluate support for them by different parsing technologies, and use this model as part of a comparison of multiple technologies for editing and parsing textual languages.

Contents

Glossary	4
Acronyms	5
1 Introduction	7
2 Background	7
2.1 Unicode	7
2.2 Parser Architecture	10
2.2.1 Grammar classifications	11
2.2.2 Parser generators	15
2.3 Common Token Type Categories	16
2.3.1 Identifiers	16
2.3.2 Keywords	16
2.3.3 Operators and Separators	16
2.3.4 White Space	16
2.3.5 Literals	17
2.3.6 Comments	17
2.4 Language Server Protocol	19
3 Language Variability Model	21
3.1 Concrete Syntax	21
3.2 Grammar Classification	22
3.3 Block Structures	22
3.3.1 Indentation Sensitive	22
3.3.2 Free-form	23
3.3.3 Section based	24
3.4 Statement Endings	24
3.5 Comments	26
3.6 Unicode	26
3.7 Language Dialects or Variations	27
4 Tools Overview	27
4.1 Spoofox	28
4.2 Xtext	28
4.3 JetBrains MPS	28
4.4 IntelliJ IDEA	28
4.5 Atom	29
4.6 Visual Studio Code	29
4.7 Python Lex-Yacc (PLY)	29
4.8 Python Lark	29

5	Criteria	29
5.1	Language Variability Support	29
5.2	Multi-language and Dynamic Parsing	29
5.3	Maintenance	30
5.4	Documentation	30
5.5	Setup	30
5.6	Support	31
5.7	Editor Features	31
5.8	Language Server Protocol	31
6	Comparison	32
6.1	Language Variability Support	33
6.2	Multi-language and Dynamic Parsing	42
6.3	Maintenance	45
6.4	Documentation	55
6.5	Setup	56
6.6	Support	58
6.7	Editor Features	59
6.8	Language Server Protocol	63
7	Conclusion & Future work	64
	References	66

Glossary

abstract syntax A representation of the internal structure of programs or models based on graphs or trees [77], [86].

Abstract Syntax Tree A tree representation of a source document after parsing. The result of performing semantic analysis on a parse tree. See Section 2.2.

Application Programming Interface A type of software interface that provides services to other software.

big endian Ordering method that puts the most significant byte of a word first when serializing the word (to storage, to the network, etc.).

chart parser A parser using the dynamic programming method, which stores partial results in a ‘chart’ structure, which allows them to be reused.

concrete syntax (textual) strings of characters that represent the abstract syntax of a model or program; **(visual)** a graphical representation of the abstract syntax of a model or program [77], [86].

Context-Free Grammar A formal grammar with production rules of the form $A \rightarrow \alpha$ with A a non-terminal and α a string of terminals and/or non-terminals. The production rules operate on non-terminals without context. See Section 2.2.1.

context-free language A language which is generated by a Context-Free Grammar, and can be recognized by a non-deterministic pushdown automaton. See Section 2.2.1.

Domain Specific Language A language that is specialized to be used in a specific application domain, such as when modelling or programming. The opposite is a General-Purpose Language (GPL), which is usable in a broad domain, and often lacks specialized features.

dynamic programming A programming method that simplifies a complicated problem by recursively breaking it down into simpler sub-problems.

Integrated Development Environment A software application for using programming or domain specific languages that provides the user with facilities to aid in usage and development. Facilities include assisted editing and build automation.

Language Server Protocol A protocol for communication between an editor or IDE (client) and a language server which provides language features to the client. See Section 2.4.

language workbench Collection of tools used to develop languages. See Section 2.2.2.

little endian Ordering method that puts the least significant byte of a word first when serializing the word (to storage, to the network, etc.).

metamodel A model for the abstract syntax of a modelling language. Also known as a Linguistic Type Model (LTM) [58], [86].

model An abstraction of a system that only contains those properties which are relevant for making predictions or inferences [58], [86].

operational semantics Specification of the semantics of a model as defined by a description of how the model operates, such as on an abstract machine or in a programming language [17], [77].

parse tree A tree representation of a source document right after parsing, without any extra semantic information. Related to Abstract Syntax Tree.

parser generator Tool used to create parsers from a grammar specification. See Section 2.2.2.

Parsing Expression Grammar A grammar specification formalism such as Context-Free Grammars, but which does not support ambiguous grammars, and supports some languages which are not context-free. See Section 2.2.1.

projectional editor A text editor which works by editing the abstract syntax directly instead of requiring parsing a text with a grammar. Results of edits are “projected” onto concrete textual syntax.

translational semantics Specification of the semantics of a model as defined by the translation of the model from one formalism to another [86].

Unicode A standard for encoding, displaying and handling text and characters in various scripts from around the world. See Section 2.1.

Acronyms

API Application Programming Interface.

ASCII American Standard Code for Information Interchange.

AST Abstract Syntax Tree.

CFG Context-Free Grammar.

DSL Domain Specific Language.

IDE Integrated Development Environment.

LSP Language Server Protocol.

PEG Parsing Expression Grammar.

UTF Unicode Transformation Format.

1 Introduction

The creation of a new language is no easy feat and requires not only the definition of a syntax, but also the internal representation and associated semantics. Additionally the aspect of user experience (UX) needs to be considered: for large or complicated languages some form of editing assistance is a must. For textual languages, this assistance includes things such as syntax highlighting, completion, refactoring, diagnostics, and more. Language workbenches allow designing Domain Specific Languages (DSLs) and provide the facilities to easily add editing assistance features. Integrated Development Environments (IDEs) also provide these facilities, but instead provide an abstract Application Programming Interface (API) which leaves implementing the actual logic to the programmer instead. Efforts exist to provide a standardized API for this, such as the Language Server Protocol for features related directly to editing, and the Debug Adapter Protocol for the debugging of languages.

The aim of this study is to review several tools such as language workbenches, Integrated Development Environment (IDE), editors, and software libraries to compare them with the end goal of producing a library or framework that (1) allows parsing multiple languages in a single document and (2) supports these languages to be defined and used immediately afterwards in the same document.

We start by providing some background information in Section 2, followed by Section 3 where we define a language variability model for comparing various languages, which can also double as a way to evaluate support for various forms of languages by parser generators and language workbenches. Finally, we build up a set of criteria partially based on our new concept and variability model, and use that to compare some tools for the creation and/or design of languages in Section 6.

2 Background

2.1 Unicode

Unicode is a standard that defines how to encode, display and handle text and characters in various scripts maintained by the Unicode Consortium [92]. The goal of the Unicode standard is to be a universal standard which supersedes other encoding schemes.

The most basic part of Unicode are *characters*, which are defined by their unique meaning or semantics. Each character is mapped to a unique *human readable name* and also to a unique code point. Hence if one knows either the character, the name of the character, or the code point of the character, all others are also known.

Each character is also assigned a *glyph*, which defines how a character should be displayed. The actual look of a character however depends on the used font, and multiple characters may be combined into a single glyph to form *ligatures* as defined by the font. Furthermore, depending on the language, the direction

of text can be from left to right such as in English, or from right to left such as in Arabic or Hebrew.

A *code point* is an integer value ranging from 0 through 1,114,111 (hexadecimal value 0x10FFFF) [92]. When referencing a Unicode character, the notation U+232C is used to represent the code point with value 0x232C (decimal value 9,004).

The Unicode standard defines a collection of encoding schemes called the Unicode Transformation Format (UTF), which define UTF-8, UTF-16, and UTF-32 [92]. These encoding schemes differ in their ‘word size’, which is the unit of operation.

- UTF-8 has a word size of 8 bits. It is known as a ‘variable-width encoding’ because the amount of bytes used to represent a code point depends on the value of the code point, Table 1 shows how this is implemented for UTF-8.
- UTF-16 has a word size of 16 bits, and like UTF-8 is a variable-width encoding. It represents code points from U+0000 to U+D7FF and from U+E000 to U+FFFF as their exact numeric value (values from U+D800 to U+DFFF are reserved, should not encode, and should be treated as errors if encountered). Code points from U+010000 to U+10FFFF are encoded using an algorithm as defined in Listing 1, which first subtracts 0x10000 from the code point and then splits it up between two words.
- UTF-32 is a fixed-width encoding with a word size of 32 bits. A code point encoded with UTF-32 has the exact same numerical value assigned to it as the numerical value of the code point itself.

For compatibility reasons, the first 256 code points in Unicode are based on the extended ASCII character set (ISO/IEC 8859-1). Because of this, the first 127 code points encode exactly the same as the original ASCII 7-bit encoding, which allows files encoded that way to be read by UTF-8 [92].

When encoding with UTF-16 or UTF-32, special care needs to be taken when writing these encoded strings to the network or non-volatile storage. For example, when writing a 16-bit number 0x36A9, we need to write two individual bytes, 0x36 (00110110) and 0xA9 (10101001). We can chose to write them in two different ways: 0x36 0xA9 or 0xA9 0x36. We call the first *big endian* (BE) because it puts the most significant byte (the byte which contributes the most value to a number) first. Similarly we call the second *little endian* (LE) because it puts the least significant byte first. The same concept applies for 32-bit numbers and higher. We call the ordering of multi-byte words the *endianness*.

Because of this, UTF-16 and UTF-32 both have three flavors: UTF-16, UTF-16BE, UTF-16LE, and UTF-32, UTF-32BE, UTF-32LE. In case no endianness is specified in an encoding, either a special Byte Order Mark (BOM) needs to be present, or the encoding is assumed to be big endian [92].

Table 2 shows two code points with example glyphs for both, as well as their name and encoding as a sequence of bytes.

1 st byte	2 nd byte	3 rd byte	4 th byte	Free bits	Highest Code Point
0 xxxxxxx				7	U+007F (127)
110 xxxxx	10 xxxxxx			11	U+07FF (2,047)
1110 xxxx	10 xxxxxx	10 xxxxxx		16	U+FFFF (65,535)
11110 xxx	10 xxxxxx	10 xxxxxx	10 xxxxxx	21	U+10FFFF (1,114,111)

Table 1: Binary representation of how UTF-8 encodes code points. Each x represents a single bit of the encoded code point. Code points are encoded using the shortest representation, as such U+003D is only representable as **00111101** and U+232C as **11100010 10001100 10101100** [92].

```

U' = yyyyyyyyyyxxxxxxxxxx // U - 0x10000
W1 = 110110yyyyyyyyyy // 0xD800 + yyyyyyyyyy
W1 = 110111xxxxxxxxxx // 0xDC00 + xxxxxxxxxxxx

```

Listing 1: Algorithm for encoding of a code point U into UTF-16 if U is between $U+010000$ and $U+10FFFF$, which turns into two words $W1$ and $W2$ [92].


	Name	Equals Sign	Hundred Points Symbol
	Example Glyph	=	
	Code point	U+003D	U+1F4AF
Encoding	UTF-8	0x3D	0xF0 0x9F 0x92 0xAF
	UTF-16LE	0x3D 0x00	0x3D 0xD8 0xAF 0xDC
	UTF-16BE	0x00 0x3D	0xD8 0x3D 0xDC 0xAF
	UTF-32LE	0x3D 0x00 0x00 0x00	0xAF 0xF4 0x01 0x00
	UTF-32BE	0x00 0x00 0x00 0x3D	0x00 0x01 0xF4 0xAF

Table 2: Comparison of the symbols **Equals Sign** and **Hundred Points Symbol** and their hexadecimal representation for different Unicode encoding schemes. Note that **Equals Sign** is on the Basic Multilingual Plane (BMP), while **Hundred Points Symbol** is on the Supplementary Multilingual Plane (SMP), which requires more bytes to represent code points in UTF-8 and UTF-16 encoding schemes.

2.2 Parser Architecture

A parser is a program that transforms a string of characters into a parse tree, which can then be processed such as for compilation, translation, or diagnostics. Usually this process is split up into two phases: a lexing phase and a parsing phase. In Figure 1 an overview of this process can be seen.

During the lexing phase (also called lexical analysis), a source string is analyzed and turned into a stream of tokens. This translation of text to tokens is called tokenization, and is generally defined by a collection of Regular Expressions mapped to a ‘token type’. Until the end of the input is reached, the lexer takes the current position in the input and attempts to get a match with each Regular Expression taking the ‘best’ match (first, longest, etc.), and then emits this token and progressing the input position to after the match. If no match is made with any defined Regular Expression, a lexical error is produced. Implementations can decide to perform lexical analysis with more advanced algorithms when required for their syntax.

The parsing phase (also called syntactic analysis) takes the token stream from the lexing phase and turns them into a parse tree. This translation is

usually specified as a Context-Free Grammar (CFG) (see Section 2.2.1), with the implementation able to choose which algorithm they use. For performance reasons, most algorithms support only a subset of all CFGs. Additionally, while for most parsers there only exists communication from the lexer to the parser, a lot of programming languages also allow communication from the parser back to the lexer. For example, the parser could tell the lexer to change the allowed token types based on its current state. The implication for this could be that the language is no longer context-free, but it still manages to use a parser based on context-free grammars for performance.

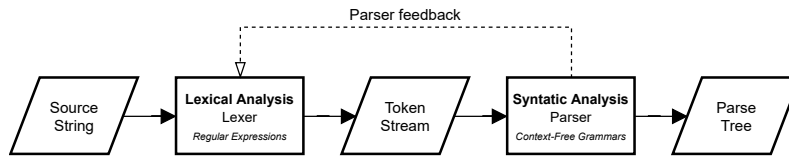


Figure 1: Overview of a parser architecture as it is usually implemented, with a lexing step (scanner/tokenizer) before the actual parsing. Usually there exists a mechanism for the parser to interact with the lexer, for example to change its state.

Some parsers omit the first phase, we call these parsers “scannerless”. The parsing phase in this case operates on the plain source string instead, with the terminal symbols being either characters or the raw bytes instead of tokens. An overview of this process can be seen in Figure 2.

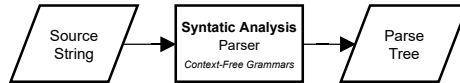


Figure 2: Overview of a parser architecture without lexing step.

2.2.1 Grammar classifications

Languages can be divided into several categories and subcategories, in [15], [16] Chomsky defined a hierarchy of four grammar types, with Type-0 giving the most freedom and Type-3 being the most restricted.

The following is an overview of the four grammar types, with production rules where:

a is a terminal;

A, B are non-terminals;

α, β a possibly empty string of terminals and/or non-terminals; and

γ a non-empty string of terminals and/or non-terminals.

Type-0 Recursively enumerable languages, recognizing a language in this type requires a Turing machine. Grammars for these languages have productions of the form $\gamma \rightarrow \alpha$.

Type-1 Context-sensitive languages, which can be recognized by linear-bounded non-deterministic Turing machines. Grammars for these languages have productions of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ and are called “context-sensitive” due to the production rule requiring context α and β to transform a non-terminal A .

Type-2 context-free languages, which can be recognized by non-deterministic pushdown automata. Grammars for these languages have productions of the form $A \rightarrow \alpha$ and are called “context-free” because the production does not require a context to transform a non-terminal A .

Type-3 Regular languages, which can be recognized by finite-state machine. Grammars for these languages have productions of the form $A \rightarrow a$ and $A \rightarrow aB$.

We focus on type-2 grammars, which are expressed with Context-Free Grammars. Additionally we look at the Parsing Expression Grammar (PEG) formalism, which is similar but does not describe the same set of languages. For example, while Listing 2 shows a PEG grammar for the *context-free* language $\{a^n b^n : n \geq 1\}$, Listing 4 shows a PEG grammar for the *context-sensitive* language $\{a^n b^n c^n : n \geq 1\}$. Note that PEGs use an ordered choice whereas CFGs use an unordered choice, and that PEGs include positive (&) and negative (!) look-ahead predicates.

$\langle S \rangle ::= \text{'a'} \langle S \rangle? \text{'b'}$

Listing 2: Parsing Expression Grammar for the context-free language $\{a^n b^n : n \geq 1\}$.

Cursor	State
aabb	$\langle S \rangle ::= * \text{'a'} \langle S \rangle? \text{'b'}$ (start)
a abb	$\langle S \rangle ::= \text{'a'} * \langle S \rangle? \text{'b'}$ (progress)
a abb	$\langle S \rangle ::= * \text{'a'} \langle S \rangle? \text{'b'}$ (check $\langle S \rangle$)
aa bb	$\langle S \rangle ::= \text{'a'} * \langle S \rangle? \text{'b'}$ (progress)
aa bb	$\langle S \rangle ::= * \text{'a'} \langle S \rangle? \text{'b'}$ (check $\langle S \rangle$)
aa bb	$\langle S \rangle ::= \text{'a'} \langle S \rangle? * \text{'b'}$ (fail, return, progress)
aab b	$\langle S \rangle ::= \text{'a'} \langle S \rangle? \text{'b'} * \text{'b'}$ (progress)
aab b	$\langle S \rangle ::= \text{'a'} \langle S \rangle? * \text{'b'}$ (return, progress)
aabb	$\langle S \rangle ::= \text{'a'} \langle S \rangle? \text{'b'} * \text{'b'}$ (progress, complete)

Listing 3: Example parse of the string ‘aabb’ for the context-free language $\{a^n b^n : n \geq 1\}$ by a PEG parser using the grammar in Listing 2. ‘|’ indicates the current input position, and ‘*’ indicates the current position in a state.

