

WEAVE: A Dynamic Multi- Language Parsing Framework

Mitchel Pyl

Promotor: Prof. Hans Vangheluwe

Thesis ingediend in Augustus 2022 bij het
Departement Informatica van de
Faculteit Wetenschappen, Universiteit Antwerpen,
ter vervulling van de vereisten voor het bekomen
van het Diploma van Master in de Wetenschappen.

Contents

Abstract	iv
Acknowledgments	v
Nederlandstalige Samenvatting	vi
1 Introduction	1
1.1 Motivation	1
1.2 Contributions	2
1.3 Structure	2
Glossary	4
Acronyms	6
I Background	7
2 Incrementality and dynamicity	8
3 Black box and White box	10
4 Island Grammars	12
5 Parsers	13
5.1 Language and Grammar Classifications	16
5.2 Parse Trees	18

6	Modelling	21
6.1	Hybrid Languages	24
II	Model Overview	26
7	Overview	27
8	Parse Trees	29
8.1	Comparison of Implementations	30
8.2	Implementation	33
8.2.1	Tree Definition: Abstract Syntax	35
8.2.2	Tree Definition: Concrete Syntax	41
8.2.3	Tree Instances	46
8.2.4	Tree Construction	50
8.2.5	Tree Construction: Operations	51
8.3	Parser Integration	60
8.3.1	Extended Backus-Naur Forms	64
8.3.2	Note	65
8.4	Grammars	66
8.5	Considered Variations	70
9	Parsers and Languages	72
10	Tracing	76
11	Multi-Language Parsing	79
11.1	Parsing	81
11.2	Concrete Syntax	82
11.2.1	Selecting a Language	83
11.3	Non-trivial Issues	84
11.3.1	Early End Sentinels	84
11.3.2	Indentation Sensitive Languages	86
11.3.3	Cross-Fragment References	86
11.4	Similar Techniques	86
11.5	Dynamic Multi-Language Parsing	88
III	Implementation	89
12	Overview	90
13	Supporting Types	91

14 Tracing	94
14.1 Source Resolvers	95
15 Parse Trees	96
15.1 Tree Definition	96
15.2 Tree Instance	97
15.3 Tree Construction	98
16 Grammar Specification	100
16.1 Embed Specifications	104
16.2 Indentation Sensitive Languages	105
17 Lark Implementation	106
17.1 Multi-Language Parsing	108
17.2 Dynamic Multi-Language Parsing	109
18 Bootstrapping	110
IV End	113
19 Conclusions	114
19.1 Future Work	114
Bibliography	117
Appendices	122
Appendix A Grammars and Models	123
Appendix B Example Models	132
Appendix C Parsing Tools Comparison	146

Abstract

In order to describe phenomena and systems in the real world, the use of models is employed. These models provide an approximate description of something complex, and are generally developed by those knowledgeable on the domain of the subject, called domain experts.

For a domain expert without programming experience to create a model, a domain specific language needs to exist or be engineered for them to work with. Such a language may describe models visually through drawings, graphs and symbols, or by writing out a description in text. In both cases, a computer needs to be able to read the model and turn it into something it understands and can work with efficiently.

Complex systems with many aspects may not be describable with a single language, languages may need to be composed in their syntax and semantics in order to describe these systems, giving us hybrid languages.

In this thesis we look at specifying and implementing a way of parsing these hybrid languages when they are specified using a textual notation. As part of this thesis, a Python implementation is provided that handles the concepts discussed in this document.

Acknowledgments

I would like to thank my promotor, Hans Vangheluwe, for providing assistance and feedback, being patient with my struggles, and generally guiding me through my research project and masters thesis during the covid pandemic. He allowed me to delve into the world of parsers, a favorite subject of mine, and gave me many challenging thoughts and ideas to try and think about.

I would also like to thank Simon Van Mierlo, my co-promotor at the start, for his initial guidance with my research project.

Lastly, I would like to acknowledge Eelco Visser, who sadly passed away unexpectedly earlier this year, and whose work I used and referenced.

Nederlandstalige Samenvatting

Om fenomenen en systemen uit de echte wereld te beschrijven maakt men gebruik van modellen. Deze modellen geven een benadering van iets complex, en worden algemeen ontwikkeld door mensen met kennis van de materie van het domein, domeinexperten genoemd.

Om als domeinexpert zonder programmeerervaring een model te creëren moet er een domeinspecifieke taal bestaan of ontwikkeld worden om mee te werken. Zo een taal kan dan modellen beschrijven aan de hand van tekeningen, grafieken en symbolen, of door een stuk geschreven text. In beide gevallen is het nodig dat een computer deze modellen kan lezen, om deze dan om te vormen naar iets dat het begrijpt en efficiënt mee kan werken.

Complexe systemen met verschillende aspecten zijn niet beschrijfbaar met één enkele taal. Talen moeten dan mogelijk samengevoegd worden met betrekking tot hun syntaxis en semantiek om deze systemen toch te kunnen beschrijven, wat ons dan hybride talen geeft.

In deze thesis kijken we naar het specificeren en implementeren van een manier om deze hybride talen te kunnen parsen wanneer zij een tekstuele notatie gebruiken. Als deel van deze thesis is er ook een Python implementatie gegeven die de concepten in beschreven in dit document gebruikt.

CHAPTER 1

Introduction

The world of Model Driven Engineering is one where complex systems are engineered by turning them into a *model* representing their essence. A model often contains facets across multiple levels of abstraction and specified using different *formalisms*, using the most appropriate formalism at the most appropriate level of abstraction. This multi-level modelling philosophy is known as Multi-Paradigm Modelling.

The combination of these formalisms is a challenge, as they need to be able to reference each other or be composed as a single *hybrid model*. This composition requires the joining of the concrete syntax, abstract syntax, semantics, and the transformation steps from each to the other.

In this chapter we will look at our motivations, previous work we did, the contributions we make in this thesis, and give a brief overview of the structure of the document.

1.1 Motivation

In order to allow a domain expert to design a model in a hybrid formalism, a concrete syntax must exist. This syntax can be either visual through drawings and graphs, or through text as a text document. In [21], [22] Mustafiz et al. described the process needed for composing the visual concrete syntax, abstract syntax and semantics of languages into a hybrid language. Similarly, Paredis et al. in [26] used a hybrid visual concrete syntax, and used the DEVS formalism as a semantic domain, skipping the abstract syntax combination step. We note that the composing of abstract syntax and semantics lies outside of our scope.

We will focus on the composing of the textual concrete syntax of multiple languages, with the goal of being able to specify the syntax for each language individually and having them parse together flawlessly, without having to resort to combining the syntaxes such that they become one, or having to perform multiple passes of parsing. An additional goal we set is to be able to define languages in a document, and then dynamically parse these languages in the same document.

1.2 Contributions

In [28] as part of a previous research project (see Chapter C), we compared several tools such as language workbenches, editors and IDEs, and parsing libraries. We concluded that current language workbenches are too rigid for handling hybrid models, and don't support dynamically defining new languages. Instead a hybrid option whereby we pick an editor that supports the Language Server Protocol (LSP) and implement a language server using a parsing library would be the best course of action. This thesis continues on this work by creating a parsing framework that can handle hybrid models.

As part of this, we propose a reusable and extensible formalism for the declaration of parse trees, and a language agnostic syntax for declaring how these parse trees should be constructed. We also provide a workflow for the parsing of hybrid models through the combination of multiple individual parsers, as well as a basic syntax for arbitrarily choosing the language of a language fragment.

Parallel to this document, we also built a new framework named WEAVE, implementing the models, algorithms and formalisms in Python. In order to support (multi-language) parsing in our framework, we used an the existing parsing library Lark and extended its functionality.

1.3 Structure

We start by introducing background information in Part I, starting off with the terms '*incremental*' and '*dynamic*' in Chapter 2, and '*black box*' and '*white box*' systems. Next we give a brief overview of the concept of '*island grammars*' in Chapter 4, followed by concepts and terms used in the world of *parsers* in Chapter 5. Lastly in Chapter 6, we provide information on the concepts used in the world of Model Driven Engineering and Multi-Paradigm Modelling.

Next in Part II, we introduce the WEAVE framework. We describe and model the required systems, formalisms, and algorithms in order to parse hybrid models, which are the building blocks of WEAVE Chapter 7 gives a more in-depth overview of this part. In

Chapter 8 we start by comparing existing implementations of parse trees, and providing a set of formalisms in order to describe, represent, and build parse trees that is language agnostic and reusable. We also provide instructions on how parsers could make use of these formalisms. Next in Chapter 9 we describe an API with which support for languages and parsers can be added and defined. Chapter 10 provides structures and an explanation on how to handle *tracing* in order to allow a program to go back to the definition of a model or model element from a textual model description. Lastly in Chapter 11 we delve into multi-language and dynamic multi-language parsing in order to be able to parse hybrid models.

In Part III we go over the process of going from the specifications from the previous part into a working software system. Chapter 12 starts by providing an overview of the implementation, with Chapter 13 we provide some information on introduced types that are not part of the modelling specification, but are used to provide information on operations such as failures, and to convey that information to the user. Next in Chapter 14 we describe the implementation of the tracing structures, and the specific handling of tracing with embedded languages as we implemented it. In Chapter 16 we describe the grammar specification language used to define grammars used by WEAVE to generate parsers, and its specific handling of language embedding and indentation sensitive languages. Following that, we describe the API provided by Lark, and how we used it to make our own parsers in Chapter 17. Lastly, in Chapter 18 we go over the process of loading the parse tree formalisms and WEAVE grammar language by using themselves as a meta-language.

Finally in Part IV Chapter 19 we list some conclusions, as well as some future to be done on the formalisms, techniques, and WEAVE architecture.

abstract syntax Description of the set of valid models for a formalism. See Chapter 6.

abstract syntax tree A tree representation of a source document after parsing. Contains annotated data relevant to the program and omits irrelevant tokens. See Chapter 5.

application programming interface A type of software interface that provides services to other software.

black box A system which does not provide information on its inner workings. See Chapter 3.

concrete syntax Description of how models in a given formalism should appear. See Chapter 6.

context-free grammar A grammar specification formalism describing a set of production rules that describe all possible strings for a language. See Chapter 5.

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, which is usable in a broad domain, but lacks specialized features.

dynamic A process that can adapt during execution and evaluation. See Chapter 2.

formal language See formalism.

formalism Set of models that can be used for modelling purposes. See Chapter 6.

-
- incremental** A process that builds on top of previous executions to reduce repetitive work. See Chapter 2.
- 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.
- island** A part of the input which interests us when using an island grammar.
- island grammar** A grammar that consists of detailed productions for constructs one is interested in (islands), and liberal productions for those one is not interested in (water). See Chapter 4.
- lake** Part of input in an island which does not interest us when using a lake grammar.
- lake grammar** An island grammar which primarily consists of detailed productions and has some liberal productions for uses such as language embedding. See Chapter 4.
- language server protocol** A protocol for for communication between an editor or IDE (client) and a language server which provides language features to the client. See [17].
- language type model** Description of the language that describes the abstract syntax of a formalism. See Chapter 6.
- metamodel** See language type model.
- model** An abstraction of an aspect of reality (as-is or to-be) that is built for a given purpose. See Chapter 6.
- modelling language** See formalism.
- parse tree** A tree representation of a source document after parsing, containing minimal data such as tokens, their position and type. See Chapter 5.
- parsing expression grammar** A grammar specification formalism like Context-Free Grammars, but which does not support ambiguous grammars, and supports some languages which are nont context-free. See Chapter 5.
- scannerless parser** A parser that takes in a character stream instead of a token stream. See Chapter 5.
- water** A part of input which does not interest us when using an island grammar.
- white box** A system which can have its inner workings inspected. See Chapter 3.

Acronyms

API Application Programming Interface.

CFG Context-Free Grammar.

DSL Domain Specific Language.

IDE Integrated Development Environment.

LSP Language Server Protocol.

PEG Parsing Expression Grammar.

Part I
Background

Incrementality and dynamicity

When talking about parsers and compilers, the words “dynamic” and “incremental” are often used and with differing meanings. We give a short overview of the different meanings as they have been used.

Dynamic linking A method of linking compiled objects and libraries to an executable at run time which loads these from separate files on the filesystem. This, as opposed to static linking which happens at compile time and includes all objects and libraries directly into the executable file [27].

Dynamic loading Similar to dynamic linking, but where dynamic linking occurs at program load time, dynamic loading can happen at any point during execution. It allows for the unloading of dynamically loaded libraries as well [27].

Incremental linking A method of compiling executable files and libraries that, instead of recreating the output file every time a change was done, only updates those parts that were changed in it [18]. This eliminates recurring computations such as reference resolution for unchanged parts and gives a boost in compilation time. This sort of linking is usually only used for debugging purposes as it produces bigger and slower programs.

Incremental compilation Like incremental linking, but more general to include this process across the entire compilation pipeline rather than just the linking step. One way to achieve this is to create and save a dependency graph between data in each step [42].

Incremental parsing Incremental parsing allows a parser to update its Parse Tree without completely reparsing an input file when it gets changed [41]. It does this by

reusing the parse tree of a previous parse and only updating those nodes that relate to changed pieces of text, while maintaining a valid parse tree that should be identical to a complete reparse.

Dynamic typing Dynamic typing relates to the semantics of an executed program. With it, ‘type’ information for variables is not known at compile time as is the case for static typing. Instead this information is only available at run-time, and may change for any name throughout the course of program execution.

Dynamic Parser Less frequently used but related to the eventual goal of our research is the term *Dynamic Parser* as defined by Cabasino et al. [6] and further extended by Boullier [3]. Dynamic parsers are a set of parsers that support “evolving grammars”, which is a set of grammars that during parsing progressively have more production rules added to them, specific to the parsed string.

Based on these we can say that the term “dynamic” is mostly referred to when speaking of processes during run-time execution and evaluation, as opposed to processes that run beforehand such as compilation or pre-processing. The term “incremental” is used when talking about processes that build on top of previous executions of the same process, adding onto or replacing pieces of the previous output to get a new output.

Black box and White box

According to Bunge in [5], a “black box” is a system which has some inputs or stimuli, and some outputs that act as a reaction to the stimuli. The construction and structure of the system are irrelevant and only the behavior is important.

However, when talking about systems, and in our case software applications and libraries, we use the terms “black box” and “white box” to denote whether or not they show their inner workings or not.

Definition 1 *A **black box** is a system that gives no knowledge about its inner workings, or that obfuscates it in such a way that attempting to understand how it works is difficult.*

Definition 2 *A **white box** is a system which allows us to see and understand the inner workings.*

Both black boxes and white boxes have some form of inputs and outputs. Additionally, a ‘contract’ can exist which specifies the valid operating conditions of a box. Figure 3.1 gives an illustration of black and white boxes. It shows how a white box allows its internal structure and operation to be seen, while a black box does not and obscures it instead.

There are two possible definitions for a contract:

Definition 3 *A **contract** is a collection of requirements (pre-conditions), guarantees (post-conditions), and invariants, which define the valid operating conditions of a system. Inputs to the system must satisfy its requirements, and in turn its outputs will satisfy their guarantees, while the invariants will always hold.*

Definition 4 *A **contract** is a collection of specifications that define relations between inputs and outputs.*

While contracts in Definition 3 only provide their guarantees if their requirements are met, there is no way to specify relations such as “if A holds, then B holds, otherwise if C holds then D holds, etc.”. Definition 4 gives more flexibility in this by allowing the definition of a relation between inputs and outputs. For example, a box that has two inputs x, y and an output z could specify the following relations, with \rightarrow indicating an ‘implies’ relationship:

$$x + y > 0 \rightarrow z < 0$$

$$x + y = 0 \rightarrow z = x$$

$$x + y < 0 \rightarrow z = y$$

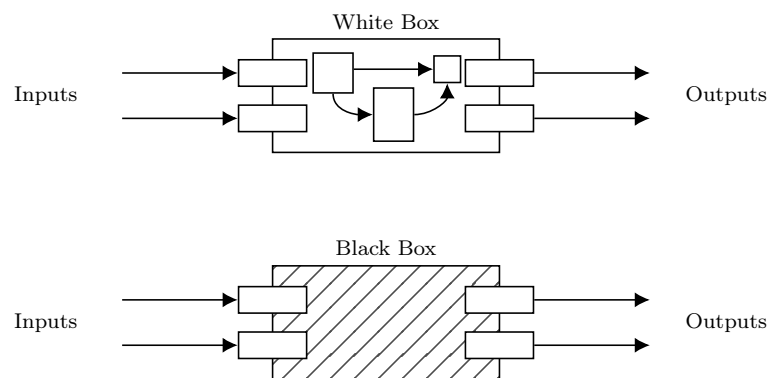


Figure 3.1: Illustration showing the difference between a white box that shows its inner workings, and a black box that shields its inner workings.

CHAPTER 4

Island Grammars

“Island grammars” are grammars that usually only parse a subset of another language, such as for finding all pieces of documentation in a source file as described by van Deursen in [10]. According to van Deursen [10] and Moonen [20], island grammars are defined as follows:

Definition 5 *An **island grammar** is a grammar that consists of (1) detailed productions for the language constructs we are interested in (called **islands**) and (2) liberal productions that catch the remainder of the input (called **water**).*

A different approach to this concept are “lake grammars”, which start with a complete grammar for a given language, and are extended with some liberal productions (**water** or **lakes**) instead. Van Deursen noted that these are useful for allowing arbitrary embedding of code [10]. They also noted that islands and water productions can be mixed to get *islands with lakes* and *lakes with islands*.

```
 $\langle start \rangle ::= \rightarrow \left\{ \begin{array}{l} \langle island \rangle \\ \langle water \rangle \end{array} \right. \longrightarrow \rightarrow$ 
```

```
 $\langle island \rangle ::= \rightarrow \text{src} - = - \langle string \rangle \longrightarrow \rightarrow$ 
```

```
 $\langle string \rangle ::= \rightarrow "[^"\\|\\|\\]" \longrightarrow \rightarrow$ 
```

```
 $\langle water \rangle ::= \rightarrow .+ \longrightarrow \rightarrow$ 
```

Listing 4.1: Example island grammar that searches for `src` tags in HTML documents. Note that for the grammar to be unambiguous the $\langle island \rangle$ rule needs to have precedence over the $\langle water \rangle$ rule (for example through priorities or ordered choice).

CHAPTER 5

Parsers

There are two ways to define what a parser is: it can be a phase in the compiler pipeline called the syntax analyzer [1], or it can be seen as something that simply takes an input string and turns it into a parse tree. We will explain what a parse tree is shortly in Section 5.2.

When looking at it from the perspective of a traditional compiler pipeline, the parser (syntax analyzer, or syntactical analysis phase) takes as input a stream of tokens. This token stream is itself derived from a string of characters, such as a file on disk, by a lexer (lexical analyzer, or lexical analysis phase). The output of the parser is a parse tree that in turn gets fed into a semantic analyzer, which checks the validity of the program and annotates the parse tree with data such as type information. See Figure 5.1 for an overview of the entire pipeline as given by Aho et al. in [1]. We note that this traditional pipeline is usually represented as having information flowing in a single direction ‘down’ the pipeline, however some compilers also allow information to flow ‘up’, for example for optimization or changing the selection of possible tokens.

Looking at it from the other direction, we can say that a parser is actually just something that transforms a character stream, such as a file on disk and turns it into a parse tree. The difference with compiler pipeline viewpoint lies in the fact that in this definition the lexical and syntactic analyzer phases of the compiler pipeline are considered to be the “parser”, rather than only the syntactic analyzer. Some programs that call themselves “parser generators” follow this viewpoint and generate not only a parser (syntactic analyzer only), but a lexer as well.

We note that that in some cases no lexical analysis is performed, and instead the parser or syntactic analyzer operates directly on the character stream rather than the token stream. These are called Scannerless Parsers [13], [40]. We provide an overview of all these different definitions and what their differences are in Figure 5.2.

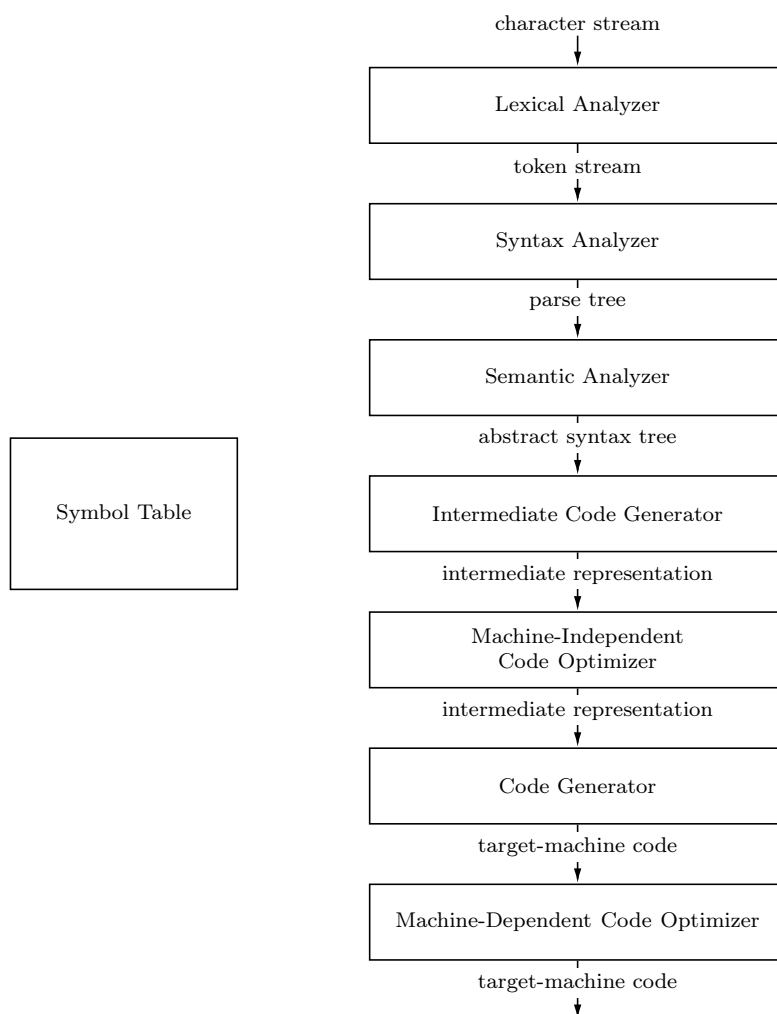


Figure 5.1: Phases of a compiler, adapted from Aho et al. [1].

For the purpose of clarity, from this point on we will follow the traditional definitions and use the following terms to avoid ambiguity:

Lexer A program that takes in a character stream and puts out a token stream.

Parser A program that takes in a token stream and puts out a parse tree.

Scannerless Parser A parser that takes in a character stream instead of a token stream.

Lexer-Parser The combination of a Lexer and a Parser as a whole, which takes in a character stream and puts out a parse tree, having a token stream in the middle.

Lexer Generator A program that takes a specification of a lexical structure and turns it into a Lexer.

Parser Generator A program that takes a syntax specification and turns it into a Parser.

Scannerless Parser Generator A program that takes a syntax specification and turns it into a Scannerless Parser.

Lexer-Parser Generator A program that takes a syntax specification and a specification of a lexical structure (combined: language specification), and turns it into a Lexer and Parser, or a Lexer-Parser.