$\langle S \rangle ::= \&(\langle A \rangle \text{'c'}) \text{'a'} + \langle B \rangle !$.

$\langle A \rangle ::= \text{'a'} \langle A \rangle? \text{'b'}$

$\langle B \rangle ::= \text{'b'} \langle B \rangle? \text{'c'}$

Listing 4: Parsing Expression Grammar for the context-sensitive language $\{a^n b^n c^n : n \geq 1\}$.

Cursor	State
aabbcc	<S> ::= * &(<A> 'c') 'a'+ !. (start)
aabbcc	<S> ::= &(* <A> 'c') 'a'+ !. (progress)
aabbcc	<A> ::= * 'a' <A>? 'b' (check <A>)
a abbcc	<A> ::= 'a' * <A>? 'b' (progress)
a abbcc	<A> ::= * 'a' <A>? 'b' (check <A>)
aa bbcc	<A> ::= 'a' * <A>? 'b' (progress)
aa bbcc	<A> ::= * 'a' <A>? 'b' (check <A>)
aa bbcc	<A> ::= 'a' <A>? * 'b' (fail, return, progress)
aab bcc	<A> ::= 'a' <A>? 'b' * (progress)
aab bcc	<A> ::= 'a' <A>? * 'b' (return, progress)
aabb cc	<A> ::= 'a' <A>? 'b' * (progress)
aabb cc	<S> ::= &(<A> * 'c') 'a'+ !. (return, progress)
aabbc c	<S> ::= &(<A> 'c' *) 'a'+ !. (progress)
aabbcc	<S> ::= &(<A> 'c') * 'a'+ !. (progress)
aa bbcc	<S> ::= &(<A> 'c') 'a'+ * !. (progress)
aa bbcc	 ::= * 'b' ? 'c' (check)
aab bcc	 ::= 'b' * ? 'c' (progress)
aab bcc	 ::= * 'b' ? 'c' (check)
aabb cc	 ::= 'b' * ? 'c' (progress)
aabb cc	 ::= * 'b' ? 'c' (check)
aabb cc	 ::= 'b' ? * 'c' (fail, return, progress)
aabbc c	 ::= 'b' ? 'c' * (progress)
aabbc c	 ::= 'b' ? * 'c' (return, progress)
aabbcc	 ::= 'b' ? 'c' * (progress)
aabbcc	<S> ::= &(<A> 'c') 'a'+ * !. (return progress)
aabbcc	<S> ::= &(<A> 'c') 'a'+ !. * (progress, complete)

Listing 5: Example parse of the string 'aabbcc' for the context-sensitive language $\{a^n b^n c^n : n \geq 1\}$ by a PEG parser using the grammar in Listing 4. '|' indicates the current input position, and '*' indicates the current position in a state.

$\langle S \rangle ::= \langle T \rangle \text{'+' } \langle T \rangle$

$\langle T \rangle ::= \text{'a'}$
 $\quad \quad \quad | \text{'b'}$

Listing 6: Simple grammar that can generate the strings **a+a**, **a+b**, **b+a**, and **b+b**.

There are several algorithms for parsing Context-Free Grammars. The difference between them lies in how the algorithm behaves:

- Most parsers are ‘left-to-right’ and read input (characters if lexerless, tokens if with lexing step) without backing up.
- Some allow looking ahead one or more characters or tokens.
- Parsers can either perform a top-down parse which attempts to construct the parse tree by looking at it from the top, or a bottom-up parse which first looks at the lowest level of the parse tree and gradually constructs it upwards.
- They can perform a leftmost derivation whereby production rules are expanded from left to right, or a rightmost derivation does so in the opposite direction. An example leftmost derivation for **a+b** from Listing 6 can be given as follows:

$$\langle S \rangle \vdash \langle T \rangle \text{ ‘+’ } \langle T \rangle \vdash \text{ ‘a’ ‘+’ } \langle T \rangle \vdash \text{ ‘a’ ‘+’ ‘b’ } \quad \square$$

The same string can be derived using a rightmost derivation as follows:

$$\langle S \rangle \vdash \langle T \rangle \text{ ‘+’ } \langle T \rangle \vdash \langle T \rangle \text{ ‘+’ ‘b’ } \vdash \text{ ‘a’ ‘+’ ‘b’ } \quad \square$$

Figure 3 shows an overview of the most common parser algorithms for Context-Free Grammars, a short explanation of their names is given here:

- **LL(*k*)**: **L**eft-to-right, **L**eftmost derivation with *k* tokens lookahead
- **LR(*k*)**: **L**eft-to-right, **R**ightmost derivation with *k* tokens lookahead
- **SLR**: **S**implified **LR** parser, based on an LR(0) parser.
- **LALR(*k*)**: **L**ook-**A**head **LR** parser with *k* tokens look-ahead, an extension to LR(0) parsers.
- **GLR**: **G**eneralized **LR** parser, an extension to the LR parser algorithm to support non-deterministic and ambiguous grammars. It can parse all context-free languages.
- **Earley**: a chart parser using dynamic programming, named after its developer Jay Earley [18]. It can parse all context-free languages including those that are ambiguous and/or non-deterministic.
- **CYK**: the **C**ocke-**Y**ounger-**K**asami algorithm [79], another chart parser. It can parse all context-free languages including those that are ambiguous and/or non-deterministic.

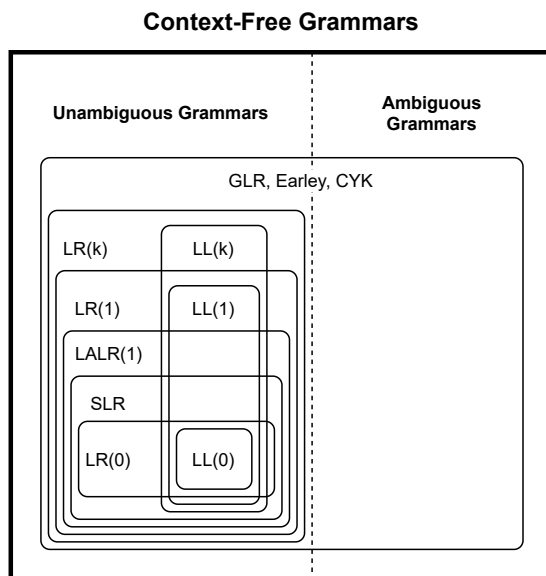


Figure 3: Hierarchical representation of Context-Free Grammar classes, adapted from [31].

Additionally, as said earlier Parsing Expression Grammars parser can parse all context-free languages and some context-sensitive languages. PEG parsers are mostly implemented as recursive descent parsers, which use recursive procedures to parse and allows both backtracking and infinite lookahead. By using memoization to store the results of recursive parsing functions, the performance of these parsers can be improved, these sorts of parsers are called ‘packrat parsers’.

2.2.2 Parser generators

A parser generator is a sort of compiler, with its own language and parser, that takes a grammar specification and turns it into a parser. Examples of parser generators are ANTLR, GNU Bison, and Yacc. Some parser generators do not generate the whole parser pipeline, and instead only generate the syntactical analysis phase. These rely on other tools such as Lex and Flex to generate the lexical analysis phase.

A language workbench contains a parser generator, but also includes functionality such as diagnostics and special editor features. Languages designed with a language workbench usually also automatically generate editors, which on their own also provide special editor features either out of the box or by the developer via tools provided by the workbench.

2.3 Common Token Type Categories

Tokens passed from the lexer to the parser have an associated type (see Section 2.2). When designing a language there are several sorts of token types that are inevitably encountered. We go over a few of them.

2.3.1 Identifiers

Identifiers are used as a way to assign a name to parts of a program or model. A name assigned to something usually needs to be unique with respect to a specific ‘scope’, which dictates where a name is valid. When referencing something named, this usually requires that the name is valid in the referencing scope.

An identifier is usually made up of some letters and numbers, but cannot begin with a number. Some languages support using Unicode characters. An example identifier could be `‘foobar’`.

2.3.2 Keywords

Keywords are reserved words which have a special meaning to the compiler. When a parser makes use of a lexer, these are usually returned as instances of their own token type. Keywords cannot be used as identifiers in most languages, though there are some languages such as for example JavaScript that allow keywords to be used as identifiers in some places, but not in others [28], [75].

Some example keywords are `‘for’`, `‘if’`, `‘float’`, `‘return’`, `‘class’`, and `‘import’`.

2.3.3 Operators and Separators

Separators are like punctuation used in natural language, in a sense that they are used to split a program into multiple parts that each have their own meaning. Examples of separators are: parentheses (`‘(’` and `‘)’`), brackets (`‘[’` and `‘]’`), braces (`‘{’` and `‘}’`), semicolons (`‘;’`), and commas (`‘,’`).

Operators are used to define a semantic operation or relation on one or more ‘operands’: for example the addition of two numbers. Examples of operators are: addition (`‘+’`), increment (`‘++’`), subtraction (`‘-’`), division (`‘/’`), assignment (`‘=’`), equality check (`‘==’`), and boolean AND (`‘&&’`).

2.3.4 White Space

White space are characters which are used for layout. These are usually the following characters: space, horizontal tab, newline, and carriage return.

Under Unicode the `White_Space` property is defined as: “Spaces, separator characters and other control characters which should be treated by programming languages as “white space” for the purpose of parsing elements.”.

2.3.5 Literals

Literals represent a constant value in source code such as numbers, booleans, strings.

Strings are usually delimited by single (‘’) or double quotes (’’), or by a sequence of characters. These sometimes have different meanings depending on the delimiters. For example, in C a double quoted string ends with a NULL byte, while a single quoted string does not. The ability to write Unicode characters is often allowed either directly or by using an escape sequence (usually as a sequence of ASCII characters) which inserts them post parse. The following is an example of a valid string in Python 3: "Привет мир!". The same string can also be represented using character escapes as:

```
"\u041f\u0440\u0438\u0432\u0435\u0442 \u043c\u0438\u0440!"
```

Booleans only have two possible values, and as such they are usually represented using keywords such as `true` and `false`.

Numbers are usually written using decimal digits 0 through 9 in programming languages. There are however other characters for representing numbers included in Unicode. For example, the number 69 in Japanese Kanji is 六十九

Some languages have a concept of a ‘null’ value or pointer, which represents the absence of a value or a reference to nothing. Examples of these are ‘null’, ‘nullptr’, ‘nil’, and ‘None’.

2.3.6 Comments

Most languages have a way of writing comments, which are a way to add non-semantic pieces of text in a source document, such as for documentation, or for temporarily disabling or removing part of the model or program. Comments can be formed in several ways. Van Tassel put them into four categories in [93]: positional, end-of-line, block, and mega comments. Additionally, some forms of comments can lie outside of this categorization. For example, some esoteric languages such as Brainfuck ignores any character which does not represent an instruction (instructions being one of ‘>’, ‘<’, ‘+’, ‘-’, ‘.’, ‘,’, ‘[’, and ‘]’), which is its way of writing comments.

Positional comments

Positional comments consider the entire line to be a comment, and require the comment indicator to be in a specific position. Languages making use of this style of comments are mostly from the age of when punch cards were still.

Examples of languages that use positional comments: FORTRAN (‘C’ in column 1), BASIC (‘REM’ at start of line, see Listing 7), COBOL (‘*’ in column 7).

```
010 REM      FIND PRIME NUMBERS LESS THAN 100
020 REM      BY DENNIE VAN TASSEL
030 REM      JULY 4, 1965
```

```
040 LET A = 1
```

Listing 7: Example partial BASIC code indicating comment lines by putting REM at the beginning of the line. Example from [93].

End-of-line Comments

End-of-line comments are slightly less restrictive than positional comments in that they allow the comment indicator to be anywhere on the line, and all columns after it are treated as comment, while the columns before are interpreted as program text.

Examples of languages that use end-of-line comments: Assembly (`;`), MySQL (`--`), C++ (`//`, see Listing 8), Java (`//`), Python (`#`).

```
int main() {  
    return 0; // Don't do anything, just return  
}
```

Listing 8: Example C++ code with an end-of-line comment starting with `//`.

Block Comments

Block comments can start and end at arbitrary positions in a file, but instead of having only one indicator, they have a start indicator and an end indicator. This allows comments that span only a short piece of a line, and also those that span multiple lines. A use case for these is the commenting out of longer pieces of code without the need for adding a comment on each line.

Examples of languages that use block comments: C/C++ (`/*` `*/`), Java (`/*` `*/`), HTML/XML (`<!--` `-->`, see Listing 9), etc.

```
<?xml version = "1.0"?>  
<dictionary>  
    <!-- Placeholder, dictionary is empty -->  
</dictionary>
```

Listing 9: Example XML with a block comment between `<!--` and `-->`.

Mega Comments

If a comment exists within another comment, we call this a nested comment. The idea behind this is the same as that which Van Tassel called mega comments in [93]. A use case for this is that you may want to disable a piece of code that already contains comments (either other code that is disabled, or documentation).

Examples of languages that support nesting of comments: Scala, Swift (see Listing 10), Dart, XML (not as actual comments, but as the marked section type IGNORE, which prevents processing in conforming parsers [89], see also Listing 11).

```
/*
 * Comments are started by /* and ended with */
 * The comment is still going on even after
 * the closing sequence due to nesting.
 */
print("/* This is not a comment */")
```

Listing 10: Example nested comments in the Swift programming language, with comments delimited by ‘/*’ and ‘*/’ and allowing nesting.

```
<?xml version = "1.0"?>
<profile>
  <![IGNORE[
  <!-- This is a regular comment in a mega comment -->
  <friends>
    <friend>Anna</friend>
    <![IGNORE[
    Disabled for testing
    <friend>Steve</friend>
    ]]>
  </friends>
  ]]>
</profile>
```

Listing 11: Example nested comments in XML, with IGNORE sections [89] being used as comments, which allows them to be nested.

2.4 Language Server Protocol

The Language Server Protocol (LSP) [13], [71] is a protocol that defines a way for editors or IDEs (“language clients”) to communicate with “language servers”, which provide editor features such as auto completion, error checking, refactoring, etc. The goal of the protocol is to allow developers to write a language server once and be able to reuse it for multiple clients, as opposed to having to implement the same features for different tools through different APIs and possibly even in different programming languages.

The general architecture is one where a language server is started by a client when it requires services provided by the server. This server only provides services to the client that started it, but the client itself can request services from multiple servers independently (see Figure 4).

Additionally, a language server can itself act as a client by starting other language servers, which it can then use to delegate requests to. This approach is one possible way of implementing embedded languages. The other is to use request forwarding but this requires the client to do the heavy lifting.

The protocol is based on the JSON-RPC protocol, which is a remote procedure call protocol encoded in JSON [56]. Clients and servers can both send either requests which require a response for either successful handling or error conditions, or notifications which do not require or even allow a response to be sent back. A full overview of the protocol can be found at [71].

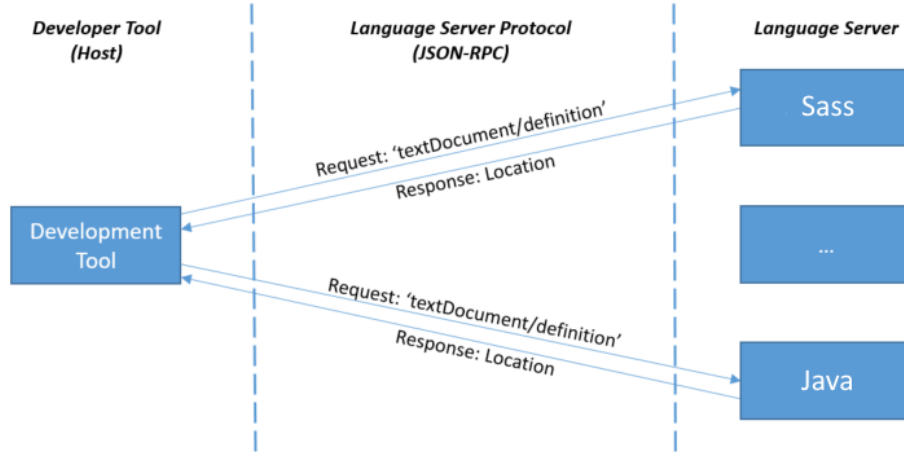


Figure 4: Illustration of the Language Server Protocol when a user using a development tool acting as the language client is working with multiple languages. The client starts multiple language servers that each process requests for their respective languages [71].

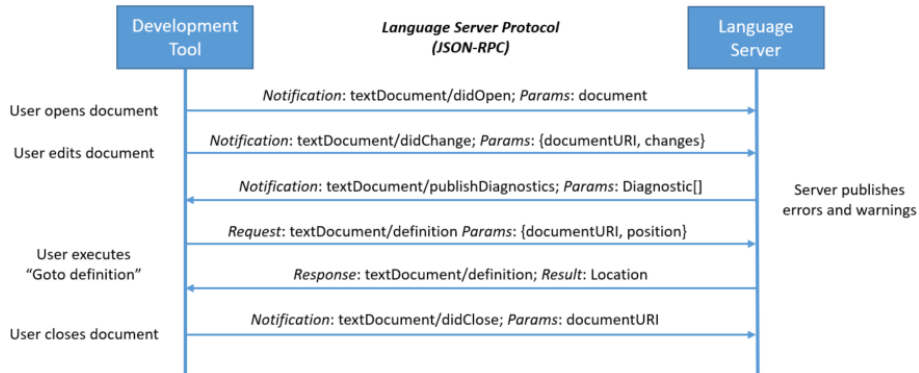


Figure 5: Illustration of the Language Server Protocol showing the communication between a language client and a single language server [71].

3 Language Variability Model

One of our goals is to have a parser that can work with many different kinds of languages and support different language configurations. In this part we'll go over some ways these languages can differ from each other in their concrete syntax by building a variability model, without looking at their abstract syntax and semantics. An overview of our variability model can be seen in Figure 6.

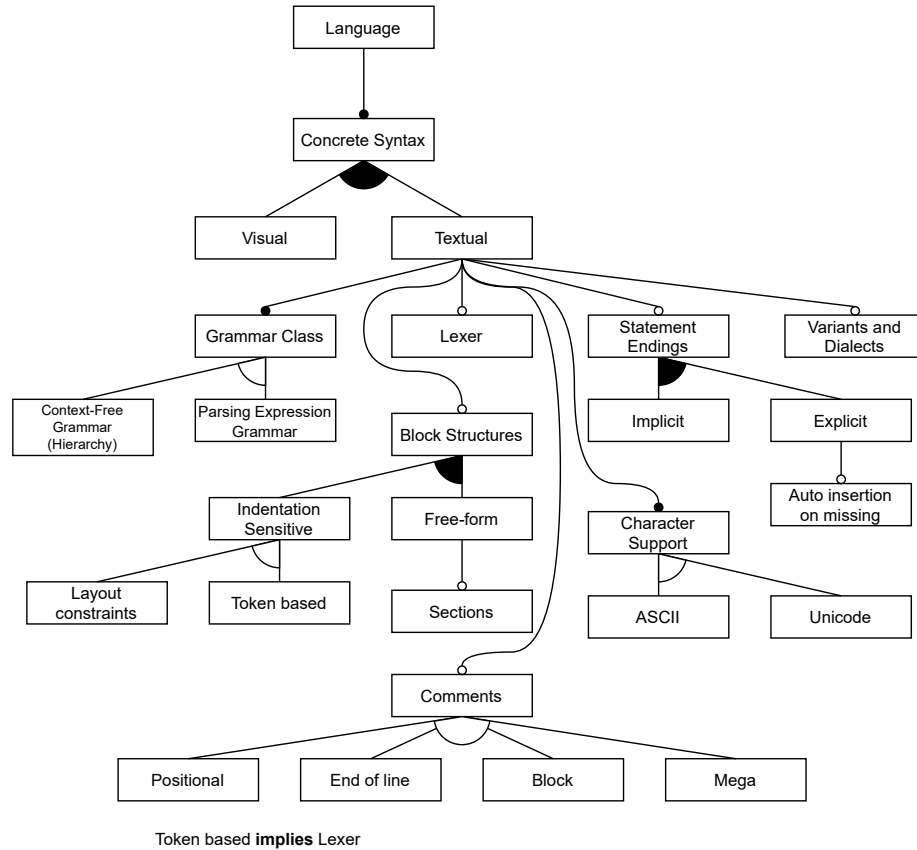


Figure 6: Visual representation of the variability model which we will use in our comparison in Section 6.

3.1 Concrete Syntax

Larkin and Simon [60] discerned two kinds of representations for problems: sentential (with sentences in a natural language describing the problem) and diagrammatic (made up of diagrams, where every component describes part of the problem). When modeling or writing code, we have a similar division in

languages: textual languages that describe a model or program with characters and words, and visual languages which describe a model or program by shapes, colors, etc.

In the context of modelling, we speak of “concrete syntax” when talking about the notation of a language, which can be either visual or textual. This visual or textual concrete syntax gets mapped to the Abstract Syntax, which is the abstract representation of a model.

We include this variability point for comparing editors mainly, due to the fact that being able to edit textual and visual languages in the same program offers users a better user experience such as having integrated environment and needing to switch context less often.

3.2 Grammar Classification

As we described in Section 2.2.1 and can be seen in Figure 3, not every context-free parser supports every Context-Free Grammar. Special care needs to be taken to ensure the tool we chose supports a wide range of grammars, while also keeping in mind the performance loss for extended support such as ambiguity or non-determinism. Furthermore, some tools use a lexer or scanner step to pre-process the parsed input, while others send the input text straight to the parser, and this needs to be taken into account when choosing a tool.

3.3 Block Structures

A block is a collection or sequence of declarations and statements that combined have some sort of semantic meaning.

In most programming languages, a block defines a sequence of instructions to perform in that order. It also provides a concept of ‘scope’ which dictates what ‘names’ or ‘identifiers’ are referenceable by the instructions in said scope.

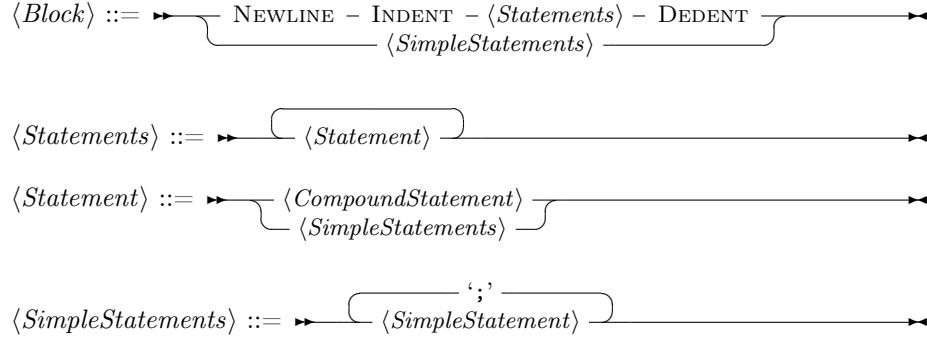
In the following sections we describe two major and one minor forms of syntax to indicate the start and end of a block.

3.3.1 Indentation Sensitive

Originally coined by Landin as the “off-side rule” in 1966 [59] when he described a fictional family of languages called ISWIM (If you See What I Mean). Nowadays these languages are referred to as “indentation sensitive” languages [1], [11], which use the indentation of the first token on a line to denote which block it belongs to.

Examples of these sorts of languages are Python [91] (see Listing 12), CoffeeScript [4], and Make files.

Implementing grammars of these sort require special support by the parser, either by having a lexer that produces tokens for increasing and decreasing the indentation (see for example Listing 12), or by putting layout constraints in the grammar [29]. Our evaluation for block structure support in our comparison in Section 6 will thus focus mainly on this.



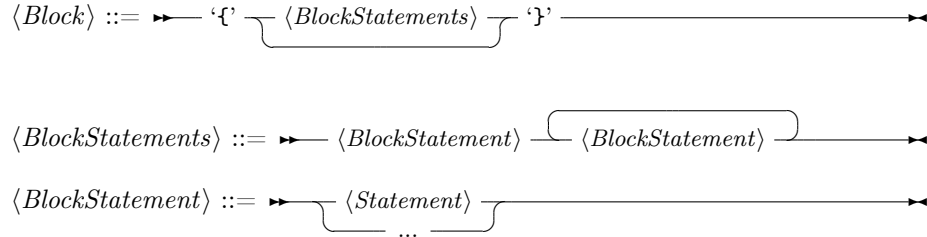
Listing 12: Block definitions for Python 3 grammars. INDENT and DEDENT are tokens generated by the lexer based on the indentation level of the source document [91].

3.3.2 Free-form

If the usage of indentation or alignment does not have any meaning in a language, we call it a “free-form language”. These sorts of languages require other ways of indicate the start and end of blocks.

A lot of modern languages such as C++ [42], Java [41], Rust [78] use curly braces (‘{’ and ‘}’) to start and end a block respectively, while ALGOL 68 uses parentheses for this purpose. Other languages use words such as ‘begin’ and ‘end’, for example Pascal [43] or SHell script (‘if’/‘fi’, ‘do’/‘done’, etc.).

Writing grammars for these constructs is trivial, as they are just tokens passed from the lexer to the parser without special processing, or in the case of a scannerless parser are parsed directly without any special processing. An example of such grammars can be taken from the latest Java language specification [41] (Java 16) and can be seen in Listing 13.



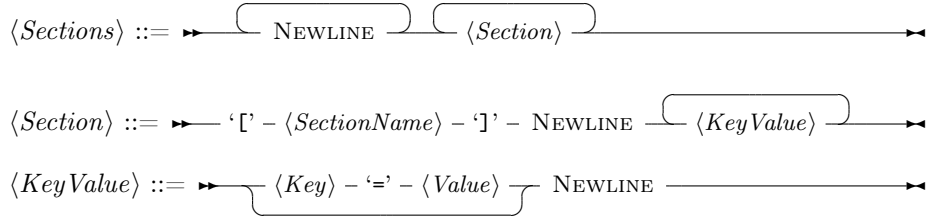
Listing 13: Block statement grammar specification for Java 16 [41] (chapter 14).

3.3.3 Section based

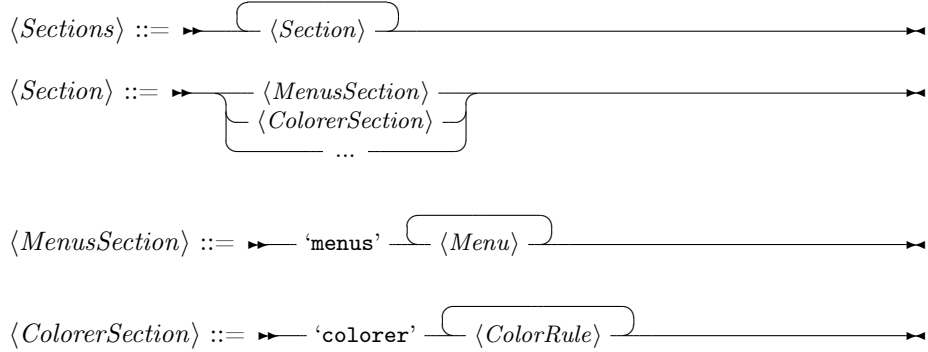
We view section based block structures as a variation to free-form languages. They have a single or series of symbols which indicate the start of a block, but have no specific symbol(s) to end a block, as they instead end implicitly.

Examples of these sorts of languages are INI (see Listing 14) configuration files, and the meta-languages used by Spoofax such as SDF3, NaBL2 and ESV (see Listing 15) [66].

Grammars for these languages are similar to those of free-form languages, except for the lack of a specific end character, and instead rely on the parser to automatically detect the end of each section. This form is most noticeable in the example in Listing 15.



Listing 14: A partial example grammar definition for INI configuration files.

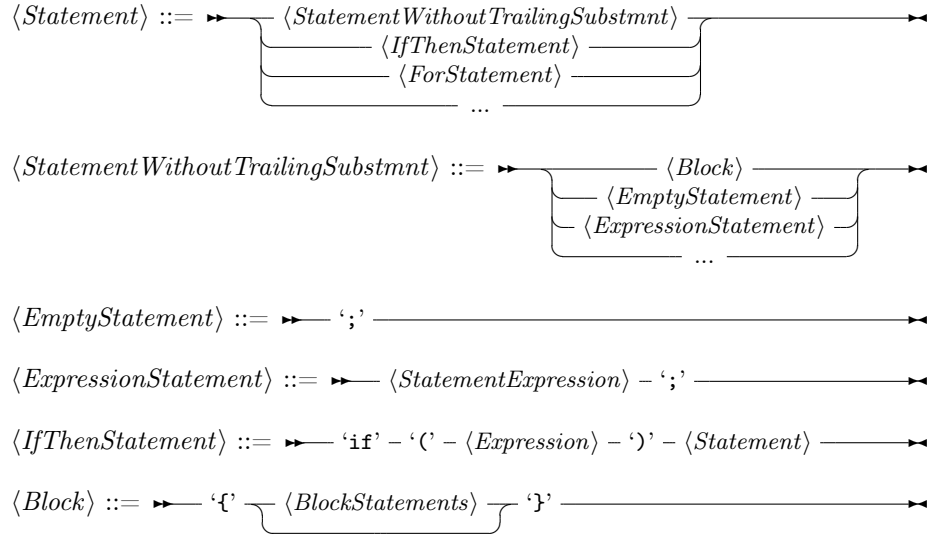


Listing 15: Part of the section based language ESV from the Spoofax Language Workbench [66].

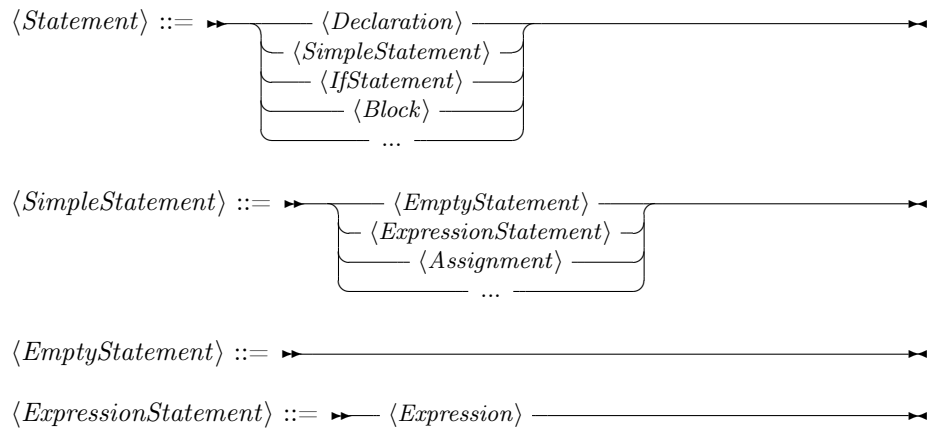
3.4 Statement Endings

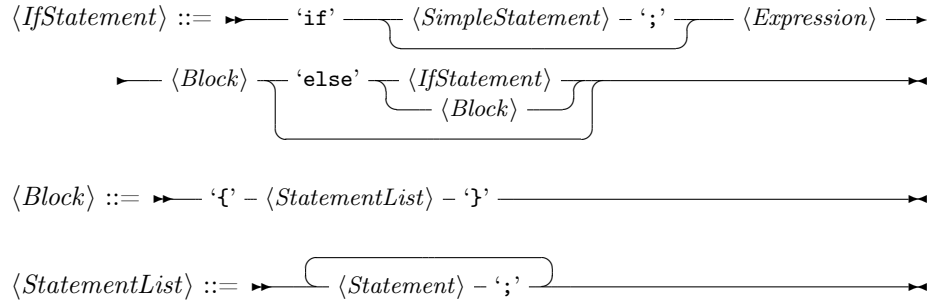
Most programming languages have a concept of statements, which denote a sequence of operations to be performed on the current program state. In a lot of those languages such as C++ [42], Java [41] (see Listing 16), Go [90] (see

Listing 17), Rust [78], etc. they are separated with a semicolon (;). Other languages such as Python [91] and SHell script do not require separation with semicolons, but allow them anyway to put multiple statements on a single line. Also in Python, statements can continue on the next line if the line ends with a backslash ('\') or if there are unclosed parentheses, brackets, or braces. Finally, there are languages where semicolons are required but can be omitted in the source file. For example, both JavaScript [28] and Go [90] insert semicolon tokens under some circumstances that would otherwise cause a parse error.



Listing 16: Partial grammar for Java 16 statements [41] (chapter 14), which puts the separator (;) in the individual statements (e.g. *EmptyStatement*, *ExpressionStatement*) that require it.





Listing 17: Partial grammar for Go statements [90], which puts the separator (';') in the *StatementList* production.