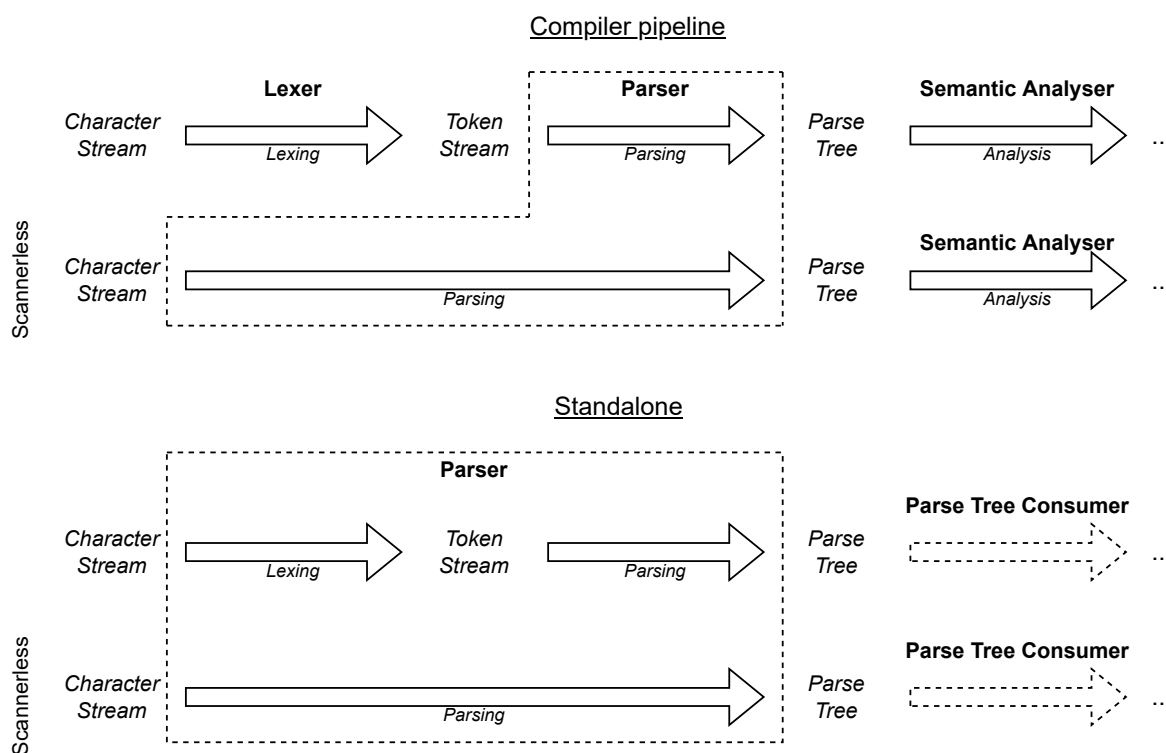


Figure 5.2: Comparison of the main definitions of “parser”. Above: the definition in a compiler pipeline, see also Figure 5.1. Below: the definition such as when using a parser generator. In both cases an instance with lexer, and a scannerless version are shown. What is understood as the “parser” in each case is encompassed by a dashed line.

5.1 Language and Grammar Classifications

Languages can be divided into several categories based on their expressiveness. Chomsky formally defined a hierarchy to divide languages into four types, named ‘Type-0’ (most expressive) through ‘Type-3’ (most restrictive) [7], [8].

Given:

a a terminal;

A, B non-terminals;

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

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

the ‘production’ relation \rightarrow indicating the left-hand side can be substituted with the right-hand side,

then the types are given as:

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 $A \rightarrow aB$.

In general, during the lexical analysis phase the simplest form of languages is used to turn text into tokens: the regular languages. Usually, this is done through the definition of ‘regular expressions’ which is a grammar specification formalism for parsing regular languages.

For the syntactical analysis phase of a parser, context-free grammars are generally used. Some parsers ‘cheat’ and allow for feedback to change their parsing process slightly,

making them not entirely context-free, but also not able to parse all context-sensitive languages.

There are several algorithms for parsing languages specified using context-free grammars, but not each algorithm is able to parse every language specified using a context-free grammar: some specified grammars can be ambiguous, or have constructs that are difficult to handle (e.g. recursion). Some algorithms are: LL, LR, SLR, LALR, GLR, Earley, CYK. Figure 5.3 shows the relationship between these algorithms and the grammars they can handle.

Additionally, there exists other algorithms that can parse context-free languages. For example, a Parsing Expression Grammar (PEG) parser can parse all context-free languages, but also some context-sensitive languages due to its ability to perform look-aheads.

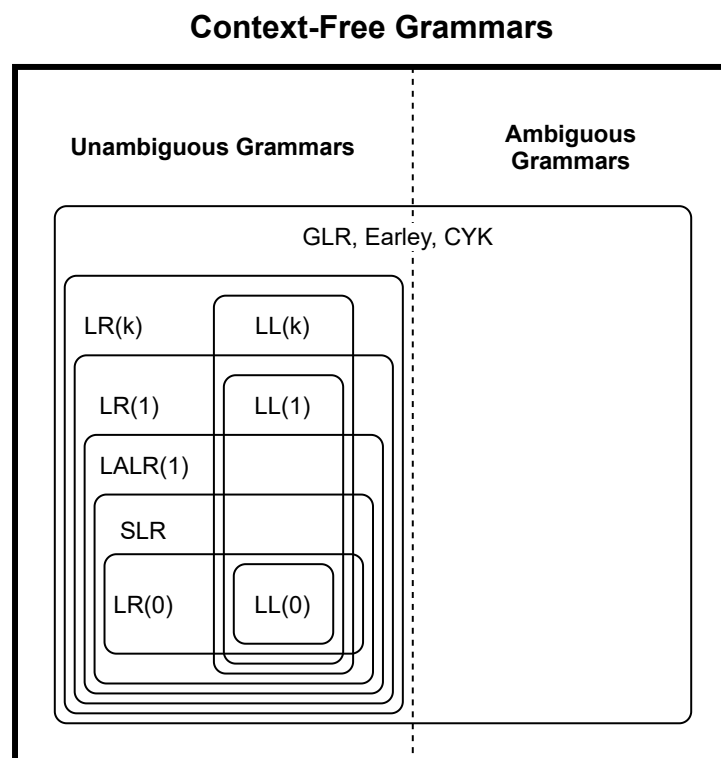


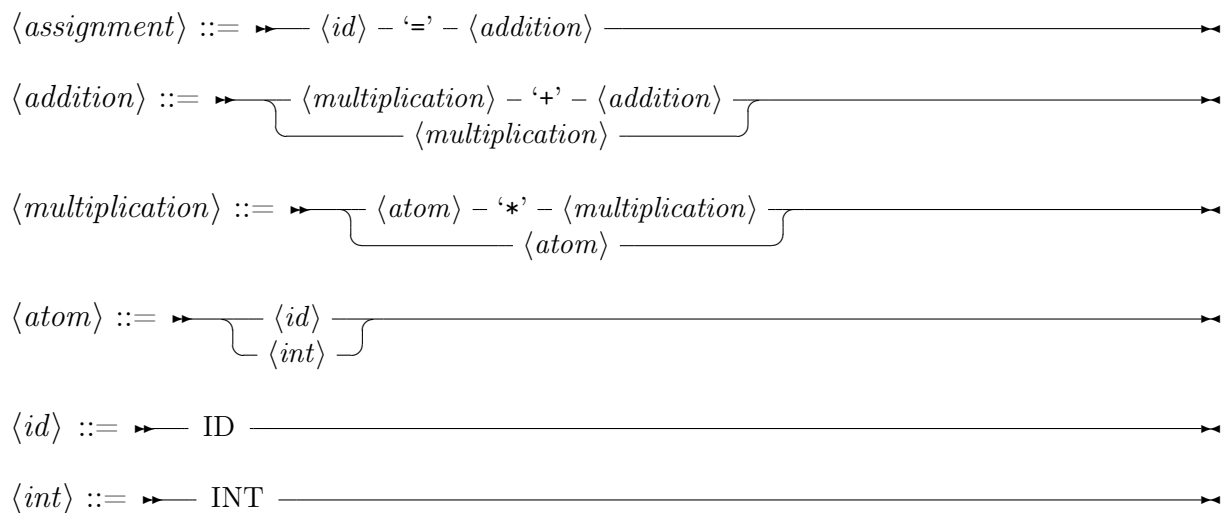
Figure 5.3: Hierarchical representation of Context-Free Grammar classes, adapted from [12].

5.2 Parse Trees

A parse tree is the output of a parser, and as the name implies is a tree structure. A basic example of a parse tree can be seen in Figure 5.4. This example conforms to the definition of a parse tree by Aho et al. in [1], which is given as follows:

1. A *Parse Tree* is a tree structure.
2. The root node is labeled by the start symbol.
3. Each leaf node is labeled by a terminal or the empty symbol ϵ .
4. Each interior node is labeled by a non-terminal.
5. If A is the non-terminal labeling some interior node with X_1, X_2, \dots, X_n the labels of the children of the node from left to right, then there must be a production $A \rightarrow X_1 X_2 \dots X_n$, with X_1, X_2, \dots, X_n each standing for either a terminal or a non-terminal.
6. If $A \rightarrow \epsilon$ is a production, then a node labeled A may instead have a single child labeled with the empty symbol ϵ .

We will sometimes also refer to a leaf node as a terminal node in this document.



Listing 5.1: Example grammar for a simple calculation language with addition and multiplication. The start symbol is $\langle \textit{assignment} \rangle$.

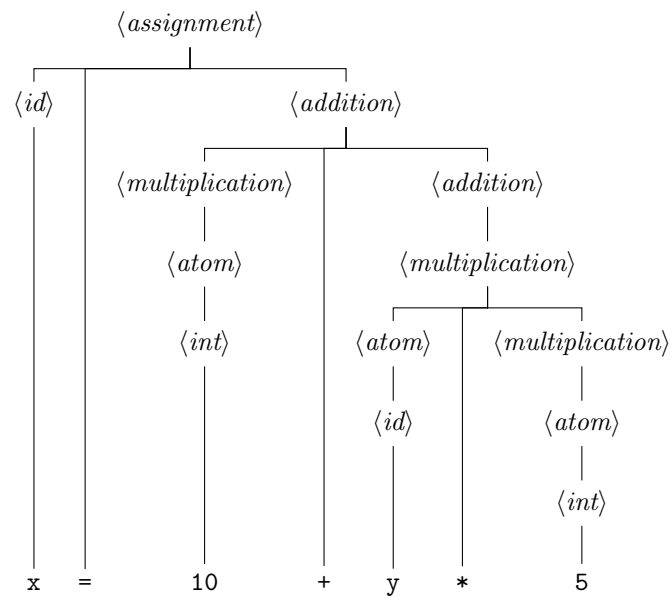


Figure 5.4: Example parse tree representation for the string $x = 10 + y * 5$ for the language defined in Listing 5.1.

Aho et al. write in [1] that parse trees resemble *Abstract Syntax Trees* (ASTs), but that instead of interior nodes representing non-terminals, the interior nodes of an AST represent programming constructs. They note that many non-terminals already represent programming constructs themselves, but that some are helpers that on a semantic level are unnecessary. To add contrast between the two, parse trees are sometimes called *Concrete Syntax Trees* instead [1]. The Object Management Group (OMG) maintains a similar distinction between parse trees and abstract syntax trees in their Abstract Syntax Tree Metamodel standard documentation [23].

We note that Figure 5.1 differs slightly from the original by Aho et al. in [1], in it both the Syntax Analyzer and the Semantic Analyzer would output syntax trees. However, based on their work we would instead say that the output of the Syntax Analyzer can more accurately be described as a *concrete syntax tree* or *parse tree*, while the output of the Semantic Analyzer is an *abstract syntax tree*.

While the definition of a parse tree by Aho et al. [1] is good to get a theoretical understanding of the concept of parse trees, for the purposes of implementing a parser it is rather constraining:

- The order of the children of a node is determined by the order in which they were matched, which is based on the definition of the matched rule in the grammar of the language.
- If a rule has alternatives, it's possible to have two or more nodes with the same labeling but with a different number of child nodes, or with the labels of their n^{th}

child not matching up (with n any number from 1 to the number of child nodes). For example, in Figure 5.4 the ‘addition’ and ‘multiplication’ nodes have either one child or three children, and ‘atom’ nodes can have either an ‘id’ or an ‘int’ node as their child.

- If changes are made to any productions in the grammar, the shape of the parse trees also changes.

This all means that anything consuming the parse tree, such as a semantic analyzer or code generator, needs to be updated almost every time a change is made to the grammar.

We would like to avoid these issues, and instead work with a nicer data structure, so in Chapter 8 we will look into how we will work with parse trees instead.

CHAPTER 6

Modelling

In this chapter we will provide some context for terms used in the discipline of Model Driven Engineering (MDE). MDE focuses on *modelling* in order to better handle essential complexities (those that are inherent to the problem or solution), and to reduce accidental complexities (those that arise from using inappropriate software or none at all) [9]. To this end, *modelling languages* (also referred to as *formal languages* or *formalisms* [14]) define a set of models that can be used to for the purposes of modelling.

According to Stachowiak in [33], a *model* is made up of the following features:

Mapping feature A model is based on an original.

Reduction feature A model only reflects a (relevant) selection of an original's properties.

Pragmatic feature A model needs to be usable in place of an original with respect to some purpose.

Or more succinctly as put by Combemale et al. in [9]: “*a model is an abstraction of an aspect of reality (as-is or to-be) that is built for a given purpose*”.

A modelling language then defines a set of models that can be used for modelling purposes. According to Combemale et al. [9] and Heinrich et al. [14] its definitions consists of:

- Syntax (also referred to as *concrete syntax*), describing how models described in the language should appear.
- Semantics, describing the meaning of each of its models. This can be through a *semantic mapping* to a *semantic domain*.

- Pragmatics, to describe how to use its models according to their purpose.

Heinrich et al. in [14] used, for a given language, \mathbb{L} for the *set of well-formed models* (also referred to as the *abstract syntax* by others), \mathbb{S} for the *semantic domain*, and \mathbb{M} for the *semantic mapping*. See Figure 6.1 for a graphical representation of these.

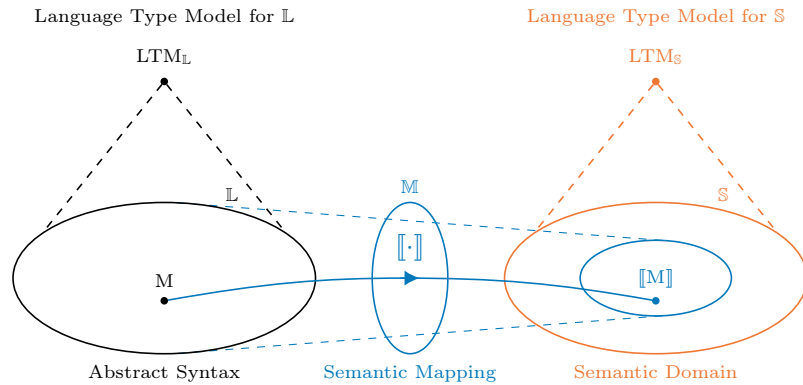


Figure 6.1: Overview of the relationship between abstract syntax and semantics for a formalism.

Note that both \mathbb{L} and \mathbb{S} are defined through a “*metamodel*” (also sometimes called the *language type model*), which is a model used to describe the abstract syntax of a language. Thus, the semantic mapping \mathbb{M} can be a conversion of one formalism to another. However, the semantic domain \mathbb{S} may also be the set of valid inputs for a state machine, or the set of possible states with inputs, etc. In general, the semantic domain is often precisely defined mathematically, such as a well known formalism like Statecharts or Petri Nets, that represent what we want to describe [9].

Also note that the semantic mapping \mathbb{M} maps the entirety (or sometimes only a portion) of the well-formed models onto only a subset of the semantic domain \mathbb{S} , because some elements of \mathbb{S} may not be described by any model in \mathbb{L} . Additionally, some models in \mathbb{L} may map to the same model in \mathbb{S} due to variations in syntax that may describe the same semantics.

Heinrich et al. in [14] say that a model *conforms* to a metamodel if each model element is an instance of a metamodel element. This “instance of” relation is need not be the same as is encountered in most Object Oriented Languages. Instead, it can be interpreted as “does this model element have all the properties and relationships required for this metamodel element, and does it satisfy all constraints?”

With regards to the concrete syntax of a formalism, the following are some ways of presenting a model [14]:

- Graphical, using diagrams (often referred to as *visual concrete syntax*);
- Tabular, with the use of tables;

- Textual, written down using text (often referred to as *textual concrete syntax*); or
- A combination of the above.

An example of a combined syntax are Statecharts, which have their main syntax as graphical to represent states and transitions, but embed pieces of textual syntax in order to describe operational semantics (see Figure 6.2).

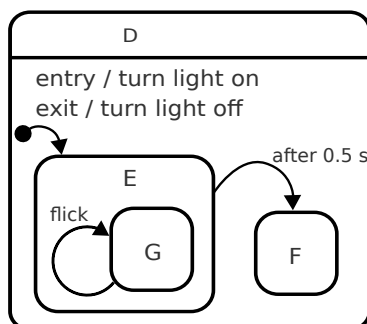


Figure 6.2: Example statechart model, showing a combined visual and textual concrete syntax, from the `statecharts.dev` project (<https://statecharts.dev/>).

A formalism may have one or more concrete syntaxes associated with it. For example, class diagrams may be specified using a graphical representation or a textual notation, or a language may have several “dialects”, such as the `Prompto` language [39]. We note that there also may languages that have very similar syntaxes, but are considered a separate language due to differences in their abstract syntax. For example, `ALGOL` has different implementations that each have their own specialized facilities.

The relationship between the concrete syntax and the abstract syntax is similar to the relationship between the abstract syntax and the semantic domain. That is, the concrete syntax for a language, its conversion into abstract syntax (parsing), and its abstract syntax itself, describe the same principles as the *set of well-formed models* \mathbb{L} , the *semantic mapping* \mathbb{M} , and the *semantic domain* \mathbb{S} respectively. In effect, the concrete syntax is a set of strings, graphs, etc. that describe valid instances of models in a language acting as the domain of the language. The mapping itself is performed through a “*parser*”, which is a specialization of semantic mapping. The only difference lies in the fact that there may be a way of turning an abstract model back into a concrete model through the use of a “*renderer*” or “*pretty-printer*”, the reverse of a parser. See Figure 6.3 for how this looks graphically.

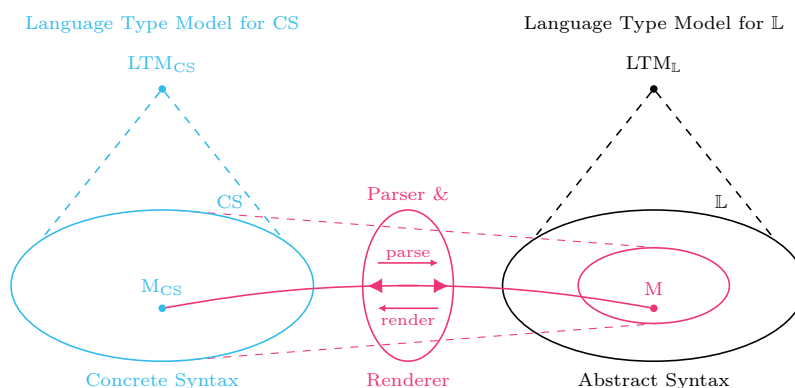


Figure 6.3: Overview of the relationship between abstract syntax and semantics for a formalism.

6.1 Hybrid Languages

Hybrid languages are a combination of one or more languages or formalisms in order to model features of complex systems, while using the most appropriate formalism for each of those features.

The combination of languages requires several aspects to be combined, some of which are:

- The abstract syntax;
- The concrete syntax;
- The semantics;
- The mapping of concrete syntax onto abstract syntax (parsing) and back (rendering);
- The semantic mapping.

For the abstract syntax, this combination (referred to as composing by Mustafiz et al. [21], [22]) might involve adding new classes, relations, properties, etc. or replacing/splitting/... parts. Likewise, for the concrete syntax, languages may need to introduce new syntaxes or modify existing ones. We note that in some cases it may be possible to forego the composing of abstract syntax, and go straight from concrete syntax to semantic domain, such as done by Paredis et al. in [26].

In the MontiCore handbook [30], the term *language embedding* is used to refer to the combining of languages that have been developed independently, but can define a model through their combination. Language embedding is a more specialized version of what

they refer to as *language aggregation*, as the latter only combines the abstract syntax, and the former also combines the concrete syntax of the languages.

In a case study Mustafiz et al. [21] provided the example combination of Timed Finite-State Automata (TFSA), used for describing reactive systems, and Causal Block Diagrams (CBDs), used for describing embedded control systems, into the hybrid TFSA-CBD language. This example (see Figure 6.4) modelled a bouncing ball that could be three distinct states (aside from initializing): (i) falling, (ii) bouncing off the floor, or (iii) getting kicked. Additionally in each state, the ball has distinct behaviour modelled for its speed and position: (i) increasing downwards speed and decreasing x position, (ii) inverting and decreasing speed, and (iii) adding speed.

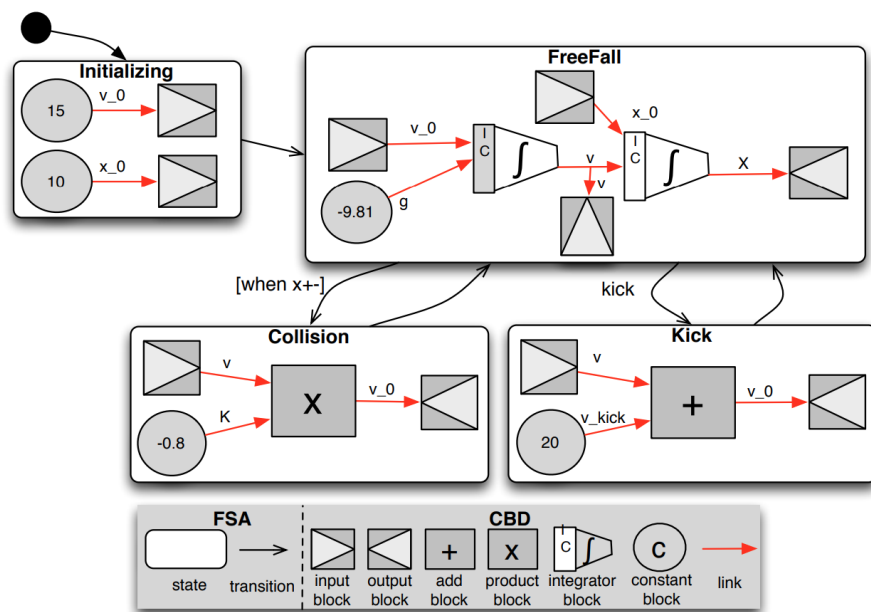


Figure 6.4: Example hybrid TFSA-CBD model, modelling a bouncing ball. From [22].

We note that in this example, the hybrid language uses a composed visual concrete syntax. For our framework, we will instead be focusing on composing textual concrete syntax.

Part II

Model Overview

CHAPTER 7

Overview

In this part we will provide an initial introduction to the WEAVE framework by introducing and detailing the models that surround it, which describe the types and operations involved with the parsing of textual models of hybrid languages. The end goal is to describe how a dynamic multi-language parser may operate, which can then be used as a description for an actual implementation, which we will cover in the next part. Figure 7.1 shows a high-level overview of WEAVE. Central to it is the “orchestrator”, whose job it is to handle the interaction between different parsers, which is the preferred way of dynamically handling multiple languages (see Section 11.5 in Chapter 11).