3.5 Comments

As written by Van Tassel in [93] (see Section 2.3.6), there are various sorts of comment types. Both end-of-line comments and block comments are easily implemented in most parsers as a regular expression in the lexical analysis phase (see Section 2.2). For mega comments, some form of extended regular expressions that allow recursion are the preferred way to implement them in the lexer, though this requires the lexer supporting this. Lastly positional comments are difficult because they work on a per-line basis, while most lexers and lexerless parsers work on a per-character basis. Implementing them would require either a pre-processing step or a custom lexer, and as such is difficult to achieve. Thankfully, these sorts of comments only exist in languages as a relic of the limitations of the past, and as such not supporting them is not a big issue.

3.6 Unicode

Unicode (see Section 2.1) support is an important requirement for us due to the interoperability it provides between programs. However, there are still languages that only support ASCII or other character sets as input, or that use basic ASCII characters for most of their language but allow using Unicode characters as well.

The following are important aspects we look at for Unicode support:

- Support for reading files in different Unicode encodings, but primarily UTF-8.
- Support for using Unicode characters in the grammar specification language, both in rule and token names, and in the actual string representation of tokens.
- Support for rendering Unicode characters, and properly taking into account characters which get encoded into multiple bytes or words based on the memory representation. An example would be to not allow putting the caret halfway through a character.

3.7 Language Dialects or Variations

Some languages have multiple ways to be written down, or change between releases. For example a programming language such as `Python` (version 2, 3 / 2.7, 3.6, 3.7, 3.8, etc.) or `Java` (version 5, 6, 7, etc.) evolves and adds new features, which require new syntax to be added. Another example would be `ALGOL`, which has a lot of different implementations and tweaks to the syntax, each to suit a use case such as I/O capability. `Prompto` [94] is language and framework which has three different dialects all in the same version of the software, with all of them being convertible between each other, allowing the same model to be specified each of the dialects without loss of data.

Important to us is the ability to select dialects at least per-extension, and preferably even per-file. Another nice thing to have is the ability to change the language/dialect on the fly without re-opening the file.

4 Tools Overview

In the following sections, we describe the tools we decided to look at for our comparison and give some background information about them. Table 3 gives an overview of the most important properties of each tool.

Name	Languages	Parser	Designation
Spoofax	Java Stratego	GSLR	Language Workbench
Xtext	Java	LL(*)	Language Workbench Software Framework
JetBrains MPS	Java	N/A	Language Workbench
IntelliJ IDEA	Java	N/A	Integrated Development Environment (IDE)
Atom	JavaScript CoffeeScript	N/A	Source Code Editor
Visual Studio Code	JavaScript TypeScript	N/A	Source Code Editor
Python PLY	Python	LALR(1)	Library
Python Lark	Python	LALR(1) Earley	Library

Table 3: An overview of all the tools looked at in this comparison.

4.1 Spoofax

Spoofax is an open-source language workbench for writing Domain Specific Languages maintained by the TU Delft Software Engineering Research Group [57], [66]. It is written in Java and uses an Eclipse based IDE to aid in the development of languages, at the time of writing an IntelliJ IDEA plugin is also under development [63].

Spoofax makes use of a Scannerless Generalized LR (SGLR) parser [95], which does not use a lexing step during parsing of files.

4.2 Xtext

Xtext [26] and Xtend [21] are an open-source framework for writing Domain Specific Language, and are maintained by the Eclipse Foundation. They are written entirely in Java and have an Eclipse based IDE to aid in developing languages, an IntelliJ IDEA plugin exists [20] but has not been updated since 2016 at the time of writing.

Xtext uses ANTLR v3 internally as its parser generator, which generates parser that can parse LL(*) grammars [76].

4.3 JetBrains MPS

MPS (Meta Programming System) is an open-source IDE for creating and working in Domain Specific Language and is developed by JetBrains [53]. It is written in both Java and a “Base Language” which is a modeled version of Java 6. Its key feature is that it works with a projectional editor which edits the Abstract Syntax Tree (AST) directly, rather than using the classic textual editor which needs a grammar specification and a parser to convert it to an AST.

When writing a Domain Specific Language with MPS, the developer defines the abstract syntax, constraints, concrete (textual) syntax, and both translational semantics and operational semantics. Additionally the developer can write ‘intentions’, which provide the user of the language with quick actions to perform to the model, refactoring operations, migration operations (for when the language gets updated), and tests for verifying the language definition.

4.4 IntelliJ IDEA

IntelliJ IDEA is a semi open-source Integrated Development Environment (IDE) for writing Java libraries and programs [48], and features a variety of plugins which add support for other languages. It is written entirely in Java and features two editions: a free “Community Edition” and a paid for “Ultimate Edition”. The paid edition features are built on top of the free edition, and include closed-source code as opposed to the free edition’s open-source code

4.5 Atom

Atom is an open-source source code editor developed by GitHub [32], it features a variety of community made plugins which add editor features or support for other languages. It is based on Electron, which is a framework for developing applications with HTML, CSS, and JavaScript. Atom itself is written in JavaScript and the CoffeeScript language [3] which compiles to JavaScript.

4.6 Visual Studio Code

Visual Studio Code [73] is a semi open-source source code editor developed by Microsoft, it features a variety of plugins made by Microsoft and the community to add various features and support for other languages. Like Atom, it is based on the Electron framework, which is why it is written in JavaScript and TypeScript (which compiles to JavaScript).

4.7 Python Lex-Yacc (PLY)

PLY [7] is an open-source Python 2 & 3 implementation of the lexer and parser generators Lex and Yacc. It makes use of an LALR(1) algorithm to parse input.

At the time of writing, a new version of PLY is being developed by the same developer under the name of SLY [6], which makes use of more modern versions of Python and drops support for Python 2.

4.8 Python Lark

Lark [81] is an open-source Python 2 & 3 parsing toolkit. It supports both an LALR(1) parser and an Earley [18] parser.

It is the successor of PlyPlus [80], which is a parser library built on top of and extending PLY.

5 Criteria

5.1 Language Variability Support

In Section 3 we described our language variability model which represents some common variation points in languages. The tool we pick should support as many variations of this model as possible, as such we will check which parts are (partially) supported or not.

5.2 Multi-language and Dynamic Parsing

As we eluded to at the beginning, our goal is to have a parsing toolchain that supports multiple grammars throughout a source document, either by changing the parser grammar or by deferring to a different parser. This is similar to

‘dynamic parsers’ as defined by Cabasino et al. [14] and Boullier [10], but instead involves completely changing the grammar, and being able to go back to the previous grammar, as opposed to modifying it without being able to revert. We’ll refer to this as *multi-language parsing* for the purposes of our comparison.

Additionally, we would like to be able to define languages while parsing, and afterwards making use of those languages, all in the same source document. This is for example similar to defining a formalism in an interactive terminal, and then performing a command to start using this formalism, but with grammars instead. We’ll refer to this as *dynamic language definitions* in our comparison.

5.3 Maintenance

Under this criterion we look at how maintained a specific tool is: do its developers still actively commit code to it to either remove bugs, add features, or modernize the codebase? Or has it been (partially) abandoned?

The importance here lies in the fact that if a tool has been abandoned or is not receiving any more bug fixes, this would incur a cost on us to either fork and fix bugs ourselves, or to replace it with a different tool/library and adapt our codebase to it.

5.4 Documentation

This criterion looks at how well each tool is documented, which is important for us to be able to properly use it. APIs provided by the tool need to be annotated such that as developers of our own library or tool we know what it expects, what it returns, etc. Any (meta-)languages provided by the tool itself should also be properly explained, for example the concrete syntax definition language (usually some form of **E**xtended **B**ackus-**N**aur **F**orm or EBNF) should have documentation on how to properly define rules and symbols.

We will focus on official manuals, guides and examples made available by the creators and/or maintainers of each tool.

5.5 Setup

Under this criterion we look at what steps are required to get started using each tool, both as a language designer and as a user of a language made with the tool. If installation is easy, it makes our project easier to use by other users and as such increases user experience.

For the best user experience, the user should be required to install as little extra dependencies as possible, preferably none. If the user were to need to install extra dependencies this should be made easy, such as the tool taking care of downloading and possibly even installing them.

Plugins (or extensions, packages) for these tools are also an important way of distributing functionality, and as such should be easy to download and install.

Preferably this would be possible inside the tool/application itself, which would configure everything on its own in order to make use of the plugin.

5.6 Support

This criterion looks at whether there is somewhere to get support for the tool, such as a forum, discussion board or issue list. These places normally have experienced users and developers of the tool that help new users getting started or resolving issues they are having with the tool.

5.7 Editor Features

Because our end goal is to have an Integrated Development Environment (IDE) that allows us to write Domain Specific Languages with arbitrary embedding of other languages, this criterion focuses on the features that make up an IDE: if they are supported or easy to add.

The following is a non-exhaustive list of IDE features:

- Syntax highlighting of semantic components.
- Completion of constructs, symbols, etc.
- Model transformations such as refactoring, renaming, etc. but also transformations of the parse tree / abstract syntax tree.
- Informative error reporting for parsing errors, semantic errors, etc.
- Linting support to detect styling issues, smells, etc.
- Jumping to declaration and references of symbols.
- Debugging operational semantics: pausing execution, stepping through, reading values in memory, etc.
- Structured overviews of a document detailing declared components.

5.8 Language Server Protocol

The Language Server Protocol (LSP) (see Section 2.4) is a protocol for editors and IDEs to make use of a language server which provides editor features. Being able to provide our tool as a language server would be a great advantage, as this would mean users of our tool would be able to choose the editing environment of their choice. With this in mind, we will look at each tool with the idea of either (1) creating a language server using a library or language workbench, or (2) for creating a language client such as for an editor or IDE.

6 Comparison

In this section we will go over each of our criteria as defined in Section 5. In Table 5 we provide an overview of this comparison to get a quick glance at the overall outcome, using Table 4 as a reference for our scores.

Score	Meaning
— — — —	Extremely terrible support
— — —	Terrible support
— —	Very bad support
—	Bad support
+	Alright support
++	Good enough support
+++	Good support
++++	Exceptionally well support
✓	Supports feature
✗	Does not support feature

Table 4: Overview of several symbols used in our comparison tables.

	Spoofox	Xtext	Jetbrains MPS	IntelliJ IDEA	Atom	Visual Studio Code	Python Lex-Yacc	Python Lark
Language Variability Support ^a	+	++	+	+	++	++	+++	+++
Dynamic language definitions	+	--	-	+	+++	+	++	+++
Multi-language parsing	--	--	++++	? ^b	? ^b	? ^b	+++	+++
Maintenance	++	++	+++	+++	-	++++	+	++
Documentation	++	++	++++	+++	++	++++	+++	++
Setup	+++	+++	+++	+++	+++	+++	+++	+++
Support	+	+++	+++	+++	++	+++	++	++
Editor Features	+++	+++	+++	++	+ ^c	+++	N/A	N/A
Language Server Protocol	--	+++	N/A	+ ^d	++	++++	+ ^d	++ ^d
Overall	++	++	++	+++	+++	+++	++	+++

^a See Table 6 for a detailed overview of this criterion per tool.

^b Support depends on used third-party libraries or toolchains.

^c Additional features provided through third-party extensions or plugins.

^d Support provided through third-party extensions, plugins, or libraries.

Table 5: Overview of the comparison result. See Table 4 for an overview of meanings for each score.

6.1 Language Variability Support

In this section we compare each tool from section Section 4 to our language variability model defined in Section 3. Table 6 provides an overview of this comparison, including scores.

	Spoofox	Xtext	Jetbrains MPS	IntelliJ IDEA	Atom	Visual Studio Code	Python Lex-Yacc	Python Lark
Textual languages	✓	✓	✓	✓	✓	✓	✓	✓
Visual languages	✗	✗	✗	✓	✓	✓	N/A	N/A
Parser Class	SGLR	LL(*)	N/A	? ^a	? ^a	? ^a	LALR(1)	LALR(1) Earley
Lexer step	✗	✓	N/A	? ^a	? ^a	? ^a	✓	Optional ^b
Indentation sensitive block structures	++++	+++	N/A	? ^a	? ^a	? ^a	+	++
Explicit Statement Endings	++	++	N/A	? ^a	? ^a	? ^a	+++	+++
Implicit Statement Endings	++	++	N/A	? ^a	? ^a	? ^a	+++	+++
Comments	+	++	N/A	? ^a	? ^a	? ^a	++	+++
Unicode Encodings	--	++	+	++++	+++	+++	++++	++++
Unicode Specification	---	+	+	? ^a	? ^a	? ^a	++	++
Unicode Editing	+++	+++	+	+++	+++	+++	N/A	N/A
Dialects Per File	---	+++	+++	--	++++	++++	N/A	N/A
Dialects Per Extension	+++	+++	N/A	+++	+++	+++	N/A	N/A
Overall	+	++	+	+	++	++	+++	+++

^a Support depends on used third-party libraries or toolchains.

^b Can be configured without lexer when using the Earley parser algorithm.

Table 6: An overview of the language variability support. See Table 4 for an overview of meanings for each score.

Spoofox

Visual or Textual Textual: ✓ Visual: ✗

Spoofox is designed for writing textual languages. As such it does not support visual languages.

Grammar classification Parser class: SGLR Lexer step: ✗

Spoofox makes use of a Scannerless Generalized LR parser [95], which does not use a lexing step.

Block structures Indentation sensitive: ++++

Spoofox supports free-form and section based languages by default due to their simple nature. It also offers support for layout sensitive languages [29], which includes indentation sensitive languages and more. It does this by allowing layout constraints to be defined in the grammar definition if configured to support this.

Statement endings Explicit: ++ Implicit: ++

Due to the simple nature of explicit statement endings, this variant is easy to implement. Optional explicit statement endings such as in Go however are not possible due to not being able to configure token injections on missing tokens. For implicit statement endings, it's possible to write a grammar that uses line endings as statement separator, allows these to be escaped, and also allows multiple statements on a single line with the user of a separator character. However, there is no way to tell the parser to start ignoring end of lines under circumstances such as in Python when there are unclosed parentheses, because these are global options to the parser.

Comments +

End-of-line comments and block comments are easily implemented as terminals. Positional and mega comments are not supported by the grammar definition, and no support exists for custom lexers due to SGLR parsers not having a lexing step.

Unicode Encodings: -- Specification: -- -- Editing: +++

At the time of writing, though the editors used by Spoofox (Eclipse, IntelliJ IDEA) support Unicode characters, Spoofox itself does not support them. The underlying parser has added support for Unicode recently, but this has not yet been propagated to the meta-languages used to engineer a language ¹.

Language dialects or variations Per File: -- -- Per Extension: +++

Spoofox does not support selecting different variants or dialects except by assigning a different file extension for each variant.

¹<https://github.com/metaborg/jsgr/pull/72>

Xtext

Visual or Textual Textual: ✓ Visual: ✗

Xtext is designed for writing textual languages. As such it does not support visual languages.

Grammar classification Parser class: LL(*) Lexer step: ✓

Xtext makes use of ANTLR3 under the hood, which uses an LL(*) parser [76]: a **Left**-to-right **Leftmost** derivation parser with infinite lookahead, and includes a lexing step.

Block structures Indentation sensitive: +++

Xtext supports free-form and section based languages by default due to their simple nature. It also supports indentation sensitive languages if configured, which it does by outputting `indent` and `dedent` tokens.

Statement endings Explicit: ++ Implicit: ++

Due to the simple nature of explicit statement endings, this variant is easy to implement. Optional explicit statement endings such as in `Go` can be implemented by providing a custom lexer, which can insert tokens if needed. For implicit statement endings, it's possible to write a grammar that uses line endings as statement separator, allows these to be escaped, and also allows multiple statements on a single line with the user of a separator character. However, there is no way to tell the parser to start ignoring end of lines under circumstances such as in `Python` when there are unclosed parentheses, because these are global options to the parser.

Comments ++

End-of-line comments and block comments are easily implemented as terminals. Neither positional nor mega comments are supported by the grammar definition, and due to abstraction and lack of access to internals it is also not possible to implement these using a custom lexer.

Unicode Encodings: ++ Specification: + Editing: +++

Xtext only supports UTF-8 as an encoding for source files. The meta-grammar supports Unicode characters directly and indirectly via Java escape sequences, but does not support them in rule and terminal names. Xtext relies on the Eclipse Platform to build its editor windows on, as such the Xtext editor properly supports Unicode glyph rendering and does not split characters which are represented by multiple bytes.

Language dialects or variations Per File: +++ Per Extension: +++

Xtext supports selecting different language variants to open a file in (named editors), but this is a side effect of it making use of the Eclipse Platform and would not necessarily work in other editors.

JetBrains MPS

Visual or Textual Textual: ✓ Visual: ✗

MPS supports both textual languages and graph based visual languages. However these graph based visual languages have been deprecated in support for an external plugin [46].

Grammar classification Parser class: N/A Lexer step: N/A

Because MPS does not parse actual text, we did not grade it for this variability point.

Block structures Indentation sensitive: N/A

Because MPS does not parse actual text, we did not grade it for this variability point.

Statement endings Explicit: N/A Implicit: N/A

Because MPS does not parse actual text, we did not grade it for this variability point.

Comments N/A

Because MPS does not parse actual text, we did not grade it for this variability point.

Unicode Encodings: + Specification: + Editing: +

MPS supports Unicode partially. However the reflective editor only properly works with characters on the Basic Multilingual Plane (characters with code points from U+0000 to U+FFFF). The meta-language supports Unicode characters directly the same way as all other languages, though named concepts are restricted to regular ASCII names. Characters outside this range need to be split up into multiple 16-bit words internally, and the editor allows the caret to be put between two 16-bit words that make up a single character. A bug report has been filed to address this issue, and it will likely be fixed in the future.

Language dialects or variations Per File: +++ Per Extension: N/A

MPS only supports a single default visual editor for a concept. However multiple dialects can be implemented by adding a property to the root concept and making it editable, which allows quick changing of the dialect.

IntelliJ IDEA

Visual or Textual Textual: ✓ Visual: ✓

IntelliJ IDEA primarily supports textual languages. It is possible to create a custom editor window, which can be used to add a visual editor.

Grammar classification Parser class: N/A Lexer step: N/A

IntelliJ IDEA itself does not specify which parser to use, it is up to developers of plugins to decide on which parser to use. We note however that the parser used in the official tutorial makes use of a Parsing Expression Grammar (PEG) parser.

Block structures Indentation sensitive: N/A

Because IntelliJ IDEA does not provide a specific parsing technology, we did not grade it for this variability point.

Statement endings Explicit: N/A Implicit: N/A

Because IntelliJ IDEA does not provide a specific parsing technology, we did not grade it for this variability point.

Comments N/A

Because IntelliJ IDEA does not provide a specific parsing technology, we did not grade it for this variability point.

Unicode Encodings: ++++ Specification: N/A Editing: +++

IntelliJ IDEA supports the following Unicode encoding formats: UTF-8, UTF-16 BE, UTF-16 LE, UTF-32 BE, UTF-32 LE. Supported glyphs depend on the system fonts installed and multi-byte or multi-word characters do not get split in half. Because IntelliJ IDEA does not provide a specific parsing technology, we did not grade this variability point for parsing/syntax definition.

Language dialects or variations Per File: -- Per Extension: +++

IntelliJ IDEA does not allow changing the language of an open document or opening a document in a specific language without configuring a file association. Extension or file pattern associations however can be configured.

Atom

Visual or Textual Textual: ✓ Visual: ✓

Atom primarily supports textual languages. There exists an API to display custom content in the editor window, and users have created packages that add diagrams.net (formerly draw.io) support to Atom ².

Grammar classification Parser class: N/A Lexer step: N/A

Atom does not specify which parser to use, it is up to developers of plugins to decide on which parser to use. We note however that the suggested parsing technology “tree-sitter” makes use of a Generalized LR (GLR) parser with a lexing step. Syntax highlighting is also supported using TextMate grammar files, which uses regular expressions to find patterns in files but does not actually parse them.

²<https://atom.io/packages/atom-drawio>

Block structures Indentation sensitive: N/A

Because Atom does not provide a specific parsing technology, we did not grade it for this variability point.

Statement endings Explicit: N/A Implicit: N/A

Because Atom does not provide a specific parsing technology, we did not grade it for this variability point.

Comments N/A

Because Atom does not provide a specific parsing technology, we did not grade it for this variability point.

Unicode Encodings: +++ Specification: N/A Editing: +++

Atom supports the following Unicode encoding formats: UTF-8, UTF-16 BE, UTF-16 LE. Supported glyphs depend on the system fonts installed and multi-byte or multi-word characters do not get split in half. Because Atom does not provide a specific parsing technology, we did not grade this variability point for parsing/syntax definition.

Language dialects or variations Per File: ++++ Per Extension: +++

Atom allows changing the language of an open document without re-opening it, though this association does not last. It also allows associating by file extensions/patterns or exact filenames, and for package authors to register language variants.

Visual Studio Code

Visual or Textual Textual: ✓ Visual: ✓

VSCoDe supports textual languages and custom editors. Users have created extensions that add diagrams.net (formerly draw.io) support ³.

Grammar classification Parser class: N/A Lexer step: N/A

VSCoDe does not provide any facilities to parse on its own, extension developers are instead expected to implement this on their own (with libraries, etc.). Syntax highlighting is also supported using TextMate grammar files, which uses regular expressions to find patterns in files but does not actually parse them.

Block structures Indentation sensitive: N/A

Because VSCoDe does not provide a specific parsing technology, we did not grade it for this variability point.

Statement endings Explicit: N/A Implicit: N/A

Because VSCoDe does not provide a specific parsing technology, we did not grade it for this variability point.

³<https://marketplace.visualstudio.com/items?itemName=hediet.vscode-drawio>

Comments N/A

Because VSCode does not provide a specific parsing technology, we did not grade it for this variability point.

Unicode Encodings: +++ Specification: N/A Editing: +++

VSCode supports the following Unicode encoding formats: UTF-8, UTF-16 BE, UTF-16 LE, it also allows automatically detecting which encoding is used, though it will default to UTF-8. Supported glyphs depend on the system fonts installed and multi-byte or multi-word characters do not get split in half. Because Atom does not provide a specific parsing technology, we did not grade this variability point for parsing/syntax definition.

Language dialects or variations Per File: ++++ Per Extension: +++

VSCode allows changing the language of an open document without re-opening it, though this association does not last. It also allows associating by file extensions/patterns or exact filenames.

Python Lex-Yacc**Visual or Textual** Textual: ✓ Visual: N/A

As PLY is a parser framework, it only supports textual languages.

Grammar classification Parser class: LALR(1) Lexer step: ✓

PLY uses an LALR(1) parser with a lexing step.

Block structures Indentation sensitive: +

PLY supports free-form and section based languages by default due to their simple nature. In order to support indentation sensitive languages, a custom lexer needs to be used.

Statement endings Explicit: +++ Implicit: +++

Due to the simple nature of explicit statement endings, this variant is easy to implement. Optional explicit statement endings such as in Go can be implemented by using a custom lexer. For implicit statement endings, it's possible to write a grammar that uses line endings as statement separator, allows these to be escaped, and also allows multiple statements on a single line with the user of a separator character. A custom lexer implementation is required to allow suspending implicit statement ends such as when there are unclosed parentheses in Python.

Comments ++

End-of-line comments and block comments are easily implemented as terminals. Positional comments can be implemented by either performing a pre-processing step before lexing occurs, replacing commented lines but losing this information in the token stream, or by providing a custom lexer which would allow these

tokens to remain in the token stream. For mega comments, we can write a token specification that checks for nesting and applies it recursively, updating the lexer state upon matching.

Unicode Encodings: ++++ Specification: ++ Editing: N/A

Lark supports all Unicode encoding formats supported by Python: UTF-8, UTF-16 BE, UTF-16 LE, UTF-32 BE, UTF-32 LE; selection of these encoding formats needs to be done by the programmer however. The meta-grammar supports Unicode characters directly and indirectly via Python escape sequences in patterns, but does not support them in rule and terminal names. Because PLY is not an editor, we did not grade it with regards to editor support for Unicode.

Language dialects or variations Per File: N/A Per Extension: N/A

Because PLY is a parser framework, it can parse multiple languages and dialects, but it is up to the user to define them and select the appropriate parser. For this reason we decided to not grade Lark on this variability point.

Python Lark

Visual or Textual Textual: ✓ Visual: N/A

As Lark is a parser framework, it only supports textual languages.

Grammar classification Parser class: LALR(1), Earley Lexer step: Optional
Lark implements three different kinds of parser:

- an LALR(1) parser with a lexing step;
- an Earley parser [18], which can be configured to work with or without a lexing step; and
- a CYK parser, which is deprecated/unsupported and therefore not included in the comparison.

Block structures Indentation sensitive: ++

Lark supports free-form and section based languages by default due to their simple nature. In order to support indentation sensitive languages, either a post-lexing step needs to be configured on the parser, or a custom lexer needs to be provided.

Statement endings Explicit: +++ Implicit: +++

Due to the simple nature of explicit statement endings, this variant is easy to implement. Optional explicit statement endings such as in Go can be implemented by using a custom lexer. For implicit statement endings, it's possible to write a grammar that uses line endings as statement separator, allows these to be escaped, and also allows multiple statements on a single line with the user

of a separator character. A custom lexer implementation is required to allow suspending implicit statement ends such as for example Python does when there are unclosed parentheses.

Comments +++

End-of-line comments and block comments are easily implemented as terminals. Positional comments can be implemented by either performing a pre-processing step before lexing occurs, replacing commented lines but losing this information in the token stream, or by providing a custom lexer which would allow these tokens to remain in the token stream. For mega comments, using the `regex` module allows defining recursive regular expressions to achieve them. An example regular expression for comments that start with `/*` and end with `*/` is given as such:

```
/\*(?: (?! (?: /* | \*/ ) ) . | (?: R) ) * \*/
```

Alternatively a custom lexer can be provided to implement mega comments.

Unicode Encodings: ++++ Specification: ++ Editing: N/A

Lark supports all Unicode encoding formats supported by Python: UTF-8, UTF-16 BE, UTF-16 LE, UTF-32 BE, UTF-32 LE; selection of these encoding formats needs to be done by the programmer however. The meta-grammar supports Unicode characters directly and indirectly via Python escape sequences in patterns, but does not support them in rule and terminal names. Because Lark is not an editor, we did not grade it with regards to editor support for Unicode.

Language dialects or variations Per File: N/A Per Extension: N/A

Because Lark is a parser framework, it can parse multiple languages and dialects, but it is up to the user to define them and select the appropriate parser. For this reason we decided to not grade Lark on this variability point.

6.2 Multi-language and Dynamic Parsing

Spoofax

Dynamic language definitions Spoofax is able to reload languages upon building them, without requiring an IDE restart or reload. However this process does not happen automatically, and it is not clear to us how parsers for new language definitions are created.

Multi-language parsing Spoofax does not support multi-language parsing due to us not being able to change the parser mid-parse. Included in the workbench is the **SPoofax Testing language (SPT)**, which is an example of a language that contains embedded fragments. While this is close to our goal, the parsing here happens as a post-processing step rather than during parsing itself and as such does not qualify as multi-language parsing.

Xtext

Dynamic language definitions In order to use a language definition, it needs to be compiled from its original specification, this also requires running a Java compiler. Theoretically it may be possible to dynamically define a language and load it in the current process, but this would at the very least require adding a class loader, including the Xtext compiler, and a the user having a JDK installed. In conclusion, it should be possible, but very complicated.

Additionally, when creating a language and testing it in a development IDE, the IDE needs to be restarted if the language is changed.

Multi-language parsing Xtext does not support changing the parser mid-parse, and as such does not support multi-language parsing.

JetBrains MPS

Dynamic language definitions In order to use a language and its editors in MPS it needs to be (re-)generated, but it does not require an IDE restart or reload. This makes it impossible to add new languages or formalisms dynamically, without re-implementing everything provided by MPS already.

Multi-language parsing While MPS does not actually parse languages, due to the fact that one can reuse (parts of) existing languages defined with MPS, working with multiple languages in a single source file is possible.

IntelliJ IDEA

Dynamic language definitions IntelliJ IDEA provides both declarative and runtime declaration of "File Types", though at the time of writing the facilities for runtime declarations are deprecated and slated for removal in favor of the declarative method [44].

Multi-language parsing Because IntelliJ IDEA itself does not specify which parser to use, we do not evaluate this criterion.

Atom

Dynamic language definitions Atom provides access to the list of currently loaded grammars via its `GrammarRegistry` [33], which can dynamically be added to and removed from by packages. Grammars added this way need to be read from the filesystem however, so this requires writing a temporary file.

Multi-language parsing Because Atom itself does not specify which parser to use, we do not evaluate this criterion.

Visual Studio Code

Dynamic language definitions Visual Studio Code currently does not provide a way to register languages at runtime. The underlying system used by the editor Monaco [70] supports dynamically adding languages⁴, but this is not exposed to extensions⁵ and instead is done declaratively in an extension’s package specification.

Multi-language parsing Because VSCode itself does not specify which parser to use, we do not evaluate this criterion.

Python Lex-Yacc

Dynamic language definitions Because PLY is an API for creating lexers and parsers from a specification, defining languages is done relatively easily. There exist some oddities however with the way this is done:

- When looking at examples and the documentation, we noticed that parser and lexer configurations are supposed to be defined in source files either directly in the file global namespace, or as members of a class. They can also be defined either separately or together.
- While not explicitly documented, because of the above if you pass a regular dictionary, a missing attribute error gets raised. Similarly, dynamically defined classes also have this issue due to not being associated with a source file.
- These missing attributes are accessed even if they are not necessary and should be used only when optimizing.

The result is that we cannot dynamically define new languages with PLY, unless we write them to a temporary file first.

Multi-language parsing Due to the large freedom of modifying PLY thanks to it being written in Python, implementing multi-language parsing should be possible, however we would need to implement this ourselves.

Python Lark

Dynamic language definitions Lark is an API for creating a lexer-parser pipeline based on a textual grammar specification. Therefore defining a grammar dynamically simply requires creating a new string representation in the right format. Alternatively the option exists for passing in a grammar definition that is already been parsed, but this is less documented and probably more likely for internal use.

⁴<https://github.com/microsoft/vscode/blob/e1f0f8f51390dea5df9096718fb6b647ed5a9534/src/vs/editor/standalone/browser/standaloneLanguages.ts#L581>

⁵<https://github.com/microsoft/vscode/blob/94c9ea46838a9a619aeafb7e8afd1170c967bb55/src/vs/workbench/api/common/extHost.api.impl.ts#L1108>

Multi-language parsing Like PLY, Lark is written in Python and this gives us great freedom for modifying its behaviour, as such implementing multi-language parsing is possible though again through implementing it ourselves.

6.3 Maintenance

We performed the following evaluations in June 2021. The accompanying contributions graphs have been recorded in August 2021 from each project’s respective GitHub repository where possible, and have only been recorded to serve as a reference frame for this document at the time of writing. The graphs represent contributions on the main branch of each repository, and exclude merge commits and bot accounts.

For an up-to-date version of these graphs one can navigate to these repositories themselves, go to the ‘Insights’ tab, and select ‘Contributors’ (assuming this will stay available for the foreseeable future).

Spoofax

Spoofax is split into multiple git repositories: the main runtime repository (`‘spoofax’`), IDE specific repositories (`‘spoofax-eclipse’` and `‘spoofax-intellij’`), and core language development language repositories.

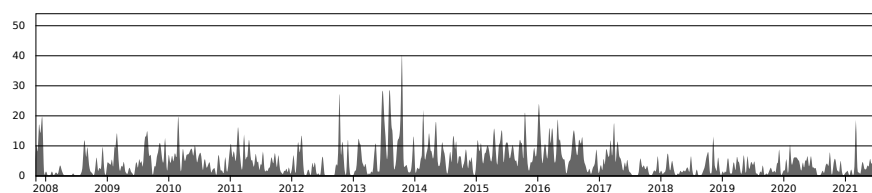


Figure 7: Contributions graph for the `metaborg/spoofax` repository on GitHub. See <https://github.com/metaborg/spoofax>.

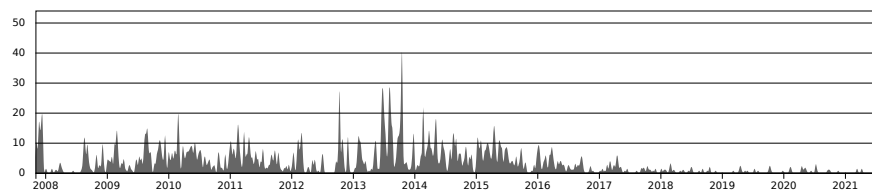


Figure 8: Contributions graph for the `metaborg/spoofax-eclipse` repository on GitHub. See <https://github.com/metaborg/spoofax-eclipse>.

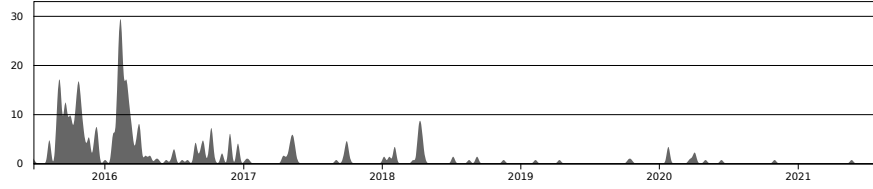


Figure 9: Contributions graph for the `metaborg/spoofax-intellij` repository on GitHub. See <https://github.com/metaborg/spoofax-intellij>.

If we look at the contributors graph for the main runtime repository (see Figure 7), we can see that commit activity has been about the same throughout its history.