The following chapters will cover how each part is modeled, starting with Chapter 8 we will describe the structures for representing parse trees, where we will introduce three formalisms that are to be used together: the Tree Definition Formalism (TDF), the Tree Instance Formalism (TIF), and the Tree Construction Formalism (TCF). In Chapter 9 we will cover the ‘Parsers & Generators’ block, which contains a collection of parsers or parser generating objects, and part of the ‘Languages’ block. This chapter describes the basic interfaces required for a parser to be used by WEAVE in order to work with it. Chapter 10 provides a framework for ‘tracing’, which is used to provide a way to store how something was made (token, parse tree, model element, etc.). Lastly, in Chapter 11 we describe how we envision the textual concrete syntax of hybrid languages (see Section 6.1), how the syntax may need to look for a basic implementation, and how parsing should be handled.

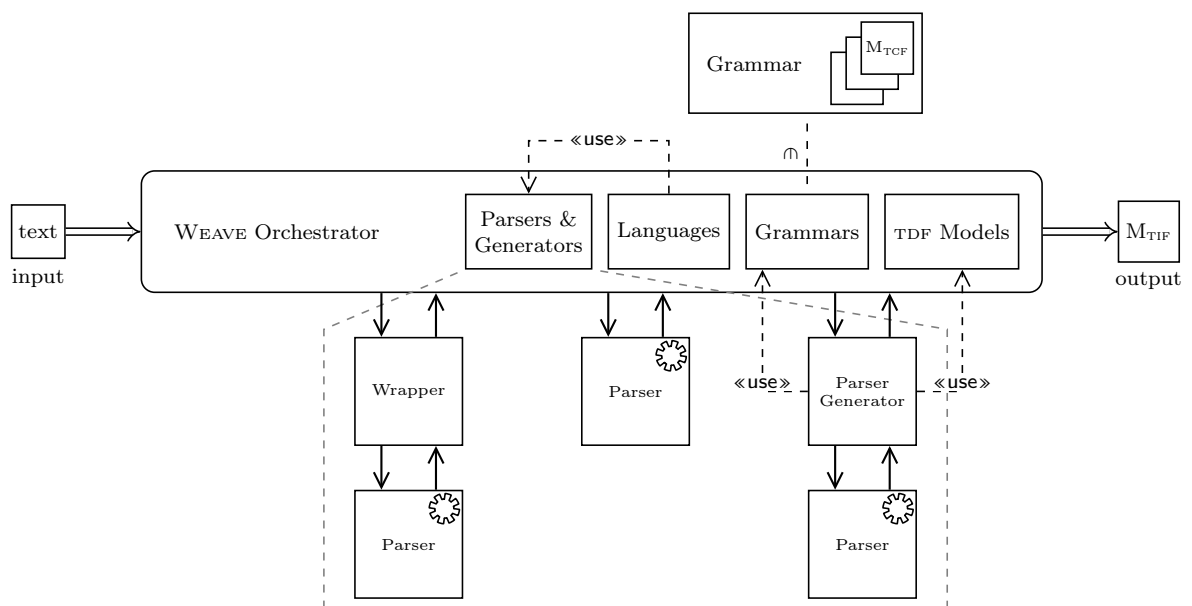


Figure 7.1: Overview of the WEAVE parsing framework.

CHAPTER 8

Parse Trees

As we mentioned near the end of Section 5.2, the classic parse tree representation is simple in its implementation, but due to its simplicity can cause complications further along the line. This is why generally a parser either doesn't generate parse trees themselves and leaves it up to the developer (Yacc, GNU Bison, Python PLY), or they implement their own tree structure (ANTLR4, Python Lark).

Due to our requirement of being able to augment languages these issues are made more apparent, as changing the concrete syntax might bring along a lot of intrusive changes elsewhere the parse tree is consumed if using the classic representation. For example: if we were to augment a single parse rule to always start with some other rule, the tree children all shift right by one, and anything consuming trees for this grammar would require changes to account for this one change. As such, we will immediately require here that any implementation we use or make should always use something that is not tied to the order of occurrence in the definition. Additionally, due to our requirement of being able to combine the parse trees of multiple parsers, a common data structure would be beneficial to the interoperability of the parsers.

We will first compare different implementations of parse trees in some popular parsing tools, and compare some available academic literature on the subject in Section 8.1. We note that we looked at GNU Bison and the Python library PLY (both based on Yacc), but these leave the construction of a tree up completely to the writer of the grammar, and so are of no use for this comparison. Aside from those, we looked at ANTLR4, and the Python library Lark as popular parsing tools. Table 8.1 shows a general comparison of the tools, as well as their supported language targets.

Over in the academic world, we looked at the Object Management Group's (OMG) Abstract Syntax Tree Metamodel (ASTM) [23], Annotated Terms (ATerms) by van den Brand et al. [4], and the markup languages JavaML, CppML, and OOML introduced by

Mamas et al. [16]. We also compared these to the theoretical implementation as given by Aho et al. in [1].

We show the specifications for our proposed parse tree structures in Section 8.2. Then in Section 8.3, we show how a parser can integrate our parse trees through example, followed by a grammar specification in Section 8.4. Finally, we provide a list of considered variations and why we decided not to add them in Section 8.5.

Tool	Generates Parse Trees	Languages	Run-time parser creation
GNU Bison	No, but runs actions on rule match	C, C++	No ^a
ANTLR4	Yes (generates class per non-terminal, tree nodes are class instances)	C#, C++, Dart, Go, Java, JavaScript, PHP, Python, Swift	No ^a
Python Lark	Yes (single tree class for all node instances)	Python	Yes
Python PLY	No, but runs code on rule match	Python	Yes, but using a workaround [28]

^a Counted as no, because it requires running one or more external programs (ex. C compiler and linker), and writing several files to disk. In effect this would require a lot of code to achieve and isn't available natively.

Table 8.1: General overview of the parser generators and parsing libraries we looked at.

8.1 Comparison of Implementations

As we mentioned earlier both **GNU Bison** and the Python library **PLY**, which are based on **Yacc**, leave construction of the parse tree up to the person writing the grammar. As such they are not included in this comparison.

ANTLR4 requires running a code generator that allows targeting different languages, these include but are not limited to **C#**, **C++**, **Java**, and **Python** (see Table 8.1 for a more complete list). Each target language has a similar interface for accessing the parse tree using classes for both interior nodes and terminal nodes. Note that each rule is automatically turned into a class definition for an interior node, while terminal nodes all share a single class definition. The interior nodes store their children in a list structure, and access is provided through both getters that are generated based on rule names, as well as direct access into the list structure (though this should probably be avoided). Making a change to the grammar definition requires running the code generator again.

Parse Tree	Node Storage	Node Type	Children	Child Access
Theoretical (Aho et al. [1])	Tree structure	Node value	Outward edges (ordered by rule)	By index (order)
ANTLR4	Class (Tree)	Class name	List of children	By rule name, by index
Python Lark	Class (Tree)	<code>data</code> field (rule name)	List of children	By rule name, by index
ASTM	Class	Class name	Class fields	Field name
ATerms	APPL term	Function symbol	Term Parameters	Pattern matching
JavaML, CppML, OoML	XML Element	Element name	Child elements (nodes) Attributes (values)	By child element or attribute name
<i>Our tree</i>	Record-like	Name	Record attributes	By name

Table 8.2: Overview of differences in parse tree structure between different tools and documents.

The Python Lark library also makes use of classes for interior nodes and terminal nodes. However, here only a single class is used for all interior node types, with the distinction being made through a field called `data` that stores which production was used to create the node. Child nodes can be accessed through index access, with interior nodes also being accessible through querying by node type. Generating the parser happens at runtime based on a grammar definition, and no files are written to disk unless the result of is saved for performance reasons.

The **Abstract Syntax Tree Metamodel** (ASTM) [23] is defined using the Unified Modeling Language (UML) as a class hierarchy and is split up into a core ‘Generalized ASTM’ (GASTM) and extensions to the core, which are combined under the umbrella term ‘Specialized ASTM’ (SASTM). The GASTM is focused on imperative and object oriented programming languages, so its contents are of little use to us as our focus does not lie towards any specific kind of language at all. Additionally, the purpose of the ASTM is not to be an in-memory representation of parse trees, but it is instead meant as a way of providing interoperability between programs. It does this by defining how the structures should be passed along. However, one of the properties of interest to us is the fact that node instances have named attributes and relationships (children).

Annotated Terms [4] is another solution for providing cross-program compatibility by defining a basic general purpose format for representing parse trees. As opposed to

Parse Tree	Positives	Negatives
ANTLR4	<ul style="list-style-type: none"> + Wide support for languages + Node shapes are fixed at parser generation + Supports typing of node instances (through turning rules into class definitions) + Uses names for accessing child nodes 	<ul style="list-style-type: none"> - Does not allow run-time parser creation
Python Lark	<ul style="list-style-type: none"> + Allows run-time creation of parsers + Uses names for accessing child nodes 	<ul style="list-style-type: none"> - Node shapes are not defined (i.e. no typing)
ASTM	<ul style="list-style-type: none"> + Class diagrams for definition + Child ‘nodes’ are attributes or relationships + Node shapes are defined in model 	<ul style="list-style-type: none"> - Focused on OOP, not general purpose - Definition in a highly technical standard - Not for in-memory representation
ATerms	<ul style="list-style-type: none"> + Simple record-like syntax + Defined operations + General purpose 	<ul style="list-style-type: none"> - Too heavy for our use case - Not for in-memory representation - Node shapes are not defined
JavaML, CppML, OOML	<ul style="list-style-type: none"> + Uses a well known storage format (XML) + Node shapes are defined in a well known format (DTD) 	<ul style="list-style-type: none"> - Slow speed - High memory usage - High disk usage - Not for in-memory representation

Table 8.3: Overview of the positives and negatives of the compared tools and papers.

ASTM, ATerms are generic and not targeted at any specific type of language. This makes it a good candidate when working with any kind of language. For example, the Spoofox Language Workbench makes use of ATerms in order to process and store parse trees.

Lastly, Mamas et al. [16] proposed using **XML** to provide interoperability between programs, and introduced three languages: Java Markup Language (JavaML), C++ Markup Language (CppML), and Object Oriented Markup Language (OOML). The advantage here lies in the usage of a well known language like XML and its Document Type Definition (DTD) language, requiring implementors of the scheme to only need an XML parser (and possibly verifier) to be able to read it. However, according to Anderson in [2] it suffers from slow speed, and high memory and disk usage as opposed to custom built binary formats. And for our use case, storing the parse tree into an XML data structure would be detrimental unless it was to be passed to a different program.

In Table 8.2 we show an overview of the implementation of parse trees in each of the above looked at tools and documents, and Table 8.3 shows the positives and the negatives

for each. Note that while ANTLR4 and Lark both have a lot of positives, they each lack an important part: ANTLR4 does not allow run-time parser creation, and while Lark does, it does not provide the type safety provided by ANTLR4 do to its run-time parser creation. In other words: Lark provides the best solution, but does not provide static or dynamic analysis.

8.2 Implementation

Having compared different existing implementations and what's available, as well as looking at some example use cases, we have the following goals for working with parse trees:

- Detachment of parse trees from the syntax definition, to allow re-use of structure by for example languages with dialects, or when migrating parser (e.g. LALR(1) to PEG).
- Using some form of identification for parse tree children.
- Allowing specification of parse tree structure, and ensuring this structure is obeyed.
- Analyzing and validating the structure of grammar specifications.

Additionally, it would be nice to have the ability to, based on the sort of parse tree node, know what children can be expected or are allowed, and what sort they themselves can be. For example: in a code block we have statements, and a statement could be an expression, assignment, conditional, etc. An advantage of this could be allowing more advanced static analysis of grammar specifications to find accidental mistakes. As such, we also introduce a basic type system to our parse trees:

- A node has a type.
- A node type can 'extend' from other node types, called the parents (the closure on node parents are called the ancestors).
- The type of a node defines the children it can have.
- The children are defined per node-type, and are inherited from all ancestors.
- Each child definition also defines the type of nodes expected.

Note that for the purposes of quick language prototyping we will allow this type system and its corresponding type checks to be disabled, though this will depend on the grammar specification language.

Based on the above goals, we will start with a formalism to define the shapes of parse trees in Sections 8.2.1 and 8.2.2, which we will call the TDF or Tree Definition Formalism. Next we will define a formalism for instances of these definitions in Section 8.2.3, which we will call the Tree Instance Formalism or TIF. We then define a formalism that constructs these instances according to their specification in Section 8.2.4 named the Tree Construction Formalism or TCF. And lastly we will explain how we expect to integrate these formalisms into a parser in Section 8.3.

Figure 8.1 shows how these three formalisms are expected to interact, with a grammar specification containing many TCF models, a TDF model being specified parallel to this, and the output of the parser being a TIF model. Removing the parser from the picture, Figure 8.2 shows a more basic overview of the interactions between the formalisms.

We note that in order to reduce the specificity and allow re-use, the definitions here will be usable without the overarching WEAVE framework. For example, the implementation of leaf nodes (terminal nodes) is left up to the implementation (as will be noted). Additionally, there will be no specific definitions for embedding parse trees from one parser into those of another, as this lies outside the scope of parse trees themselves. Instead, we will introduce extensions to the structures introduced here in Chapter 11 for those concepts specific to our use-case.

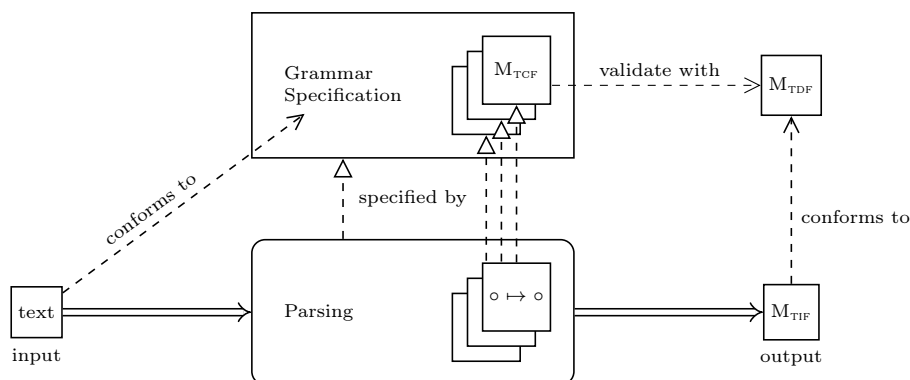


Figure 8.1: Graphical overview of the intended interactions between our parse tree formalisms, grammar specifications and generated parsers.

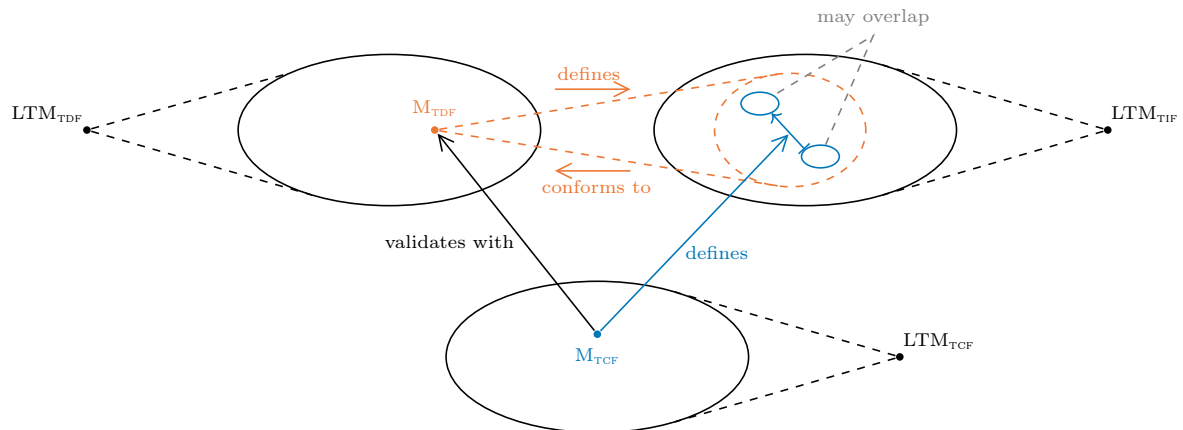


Figure 8.2: Reduced overview of the interactions between TDF, TIF, and TCF.

8.2.1 Tree Definition: Abstract Syntax

Figures 8.3 and 8.4 show an overview of the abstract syntax of our parse tree definition formalism, which we will call TDF (Tree Definition Formalism). The base interface is `NodeType`, which is the base type for all node kinds. `NodeType` and its basic derived types are visualized in Figure 8.3. Every node type has a ‘name’ which should be unique in the context of a language definition.

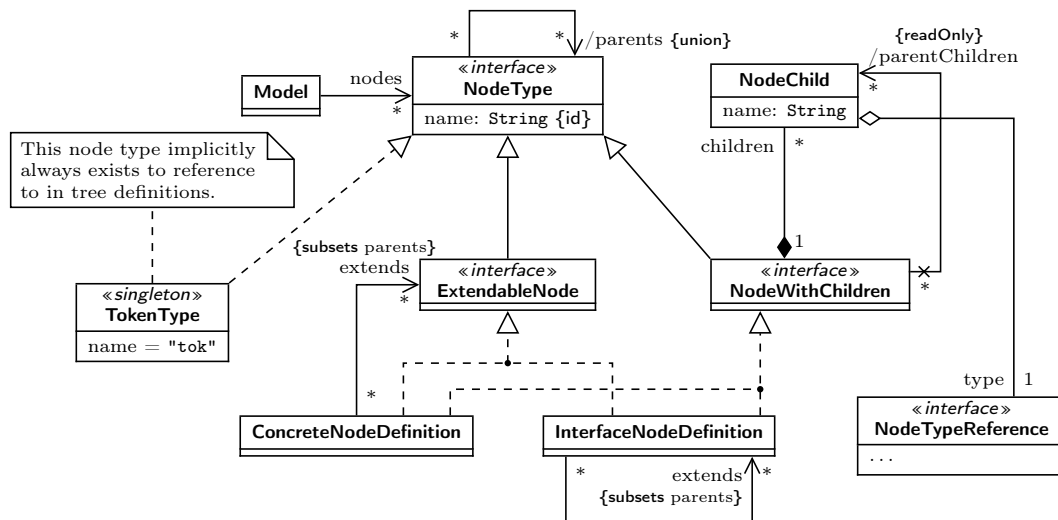


Figure 8.3: Class diagram showing the abstract syntax of the base building blocks of parse tree definitions.

`TokenType` is a node type with the name `tok` that implicitly always exists and has no children, instead realizations of this node type have a ‘value’ that in general represents parts of the input of the parser. For example the identifier ‘x’ in Figure 5.4 is a token

of type 'ID' with value "x" and could serve as a realization of `TokenType`. In the case of a scannerless parser, `TokenType` could represent a sequence of characters from the input stream instead. It could even represent a “plain string”, meaning only the text without any metadata attached by a lexer or parser, for example if loaded from a pre-defined TIF model (see Section 8.2.3) from disk or memory, or constructed by hand.

`ConcreteNodeDefinition` and `InterfaceNodeDefinition` both allow language designers to define their own shapes of parse trees. ‘Concrete’ nodes define the shape a node must conform to, and while ‘interface’ nodes do this as well, conceptually they serve a different kind of construct.

The idea behind ‘interface’ nodes is that they cannot be realized themselves (like interfaces in many programming languages) and instead define a format that other nodes need to conform to. This can optionally include defining some children that are required. Take for example the grammar of Lark grammars [31], [32], which at its root allows the following constructs: parser rules, token definitions, ignore statements, import statements, override statements, and declare statements. If we were to model this in TDF, each of those constructs would have their own node definition, but the root node would not need to have knowledge of each different kind of construct. This is where ‘interface’ nodes would come in: we would define an ‘interface’ node named `RootElement`, which the node definition for each root construct would “extend” or “inherit” from. The root node then would only need to have a single child: a list of `RootElement` nodes. We could even go further and define an ‘interface’ node named `RootStatement` that “extends” `RootElement`, which the four statements node definitions would instead inherit from.

The existence of the `ExtendableNode` interface is to facilitate the ‘extends’ relationship for ‘concrete’ nodes, which can extend either other ‘concrete’ nodes or ‘interface’ nodes, but not say for example `TokenType`. The `NodeWithChildren` interface on the other hand defines that a node type can have children, which are instances of `NodeChild`, are identified through a ‘name’ attribute and have a ‘type’ relation to a `NodeTypeReference`.

We note that there are two ‘extends’ relations in Figure 8.3, both are annotated as “{subsets parents}”, which reference the derived union ‘parents’ relationship on `NodeType`. This is UML syntax that indicates that each instance of the ‘extends’ relationship also causes an instance of the ‘parents’ relationship to exist from and to the same source and target respectively of the ‘extends’ relationship [25] (i.e. they are subsets).

We also note that there exists the derived relationship ‘parentChildren’ from `NodeWithChildren` to `NodeChild`, for which the implementation is given in OCL in Listing 8.1, as well as the implementation of the ‘isEquivalent’ operation used for constraint validation further along in Listing 8.2.

The `NodeTypeReference` is itself an interface as well, allowing for different structures to be contained in a node. In our implementation we decided to go with the following kinds:

- `BaseTypeReference` requires there to be exactly one child of the referenced type under its given name.

```

-- Collect all defined children for a NodeWithChildren
context NodeWithChildren::parentChildren : Set(NodeChild)
derive: self.parents.children->flatten()->union(self.parents.parentChildren->flatten())

-- Equivalency for BaseTypeReference
context BaseTypeReference::isEquivalentent(other : NodeTypeReference) : Boolean
body: other.ocIsTypeOf(BaseTypeReference) and self.type = other.type

-- Equivalency for OptionalTypeReference
context OptionalTypeReference::isEquivalentent(other : NodeTypeReference) : Boolean
body: other.ocIsTypeOf(OptionalTypeReference) and self.type = other.type

-- Equivalency for ListTypeReference
context ListTypeReference::isEquivalentent(other : NodeTypeReference) : Boolean
body: other.ocIsTypeOf(ListTypeReference) and self.type = other.type

```

Listing 8.1: Definitions for the abstract syntax of TDF defined in OCL [24].

- `OptionalTypeReference` requires there to be either one child of the referenced type, or no child at all, under its given name.
- `ListTypeReference` requires there to be a list of children of the referenced type under its given name. The list can have zero, one, or more children in it.

Figure 8.4 shows an overview their definition in terms of their abstract syntax. We note that it should be possible to add more kinds of `NodeTypeReference`, but for simplicity we decided to only go for those that directly translate to patterns in Extended Backus-Naur Form grammar definitions (repetition and optionality).

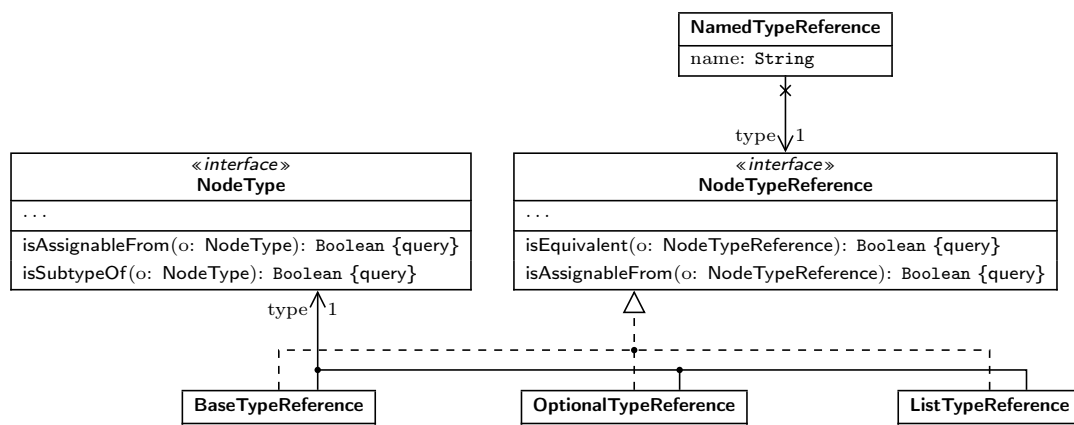


Figure 8.4: Class diagram showing the implementations of the `NodeTypeReference` interface.