Looking at the IDE integration for Spoofax, we can see that the Eclipse repository (see Figure 8) has had reduced activity since 2017. The IntelliJ integration (see Figure 9) is less old, only starting halfway 2015, but also only receiving most of its activity until 2017 after which it has been mostly silent.

We also take a quick look at the repositories for core components of the Spoofax language workbench:

SDF2/SDF3 (`'spoofax-sdf'`) See Figure 10. This repository has had continuous activity since its start until now (with a small dip in 2018).

NaBL/Statix (`'spoofax-nabl'`) See Figure 11. This repository has been active since its start, but starting halfway 2016 its activity increased significantly, staying at the same pace even still.

Stratego1/Stratego2 (`'spoofax-stratego'`) See Figure 12. Activity in this repository has seen an increase since halfway 2018, before that there was only limited activity.

ESV (`'spoofax-esv'`) See Figure 13. This component has received little activity (seeing only at most 5 commits throughout a month), though this activity has been consistent throughout time. Activity has decreased since circa 2017 though.

SPT (`'spoofax-spt'`) See Figure 14. Aside from a dip in activity in 2012 and 2019, this component has had about the same activity throughout its lifetime.

We conclude that Spoofax is being actively developed by its developers.

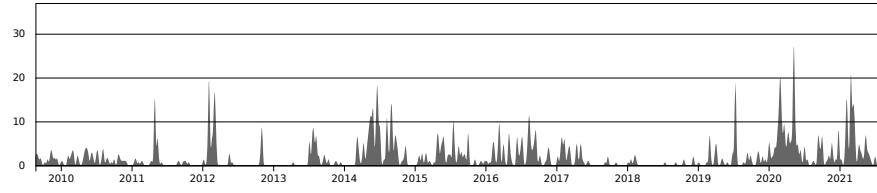


Figure 10: Contributions graph for the `metaborg/spoofax-sdf` repository on GitHub. See <https://github.com/metaborg/spoofax-sdf>.

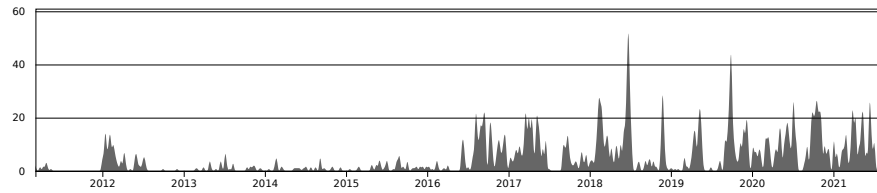


Figure 11: Contributions graph for the `metaborg/spoofax-nabl` repository on GitHub. See <https://github.com/metaborg/spoofax-nabl>.

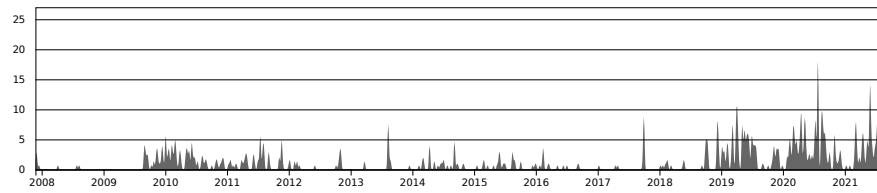


Figure 12: Contributions graph for the `metaborg/spoofax-stratego` repository on GitHub. See <https://github.com/metaborg/spoofax-stratego>.

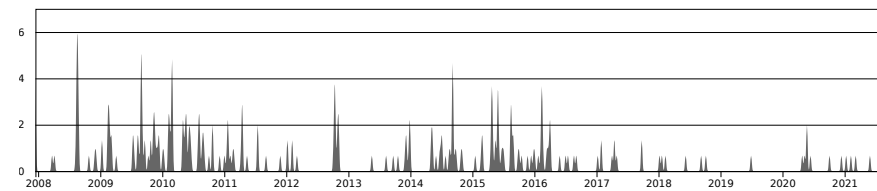


Figure 13: Contributions graph for the `metaborg/spoofax-esv` repository on GitHub. See <https://github.com/metaborg/spoofax-esv>.

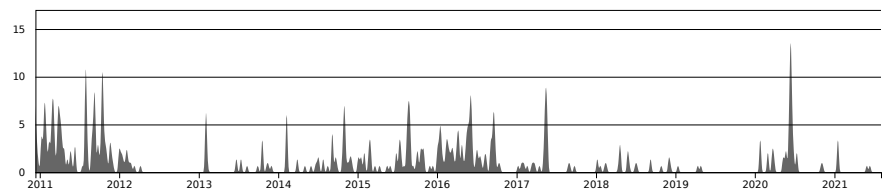


Figure 14: Contributions graph for the `metaborg/spoofax-spt` repository on GitHub. See <https://github.com/metaborg/spoofax-spt>.

Xtext

Xtext is split over multiple git repositories:

xtext See Figure 15. The main repository until 2016 after which it was split up into several repositories, which is now used to host the online documentation.

xtext-core See Figure 16. Contains the main/core framework of Xtext.

xtext-lib See Figure 17. This repository contains the standard library for Xbase languages, which allows interaction with Java.

xtext-extras See Figure 18 on page 50. Contains addons such as Xbase

xtext-xtend See Figure 19 on page 50. This repository contains the code for Xtend, which allows writing Java-like programs usually based on some source model written by the user.

xtext-eclipse See Figure 20 on page 51. Contains all code related to the Eclipse IDE integration.

xtext-idea See Figure 21 on page 51. Contains all code related to the IntelliJ IDEA integration. Support for this has been dropped since 2019.

xtext-web See Figure 22 on page 51. Contains all code related to the web editor for Xtext.

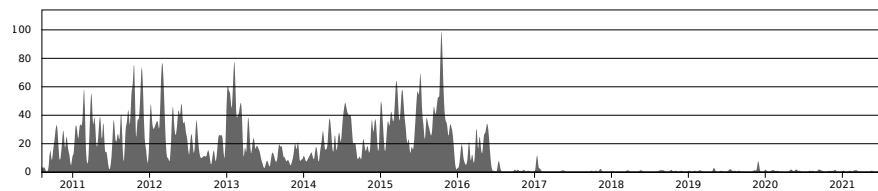


Figure 15: Contributions graph for the `eclipse/xtext` repository on GitHub. See <https://github.com/eclipse/xtext>.

Looking at all the graphs for these repositories, we can say that activity has remained roughly the same throughout from the start in 2010 up until today. We note that there is a significant spike around 2015-2016, which is when the main repository got split into multiple smaller ones.

In conclusion, Xtext has been and is continuing to be actively maintained, with exception of the IntelliJ IDEA integration.

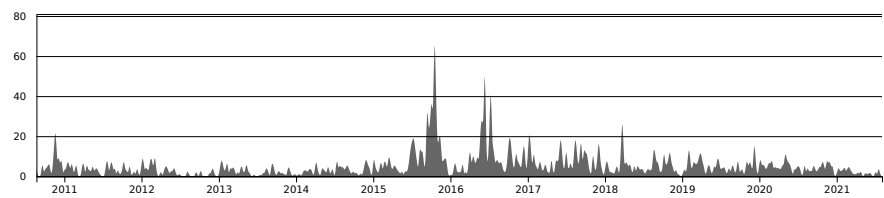


Figure 16: Contributions graph for the `eclipse/xttext-core` repository on GitHub. See <https://github.com/eclipse/xttext-core>.

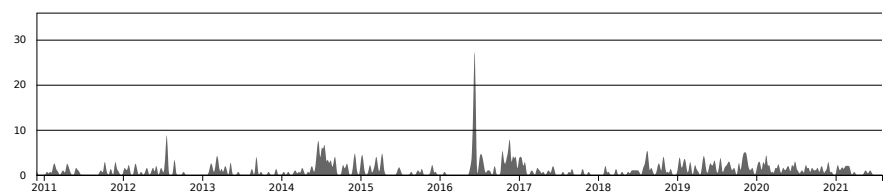


Figure 17: Contributions graph for the `eclipse/xttext-lib` repository on GitHub. See <https://github.com/eclipse/xttext-lib>.

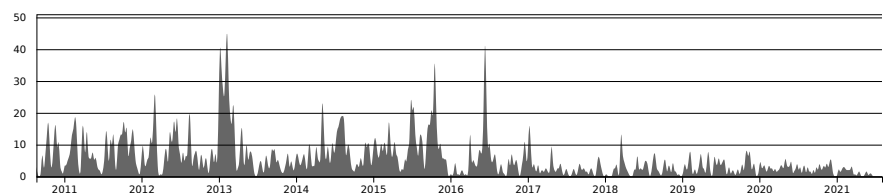


Figure 18: Contributions graph for the `eclipse/xttext-extras` repository on GitHub. See <https://github.com/eclipse/xttext-extras>.

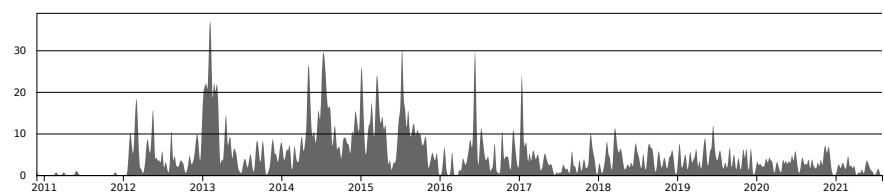


Figure 19: Contributions graph for the `eclipse/xttext-xtend` repository on GitHub. See <https://github.com/eclipse/xttext-xtend>.

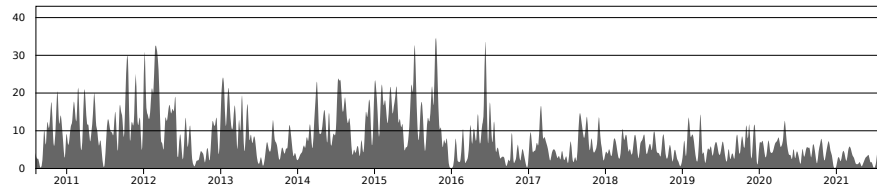


Figure 20: Contributions graph for the `eclipse/xtext-eclipse` repository on GitHub. See <https://github.com/eclipse/xtext-eclipse>.

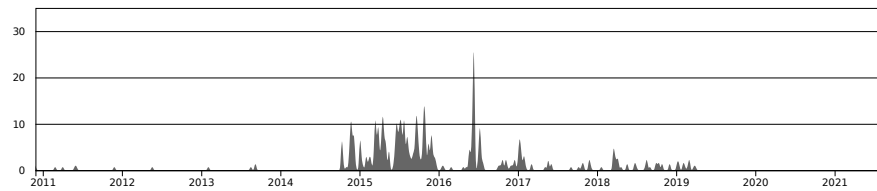


Figure 21: Contributions graph for the `eclipse/xtext-idea` repository on GitHub. See <https://github.com/eclipse/xtext-idea>.

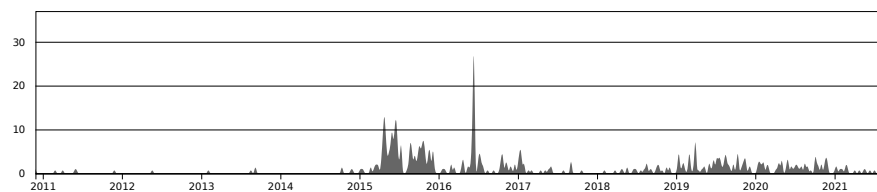


Figure 22: Contributions graph for the `eclipse/xtext-web` repository on GitHub. See <https://github.com/eclipse/xtext-web>.

JetBrains MPS

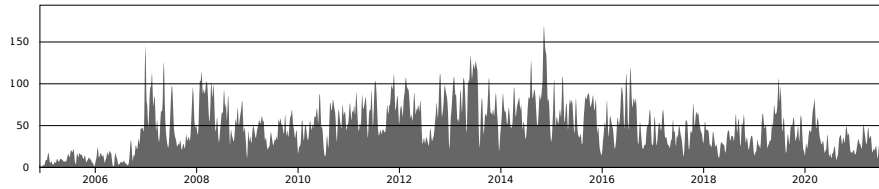


Figure 23: Contributions graph for the `JetBrains/MPS` repository on GitHub. See <https://github.com/JetBrains/MPS>.

MPS is contained in a single repository, which simplifies determining whether it is actively maintained or not. Looking at the contributions graph (see Figure 23), we can say that while development started in 2004, active development only started in 2007 and has continued to this day, only slowly lowering in activity since around 2015-2016.

In conclusion: we can say that MPS is being actively developed.

IntelliJ IDEA

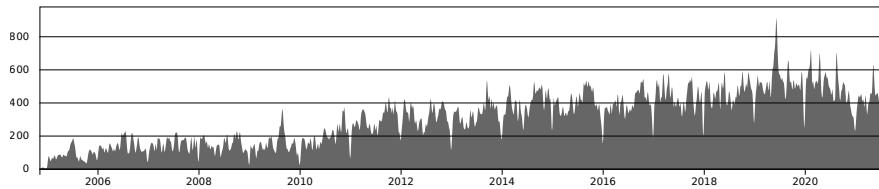


Figure 24: Contributions graph for the `JetBrains/intellij-community` repository on GitHub. See <https://github.com/JetBrains/intellij-community>.

The community edition of IntelliJ IDEA is open source and contained in a single repository, we can thus look at its contributions graph (see Figure 24) to see how active development on it is. Looking at this graph, we see that since the start in 2004 (first git commit, though their first release was in 2001) activity has seen a steady increase. Because the paid edition of IntelliJ IDEA is closed source, we can only say that it is very likely to be as or more actively developed than the free edition.

Based on this, we can conclude that IntelliJ IDEA is actively maintained.

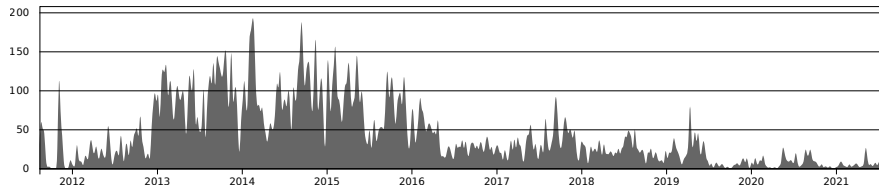


Figure 25: Contributions graph for the `atom/atom` repository on GitHub. See <https://github.com/atom/atom>.

Atom

Atom has been developed since 2011. Looking at the contribution graph (see Figure 25), it looks like the most active time was from 2013 to halfway 2016. Afterwards activity decreased, and halfway 2019 this has decreased even further. This indicates that active development has stalled at least somewhat, though this can just be an indicator that the tool is feature complete (leaving new features to be added as packages). Furthermore, the official blog has not received a new post for almost 2 years now, with the last post being in July of 2019 announcing the release of Atom 1.39 [34]. Lastly GitHub, the creator of Atom, got bought by Microsoft in 2018 [67] which owns competing editor/IDE Visual Studio Code. GitHub also announced GitHub Copilot [38] (an AI driven code generation tool) in 2021 which only supports VSCode, with at the time of writing no plans to bring this to other platforms [38].

While there has been no official announcement of Atom losing support, all signs point to it already having lost a significant part and may only be in maintenance mode, and possibly being abandoned in the near future.

Visual Studio Code

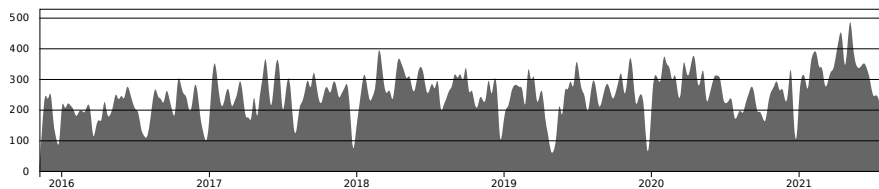


Figure 26: Contributions graph for the `microsoft/vscode` repository on GitHub. See <https://github.com/microsoft/vscode>.

Visual Studio Code is a relatively new (2015) editor/IDE, looking at the contributions graph (see Figure 26) it looks like activity has been constant or

even maybe slightly increasing since then. We can say that Visual Studio Code will most likely stay actively maintained the following few years.

Python Lex-Yacc

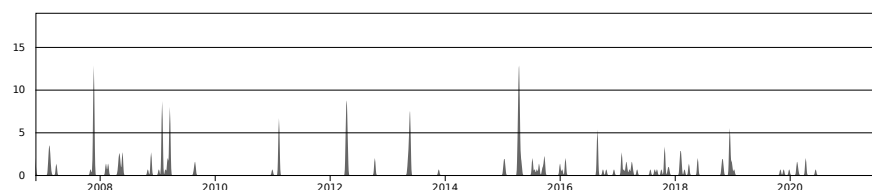


Figure 27: Contributions graph for the `dabeaz/ply` repository on GitHub. See <https://github.com/dabeaz/ply>.

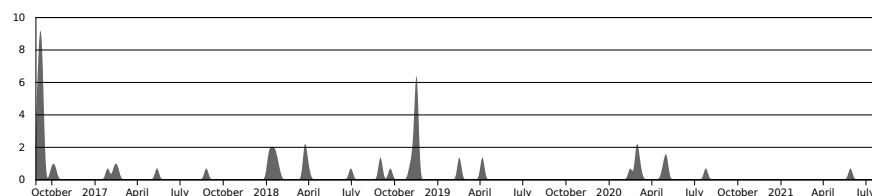


Figure 28: Contributions graph for the `dabeaz/sly` repository on GitHub. See <https://github.com/dabeaz/sly>.

According to PLY homepage [7], PLY is no longer maintained as an installable package (last release being from 2018 [8]), and new features are no longer being added. However it states that it is still maintained and modernized, though in effect this is for bugfixes only. It also links to a more modern parser/reimplementation of PLY which is maintained by the same developer called SLY [6], [9], though this is currently still a work in progress and has not yet seen a public release.

Looking at the contributions graph for both PLY and SLY (see Figures 27 and 28), not much can be gathered due to the activity being very low.

Python Lark

Lark is a relatively new library (2017). Looking at its contributions graph (see Figure 29) activity has been about the same throughout its lifetime up until the time of writing.

As it currently stands, it looks like Lark will keep being actively maintained in the near future, with new features still being added.

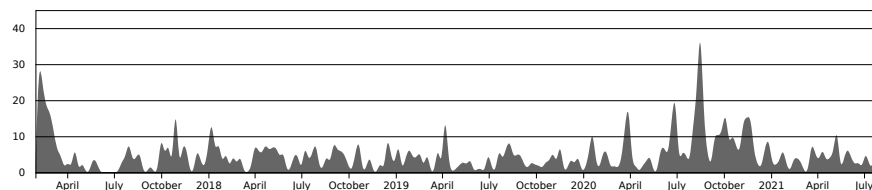


Figure 29: Contributions graph for the `lark-parser/lark` repository on GitHub. See <https://github.com/lark-parser/lark>.

6.4 Documentation

Spoofax

Extensive documentation for Spoofax is available on the MetaBorg site [66]. This includes all the languages used to for developing your own languages and usage of the API. Some parts of this are still a work in progress at the time of writing, but placeholders exist.

Xtext

Xtext provides a tutorial to get familiar with most language specification features in 15 minutes [22]. More in-depth information including the exact syntax, services provided, and examples of common language configurations are also provided on their online documentation.

JetBrains MPS

MPS has an extensive manual [52] available for creating and working with languages, going over every aspect that is available to the designer and providing information for implementing common language patterns.

IntelliJ IDEA

Extensive documentation for developing a plugin for IntelliJ IDEA is provided [49], including a whole section about adding new languages [45], though these only focus on a single Lexer/Parser technology (any technology can be used however, as long as they implement the API).

Atom

Atom calls itself the “hackable editor” because as a user you can run their own code in it with full access to the API, with the documentation [33] having a whole chapter dedicated to the different ways one can “hack” their editor. Included are sections about defining grammars for syntax highlighting [35] with Tree-sitter [12] or TextMate (legacy) [61]. Adding other parts that are part of

an IDE experience requires other packages that are provided by the community, and their documentation is provided on their respective repositories.

Visual Studio Code

Visual Studio code has an extensive amount of guides on how to write extensions to implement IDE features for custom languages using their extension API [68]. They include specific documentation on how to have embedded languages.

Python Lex-Yacc

PLY features an extensive documentation on how to use it [7]. It includes guides on error recovery during parsing/lexing.

Python Lark

Lark has growing amount of documentation [83] aimed at teaching the user how to use it mostly based on examples. They also have a set of examples with common language recipes such as handling indentation with Python-like languages.

6.5 Setup

Spoofax

The Spoofax language workbench mainly works as an Eclipse IDE plugin. Instructions on how to install Spoofax are provided on the documentation site [65]. Specific support for Mac is available via the Homebrew package manager. For other platforms a manual download is required, these downloads contain a prebuilt Eclipse IDE installation with Spoofax added.

Additionally, an IntelliJ IDEA plugin is also available, though this one does not contain all functionality yet. Using this plugin is as simple as downloading it and adding it to an existing IntelliJ installation.

Xtext

Xtext works as a plugin for the Eclipse IDE, installing it works via the built-in plugin mechanism. An IntelliJ IDEA plugin also exists, but has been abandoned (last updated in 2016). Installation instructions for the Eclipse plugin can be found on the Xtext site [25].

JetBrains MPS

MPS supports all major platforms on its download page [53]:

- Linux binaries are prebuilt and provided as a tarball (`.tar.gz` file) which the user needs to place somewhere. No support for any package managers is provided.

- Mac support is provided for both Intel based platforms and Apple Silicon based platforms in the form of app container `.dmg` files.
- For Windows an executable installer file is provided.

An alternative option to install MPS and keep it updated is by downloading the JetBrains Toolbox application [50]. The same platform support applies here, but it allows updating MPS easily.

IntelliJ IDEA

IntelliJ IDEA supports the same platforms on its download page [48] as MPS does, and is also supported by the JetBrains Toolbox application [50], as they are both developed by JetBrains.

Atom

Atom provides installation instructions for all current major platforms in its documentation [39]:

- Support for installing on the following Linux distributions is provided: `apt` (Ubuntu, Debian), `yum` (Red Hat, CentOS), `dnf` (Fedora), `zypp` (OpenSUSE).
- On Mac, Atom provides a zip file which can be installed into the Applications folder.
- For Windows it includes an installer executable file.

Atom provides a way for publishing packages publicly to `atom.io` and installing them from within the application itself.

Visual Studio Code

Visual Studio Code has installation instructions for all current major platforms in its documentation [72]:

- Support for installing on the following Linux distributions is provided: `Snap` (Ubuntu), `apt` (Ubuntu, Debian), `yum` (Red Hat, CentOS), `dnf` (Fedora), `zypp` (OpenSUSE), `AUR` (Arch Linux), `nix` (NixOS).
- A `.app` application file is provided for installing on Mac.
- An installer executable is provided for installing on Windows.

Visual Studio Code allows developer to publish extensions to the Visual Studio Code Marketplace, which allows them to be installed from the application itself. However, this marketplace is only available in builds provided by Microsoft themselves and usage by non-official builds is prohibited.

Python Lex-Yacc

Installation of Python Lex-Yacc can be done by using the Python package manager `pip` and installing the `ply` package [8]. If we were to use this library for our tool, it would be downloaded automatically as a dependency on installation.

Python Lark

Installation of Lark can be done by using the Python package manager `pip` and installing the `lark-parser` package [82]. If we were to use this library for our tool, it would be downloaded automatically as a dependency on installation.

6.6 Support

Spoofax

On its support page [62] Spoofax details an issue tracker for issues and feature requests [85]. Furthermore there exists a Slack organization for user support which one can get access to upon request.

Xtext

Xtext on its community page [23] links to the Xtext forum [19] for getting help in case you are stuck. They also link to their GitHub for reporting bugs and feature requests.

JetBrains MPS

On its product page [53] near the bottom are links to the MPS community forum [55] for receiving support with using it. It also has a link to the JetBrains issue tracker for MPS [51] for reporting issues and submitting feature requests.

IntelliJ IDEA

On the IntelliJ IDEA product page [48] near the bottom are links to the community forum [54] for receiving support, and to its issue tracker [47] for reporting issues and submitting feature requests.

Atom

Atom used to have a Discourse discussion forum for user support but now uses GitHub discussions instead [36]. Bug reports and feature requests can be done on the same GitHub repository.

Visual Studio Code

At the bottom of the FAQ for Visual Studio Code [74] it is mentioned users can submit bug reports and feature requests on the GitHub repository [69] and using Stack Overflow for getting support [87].

Python Lex-Yacc

The homepage for PLY [7] notes the GitHub repository [5] as the place to get support with issues and bug reports.

Python Lark

The main Lark repository [81] notes that GitHub issues or Gitter [40] should be used for questions or issue reporting. Additionally there exists a GitHub discussions page [37] where questions can be asked, though this is not directly mentioned in the documentation.

6.7 Editor Features

Spoofax

Based on the official Spoofax documentation [66] the Spoofax Language Workbench supports the following list of editor features:

- Provided by the Editor SerVice language (ESV):
 - Syntax highlighting/coloring.
 - Line and block comment declaration for comment and uncomment shortcuts.
 - Parentheses, brackets, and braces (called fences) matching and highlighting.
 - Menus for language actions
 - File outline view.
 - Hover tooltips.
 - Compile on save.
 - Model validation and analysis.
 - Text formatting, provided as a menu action, and output to a new file.
- Provided by the Syntax Definition Formalism 3 language (SDF3):
 - Text completion.
- Provided by the Name Binding Language (NaBL2):
 - Reference resolution.
 - Model validation and analysis.
- Provided by the IDE (Eclipse or IntelliJ IDEA):
 - Version control integration.

Xtext

According to the official Xtext documentation [24], the following editor features are supported:

- Syntax highlighting/coloring (lexical and semantic).
- Model refactoring.
- Run-time debugging.
- Comment and uncomment shortcuts.
- Language specific menus
- File outline view.
- Hover tooltips.
- Compile on save.
- Automatic editing (auto-closing quotes, parentheses, etc.).
- Automatic indentation.
- Model validation, analysis, and quick fixes.
- Reference resolution and highlighting.
- Text completion.
- Text formatting.
- Text folding.
- Inline annotations.
- Version control integration.

JetBrains MPS

Based on the official documentation [52] MPS supports the following editor features:

- Syntax coloring and formatting.
- Editor actions.
- Run-time debugging.
- Editor keybindings.
- File outline view (structure view, undocumented).
- Model validation, analysis, and quick fixes.

- Reference resolution and highlighting.
- Text completion.
- Automatic formatting.
- Model inspection.
- Model refactoring.
- Version control integration.

IntelliJ IDEA

IntelliJ IDEA supports the following editor features based on the official documentation [45]:

- Syntax highlighting/coloring.
- Editor actions.
- Run-time debugging.
- Comment and uncomment shortcuts.
- File outline view.
- Model validation, analysis, and quick fixes.
- Reference resolution and highlighting.
- Text completion.
- Automatic and manual formatting.
- Model inspection.
- Text folding.
- Version control integration.

Atom

Based on the official documentation [33] Atom supports the following editor features out of the box:

- Syntax highlighting/coloring.
- Comment and uncomment shortcuts.
- Basic text completion.
- Text folding.

- Version control integration.

Additionally the following editor features are provided through extensions such as `atom-ide-ui` [30]:

- Smart text completion.
- Run-time debugging.
- File outline view.
- Model validation and analysis.
- Reference resolution and highlighting.
- Hover tooltips.
- Text formatting.

Visual Studio Code

Visual Studio Code supports the following editor features based on the official documentation [68]:

- Syntax highlighting/coloring.
- Editor actions.
- Run-time debugging.
- Comment and uncomment shortcuts.
- Model validation and analysis.
- Hover tooltips.
- Reference resolution and highlighting.
- Text completion.
- Text formatting.
- Text folding.
- Version control integration.

Python Lex-Yacc

As `PLY` is not an editor but a parsing library, we do not consider this part for our comparison.

Python Lark

As Lark is not an editor but a parsing library, we do not consider this part for our comparison.

6.8 Language Server Protocol

Spoofax

Spoofax does not currently have any documented support for the language server protocol. There exists a repository that has a skeleton for implementing a language server [64] but it has not received any updates for 4 years at the time of writing.

Xtext

Xtext has built-in support for creating a language server when setting up a new Xtext project. It includes a basic instructions page to guide the user through this [27].

JetBrains MPS

MPS does not provide language server protocol support. Because as of right now the protocol is purely for textual documents, and MPS models are represented and saved internally as a tree, the protocol is not compatible with MPS at this time.

IntelliJ IDEA

No official support for the language server protocol exists in IntelliJ IDEA. A community plugin that allows IDEA to act as a language client exists, but has not been updated since February 2020 due to the pandemic at the time of writing [88].

Atom

A community (formerly official) package exists to help with implementing a language client in Atom [2], which we could use to add support to Atom if we were to write a language server.

Visual Studio Code

The language server protocol was originally developed for use with Visual Studio Code, and while it has since been opened up as a standardized protocol, to this day development of the two is connected.

Python Lex-Yacc

PLY has no facilities for building a language server with it. One would have to manually write one either from scratch or based on a library that implements most of the LSP protocol already, and use PLY as the parser.

Python Lark

Lark has no built-in language server features in its API. One would have to manually write one either from scratch or based on a library that implements most of the LSP protocol already, and use Lark as the parser.

The Lark grammar language itself does not have a language server yet, but work is being done on creating one at the time of writing [84]. This would make use of Lark itself as the parser for the language server and would be a good base to create our own language server with Lark.

7 Conclusion & Future work

We investigated common patterns of notation for textual computer languages. Based on this we made a variability model which we described in Section 3. In it we described (1) properties important for the parser such as the grammar classification, whether it has a lexical analysis phase, support for Unicode, and the possibility of specifying alternative syntaxes; (2) properties of the structure of the language such as how statements and blocks are delimited and structured, and what comments look like; and (3) properties of the language structure, but which require specific support from the parser such as indentation sensitive or layout sensitive constructs, and auto-insertion of missing tokens in cases where semantically their omission does not matter. Possible future work on this model could involve additional variability points which we could not think of. Furthermore, a suite of example grammar definitions and test documents for each possible variation point could be developed, with which a standardized test can be developed.

In Section 6.1 we took our variability model and tried to match it to a selection of language workbenches, editors, and parsing libraries. We conclude that for supporting our variability model, using a parsing library is the best way to get as much support as possible. While language workbenches are very versatile for writing domain-specific languages (DSLs), they also have some limitations in important aspects: (1) dynamically defining new grammars is either not possible, slow, or difficult; (2) using multiple languages in a single source file is impossible or only implementable as a workaround; and (3) implementing unsupported language constructs is impossible because they do not allow modifying the parser technology. When looking at support for our variability model by the editors we looked at, we note that a lot of aspects are dependent on the underlying parser but that most of them also provide support for the Language Server Protocol.

We conclude that the best option is to take the best of two worlds: to use a parsing library for the best support of our variability model which provides the parsing facilities, and to use an editor which has extensive support for the Language Server Protocol. With this option, there is still the requirement of implementing our own language server, but there exist libraries to aid in this. Additionally a language client would need to be written for each editor, though with existing libraries this is trivial.

References

- [1] M. D. Adams, “Principled parsing for indentation-sensitive languages: Revisiting landin’s offside rule,” in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13, Rome, Italy - January 23 - 25, 2013*, R. Giacobazzi and R. Cousot, Eds., ACM, 2013, pp. 511–522. DOI: 10.1145/2429069.2429129. [Online]. Available: <https://doi.org/10.1145/2429069.2429129>.
- [2] Atom Community. “GitHub - atom-community/atom-languageclient: Provide integration support for adding Language Server Protocol servers to Atom.” [Online]. Available: <https://github.com/atom-community/atom-languageclient> (visited on 07/27/2021).
- [3] various authors. “CoffeeScript.” (Feb. 12, 2004), [Online]. Available: <https://coffeescript.org/> (visited on 02/19/2021).
- [4] various authors. “grammar.coffee,” [Online]. Available: <https://coffeescript.org/v2/annotated-source/grammar.html> (visited on 08/09/2021).
- [5] D. Beazley *et al.* “GitHub - dabeaz/ply: Python Lex-Yacc,” [Online]. Available: <https://github.com/dabeaz/ply> (visited on 07/27/2021).
- [6] D. Beazley *et al.* “GitHub - dabeaz/sly: Sly Lex Yacc,” [Online]. Available: <https://github.com/dabeaz/sly> (visited on 07/27/2021).
- [7] D. Beazley. “PLY (Python Lex-Yacc),” [Online]. Available: <https://www.dabeaz.com/ply/> (visited on 07/27/2021).
- [8] D. Beazley *et al.* “ply · PyPI,” [Online]. Available: <https://pypi.org/project/ply/> (visited on 06/11/2021).
- [9] D. Beazley. “SLY (Sly Lex Yacc) — sly 0.0 documentation,” [Online]. Available: <https://sly.readthedocs.io/en/latest/> (visited on 08/03/2021).
- [10] P. Boullier, “Dynamic grammars and semantic analysis,” INRIA, Research Report RR-2322, 1994, Projet CHLOE. [Online]. Available: <https://hal.inria.fr/inria-00074352>.
- [11] L. Brunauer and B. Mühlbacher, “Indentation sensitive languages,” 2006.
- [12] M. Brunsfeld *et al.* “Tree-sitter | Introduction,” [Online]. Available: <https://tree-sitter.github.io/tree-sitter/> (visited on 08/04/2021).