Next, we have some constraints that any valid TDF model should conform to in Listing 8.2. First, there mustn't be any loops in the node type hierarchy, i.e. a node mustn't be its own ancestor. Second, children must have a unique name, both within a single node definition, and compared with all ancestors. And if a node re-declares a child with the same name, the definition must be “compatible” (for now this is interpreted as having the exact same type, but in the future this could for example be further refined to allow sub-types to be considered compatible).

Implementation note We note that the theoretical implementation in Listing 8.2 might be okay with a simple closure algorithm for detecting loops in the type hierarchy, but recommend an algorithm that finds strongly connected components (with vertices node types, and edges the “parents” relation). The advantage of using strongly connected components is that, unlike the closure algorithm or topological sort (which does not produce a sort if there is a loop) and unlike a depth-first or breath-first search (which only return parts of the loop), the resulting strongly connected components with more than one element give all type loops and gives all types involved in each loop. The only type loop not discovered using this sort of algorithm are self-loops, where a type is defined as extending itself, a simple check should suffice to cover this case.

```

-- Ensure there are no cycles in the node type hierarchy.
-- Note that the parents association only does direct ancestors
context NodeType
inv noCycles:
  self.parents->closure(parents)->excludes(self)

-- 1. Ensure that no two childs have the same name
-- 2. Ensure that no child shadows a child in a parent
context NodeWithChildren
inv noDuplicateNames:
  self.children->forall(c1, c2 : NodeChild |
    c1 <> c2 implies c1.name <> c2.name)
inv noShadowChildren:
  self.children->forall(c1 : NodeChild |
    self.parentChildren->forall(c2 : NodeChild |
      c1.name = c2.name implies c1.type.isEquivalent(c2.type))

```

Listing 8.2: Constraints for the abstract syntax of TDF defined in OCL [24].

Lastly, we have some operations that support the typing system:

- In Listing 8.3 we define some operations ‘isAssignableFrom’ on ‘NodeType’ and derivatives, and on ‘NodeTypeReference’ and derivatives. This operation defines a relation “X isAssignableFrom Y” that holds if a value of type Y can be given where one of type X is expected.

- In Listing 8.4 we define the operation ‘isSubtypeOf’ on ‘NodeType’ and derivatives. This operation checks whether the given ‘other’ type is an ancestor in the type hierarchy, and is for example used by the ‘isAssignableFrom’ operation mentioned above, and the ‘type intersection’ algorithm in the next item.
- In Listing 8.5 we define an algorithm to get the ‘type intersection’ as the most specific common ancestors of a set of given types. This algorithm is used to deduce the type of construction operations as we will see in Sections 8.2.4 and 8.2.5.

```

-- Assignability for NodeType
context NodeType::isAssignableFrom(other : NodeType) : Boolean
body: other.isSubtypeOf(self)

-- Assignability for TokenType
context TokenType::isAssignableFrom(other : NodeType) : Boolean
body: other.ocIsTypeOf(TokenType)

-- Assignability for BaseTypeReference
context BaseTypeReference::isAssignableFrom(other : NodeTypeReference) : Boolean
body: other.ocIsTypeOf(BaseTypeReference) and self.type.isAssignableFrom(other.type)

-- Assignability for OptionalTypeReference
context OptionalTypeReference::isAssignableFrom(other : NodeTypeReference) : Boolean
body: (other.ocIsTypeOf(OptionalTypeReference) or other.ocIsTypeOf(BaseTypeReference))
and self.type.isAssignableFrom(other.type)

-- Assignability for ListTypeReference
context ListTypeReference::isAssignableFrom(other : NodeTypeReference) : Boolean
body: other.ocIsTypeOf(ListTypeReference) and self.type.isAssignableFrom(other.type)

```

Listing 8.3: Specification of assignability operations of TDF defined in OCL [24].

```

-- Subtyping for NodeType
context NodeType::isSubtypeOf(other : NodeType) : Boolean
body: False

-- Subtyping for ConcreteNodeDefinition
context ConcreteNodeDefinition::isSubtypeOf(other : NodeType) : Boolean
body: self = other or self.extends->exists(parent | parent.isSubtypeOf(other))

-- Subtyping for InterfaceNodeDefinition
context InterfaceNodeDefinition::isSubtypeOf(other : NodeType) : Boolean
body: self = other or self.extends->exists(parent | parent.isSubtypeOf(other))

```

Listing 8.4: Specification of the subtyping operations of TDF defined in OCL [24].

```

algorithm typeIntersection
  input types: Set(NodeType)
  output intersection: Set(NodeType)
  require: types.notEmpty()

  -- 1. Collect all ancestors as subgraphs of the type hierarchy.
  for each t: NodeType in types do
     $A_t$ : Set(NodeType) = all ancestors of t
  end

  -- 2. Get the intersection of the ancestors, giving us the shared ancestors.
  -- This gives us a new subgraph of the type hierarchy.
   $I$ : Set(NodeType) =  $\bigcap_t S_t$ 

  -- 3. Get all types that do not have child types in the shared ancestors set.
  -- Equivalent to getting the vertices in our subgraph with no incoming edges.
  for each a: NodeType in I do
    if not exists b: NodeType in I
      where a <> b and b.isSubtypeOf(a) then
        intersection.put(a)
    end
  end

```

Listing 8.5: Algorithm used for getting the most specific common ancestors of some nodes.

The TDF formalism is very similar to the UML class diagrams formalism [25], but works with a simpler definition that does not map completely on class diagrams. Later on we have an example TDF model instance (Figure 8.7), and we show how this would map to UML class diagrams in Figure 8.8. Table 8.4 shows a comparison of TDF concepts with class diagram concepts. A non-exhaustive list of things that are different to or are missing from TDF compared to class diagrams is:

- Multiplicity of properties is limited to either one, one or zero, or zero or more.
- Constraints cannot be specified.
- Nodes do not have any behavior such as operations or signals attached to them. They are a pure data storage formalism.
- The only supported primitive type is String (`TokenType` represents a string from the input).

Class Diagrams	TDF
Classifier	<code>NodeType</code>
Class Definition	<code>ConcreteNodeDefinition</code>
Interface Definition	<code>InterfaceNodeDefinition</code>
Class Generalization and Interface Realization	<code>ConcreteNodeDefinition</code> extends relation
Interface Generalization	<code>InterfaceNodeDefinition</code> extends relation
Class / Interface Properties	<code>NodeWithChildren</code> children association
String Primitive	<code>TokenType</code>
Property Multiplicity 1	<code>BaseTypeReference</code>
Property Multiplicity 0..1	<code>OptionalTypeReference</code>
Property Multiplicity 0..*	<code>ListTypeReference</code>

Table 8.4: Relation between the class diagrams formalism and the TDF formalism.

8.2.2 Tree Definition: Concrete Syntax

Having defined the abstract syntax and its intended meaning above, we'll now define a visual and a textual concrete syntax for TDF. The goal for the visual syntax is to be usable in this document, while the textual syntax will be used for implementations.

Figures 8.5 and 8.6 show a fairly complete overview of the both the visual and textual syntax with the abstract syntax being shown centrally. The concrete visual syntax is

shown through $\text{--}\langle\langle\text{syntax}\rangle\rangle\text{--}\rightarrow$ associations under the abstract syntax, and the concrete textual syntax through a note above. In each image, extra information such as surrounding type definitions are grayed out to not distract from what each image is supposed to convey.

- The node type is placed first in both syntaxes, with *node* being used for instances of `ConcreteNodeDefinition`, and *interface* for instances of `InterfaceNodeDefinition`. For ‘concrete’ instances the *node* specifier can be omitted in the textual syntax.
- Next is the name of the type definition.
- The “children” relation is shown in the textual syntax with parentheses surrounding them and being separated by commas. In the visual syntax this is done by putting them all in a ‘box’, each relation taking up a line.
- For each child relation, the following holds:
 - The type of a node is given after the name of a node, and is separated from it through a colon ‘:’.
 - Regular or ‘base’ “always one child” children just have the node type.
 - Optional children have a question mark ‘?’ prepended to their type.
 - List children have square brackets ‘[]’ appended to their type.
- The “extends” relations in the textual syntax are denoted by putting a colon ‘:’ after the children definition, and a list of the ‘extended’ types separated by commas.
- In the visual syntax, the “extends” relations can be shown by either using UML generalization associations ($\text{---}\triangleright$) such as on the left side, or by putting the extended node type(s) below the node name such as on the right side. Both versions are equivalent and can be used as is felt appropriate (for example, to reduce visual clutter or emphasis a relationship).

Because the TDF formalism is very similar to the UML class diagrams formalism [25], we chose to keep the visual syntax similar as well. This way, it should help the reader more easily understand the diagrams in this document. The textual syntax on the other hand only slightly resembles Python function and class definitions and Java record type definitions, but is otherwise not based off any existing language or syntax. In Section 8.4, Figure 8.27 we give a grammar specification for the textual syntax.

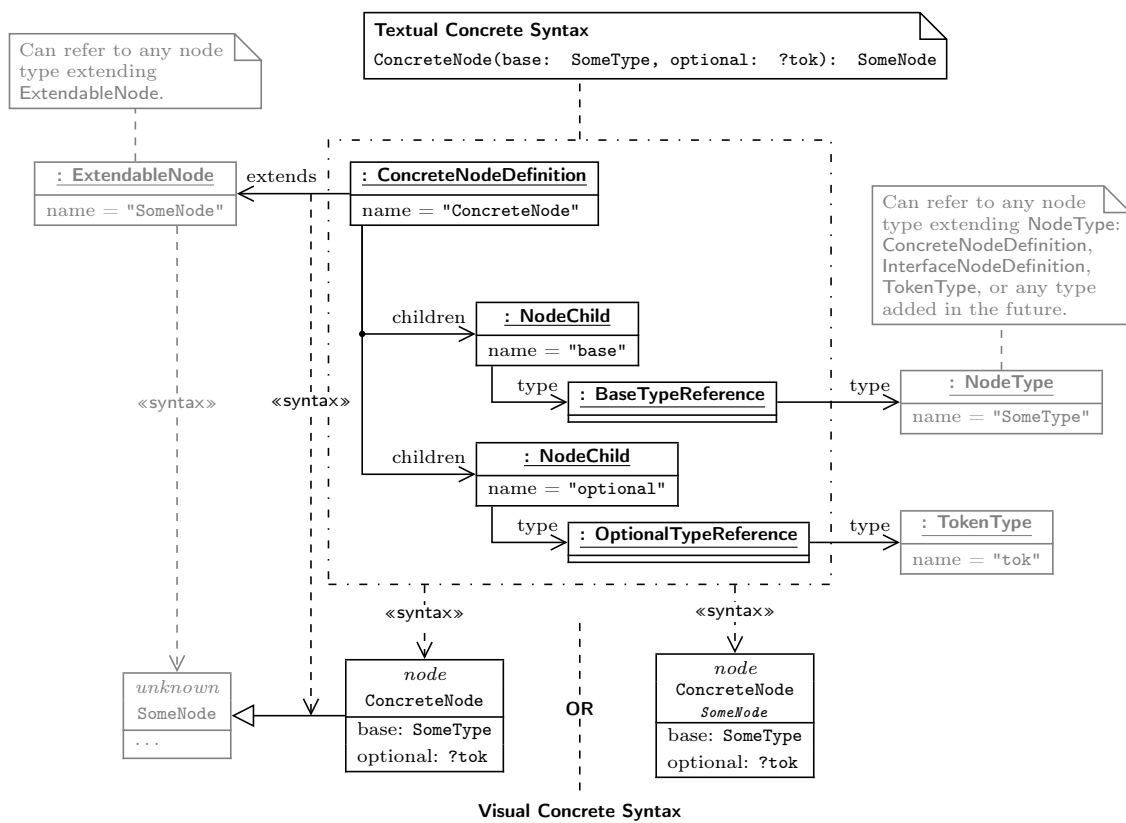


Figure 8.5: Showcase of the visual and textual concrete syntax of some an example `ConcreteNodeDefinition` named `ConcreteNode` and its object diagram representation.

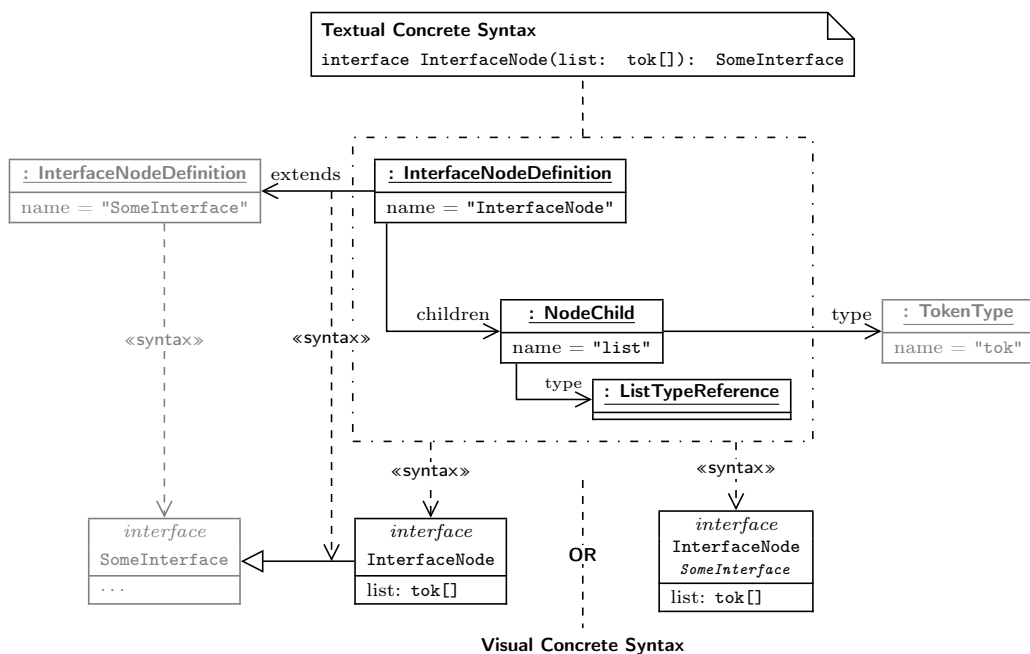


Figure 8.6: Showcase of the visual and textual concrete syntax of some an example `InterfaceNodeDefinition` named `InterfaceNode` and its object diagram representation.

Example

To further show this syntax, we show an example parse tree definition using TDF based on the grammar defined in Listing 5.1. Listing 8.6 shows how this would be defined using the textual syntax, while Figure 8.7 shows the same definition using the visual syntax. Note that the `Int` node has its value as a `str` type, indicating that it doesn't get parsed to an actual value but just represents a part of the input. Note as well that while `Addition` has two children `lhs` and `rhs` (respectively left- and right-hand side), and `Multiplication` instead has one child `operands`. Using a list like this allows rolling up several operations into a single node as opposed to requiring a chain of nodes of the same type.

```
Assignment(lhs: Id, value: Operation)

interface Operation()
Addition(lhs: Operation, rhs: Operation): Operation
Multiplication(operands: Operation[]): Operation

interface Atom(): Operation
Id(name: tok): Atom
Int(value: tok): Atom
```

Listing 8.6: Example textual parse tree definition for the language defined in Listing 5.1 using the TDF formalism.

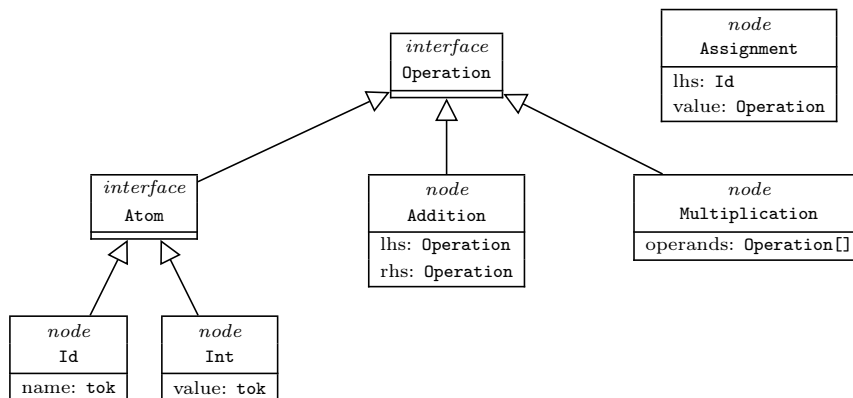


Figure 8.7: Example visual parse tree definition for the language defined in Listing 5.1 using the TDF formalism.

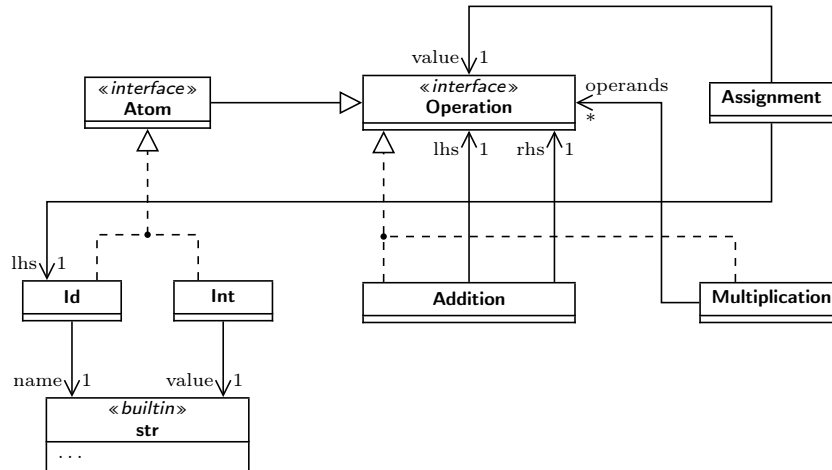


Figure 8.8: Figure 8.7 turned into an equivalent class diagram.

8.2.3 Tree Instances

As we mentioned in the previous section, TDF looks a lot like the UML class diagrams formalism, and indeed we will continue this parallel with our Tree Instance Formalism (TIF), are very similar to the UML object diagrams formalism. Due to the simplicity of this formalism, we will keep this section short.

Figure 8.9 and Listing 8.7 show the abstract syntax and constraints, with `NodeInstance` instances representing a node that's been constructed (`ConcreteInstance`) or parser input (`TokenInstance`). Instances of `ConcreteInstance` themselves can have children that have `NodeInstances` themselves, the types `BaseValue`, `OptionalValue` and `ListValue` are the realizations of the `BaseTypeReference`, `OptionalTypeReference` and `ListTypeReference` types from TDF respectively (not pictured).

We note that we leave the implementation of `TokenInstance` up to the parser, as the scope of our parse tree formalisms is limited on the construction of parse trees and its structure, but not on the leaf nodes, i.e. the terminals.

In Figure 8.10 we see the same parse tree as in Figure 5.4 for the string `x = 10 + y * 5` being represented visually, but making use of the TDF model in Listing 8.6 and Figure 8.7. The same tree is also represented textually in Listing 8.8. We say that both the textual and visual trees conform to the TDF parse tree definition in Listing 8.6 (textual) or Figure 8.7 (visual).

Not shown in our examples is the syntax for an optional child. In the case of there being a value, its syntax is the same as a mandatory child. If there is however no value, the child assignment is omitted (textual) or the association edge is left out (visual).

Just as for TDF, Figure 8.28 in Section 8.4 contains a grammar specification for the concrete textual syntax of TIF.

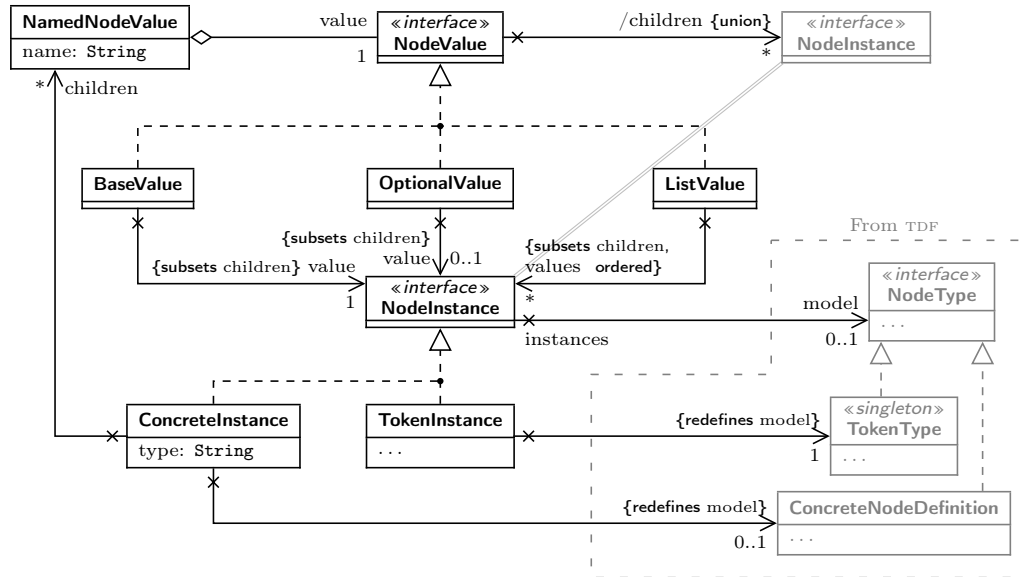


Figure 8.9: Abstract syntax definition for TIF. Shaded and enclosed in a dashed region are types from TDF.

-- *Ensure there are no cycles, and children contains no duplicate names.*

context ConcreteInstance

inv noCycles:

```

self.children.children->flatten()
  ->selectByKind(ConcreteInstance)
  ->closure(c: ConcreteInstance | c.children.children->flatten()
    ->selectByKind(ConcreteInstance))
  ->excludes(self)

```

inv noDuplicateNames:

```

self.children->forAll(c1, c2 : NodeValue |
  c1 <> c2 implies c1.name <> c2.name)

```

Listing 8.7: Constraints for the abstract syntax of TIF defined in OCL [24].

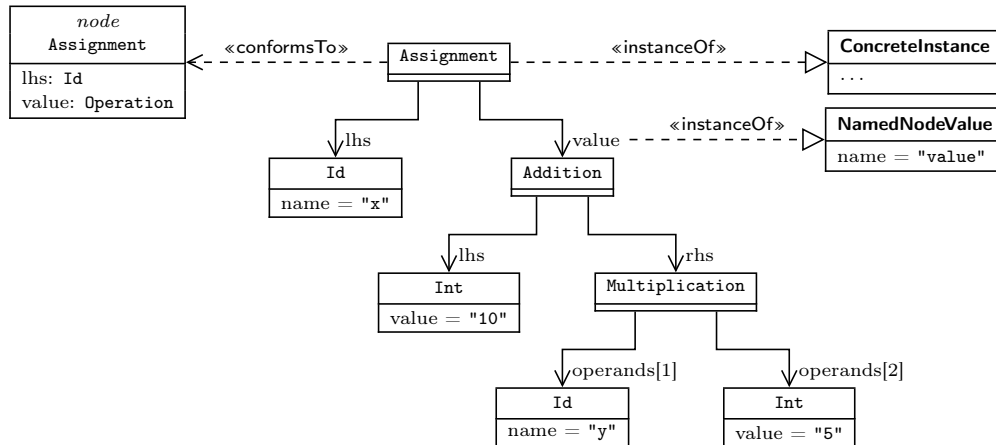


Figure 8.10: Example visual parse tree representation for the parse tree in Figure 5.4 with the parse tree definition from Figure 8.7, showing the relation to TDF and the abstract syntax of TIF. The original text is $x = 10 + y * 5$.

```

Assignment (
  lhs = Id(name = "x"),
  value = Addition(
    lhs = Int(value = "10"),
    rhs = Multiplication(operands = [
      Id(name = "y"),
      Int(value = "5")
    ])
  )
)

```

Listing 8.8: Example textual parse tree representation for the parse tree in Figure 5.4 with the parse tree definition from Figure 8.7. The original text is $x = 10 + y * 5$.

Graph Representation

An alternative representation for TIF models can be done through labeled graphs, the purpose of this representation is solely for representing operations in the TCF formalism in the next section. Figures 8.11 and 8.12 show how these would look like. Note that in Figure 8.12b **OptionalValue** can either have an outgoing edge labeled ‘node’ or no outgoing edge to indicate the presence or absence of a ‘value’, and in Figure 8.12c **ListValue** is represented as a repeating data structure, with the base case being a node without value. This repeating nature is similar to a linked list data structure, and can be envisioned similar to the way users of **Prolog** or **Haskell** work with lists.

Also note that there are different shapes for the nodes in the graph, these are used solely as a quick indicator of what sort of data it represents, and what outgoing edges can be expected from it. These shapes are: **NodeValue** \square , **NodeInstance** \circ , **ListValue** list bits \blacklozenge , or generic nodes \bullet , as well as nodes with text in them.

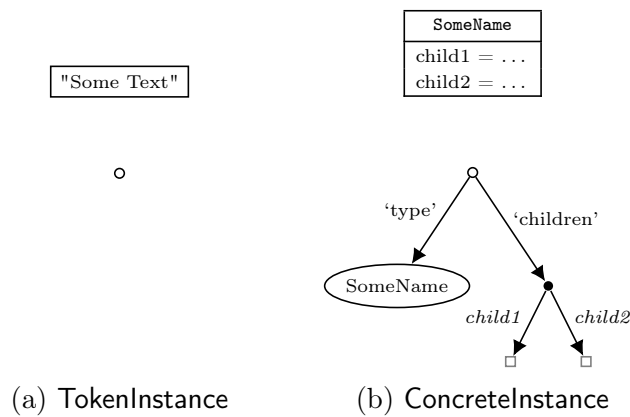


Figure 8.11: Graph based representations for node instances.

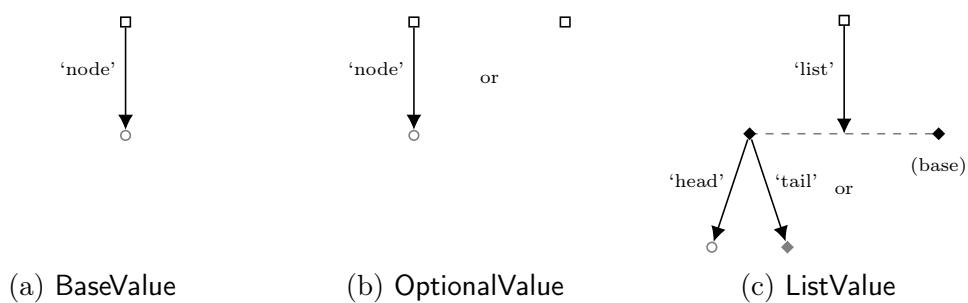


Figure 8.12: Graph based representations for node values. Note that **ListValue** is represented as a repeating structure with the head being the n^{th} element (at depth n) and the tail holding all next elements, and the base case being a node with no outward edges.

8.2.4 Tree Construction

Having now defined a way to define parse tree structures with TDF in Sections 8.2.1 and 8.2.2, and a way to represent instances of parse trees with TIF in Section 8.2.3, we will now look at constructing instances of TIF models with the Tree Construction Formalism (TCF).

A TCF model defines a mapping from a collection of TIF `NamedNodeValues` to a `NodeValue`, these mappings may take any, all, or no parts of the input model to construct the output model. The idea then is for a parser making use of our formalisms to not have a singular TCF model for the entire parse process, but to have one for each non-terminal, for each top-level choice in a non-terminal, or maybe at some other level, depending on the parser.

TCF focuses solely on the construction of nodes through transformation at the end of a successful match. As such, there are some gaps that need to be filled in by any parser making use of it, such as when a production contains Extended Backus-Naur Form (eBNF) concepts. For example, $\langle rule \rangle^*$ would need to aggregate the values returned by each $\langle rule \rangle$ invocation into a `ListValue`. In Section 8.3 we explain how this should be done.

Figure 8.13 shows the abstract syntax of the base interface for all possible operations. Note that the `makeBuilder` operation defined in `ConstructionOperation` creates instances of `TreeBuilder` objects, we will describe the behaviour of these for each realization of `ConstructionOperation` in the next section. Figure 8.14 shows the realizations for operations that work on plain values or nodes, and Figure 8.15 shows the realizations for operations that operate on lists.

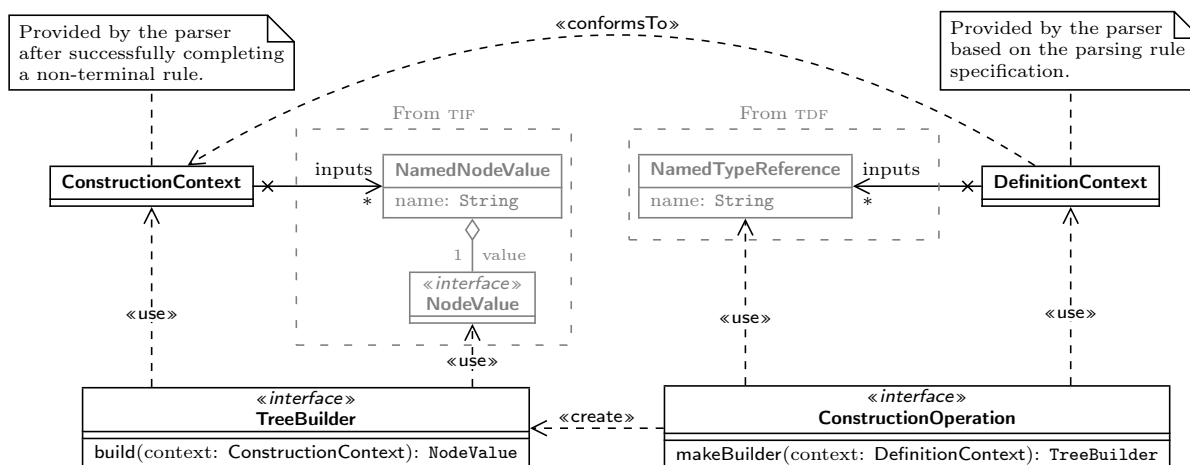


Figure 8.13: The base abstract syntax for the `ConstructionContext` of TCF.

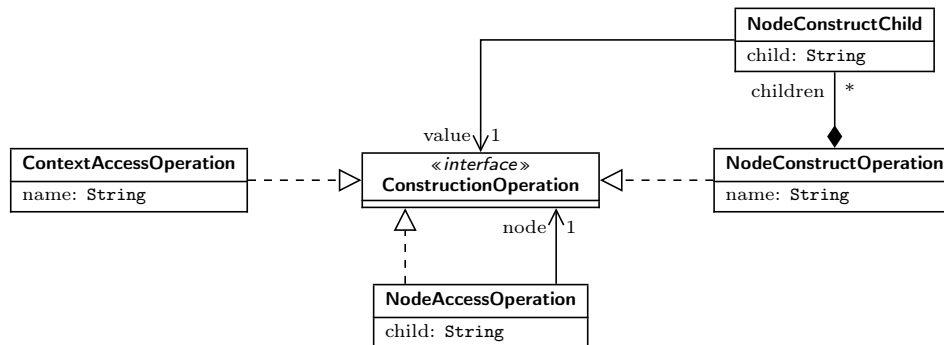


Figure 8.14: Abstract syntax for operations that access or make nodes in TCF.

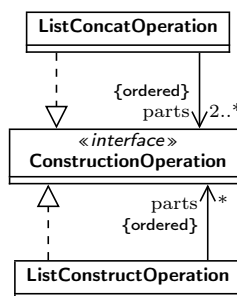


Figure 8.15: Abstract syntax for operations dedicated to lists in TCF.

8.2.5 Tree Construction: Operations

We will explain the defined operations in this section based on:

- Prose explaining the operation.
- The textual concrete syntax.
- In-place graph based transformations, such as used by Van Tendeloo et al. in [38]:
 - Solid black $\xrightarrow{\text{'edge'}} \bullet$: matched vertices and edges.
 - Thick solid green $\xrightarrow{\text{'edge'}} \bullet$: added vertices and edges.
 - Dashed blue $\xrightarrow{\text{'edge'}} \bullet$: removed vertices and edges.
 - Red dotted $\xrightarrow{\text{'edge'}} \bullet$: negative matched vertices and edges (if matched, the transformation fails).
- Execution schedules for the graph transformations, given using the MoTif (Modular Timed graph transformation) language by Syriani et al. [35].

The starting state of the graph transformation for any given TCF model will conform to the shape as given in Figure 8.16a, and the expected end shape is given in Figure 8.16b.

Note that the only difference between the expected start and end shapes is the presence of a ‘value’ edge on the vertex pointed to by ‘current’. I.e. the “result” will be stored under this ‘value’ edge.

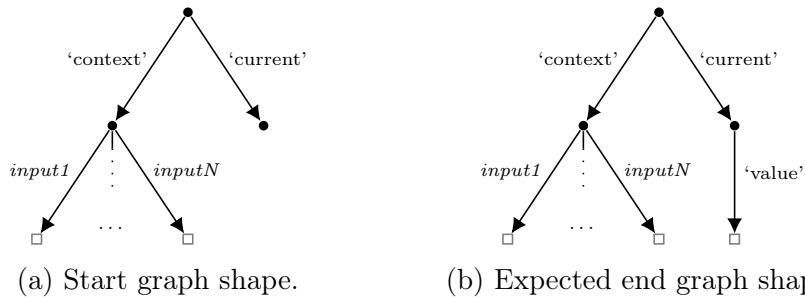
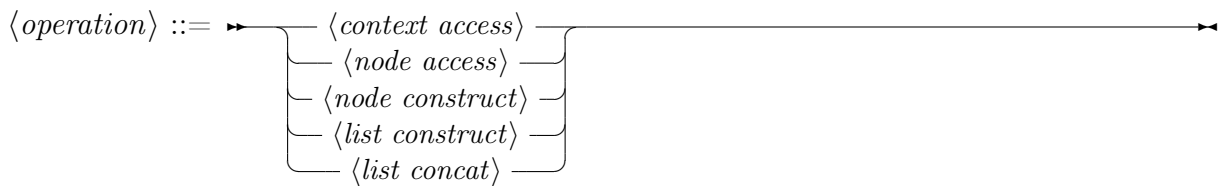


Figure 8.16: Graph based representation of the `ConstructionContext`, and its expected start and end shape.

For the textual concrete syntax, the base grammar rule $\langle operation \rangle$ is defined in Listing 8.9. This rule serves as the start symbol for any TCF model defined textually. As with TDF and TIF, the complete grammar specification for TCF can be found in Section 8.4 Figure 8.28.



Listing 8.9: Root grammar for the textual concrete syntax of TCF.

ContextAccessOperation

Shown in Figure 8.17. Takes a name/value from the definition/construction context, i.e. the inputs of the rule, or the outputs of the child rules.

It has one parameter: the **name** of the value to get. If there exists a definition in the context with the given **name**, then the type of this operation is the type in the definition context, and the value is taken from the construction context. Otherwise, this operation is invalid.

Represented graphically, this operation can be described as adding a ‘value’ edge to the given name under the context (see Figure 8.17b). The execution schedule is given in Figure 8.17c.

The textual concrete syntax is given in Listing 8.10.



Listing 8.10: Textual concrete syntax for `ContextAccessOperation`.

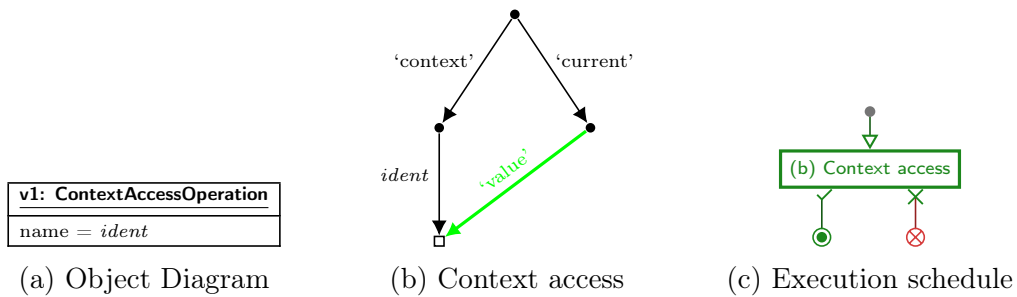


Figure 8.17: Graph based representation of the ContextAccessOperation and accompanying execution schedule.

NodeAccessOperation

Shown in Figure 8.18. Accesses a name/value from a sub-operation, e.g. from a ContextAccessOperation.

It has two parameters: the sub-operation **node**, and the **name** of the child to access. If the type of the sub-operation is a BaseTypeReference referencing to a NodeWithChildren, and the type has a child named by **name**, then the type of this operation is the type of that child, and the value is taken from the sub-operation. Otherwise, this operation is invalid.

Represented graphically, this operation can be described as moving the 'value' edge to a child name (see Figure 8.18b). The execution schedule is given in Figure 8.18c.

The textual concrete syntax is given in Listing 8.11.

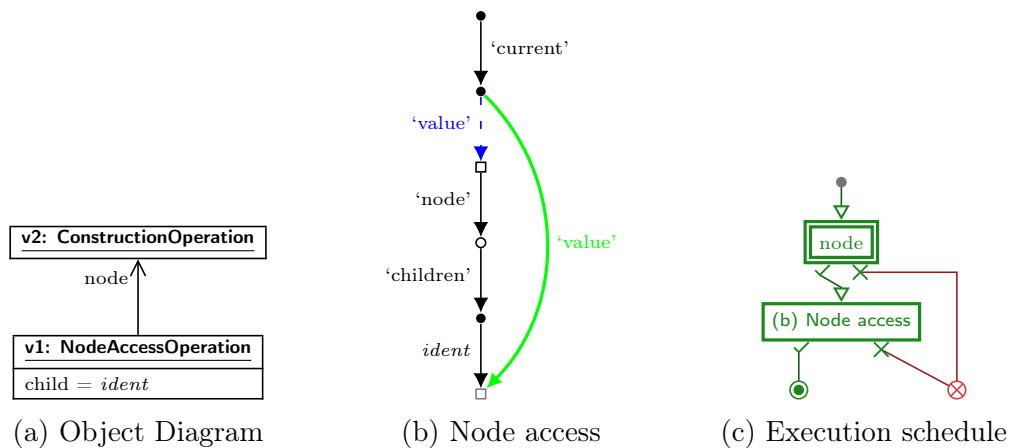


Figure 8.18: Graph based representation of the NodeAccessOperation and accompanying execution schedule.

$\langle context\ access \rangle ::= \blacktriangleright \langle operation \rangle - \text{'.'} - ID \blacktriangleright$

Listing 8.11: Textual concrete syntax for `NodeAccessOperation`.

NodeConstructOperation

Shown in Figure 8.19. Creates a new `ConcreteInstance`.

It has two parameters: the type of the node `name`, and the `children` to put under it as sub-operations qualified by `child` names. The following must hold:

- The type referenced by `name` exists.
- There are no `children` given that are not defined in the type (i.e. no excess children).
- For every child in the type not given in `children` the child type is an `OptionalTypeReference`.
- For every child in `children`, the type of `child` in the type definition must be “assignable from” the type of the sub-operation qualified by `child` (“assignable from” is specified in Listing 8.3).

If all these hold, then the type of this operation is a `BaseTypeReference` with the referenced type given by `name`, and the value will be a newly constructed `ConcreteInstance` of the given type with the given `children` set. Otherwise, this operation is invalid.

This operation can be represented graphically as a multi-phase operation with the execution schedule given in Figure 8.19f:

- Setup the resulting node (Figure 8.19b).
- For each child in the type, either run Figure 8.19g if specified in `children` or Figure 8.19h if not.
- Put the resulting node as the ‘value’ (Figure 8.19e).

Note that the execution schedule in Figure 8.19f is generated based on operation specification, and should therefore be considered as a “template” rather than a fixed unit.

The textual concrete syntax is given in Listing 8.12. Note that the second production in $\langle node\ construct\ child \rangle$ is a shorthand for a `ContextAccessOperation` with ID assigned to ID.

$$\langle node\ construct \rangle ::= ID - '(' - \overbrace{\langle node\ construct\ child \rangle}^{','} - ')'$$

$$\langle node\ construct\ child \rangle ::= \overbrace{ID - '=' - \langle operation \rangle}^{ID}$$

Listing 8.12: Textual concrete syntax for `NodeConstructOperation`.

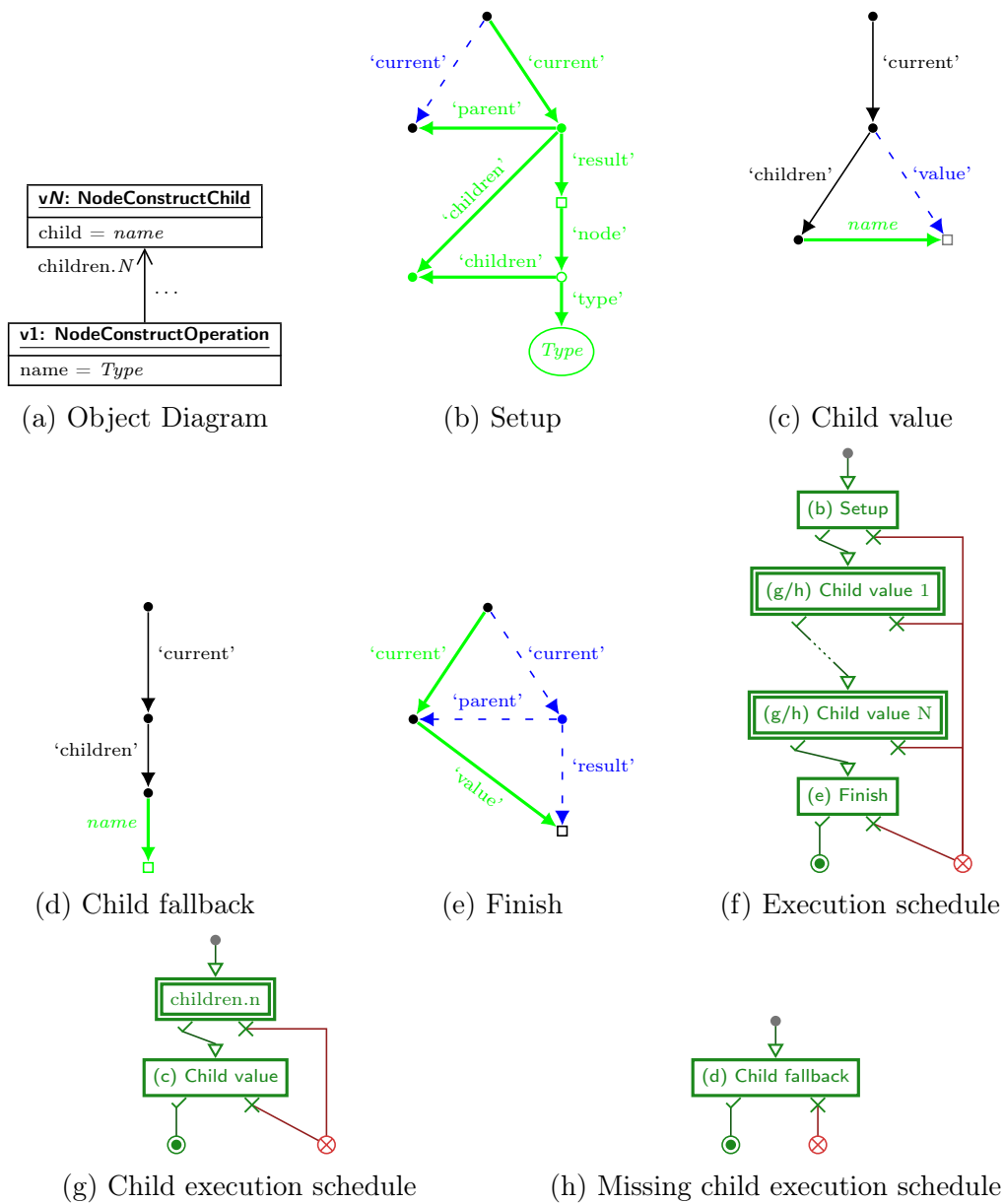


Figure 8.19: Graph based representation of the phases of a NodeConstructOperation and accompanying execution schedules.

ListConstructOperation

Shown in Figure 8.20. Creates a new `ListValue` from a list of `BaseValues`.

It has one parameter: the `parts` that make up the list as a list of sub-operations. For it to be valid, the type of each sub-operation must be a `BaseTypeReference`, and there must be a unique overlapping type for the referenced types, i.e. the type intersection of the referenced types for each part must be exactly one type. To calculate this type intersection, we use the ‘typeIntersection’ algorithm defined in Section 8.2.1 Listing 8.5.

The type intersection is required in order to assign a type to the expression, and reduce complexity that may be introduced if more than one most specific shared supertype exists. Future improvements may relax this restriction or change typing to remove this issue completely.

The type of this operation is a `ListTypeReference` with the referenced type the unique most specific shared supertype of the sub-operations.

This operation can be represented graphically as a multi-phase operation with the execution schedule given in Figure 8.20e:

- Setup the resulting node (Figure 8.20b).
- For each element in `parts`, run the sub-operation then append the value (Figures 8.20c and 8.20f)
- Put the resulting node as the ‘value’ (Figure 8.20d).

Note that the execution schedule in Figure 8.20f is generated based on operation specification, and should therefore be considered as a “template” rather than a fixed unit.

The textual concrete syntax is given in Listing 8.13.

$$\langle list\ construct \rangle ::= \text{[' } \overbrace{\langle operation \rangle}^{\text{' , '}} \text{]}$$

Listing 8.13: Textual concrete syntax for `ListConstructOperation`.