- [13] H. Bünder, “Decoupling language and editor - the impact of the language server protocol on textual domain-specific languages,” in *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019*, S. Hammoudi, L. F. Pires, and B. Selic, Eds., SciTePress, 2019, pp. 129–140. DOI: 10.5220/0007556301310142. [Online]. Available: <https://doi.org/10.5220/0007556301310142>.
- [14] S. Cabasino, P. S. Paolucci, and G. M. Todesco, “Dynamic parsers and evolving grammars,” *ACM SIGPLAN Notices*, vol. 27, no. 11, pp. 39–48, 1992. DOI: 10.1145/141018.141037. [Online]. Available: <https://doi.org/10.1145/141018.141037>.
- [15] N. Chomsky, “Three models for the description of language,” *IRE Trans. Inf. Theory*, vol. 2, no. 3, pp. 113–124, 1956. DOI: 10.1109/TIT.1956.1056813. [Online]. Available: <https://doi.org/10.1109/TIT.1956.1056813>.
- [16] N. Chomsky, “On certain formal properties of grammars,” *Inf. Control.*, vol. 2, no. 2, pp. 137–167, 1959. DOI: 10.1016/S0019-9958(59)90362-6. [Online]. Available: [https://doi.org/10.1016/S0019-9958\(59\)90362-6](https://doi.org/10.1016/S0019-9958(59)90362-6).
- [17] Y. Deng, “The formal semantics of programming languages,” *Shanghai Jiaotong University, Tech. Rep.*, vol. 16, 2010.
- [18] J. Earley, “An efficient context-free parsing algorithm,” *Commun. ACM*, vol. 13, no. 2, pp. 94–102, 1970. DOI: 10.1145/362007.362035. [Online]. Available: <https://doi.org/10.1145/362007.362035>.
- [19] Eclipse. “Eclipse Community Forums: TMF (Xtext),” [Online]. Available: https://www.eclipse.org/forums/index.php?t=thread&frm_id=27 (visited on 07/27/2021).
- [20] Eclipse. “GitHub - eclipse/xtext-idea: xtext-idea,” [Online]. Available: <https://github.com/eclipse/xtext-idea> (visited on 07/27/2021).
- [21] Eclipse. “Xtend - Modernized Java,” [Online]. Available: <https://www.eclipse.org/xtend/> (visited on 07/27/2021).
- [22] Eclipse. “Xtext - 15 Minutes Tutorial,” [Online]. Available: https://www.eclipse.org/Xtext/documentation/102_domainmodelwalkthrough.html (visited on 06/11/2021).
- [23] Eclipse. “Xtext - Community,” [Online]. Available: <https://www.eclipse.org/Xtext/community.html> (visited on 07/27/2021).
- [24] Eclipse. “Xtext - Documentation,” [Online]. Available: <https://www.eclipse.org/Xtext/documentation/index.html> (visited on 08/04/2021).
- [25] Eclipse. “Xtext - Download,” [Online]. Available: <https://www.eclipse.org/Xtext/download.html> (visited on 06/11/2021).

- [26] Eclipse. “Xtext - Language Engineering Made Easy!” [Online]. Available: <https://www.eclipse.org/Xtext/> (visited on 07/27/2021).
- [27] Eclipse. “Xtext - LSP Support,” [Online]. Available: https://www.eclipse.org/Xtext/documentation/340_lsp_support.html (visited on 07/27/2021).
- [28] ECMA International, *Standard ECMA-262 - ECMAScript® 2020 Language Specification*. 2020. [Online]. Available: <https://www.ecma-international.org/wp-content/uploads/ECMA-262.pdf>.
- [29] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann, “Layout-sensitive generalized parsing,” in *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, K. Czarnecki and G. Hedin, Eds., ser. Lecture Notes in Computer Science, vol. 7745, Springer, 2012, pp. 244–263. DOI: 10.1007/978-3-642-36089-3_14. [Online]. Available: https://doi.org/10.1007/978-3-642-36089-3_14.
- [30] Facebook Inc. “atom-ide-ui,” [Online]. Available: <https://atom.io/packages/atom-ide-ui> (visited on 08/04/2021).
- [31] A. M. Fard, A. Deldari, and H. Deldari, “Quick grammar type recognition: Concepts and techniques,” *CoRTA'2007*, p. 51, 2007.
- [32] GitHub. “Atom,” [Online]. Available: <https://atom.io/> (visited on 07/27/2021).
- [33] GitHub. “Atom,” [Online]. Available: <https://flight-manual.atom.io/> (visited on 07/27/2021).
- [34] GitHub. “Atom Blog | A hackable text editor for the 21st Century,” [Online]. Available: <https://blog.atom.io/> (visited on 08/03/2021).
- [35] GitHub. “Creating a Grammar,” [Online]. Available: <https://flight-manual.atom.io/hacking-atom/sections/creating-a-grammar/> (visited on 07/27/2021).
- [36] GitHub. “Discussions · atom/atom · GitHub,” [Online]. Available: <https://github.com/atom/atom/discussions/> (visited on 07/27/2021).
- [37] GitHub. “Discussions · lark-parser/lark · GitHub,” [Online]. Available: <https://github.com/lark-parser/lark/discussions> (visited on 07/27/2021).
- [38] GitHub. “GitHub Copilot · Your AI pair programmer,” [Online]. Available: <https://copilot.github.com/> (visited on 08/04/2021).
- [39] GitHub. “Installing Atom,” [Online]. Available: <https://flight-manual.atom.io/getting-started/sections/installing-atom/> (visited on 06/11/2021).
- [40] Gitter. “lark-parser/Lobby - Gitter,” [Online]. Available: <https://gitter.im/lark-parser/Lobby> (visited on 07/27/2021).

- [41] J. Gosling, B. Joy, G. Steele, *et al.*, *The java language specification. Java SE 16 Edition*, Oracle America, Inc., 2021.
- [42] ISO/IEC 14882:2020, “Programming languages – C++,” International Organization for Standardization, Geneva, CH, Standard, 2020.
- [43] K. Jensen and N. Wirth, *Pascal user manual and report - ISO Pascal standard, 4th Edition*. Springer, 1991, ISBN: 978-0-387-97649-5.
- [44] JetBrains. “2. Language and File Type | IntelliJ Platform Plugin SDK,” [Online]. Available: <https://plugins.jetbrains.com/docs/intellij/language-and-filetype.html> (visited on 08/04/2021).
- [45] JetBrains. “Custom Language Support | IntelliJ Platform Plugin SDK,” [Online]. Available: <https://plugins.jetbrains.com/docs/intellij/custom-language-support.html> (visited on 07/27/2021).
- [46] JetBrains. “Diagramming Editor | MPS,” [Online]. Available: <https://www.jetbrains.com/help/mps/diagramming-editor.html> (visited on 08/14/2021).
- [47] JetBrains. “IntelliJ IDEA (IDEA) - JetBrains YouTrack,” [Online]. Available: <https://youtrack.jetbrains.com/issues/IDEA> (visited on 07/27/2021).
- [48] JetBrains. “IntelliJ IDEA: The Capable & Ergonomic Java IDE by JetBrains,” [Online]. Available: <https://www.jetbrains.com/idea/> (visited on 06/11/2021).
- [49] JetBrains. “IntelliJ Platform SDK | IntelliJ Platform Plugin SDK,” [Online]. Available: <https://plugins.jetbrains.com/docs/intellij/welcome.html> (visited on 07/27/2021).
- [50] JetBrains. “JetBrains Toolbox App: Manage Your Tools with Ease,” [Online]. Available: <https://www.jetbrains.com/toolbox-app/> (visited on 06/11/2021).
- [51] JetBrains. “MPS (MPS) - JetBrains YouTrack,” [Online]. Available: <https://youtrack.jetbrains.com/issues/MPS> (visited on 07/27/2021).
- [52] JetBrains. “MPS User’s Guide | MPS,” [Online]. Available: <https://www.jetbrains.com/help/mps/mps-user-s-guide.html> (visited on 07/27/2021).
- [53] JetBrains. “MPS: The Domain-Specific Language Creator by JetBrains,” [Online]. Available: <https://www.jetbrains.com/mps/> (visited on 06/11/2021).

- [54] JetBrains. “Topics – IDEs Support (IntelliJ Platform) | JetBrains,” [Online]. Available: <https://intellij-support.jetbrains.com/hc/en-us/community/topics> (visited on 07/27/2021).
- [55] JetBrains. “Topics – MPS Support | JetBrains,” [Online]. Available: <https://mps-support.jetbrains.com/hc/en-us/community/topics> (visited on 07/27/2021).
- [56] JSON-RPC Working Group, “JSON-RPC 2.0 Specification,” JSON-RPC Working Group, Tech. Rep., 2010. [Online]. Available: <https://www.jsonrpc.org/specification>.
- [57] L. C. L. Kats and E. Visser, “The spoofax language workbench,” in *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, W. R. Cook, S. Clarke, and M. C. Rinard, Eds., ACM, 2010, pp. 237–238. DOI: 10.1145/1869542.1869592. [Online]. Available: <https://doi.org/10.1145/1869542.1869592>.
- [58] T. Kühne, “Matters of (meta-)modeling,” *Softw. Syst. Model.*, vol. 5, no. 4, pp. 369–385, 2006. DOI: 10.1007/s10270-006-0017-9. [Online]. Available: <https://doi.org/10.1007/s10270-006-0017-9>.
- [59] P. J. Landin, “The next 700 programming languages,” *Commun. ACM*, vol. 9, no. 3, pp. 157–166, 1966. DOI: 10.1145/365230.365257. [Online]. Available: <https://doi.org/10.1145/365230.365257>.
- [60] J. H. Larkin and H. A. Simon, “Why a diagram is (sometimes) worth ten thousand words,” *Cogn. Sci.*, vol. 11, no. 1, pp. 65–100, 1987. DOI: 10.1111/j.1551-6708.1987.tb00863.x. [Online]. Available: <https://doi.org/10.1111/j.1551-6708.1987.tb00863.x>.
- [61] MacroMates Ltd. “TextMate: Text editor for macOS,” [Online]. Available: <https://macromates.com/> (visited on 08/04/2021).
- [62] MetaBorg. “Getting Support — Spoofax documentation,” [Online]. Available: <http://www.metaborg.org/en/latest/source/support.html> (visited on 07/27/2021).
- [63] MetaBorg. “GitHub - metaborg/spoofax-intellij,” [Online]. Available: <https://github.com/metaborg/spoofax-intellij> (visited on 07/27/2021).
- [64] MetaBorg. “GitHub - metaborg/spoofax-lsp: Language Server Protocol support for Spoofax,” [Online]. Available: <https://github.com/metaborg/spoofax-lsp> (visited on 07/27/2021).
- [65] MetaBorg. “Installing Spoofax - Spoofax documentation,” [Online]. Available: <http://www.metaborg.org/en/latest/source/install.html> (visited on 06/11/2021).

- [66] MetaBorg. “The Spoofox Language Workbench — Spoofox documentation,” [Online]. Available: <http://www.metaborg.org/en/latest/> (visited on 07/27/2021).
- [67] Microsoft, “Microsoft to acquire github for \$7.5 billion,” *Microsoft News Center*, Jun. 4, 2018. [Online]. Available: <https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/> (visited on 08/05/2021).
- [68] Microsoft. “Extension API | Visual Studio Code Extension API,” [Online]. Available: <https://code.visualstudio.com/api> (visited on 07/27/2021).
- [69] Microsoft. “GitHub - microsoft/vscode: Visual Studio Code,” [Online]. Available: <https://github.com/microsoft/vscode> (visited on 07/27/2021).
- [70] Microsoft. “Monaco Editor,” [Online]. Available: <https://microsoft.github.io/monaco-editor/> (visited on 08/05/2021).
- [71] Microsoft. “Official page for Language Server Protocol,” [Online]. Available: <https://microsoft.github.io/language-server-protocol/> (visited on 08/04/2021).
- [72] Microsoft. “Setting up Visual Studio Code,” [Online]. Available: <https://code.visualstudio.com/docs/setup/setup-overview> (visited on 06/11/2021).
- [73] Microsoft. “Visual Studio Code - Code Editing. Redefined,” [Online]. Available: <https://code.visualstudio.com/> (visited on 07/27/2021).
- [74] Microsoft. “Visual Studio Code Frequently Asked Questions,” [Online]. Available: <https://code.visualstudio.com/docs/supporting/faq> (visited on 07/27/2021).
- [75] J. Orendroff. “Js syntactic quirks.” (Jun. 3, 2020), [Online]. Available: <https://github.com/mozilla-spidermonkey/jsparagus/blob/master/js-quirks.md> (visited on 06/04/2021).
- [76] T. Parr and K. Fisher, “Ll(*): The foundation of the ANTLR parser generator,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds., ACM, 2011, pp. 425–436. DOI: 10.1145/1993498.1993548. [Online]. Available: <https://doi.org/10.1145/1993498.1993548>.
- [77] B. C. Pierce, *Types and programming languages*. MIT Press, 2002, ISBN: 978-0-262-16209-8.
- [78] Rust Team. “The rust reference,” [Online]. Available: <https://doc.rust-lang.org/reference/> (visited on 08/09/2021).

- [79] I. Sakai, *Syntax in universal translation*. Her Majesty's Stationary Office, 1962.
- [80] E. Shinan. "GitHub - erezsh/plyplus: a friendly yet powerful LR-parser written in Python," [Online]. Available: <https://github.com/erezsh/plyplus> (visited on 07/27/2021).
- [81] E. Shinan *et al.* "GitHub - lark-parser/lark: Lark is a parsing toolkit for Python, built with a focus on ergonomics, performance and modularity.," [Online]. Available: <https://github.com/lark-parser/lark> (visited on 07/27/2021).
- [82] E. Shinan *et al.* "lark-parser · PyPI," [Online]. Available: <https://pypi.org/project/lark-parser/> (visited on 06/11/2021).
- [83] E. Shinan *et al.* "Welcome to Lark's documentation! — Lark documentation," [Online]. Available: <https://lark-parser.readthedocs.io/en/latest/> (visited on 07/27/2021).
- [84] E. Shinan and R. McGregor. "GitHub - lark-parser/lark-language-server: Provides a language server for grammars based on Lark," [Online]. Available: <https://github.com/lark-parser/lark-language-server> (visited on 07/27/2021).
- [85] Software Language Design and Engineering Group TU Delft. "Spoofax on Yellowgrass.org," [Online]. Available: <https://yellowgrass.org/project/Spoofax> (visited on 07/27/2021).
- [86] J. Sprinkle, B. Rumpe, H. Vangheluwe, and G. Karsai, "Metamodelling - state of the art and research challenges," in *Model-Based Engineering of Embedded Real-Time Systems - International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007. Revised Selected Papers*, H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz, Eds., ser. Lecture Notes in Computer Science, vol. 6100, Springer, 2007, pp. 57–76. DOI: 10.1007/978-3-642-16277-0_3. [Online]. Available: https://doi.org/10.1007/978-3-642-16277-0_3.
- [87] Stack Exchange Inc. "Recently Active 'visual-studio-code' Questions - Stack Overflow," [Online]. Available: <https://stackoverflow.com/questions/tagged/visual-studio-code> (visited on 07/27/2021).
- [88] G. Tâche. "LSP Support - plugin for IntelliJ IDEs | JetBrains," [Online]. Available: <https://plugins.jetbrains.com/plugin/10209-lsp-support> (visited on 07/27/2021).
- [89] The FreeBSD Foundation. "Freebsd documentation project primer for new contributors." version eab1c5d1f6. (Jan. 12, 2021), [Online]. Available: <https://download.freebsd.org/ftp/doc/en/books/fdp-primer/book.pdf> (visited on 02/20/2021).

- [90] The Go Authors. “The go programming language specification.” (Feb. 10, 2021), [Online]. Available: <https://golang.org/ref/spec> (visited on 06/09/2021).
- [91] The Python Software Foundation. “10. Full Grammar specification — Python 3.9.6 documentation,” [Online]. Available: <https://docs.python.org/3/reference/grammar.html> (visited on 08/09/2021).
- [92] The Unicode Consortium, *The Unicode[®] Standard - Version 13.0 - Core Specification*. 2020.
- [93] D. Van Tassel. “Comments in programming languages.” (Feb. 12, 2004), [Online]. Available: <http://www.gavilan.edu/csis/languages/comments.html> (visited on 02/19/2021).
- [94] E. Vergnaud. “The prompto platform.” <http://prompto.org/>, Accessed on 2021-02-25., [Online]. Available: <http://prompto.org/> (visited on 02/25/2021).
- [95] E. Visser, *Scannerless generalized-LR parsing*. Universiteit van Amsterdam. Programming Research Group, 1997.