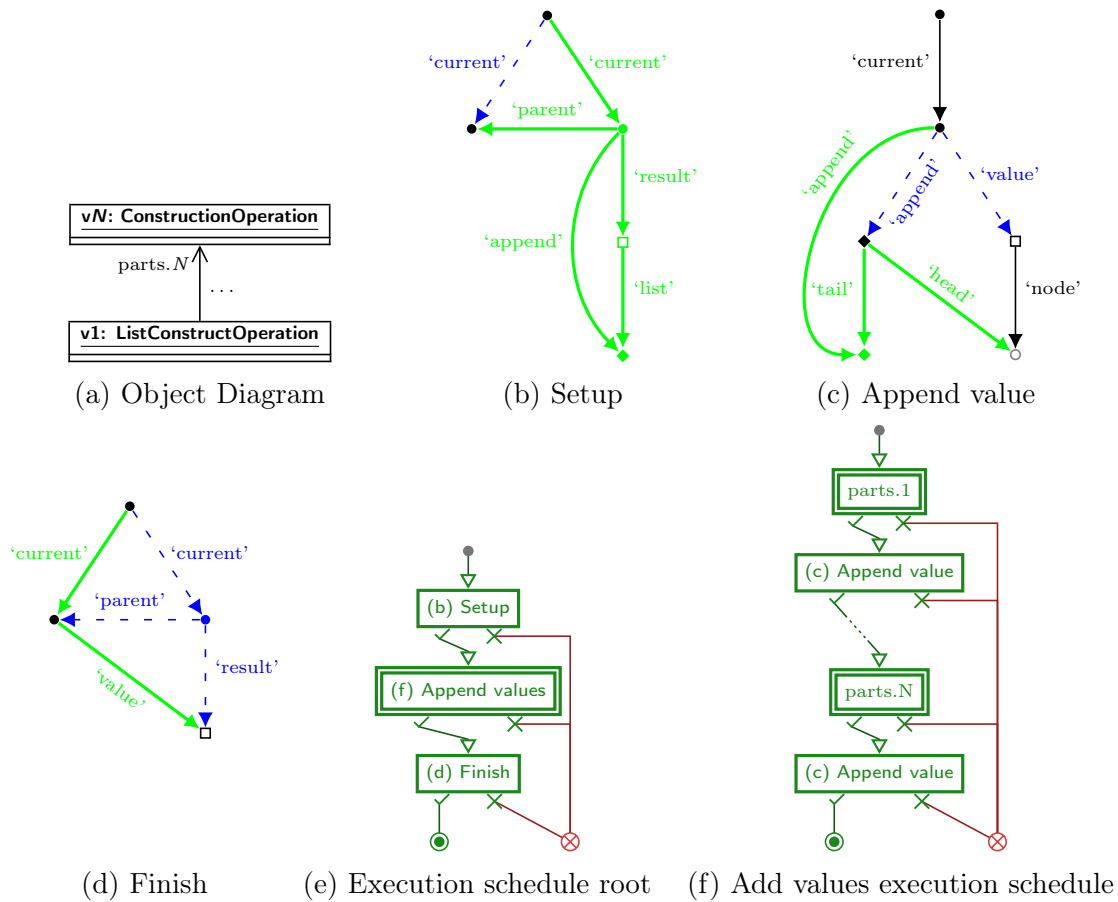


Figure 8.20: Graph based representation of the phases of a ListConstructOperation and accompanying execution schedules.

ListConcatOperation

Shown in Figure 8.21. Creates a new ListValue from a list of ListValues.

It has one parameter: the **parts** that make up the list as a list of sub-operations. For it to be valid, the type of each sub-operation must be a ListTypeReference, and there must be a unique overlapping type for the referenced types, i.e. the type intersection of the referenced types for each part must be exactly one type (using the same type intersection algorithm from Listing 8.5 as used by ListConstructOperation above).

The type of this operation is a ListTypeReference with the referenced type the unique most specific shared supertype of the sub-operations.

This operation can be represented graphically as a multi-phase operation with the execution schedule given in Figure 8.21g:

- Setup the resulting node (Figure 8.21b).
- For each element in **parts**, run the sub-operation then append the list to the queue (Figures 8.21c and 8.21h)

- For each list in the queue, add the list's values to the resulting list then go to the next list (Figures 8.21d and 8.21e).
- Put the resulting node as the 'value' (Figure 8.21f).

Note that the execution schedule in Figure 8.21h is generated based on operation specification, and should therefore be considered as a "template" rather than a fixed unit.

The textual concrete syntax is given in Listing 8.14.

$$\langle list\ concat \rangle ::= \blacktriangleright \langle operation \rangle - '...' - \left[\langle operation \rangle \right] \blacktriangleright$$

Listing 8.14: Textual concrete syntax for ListConcatOperation.

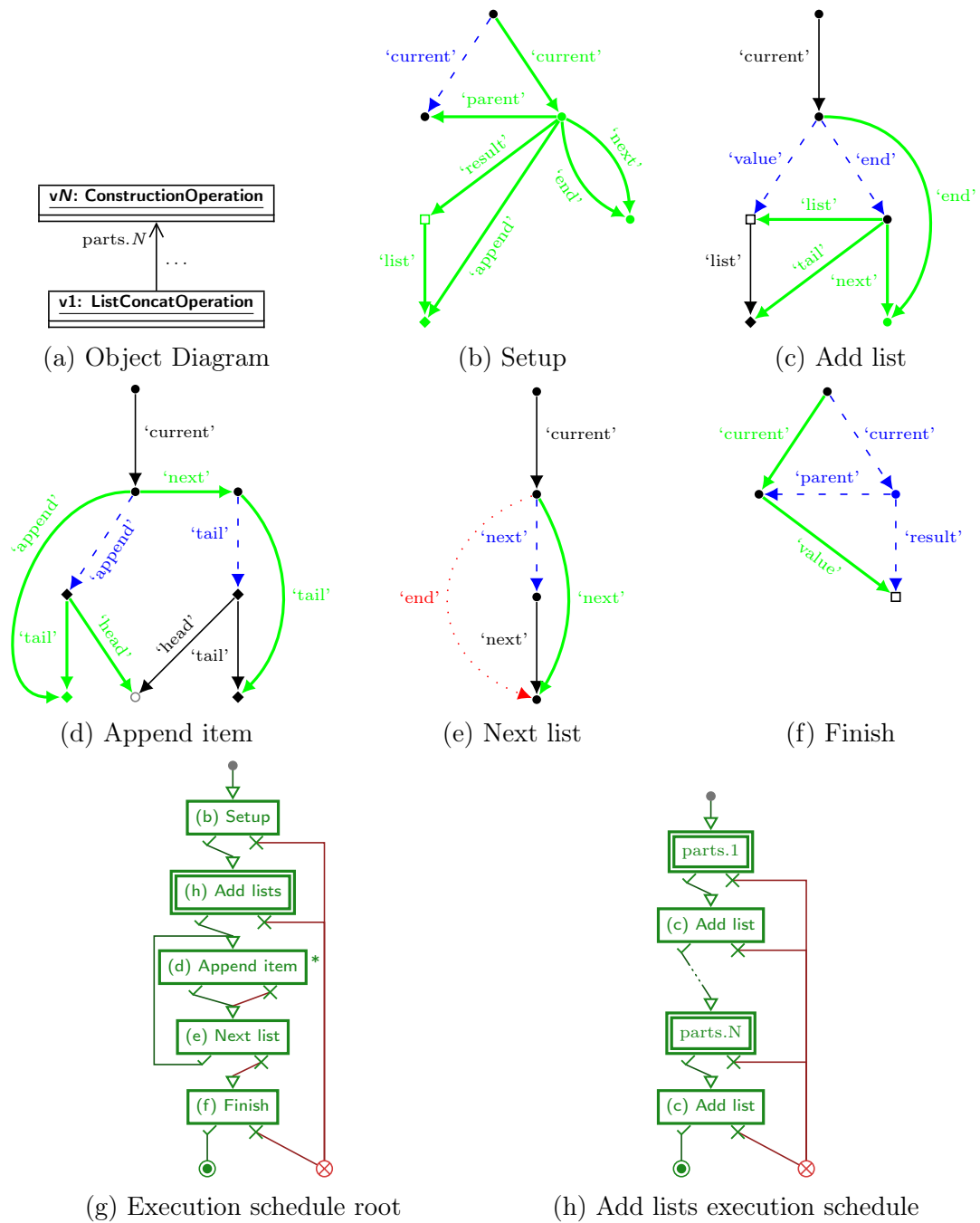


Figure 8.21: Graph based representation of the phases of a ListConcatOperation and accompanying execution schedules.

8.3 Parser Integration

In this section we will describe how a parser implementation should use the TDF, TIF and TCF formalisms introduced in the previous sections. As we mentioned in Section 8.2.4, the idea of TCF models is they are specified at some level of a non-terminal specification. As such, we will start by looking at how these two are connected by putting TCF models next to Extended Backus-Naur Form (eBNF) non-terminal specifications. Note that we will use a postfix ‘?’ as the optional/option rule, a postfix ‘*’ as the zero or more repetitions rule, a postfix ‘+’ as the one or more repetitions rule, and an infix ‘o’ operator to denote an optional separator for repetitions. See Section 8.3.1 for the equivalent BNF forms for each of these operators.

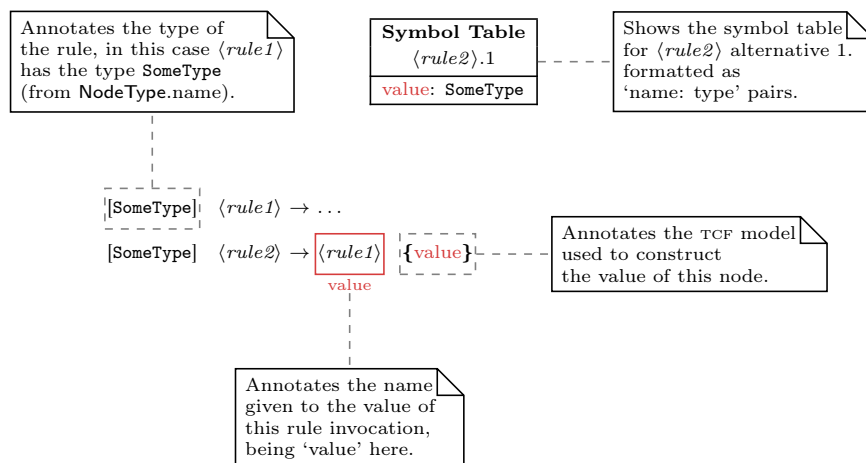


Figure 8.22: Example showing a high-level overview of a ContextAccessOperation.

The first example we will look at is Figure 8.22, in which $\langle rule2 \rangle$ takes the result of $\langle rule1 \rangle$ and returns it instead. The type of $\langle rule1 \rangle$ is given, and the definition is omitted. It also shows the concrete syntax we will use for the rest of this section:

- The ‘type’ of a non-terminal is put to the left of the non-terminal between brackets. This defines the *expected* result of the non-terminal and can be used to get the type of a non-terminal on the right-hand side of a non-terminal definition without having to evaluate the TCF model itself.
- The TCF model is put to the right of each production rule between braces, and defines how the right-hand side gets transformed into a TIF model.
- Below the $\langle rule1 \rangle$ invocation in the right-hand side of $\langle rule2 \rangle$ we put the associated ‘name’ for the result of this invocation. Implementations may choose their own syntax

for this name assigning (or make it automatically deduced for example), and it has been put here like this to avoid introducing more grammars.

Note that the TCF model for $\langle rule2 \rangle$ uses a single `ContextAccessOperation`, which receives its context from the parser itself, and accesses the name ‘value’ from it.

$$\begin{array}{l} [\text{SomeType}] \langle rule1 \rangle \rightarrow \dots \\ [\text{tok?}] \langle rule2 \rangle \rightarrow \boxed{\langle rule1 \rangle} \{ \text{value.token} \} \\ \quad \quad \quad \text{value} \end{array}$$

(a) The grammar and TCF specification.

<i>node</i>
SomeType
token: ?tok

(b) Node definition for `SomeType`.

Symbol Table
$\langle rule2 \rangle.1$
value: SomeType

(c) The symbol table.

Figure 8.23: Example showing a high-level overview of a `NodeAccessOperation`.

The next example is Figure 8.23, which is very similar to the previous one except using a `NodeAccessOperation` to access the value of the ‘token’ field on the context value of ‘value’. Note that the expected type of $\langle rule2 \rangle$ is an optional value, but the type of value returned by the TCF is a ‘base’ value: this is not an issue because a ‘base’ value can be implicitly converted to an ‘optional’ value (i.e. optional values are assignable from base values, see Listing 8.3).

$$\begin{array}{l} [\text{OtherType}] \langle rule1 \rangle \rightarrow \dots \\ [\text{SomeType}] \langle rule2 \rangle \rightarrow \boxed{\langle rule1 \rangle?} \boxed{\text{TOK}^*} \{ \text{SomeType}(\text{tokens}, \text{other} = \text{value}) \} \\ \quad \quad \quad \text{value} \quad \quad \text{tokens} \end{array}$$

(a) The grammar and TCF specification.

<i>node</i>
SomeType
tokens: tok[]
other: OtherType?

(b) Node definition for `SomeType`.

Symbol Table
$\langle rule2 \rangle.1$
value: SomeType?
tokens: tok[]

(c) The symbol table.

Figure 8.24: Example showing a high-level overview of a `NodeConstructOperation`.

The example in Figure 8.24 is slightly more complex because it involves a `NodeConstructOperation`, as well as an ‘optional’ and a ‘repetition’ operator in the grammar. The ‘optional’ operator turns the value from $\langle rule1 \rangle$ from a ‘base’ type to an ‘optional’ type, while the ‘repetition’ operator makes the ‘tokens’ value a list of tokens. See also the symbol table in Figure 8.24c, and the conversion table in Table 8.5. Note as well that this example uses the shorthand notation for assigning the child `tokens` for `SomeType` to

the value of `tokens` from the symbol table. This shorthand can only be used for `ContextAccessOperations`.

Type of	Grammar operation			
	$\langle rule \rangle$	$\langle rule \rangle?$	$\langle rule \rangle^*$	$\langle rule \rangle^+$
	B	O	L	L
	O	O	*	*
	L	*	*	*

B: `BaseTypeReference`

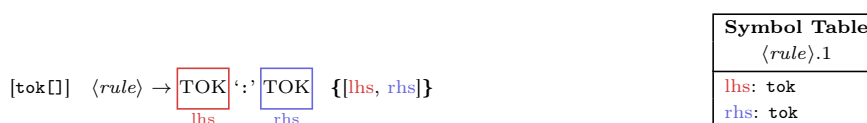
O: `OptionalTypeReference`

L: `ListTypeReference`

*: Undefined/error

Table 8.5: Table showing suggested type conversions for parsers using our parse tree formalisms.

Note that the conversion table in Table 8.5 leaves some combinations as ‘undefined/error’, the reason for this is an inability to combine types (i.e. a list of lists, or a list of optional values, or an optional list). For example, a type reference ‘`Foo [] []`’ is not valid but can be implemented through an intermediate node ‘`Bar(foos: Foo [])`’, the type reference becoming ‘`Bar []`’. Future work might involve changing the type system to allow for more flexibility, though as it is right now it can be worked around with intermediate node types as mentioned, which might arguably better reflect the intention of a language designer by somebody reading the definition, if the designer accurately describes the meaning of the intermediate node type.



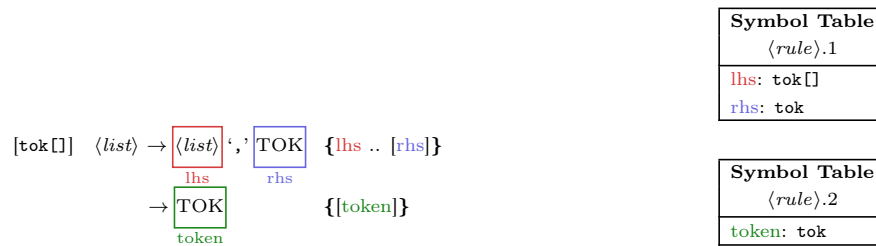
(a) The grammar and TCF specification.

(b) The symbol table.

Figure 8.25: Example showing a high-level overview of a `ListConstructOperation`.

Figure 8.25 shows another simple example, this time with a `ListConstructOperation`. Not much else can be said about this example.

In Figure 8.26 we have a rule $\langle list \rangle$ that has two alternatives. The first alternative makes use of the `ListConcatOperation` to recursively build a list of tokens separated by commas, the second alternative is a base case which makes a single-element list. Note here that the type definition for $\langle list \rangle$ is attached to the rule once: if we would be able to



(a) The grammar and TCF specification.

(b) The symbol tables.

Figure 8.26: Example showing a high-level overview of a ListConcatOperation.

define the type multiple times, this might allow the type to be different for each alternative. Of course, because this is in the domain of the parser it is up to the implementation to decide on what it allows, as this section is only meant as a guideline, however in this document and in our implementation we will keep this limitation as such.

Given all these example cases, the following is a general overview of the requirements for implementing the TDF, TIF and TCF formalisms into a parser:

1. Annotating non-terminals with expected type specifications from TDF.

Alternatively, an implementation may chose to derive the type of a non-terminal through the TCF models attached to its productions.

2. Annotating production rules with TCF models.
3. Adding or augmenting the ‘symbol’ information for elements in production rules to include ‘type’ information from TDF.
4. Adding a validation step verifying the validity of each TCF model using the symbol information and expected types.
5. Replacing tree construction to use TCF models.
6. Making an implementation of `TokenInstance`.

For a parser with lexical analysis phase, this would generally include a ‘type’ and ‘value’ field, one for the sort of token, and another for the text matched, respectively.

7. Generating instances of `TokenInstance` for any terminals encountered.

We note that an implementation may chose to generate code based on the given grammar, TDF, and TCF specifications that then needs to be compiled or interpreted, or generate runtime instances of constructs that handle the required semantics, or possibly something else. Which form used (generate code, runtime instances, etc.) may depend on the requirements of the tool using the formalisms.

For our implementation we used the grammar used by the Lark Python library, and modified it according to the above steps (see Chapter 16), as well as using the Lark parser itself as a backend for parsing languages (see Chapters 9 and 17).

8.3.1 Extended Backus-Naur Forms

Extended Backus-Naur Form is not a single syntax, but a family of syntaxes extending on the core BNF syntax. Each eBNF syntax may have its own way of writing common concepts such as optional elements, grouping, and repetition [43]. We will show the equivalency for each eBNF syntax concept used in Section 8.3 to its respective BNF:

Optional The postfix optional operator ‘?’, indicating zero or one instance of its operand, used as

$$[\text{tok?}] \langle \text{rule} \rangle \rightarrow \boxed{\text{TOK ?}} \{\text{value}\}$$

value

is equivalent to

$$[\text{tok?}] \langle \text{rule} \rangle \rightarrow \boxed{\text{TOK}} \{\text{value}\}$$

value

$$\rightarrow \epsilon \{-\}$$

Star repetition The postfix repetition operator ‘*’, indicating zero or more instances of its operand, used as

$$[\text{tok[]}] \langle \text{rule} \rangle \rightarrow \boxed{\text{TOK *}} \{\text{values}\}$$

values

is equivalent to

$$[\text{tok[]}] \langle \text{rule} \rangle \rightarrow \epsilon \{\{\}\}$$

$$\rightarrow \boxed{\langle \text{rule} \rangle} \boxed{\text{TOK}} \{\text{front} \dots \text{end}\}$$

front end

Dot-star repetition The postfix repetition operator ‘*’ combined with the infix separator operator ‘◦’, indicating zero or more instances of its second operand, separated by its first operand, used as

$$[\text{tok[]}] \langle \text{rule} \rangle \rightarrow \boxed{\text{SEP} \circ \text{TOK *}} \{\text{values}\}$$

values

is equivalent to

$$\begin{aligned}
[\text{tok}\square] \langle rule \rangle &\rightarrow \epsilon \{ \square \} \\
&\rightarrow \boxed{\text{TOK}} \langle rule \rangle \{ [\text{head}] \dots \text{tail} \} \\
&\quad \text{head} \quad \text{tail} \\
[\text{tok}\square] \langle rule \text{ more} \rangle &\rightarrow \epsilon \{ \square \} \\
&\rightarrow \text{SEP} \boxed{\text{TOK}} \langle rule \text{ more} \rangle \{ [\text{head}] \dots \text{tail} \} \\
&\quad \text{head} \quad \text{tail}
\end{aligned}$$

Plus repetition The postfix repetition operator ‘+’, indicating one or more instances of its operand, used as

$$[\text{tok}\square] \langle rule \rangle \rightarrow \boxed{\text{TOK} +} \{ \text{values} \}$$

is equivalent to

$$\begin{aligned}
[\text{tok}\square] \langle rule \rangle &\rightarrow \boxed{\text{TOK}} \{ [\text{value}] \} \\
&\quad \text{value} \\
&\rightarrow \langle rule \rangle \boxed{\text{TOK}} \{ \text{front} \dots \text{end} \} \\
&\quad \text{front} \quad \text{end}
\end{aligned}$$

Dot-plus repetition The postfix repetition operator ‘+’ combined with the infix separator operator ‘o’, indicating one or more instances of its second operand, separated by its first operand, used as

$$[\text{tok}\square] \langle rule \rangle \rightarrow \boxed{\text{SEP} \circ \text{TOK} +} \{ \text{values} \}$$

is equivalent to

$$\begin{aligned}
[\text{tok}\square] \langle rule \rangle &\rightarrow \boxed{\text{TOK}} \{ [\text{value}] \} \\
&\quad \text{value} \\
&\rightarrow \boxed{\text{TOK}} \text{SEP} \langle rule \rangle \{ [\text{head}] \dots \text{tail} \} \\
&\quad \text{head} \quad \text{tail}
\end{aligned}$$

8.3.2 Note

As it stands, there is no guidance on how to handle grouping such as encountered in many eBNF syntaxes. The lack of guidance is due to the fact that a grouping is effectively equivalent to defining an anonymous non-terminal production rule [43], introducing possible branching points within the production rule that are non-trivial (branching on a single terminal or non-terminal are considered trivial). Thus, due to the complexity of these possibly non-trivial branches, we have left this open as future work.

8.4 Grammars

We show the grammar for TDF defined using a TDF model (Listing 8.15) and an annotated grammar specification as shown in Section 8.3 (Figure 8.27). Likewise, we do the same for TIF in Listing 8.16 and Figure 8.28, and for TCF in Listing 8.17 and Figure 8.29.

```
interface DefinitionElement()

interface NodeDefinition(name: tok): DefinitionElement
ConcreteDefinition(children: NodeChild[],
    super_types: tok[]): NodeDefinition
InterfaceDefinition(children: NodeChild[],
    super_types: tok[]): NodeDefinition

NodeChild(name: tok, type: TypeReference)

interface TypeReference()
BaseTypeReference(name: str)
OptionalTypeReference(name: str)
ListTypeReference(name: str)
```

Listing 8.15: Simple TDF model describing the parse trees for TDF itself.

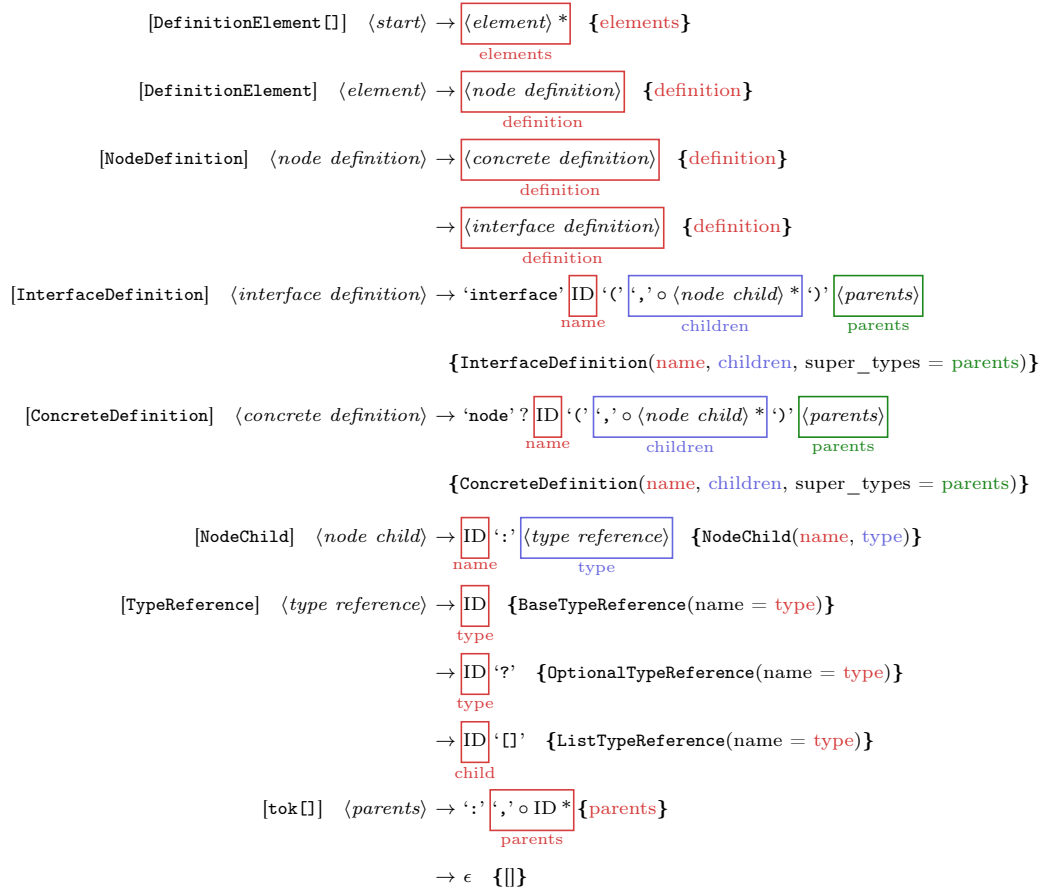


Figure 8.27: Grammar specification for the concrete textual syntax of TDF annotated with TCF models and expected types. The start symbol is $\langle \textit{start} \rangle$.

```

interface NodeInstance
ConcreteInstance(type: tok, children: NodeValue[]): NodeInstance
TokenInstance(value: tok): NodeInstance

interface NodeValue(name: tok)
BaseValue(value: NodeInstance): NodeValue
-- OptionalValue cannot be represented directly
-- -> either there is a value, which gets parsed as a BaseValue,
--    or there is none and it doesn't get parsed
ListValue(values: NodeInstance[]): NodeValue

```

Listing 8.16: TDF model describing the parse trees for TIF.

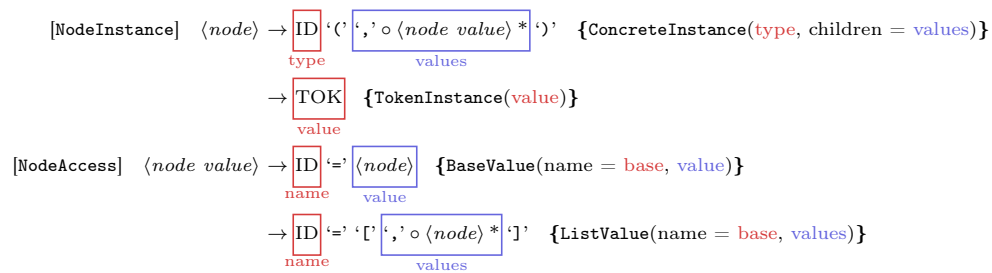


Figure 8.28: Grammar specification for the concrete textual syntax of TIF annotated with TCF models and expected types. The start symbol is $\langle node \rangle$.

```

interface Operation
ContextAccess(name: tok): Operation
NodeAccess(base: Operation, name: tok): Operation
NodeConstruct(type: tok, children: NodeConstructChild[]): Operation
ListConstruct(elements: Operation[]): Operation
ListConcat(lists: Operation[]): Operation

NodeConstructChild(name: tok, value: Operation)

```

Listing 8.17: TDF model describing the parse trees for TCF.



Figure 8.29: Grammar specification for the concrete textual syntax of TCF annotated with TCF models and expected types. The start symbol is $\langle operation \rangle$.

8.5 Considered Variations

We make no claim that the design of our parse tree formalisms is perfect, and in this section we will briefly go over some alternatives that we considered but decided not to add, together with the reasoning for why.

Array vs. List We originally called what we now call the `ListTypeReference` type the “`ArrayTypeReference`” type. The naming was changed because of the semantics behind the words list and array: an array in programming is generally considered to be a fixed size ordered collection of items in a continuous area of memory, while a list is generally considered to be dynamic in size instead using pointers referencing to the next (and previous) elements. Additionally, for non-programmers the word ‘array’ may not necessarily evoke the necessary mental model, while ‘list’ may do. And as for the data contained within, the order of elements is important, rather than the position they are in, which we consider another argument for using ‘list’ over ‘array’.

Tuple Types We considered adding a `TupleTypeReference` (extending `TypeReference`) to TDF (and related types in TIF and TCF), which would contain a fixed format of nodes. However, we decided not to because these tuples would be semantically identical to a concrete node definition, with the exception of children being unnamed, and the added disadvantage that they do not convey their meaning well. Instead a concrete node definition should be used.

Mapping Types A `MappingTypeReference` extending `TypeReference` was considered, parameterized with a single type. This type would allow a sort of “map” or “dictionary” to be stored in a node, keyed by a token and values constrained by the specified type. However, it was decided against because it could lead to hidden problems and raised questions about how it should be implemented. For example, would a token as key use its unique instance/address, or its value, or a combination of type and value? A problem would be that entries may accidentally be replaced, being removed from the parse tree even though they may influence the semantics of a parsed model. For example, a duplicate name definition (`Foo(a: tok, a: Bar)`) could be hidden if stored using this method.

Type Unions Type unions as considered would be similar to “union” types as found in C/C++, that is: the value contained by instances of the type can be one of the types defined within the union type. We decided not to add this feature as interface types are a better candidate to fulfill most use cases of union types, with the added bonus of more easily extending possible values.

Abstract Types While not really considered, it is worthwhile to mention these. The interface types we have defined are similar to interface non-terminals as defined in MontiCore [15]. From this stems the comparison with “abstract” types, as MontiCore includes abstract non-terminals, which are similar to interface non-terminals but allow defining methods on the base non-terminal. This seems to stem from the fact that MontiCore runs on the Java Virtual Machine, which has not allowed defining methods with implementations on interface types until Java 8 introduced “default” methods.

List Cardinality Cardinality specifications on lists like they are found on UML class diagrams were considered. However, due to the simple nature of parse trees, and the existing ability to have a cardinality of 1 or 0..1 with base and optional type references respectively, we decided not to add this feature.

Combinable Type References As we mentioned in Section 8.3, it’s not possible to combine type references to achieve a list of lists, or a list of optionals, or an optional list. While this limitation is a result of the abstract syntax of TDF, it also does not pose a significant problem as workarounds improve verbosity, which makes it easier for readers of TDF models to understand the intended meaning.

Parsers and Languages

Part of the WEAVE framework and our eventual goal is combining the operation of multiple parsers on a single source text. Figure 9.1 highlights the relevant parts, showing that the orchestrator has a heterogeneous collection of *parsers* and *parser generators* (see Chapter 5) that it interacts with, and that in turn also interact with the orchestrator themselves. A parser or parser generator is required to implement a common interface in order for WEAVE to be able to communicate with it.

The following are required operations for *parsers*:

- A ‘parse’ operation: the essence of a parser. This operation should work on strings in memory to allow the framework to handle embedded models (see Chapter 11).
- Error reporting functionality: the parser should be able to signal errors to the framework so that it can handle and display them. The parser itself should refrain from outputting errors to the console.
- A way to report “islands” (see Chapter 4) in its grammar to aid in the detection of language boundaries (see Section 11.3 in Chapter 11 for why this is necessary). These islands may be constructs at the lexical level such as “string literals”, block comments, etc., or at the syntactical level such as block delimiters, etc., or may be further embedded models.

Other than the above, no requirements are put on parsers making use of our framework: a parser can decide itself to make use of the framework, or not use it at all. Figure 9.2 shows the required interface as a UML class diagram.

As shown in Figure 9.1, a parser may require a “wrapper” to implement the required interface, for example if it is an executable, “black box” (Chapter 3), or any other existing

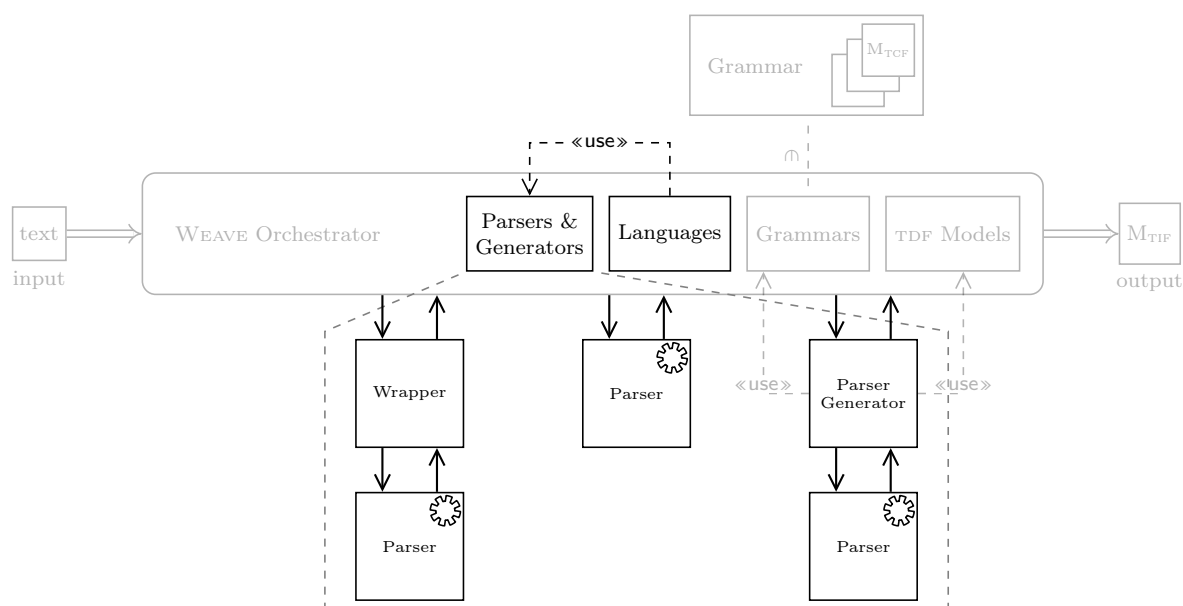


Figure 9.1: Focused view of Figure 7.1, highlighting the relevant parts of the WEAVE framework for this chapter.

implementation not written with the WEAVE framework in mind. The following are some things that may be incompatible with the interface that a wrapper might ‘fix’:

- The parser does not support reading from strings in memory (e.g. it requires a file on disk). **Possible solution:** The wrapper makes a temporary file for the parser to read.
- The parser has no pluggable error reporting facilities and would instead output to standard output/error. **Possible solution:** The wrapper captures the output and parses it to convert it into the expected format.
- The parser has no grammar specification available. **Possible solution:** The author of the wrapper manually specifies possible islands.
- The grammar is in an unsupported format or uses unconventional constructs (e.g. indentation awareness). **Possible solution:** The author manually or programmatically converts the grammar to a compatible format.

A *parser generator* can likewise implement the parser interface, and pass the required operations on to the generated parser. In a sense, this works in a similar way to a parser wrapper. Allowing parser generators to be invoked allows for WEAVE to generate new parsers on the fly as necessary. In Chapter 17 we will look into how we use *Lark* as a parser generator.

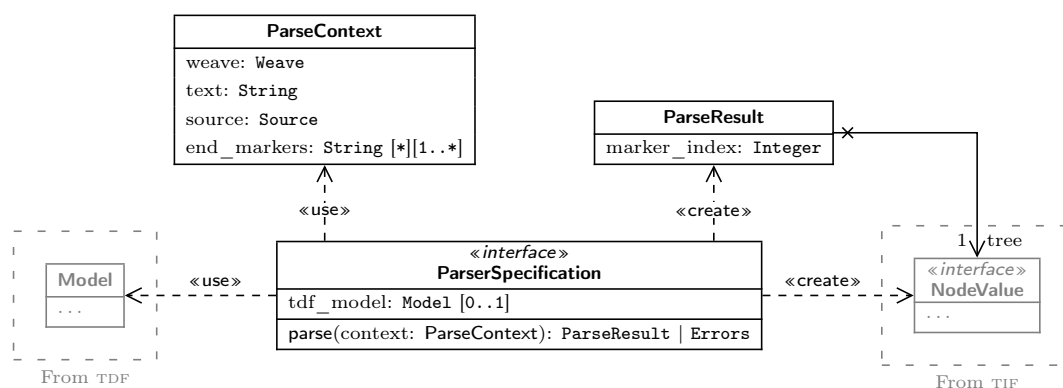


Figure 9.2: Class diagram showing the common parser interface required by the WEAVE framework, named `ParserSpecification`.

Figure 9.3 shows the abstract syntax for two parser implementations we use in WEAVE: a black box parser that provides the basis for calling non-compatible parsers, and a parser generator making use of a grammar and TDF specification to generate a parser on the fly. An example sequence diagram showing how the parser generator works is shown in Figure 9.4. How these parsers can work together to parse hybrid languages will be shown in Chapter 11.

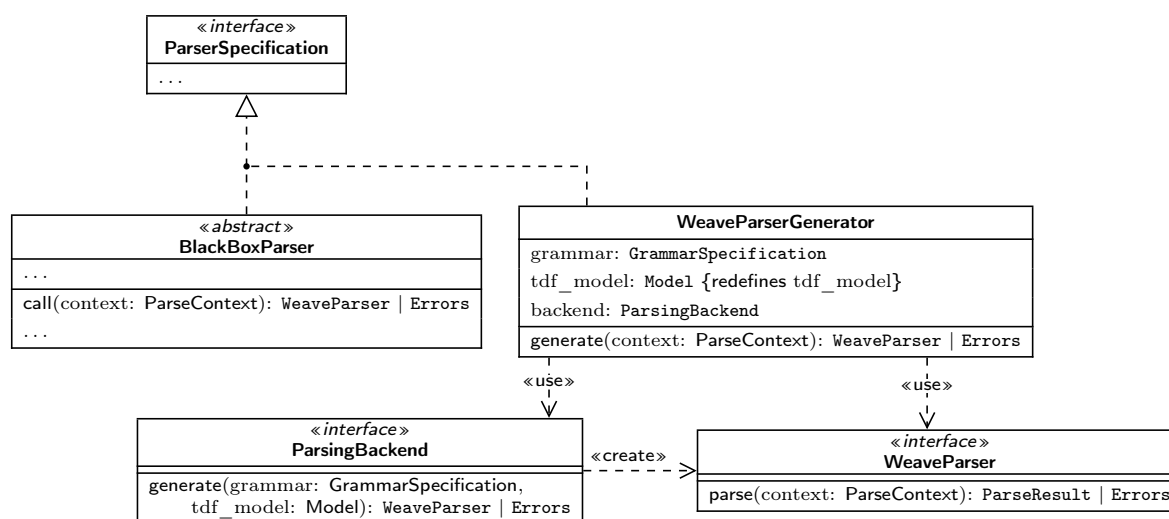


Figure 9.3: Class diagrams of possible implementations of `ParserSpecification`.

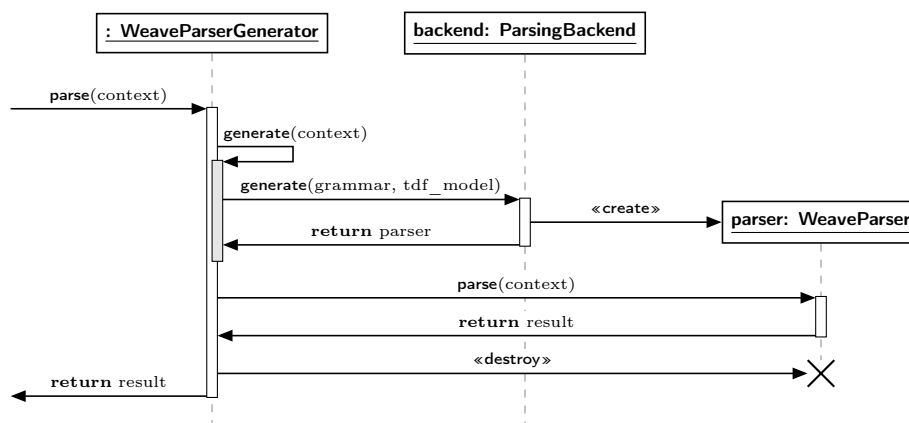


Figure 9.4: Sequence diagram describing the operation of the built-in `WeaveParserGenerator` parser specification.

We consider parser specifications to be a separate entity to *languages*. As mentioned in Chapter 6, a language is generally made up of abstract syntax, visual and/or textual concrete syntax, and semantics. For this reason, we separate the concepts of language and parser specification into different classes as can be seen in Figure 9.5, but without including the concrete syntax or semantics, as this lies outside our scope.

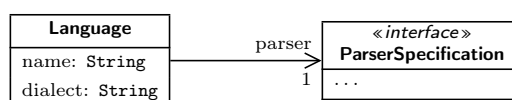


Figure 9.5: Abstract syntax of languages.

Aside from a name, a language is also identified by a dialect: some languages have multiple dialects, variations or versions [20], [28] that each require a separate parser specification, even though they are otherwise virtually identical. Thus, allowing to identify which dialect a language allows for the appropriate handling of its specific features.

We do not propose any specific format for naming languages and their dialects. However, for the purposes of implementing a proof of concept, we will use a simple syntax for specifying a language and dialect: ‘ $\langle language \rangle : \langle dialect \rangle$ ’.

CHAPTER 10

Tracing

In important part of transforming text into a model, and then validating, interpreting and/or executing that model is *tracing*. Depending on the context, a ‘trace’ may refer to different things. For example, when executing a finite state automaton, an *execution trace* could capture the states and transitions taken. For our use case, we specifically look at *source traces* instead.

A source trace can be seen as a sort of path from an object that is being inspected to the source of the object, and allows for debugging of a program or model in case of some unexpected or unwanted behaviour. This path may be a straight line from the object to some position in a text file, or may be more indirect, containing intermediate versions that were used to get to the eventual object. Examples of these steps may be:

- Preprocessor operation (on text, tokens, or parse trees) such as file inclusion, macro expansion, etc.
- Lexing: turning text into tokens.
- Parsing: turning tokens or text into parse trees.
- Model generation from parse trees.
- Model optimization: adding, removing, or replacing parts of a model for performance.
- Model transformation: transforming a model from one formalism to another, e.g. to petri nets for formal validation, or as part of model execution.
- Binary code generation: turning a model into assembly code, or virtual machine instructions (e.g. Java).

- Just-in-time compilation of virtual machine instructions to assembly code (e.g. Java Virtual Machine).

Our scope is limited to the second, third, and fourth items in this list, i.e. the steps involved in going from a string and turning it into a model. The first item (a preprocessor) will not be implemented in our work, however we will keep it in mind as it may exist in derivative or future work. Additionally, we will propose a ‘sort of’ preprocessor as future work at the end of this document (see Section 19.1 in the conclusion), which will perform processing on embedded text fragments (see Chapter 11) in order to ‘fix’ embedded language fragments that would otherwise not parse (e.g. with indentation aware languages).

We note that when tracing where an object came from, there are different ways of looking at this. For example, when calling a function ‘foo’ with a local variable ‘bar’ as parameter:

- foo is declared at some location *A*.
- The conceptual “function call” to foo is at some other location *B*.
- bar is declared at some location *C* close to *B*.
- The “reference” to bar as a parameter to the function call is at some other location *D* (even closer to *B*).
- Additionally, foo may also be *defined* (as opposed to declared) at some location *E* that may not be the same as *A*.

In our case when it comes to references, we will always work with the location where the referencing occurs, and not the location of what is referenced, as this would fall under the semantic domain.

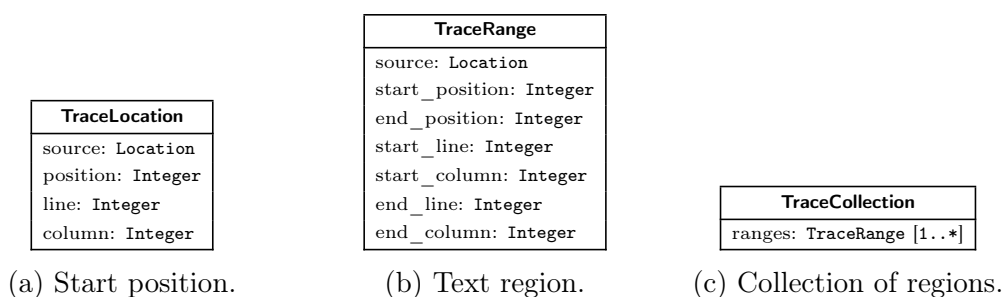


Figure 10.1: Representation of different kinds of tracing location sorts for referencing a location in a source text.

The most basic information that is relevant to tracing for us then is:

- what file (or location, or buffer, etc.) is the source text located in/at; and

- at what position in the source text is the token/parse tree/model element defined at: raw character position, and/or line and column.

Additionally, the position in the source text may be either:

- where the token/parse tree/model element starts (Figure 10.1a);
- a region defining the start and end (Figure 10.1b); or
- in some more exotic cases, a collection of regions (Figure 10.1c).

We note that in the last case, these regions may not necessarily be in a single source text, but could be spread out over multiple. For example, a partial class (a way to split large class definitions) in C# may be defined spread out over multiple files.

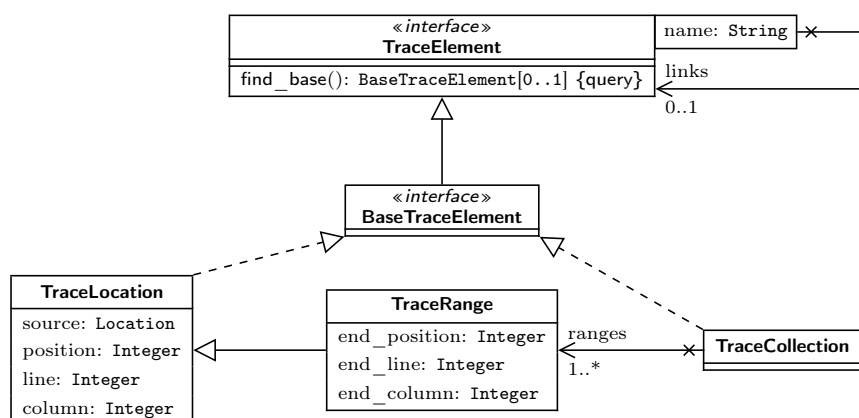


Figure 10.2: Abstract syntax for tracing elements.

Simplifying/unifying the elements in Figure 10.1, and adding a `TraceElement` interface gives us Figure 10.2. In it, the most basic elements all implement the `BaseTraceElement` interface, and objects that may be traceable can implement the `TraceElement` interface. Example constructs that could implement this interface seen so far are:

- `NodeType` (see Figure 8.3 on page 35) instances, as created by parsing a TDF specification;
- `NodeInstance` or `TokenInstance` (see Figure 8.9 on page 47) instances, created through parsing or TCF models;
- `ConstructionOperation` (see Figure 8.13 on page 50) instances, specified as part of grammar specifications;

Additionally, this may also be used for grammar specifications themselves, as for model instances created based on TIF models, which can either inherit the trace links or reference the TIF model elements themselves if they are a `TraceElement`.

CHAPTER 11

Multi-Language Parsing

As was explained in Section 6.1, “hybrid languages” are languages which combine abstract syntax, concrete syntax, semantics, etc. in order to model complex systems using the most appropriate formalism. For the WEAVE framework, the focus lies on the parsing of text files defined in several separate languages, thus we will only look at the composition of the textual concrete syntax of formalisms in this chapter.

A model that is described using textual syntax in a hybrid language has:

- a *root language*, which is the base language of the entire document;
- some *fragments*, non-overlapping continuous pieces of text in the document that are in a different language than the base language; and
- optionally, *nested fragments* contained within other fragments, if the language of the fragment allows it.

The non-overlapping constraint is put in place in order to exactly define which character is part of which fragment, i.e. there can be no shared ownership. The same is true for hybrid visual syntaxes: each element has a specific formalism that describes its meaning. Figure 11.1 shows an example excerpt from a DFA hybrid with actions on transitions using some neutral action language illustrates this ownership relationship further

```
... transition from A to B on x do report ( "x" ) ; ...  
..... DFA ..... action language ..... DFA
```

Figure 11.1: Example DFA hybrid model with an action language fragment.

In order to parse these sorts of languages, we need a *multi-language parser*: a parser that can switch to parsing a different language on the fly. The way we achieve this is by building a parser that defers the parsing to different sub-parsers, performing translation of input and output in order to make this switching as transparent as possible to the involved parsers. The goal is: a parser should be unaware that it is parsing inside a fragment, and should only be concerned with switching if it itself allows fragments contained within its syntax, without having to concern itself with the switching back at the end of the fragment. Figure 11.2 gives a high-level operational diagram for language switching as defined here.

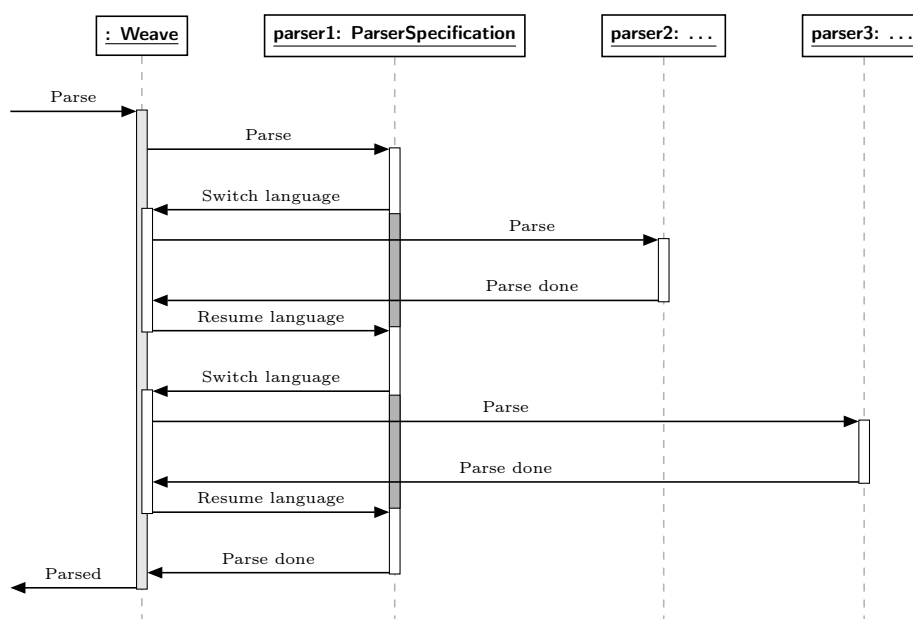


Figure 11.2: High-level overview of how language switching should work in a multi-language parser.

With all of the above said, if we were to look at the structure of a textual hybrid model, without considering semantics or references, we would end up with a tree structure. The root of this tree would be the root model parsed in the root language, and each branch would represent a fragment that can recursively have branches for nested fragments. While this tree structure is not directly accessible, it does serve as a good mental model in order to envision the structure.

In Section 11.1 we will look at how fragments can be turned from text into being part of the resulting parse tree. Next, in Section 11.2 we will explore the concrete syntax for switching to a different formalism, and some issues related to switching. In Section 11.3 we list some non-trivial issues with multi-language parsing in general, and in Section 11.4 we go over some instances of parsing that are similar to multi-language parsing. Finally, in Section 11.5 we take a brief look at *dynamic* multi-language parsing. Later on in Chapter 17 we will look at how we implemented this.

11.1 Parsing

As seen in Chapter 5, in order to turn a text document into something analyzable and executable by a computer, several steps need to be done. We will focus on the first two, lexing and parsing, as this will provide a minimal data structure usable by a computer: the parse tree.

We know that the model is given as a text document, and the goal is to incorporate the parse result of each fragment into the eventual parse tree. This gives us two questions:

1. How do we store fragments in the parse tree?
2. Where do we handle the parsing of fragments?

The answer to the first question could be to directly store the tree of the fragment as a child of the containing branch, or to provide a specialized ‘terminal’ type containing the tree as its value, rather than a string directly. In our case, we use the TDF language defined in Chapter 8. We’ll use the fact that it’s easy to extend to introduce a new node type: `EmbedType` (see Figure 11.3a). In order to support this new node type, a new instance type is also introduced: `EmbedInstance` (see Figure 11.3b). No additions are required to the concrete syntaxes of TDF or TIF. For TCF, no changes are necessary.

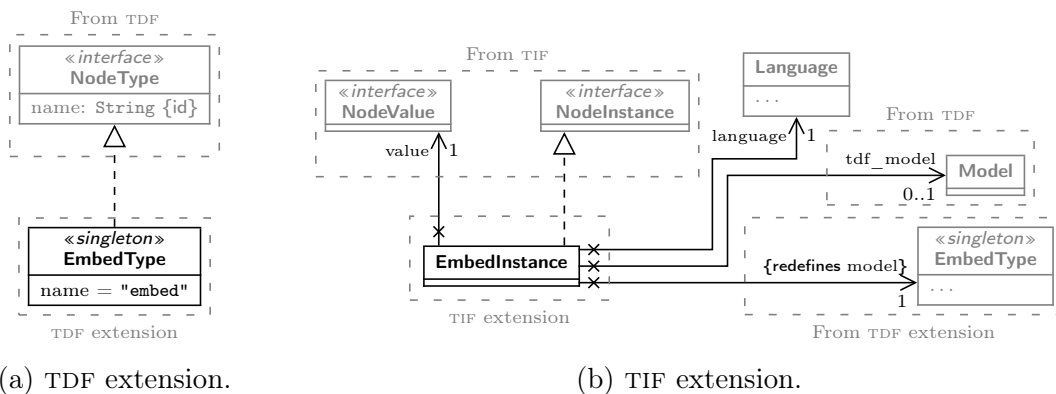


Figure 11.3: Extensions to the TDF and TIF languages in order to support embedded fragments.

The second question is more complicated: the obvious answer is to handle fragments in the lexer, as it directly works on the input text, turning it into tokens. On the other hand, the specification of *where* an embed may happen is done in on the syntactical level, i.e. is handled by the parser.

Thus, on the one hand we have the lexer that should decide on the switching of languages, and on the other we have the parser that knows when these switches should

happen. This creates a requirement for the parser to communicate back to the lexer about its state, as well as the possible requirement for the lexer to be able to backtrack: if the parser has to backtrack, and figures a language switch should happen at a point the lexer already passed, the lexer may need to discard these tokens and instead attempt switching languages.

In order to avoid this difficult situation, we instead propose a more verbose syntax that allows the lexer to determine if it should initiate a language switch. This would allow a, be it limited, multi-language parser to be written, and could serve as a stepping stone for a more powerful implementation. We will explain the requirements for this syntax in the next section.

A question may be: why not combine grammars of the languages directly, thereby removing the need for language switching and having multiple lexers and parsers? While this would be nice, it has the following issues:

- Each language has its own definitions of non-terminals, which may conflict with each other [34].
- Combining large grammars together would create large amounts of states a parser may be in, possibly heavily increasing memory usage. This may become a bigger issue if languages may be recursively embedding or have many possible embedded languages.
- Some languages are indentation sensitive, while others are not (see [28], [34] and Section 11.3). Combining these is non-trivial.
- Likewise, a grammar may be parsable by an LL parser, an LR parser, LALR, SLR, etc., or it may be ambiguous, contain left recursion or right recursion, be a Context-Free Grammar (CFG) or a Parsing Expression Grammar (PEG), etc. [28]. All these different points of variability combined may mean that there may be no one parser that can parse all those languages together.
- The list of languages that can be parsed would be fixed at the start of parsing. This limits one of our goals of being able to dynamically define and parse languages in a single file.

11.2 Concrete Syntax

As we mentioned, we would like language switching to be as seamless as possible. A (partial) example of a seamless multi-language text model was given at the beginning of this chapter (see Figure 11.1): the neutral action language code on a DFA transition is a direct part of the DFA text. However, this requires possibly complicated interactions between the parser and the lexer.

Instead, we require the syntax to be slightly more verbose by requiring a terminal at the start of a fragment. When the lexer then produces a token of such a terminal, it then knows a language switch needs to happen after the token. A caveat to this method is unfortunately the inability to use this token for anything else anymore: its sole purpose is turned into being a marker or *sentinel*, guarding the start of the fragment.

This solves the issue of finding the start of fragments, but there is another issue: knowing where a fragment ends. For example, say the neutral action language also allows function calls with no parameters to omit the parentheses, the example in Figure 11.1 could be erroneously parsed such that `report` is considered part of the fragment, and the parameters are not. This would subsequently result in a syntax error in the DFA parser, as it does not know what to do with the tokens that were supposed to be part of the action language fragment.

Likewise, a parser may accidentally try to consume more input than is intended. This can be the case for any parser that is not designed to ignore trailing input it does not recognize. For example, in our implementation we use the `Lark` parser as a parsing backend (see Chapter 17), which supports “interactive” parsing that allows us to feed it a fake *End Of File* (EOF) token on unexpected input, but this requires a specific parser configuration and is not the intended use of the mode.

We solve these issues in the same way: we will require that a fragment be followed by a terminal. This will then allow the framework to determine where to cut the input to the fragment parser, and combined with the start terminal serve as a visual separator between fragments and the text they are contained in. See Figure 11.4 for how this would look with the example from Figure 11.1.

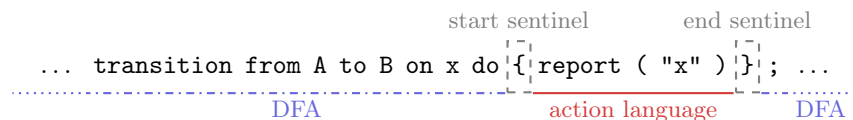


Figure 11.4: Modified version of Figure 11.1 with start and end sentinels around the action language fragment.

We note that the terminal used as end sentinel can be used for other purposes in the grammar, as opposed to the terminal used as start sentinel, which can only serve a single purpose. However, in practice it may end up being unusable anyway as a matching style will be preferred between the start and end sentinels. For example, using an open parentheses to indicate the start, it would be logical to use close parentheses to indicate the end.

11.2.1 Selecting a Language

While the above syntax is fine for fragments where we know which language to expect, we would also like to be able to specify which language the fragment is in as part of the textual model. This allows for a more generic way of composing languages in models,

or adding support for a embedding a formalism without having to modify or extend the base language. The combined semantics and abstract syntax of such generically composed models may be more complicated, but this is out of scope for this document, as we only focus on the composing of the concrete syntax.

We propose a small extension to the above syntax with sentinels to include a “language identifier” that indicates the language of the fragment, and an additional sentinel separating this identifier from the actual language. An example of how this could look is shown in Figure 11.5.

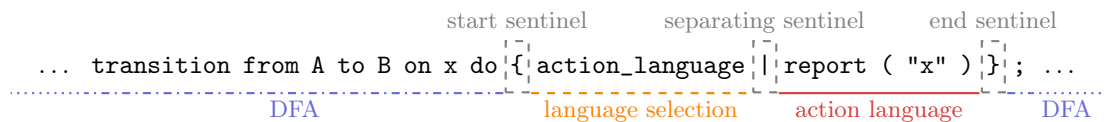


Figure 11.5: Modified version of Figure 11.4 with the ability to select which language appears in the fragment.

We note that the language selection syntax is dictated by WEAVE, in order to have a unified format. This gives the additional benefit of not having to worry about the format, and allowing it to be updated in the future without having to update every language using it (though care should be taken to not introduce breaking changes).

11.3 Non-trivial Issues

Aside from the issues with parsing fragments mentioned above, there are some other issues that can appear. In this section, we will give an overview of some possible issues we see, and some possible solutions to them.

11.3.1 Early End Sentinels

As the end sentinel is used to determine the end of a fragment in the above strategy, the risk exists that a fragment may contain the same pattern as the sentinel, causing premature ending of the fragment and syntax errors. Figure 11.6 shows a modified version of Figure 11.4, where the DFA syntax was changed to use parentheses instead of braces. Another possibility is the end sentinel appearing inside a string literal in the fragment, such as in Figure 11.7.

While for the human reader it is obvious that the second closing parenthesis should be the end of the fragment, the parser requires knowledge of the language and an appropriate algorithm to know the first one is still part of the fragment. We propose two possible solutions to this problem.

The first solution requires that we can modify the grammar specification of the fragment language: given the fragment grammar $G = (V, \Sigma, R, S)$ and the end sentinel e ,

```

... transition from A to B on x do { (report ( "x" ) ) }; ...
                                sentinel start      mismatched end expected end

```

Figure 11.6: Modified version of Figure 11.4, but with the end sentinel appearing inside a string literal.

```

... transition from A to B on x do { { report ( "}" ) } }; ...
                                sentinel start      mismatched end expected end

```

Figure 11.7: Modified version of Figure 11.4, but with the sentinels declared using parentheses instead of braces.

introduce a new grammar $G' = (V \cup \{S'\}, \Sigma \cup \{e\}, R \cup \{S' \rightarrow S e\}, S')$, and use this grammar to parse the fragment instead. Upon reaching a state where the parser can accept, stop parsing and return to the previous language.

Unfortunately, this solution has the drawback that we need to be able to modify the grammar, and as such is not usable for parsers where we do not have that option (i.e. black box parsers). Additionally, it increases the amount of cases that white box parsers need to handle, as well as needing to have a parser for embedded parsing and non-embedded (root) parsing.

The second solution is to use an island parser (see Chapter 4) as a light-weight scanner. The grammar for this island parser would be generated automatically based on the information available on the fragment language.

A basic implementation for example would use information about grouping separators (i.e. tokens that always appear in pairs to group some content) in the language, as well as terminals that may (partially) match the end sentinel, or that may contain the end sentinel. See Figure 11.8 for an example of how such an island grammar may look like. Note the added WATER terminal, which matches anything (dot) as a last resort in case no other terminals match.

$\langle start \rangle \rightarrow \langle fragment \rangle * \langle \rangle$	$STRING ::= \langle \rangle (\langle \backslash \rangle \langle \backslash \backslash \rangle [\langle \wedge \rangle \langle \wedge \rangle]) * \langle \rangle$ (ignored)
$\langle fragment \rangle \rightarrow \langle \langle \rangle \langle fragment \rangle * \langle \rangle$	$WATER ::= .$ (ignored)
(a) Non-terminal specifications.	(b) Terminal specifications.

Figure 11.8: Example island grammar specification for a neutral action language.

11.3.2 Indentation Sensitive Languages

Some languages (for example Python [36]) use the indentation at the start of a line as a sort of delimiter to indicate the start and end of code blocks [28], [34], or have otherwise specific layout constraints applied to them based on the indentation [11].

In order to parse these sorts of languages as a fragment, care needs to be taken when switching to one. As most languages implicitly assume the first line is always at indentation level zero (not indented at all), these parsers would either need to be able to receive information on the current indentation level before switching, or the fragment text would need to be transformed internally before being passed along to the fragment parser.

While the first option would be preferable as it requires less steps to parse, it is incompatible with black box parsers, as these would generally not support the passing of this information to them.

11.3.3 Cross-Fragment References

As part of hybrid languages, links need to be able to be made from fragments in one formalism to fragments in another formalism. While for visual concrete syntaxes this is more easily done by physically drawing a link between elements from different formalisms, in the textual world this is less straightforward.

For starters, in order to create a link between two elements in text, those elements would need to either be right next to each other, with a “linking” operator between the two. Or, one or both elements would need to have a name, i.e. an *identifier*, to reference by.

The first case can be unwieldy and difficult to manage in hyper-connected graphs, and doesn’t allow linking to a sub-element of a defined element. Thus the second case is a more appealing alternative. However, in order to be able to reference an element from one formalism in another, the grammars need to be compatible with regards to the format of identifiers. For example, one formalism may declare identifiers to support all Unicode letters [37], while the other may limit it to alpha-numeric characters. Referencing an element from the former in the latter may thus be severely limited or impossible.

A possible solution to this could be the introduction of a mini-language that only serves as a way to reference otherwise unreferenceable elements. This however would also require additional effort in order to properly support in the language.

11.4 Similar Techniques

In this section we will describe some techniques for parsing that involve multiple formalisms, but are fundamentally different from multi-language parsing such as we define it.

PHP The PHP (*PHP: Hypertext Preprocessor*) language is a widely used programming language for developing websites, and for the construction of web pages. A common pattern is to write HTML pages with fragments of PHP interspersed to dynamically create page content when a web client requests a page from a server. In essence, this is the combination of HTML and PHP, where the combined pieces form the semantics of a web document (i.e. the structure of the page).

However, as the name for PHP implies, this is done through a pre-processor, which only looks for fragments of PHP code. Text outside of fragments is sent verbatim to the client, while text inside the fragments is parsed, and output is only sent to the client through `echo` and `print` expressions (and similar).

This means that for a web server using PHP, the server only cares about the parsing of PHP, while the client is concerned with the parsing of the output of the preprocessor, usually in HTML. Thus in effect, this is a two-step parsing process (not considering CSS or JavaScript embedded in the HTML), performed on different machines.

C Preprocessor Similar to the PHP preprocessor, the C language (and by extension C++) contains a preprocessing step that is run before lexers are run. This preprocessor has more limited functionality than found in PHP, and instead works on string replacement. Additionally, it is considered part of the programming language itself, and as such may be considered to not be a hybrid language.

HTML The HTML (*HyperText Markup Language*) language, being derived from XML (*Extensible Markup Language*), is a markup language, meaning it controls the structure, formatting, and relationships between elements defined in it. HTML parsers are highly specialized and optimized, written to be both fast but also extremely forgiving for syntactical errors.

Two special elements, `<script>` and `<style>`, have a special meaning whereby they hold JavaScript and CSS fragments respectively. For scripts, the parsing of its content happens as they are encountered in the source document (specifically, upon parsing the `</script>` close tag, the script is evaluated). Thus, for example, the parsing of a document may be halted for as long as the script is running. Alternatively, a `<script>` element may define an external resource as the script to be parsed, which can be specified to run synchronously, asynchronously, or deferred up until the point the rest of the document has been parsed.

For style elements, a similar mechanism is run for the parsing of its contents. External styles are loaded through a separate mechanism (a `<link>` element), but this does not block parsing of the document.

Given all this, HTML has a similar mechanism for multi-language parsing, but lacks a mechanism for dynamically selecting a language to parse, as well as no facilities to dynamically create new languages.

Templating Languages Templating languages are similar to PHP and preprocessors, as they parse a language that gets transformed to some other language that then gets parsed by some other parser. They may be a simple “transform to text” language, or they may be built on top of the expected output language, having their syntax conform to a modified version (for example, based on HTML with extra syntax).

Language Injection A feature provided in IDEs developed by JetBrains is the ability to “inject a language or reference”. This feature allows marking a text element in a source file as containing a different language and is used solely for providing the assistive features of the IDE in a string that would otherwise not benefit from them.

This method works by transforming the text content, removing any character escapes, and then running it through a separate parser. As such, there is no connection between the parsing of the outer language and the contained fragments.

11.5 Dynamic Multi-Language Parsing

What we call *dynamic multi-language parsing* is an extension to the multi-language parsing concept, and is one of the goals of this project. Conceptually, the dynamic (see Chapter 2) part indicates the ability to, at run-time, add languages to the list of languages that can be parsed by the multi-language parser. This means that for example, a language engineer could write the definition of a language, and within the same file then start using that definition to parse a fragment of the language. We will look further into how we implement this in Section 17.2, but we note some difficulties that are associated with dynamic multi-language parsing.

The biggest issue lies in the fact that, in order to create a parser from a specification, the specification needs to be parsed *and* accessible. This means that the specification always needs to happen before the usage, similar to how some programming languages (such as C and Java) require variables to be declared before they can be referred to. It also means that we need to keep track of parsed fragments in parallel to the parse tree of the fragment we are currently parsing, as the parse tree may be fragmented and incomplete while the parser decides what action to take. Additionally, it requires the parser to evaluate the semantics of a sub-tree, even if it was given by a different parser, which increases complexity of the parser.

Part III
Implementation

CHAPTER 12

Overview

In this part we will take a rough look at the implementation of the WEAVE framework using the models given in Part II. Our implementation itself uses Python 3.10 as a programming language, and makes use of the Lark parsing library in order to handle basic functionality of parsing languages. Specification of parsers is done through the use of a WEAVE grammar language, which has its semantics implemented as code.

Chapter 13 gives information on some basic constructs and types used in the implementation, that do not do much on their own, but are to be used by other parts of our implementation. Following that, Chapter 14 gives details on how tracing information is saved, and how it is used to give feedback to a language engineer.

Chapter 15 details information on the parse tree formalisms TDF, TIF, and TCF, including errors that may be returned during semantic analysis of invalid models. Next, in Chapter 16 we provide details on the WEAVE grammar language such as the syntax, and for defining a language as being a hybrid language (i.e. can contain other fragments). This is then followed by an explanation of the implementation of the Lark backend, and how the Lark library is used in order to achieve dynamic multi-language parsing. Lastly, we provide an overview of the bootstrapping process in Chapter 18, where we recursively define the grammars and tree models for the formalisms introduced using themselves.

CHAPTER 13

Supporting Types

In this chapter we will look at some supporting types that are used by the implementation, but do not fit under any of the other chapters.

The WEAVE framework extensively makes use of a generic `Result` type, which can either hold a ‘success’ value, or an ‘error’ value. The type is modeled after the `Result` type found in the Rust programming language as an enum type (see Listing 13.1). The goal of this type is to limit the amount of exception handling code, as well as allowing to return multiple errors at once. Generally, results are typed as:

```
Result[T, List[ModelError]]
```

```
enum Result<R, E> {  
    Ok(T),  
    Err(E),  
}
```

Listing 13.1: Definition of the Rust `Result` type.

The `ModelError` type is used as a base type for all WEAVE errors, and provides the ability to assign a `TraceElement` (see Chapter 10) to it in order to point at what the error specifically relates to. It additionally provides the facilities to report the error to a `MessageReporter`, which can report the error to the user. Right now, the only reporter implementation outputs to the console, but it should be possible to output to for example an IDE through the implementation of a Language Server [17]. See Table 13.1 for a list of direct subtypes for `ModelError`. Subtypes of `ModelDefinitionError` and `ModelConstructionError` are listed in the sections detailing the implementation of parse trees.

ModelError Subtype	Description
<code>CustomError</code>	Used in the absence of a specific subtype
<code>InternalError</code>	Returned if an unexpected internal failure occurred
<code>NoImplementation</code>	Indicates a specific feature has not yet been implemented
<code>UnexpectedToken</code>	Returned if an unexpected token was encountered
<code>UnexpectedCharacter</code>	Returned if an unexpected character was encountered
<code>UnexpectedEndOfFile</code>	Returned if the end of the file is reached prematurely
<code>UnexpectedEndOfFragment</code>	Returned if the end of a fragment is reached prematurely
<code>ModelDefinitionError</code>	Subtypes returned for errors in TDF models, see Table 15.1
<code>ModelConstructionError</code>	Subtypes returned for errors in TCF models, see Table 15.2

Table 13.1: List of direct subtypes of `ModelError`.

The `TracePrinter` class is an implementation of `MessageReporter` and reports messages to the console (see Listing 13.2 for an example). Its job is to give a human-readable report that allows for easily determining what went wrong. For example, if a syntax error is encountered, or an invalid reference is found, it should display information on *what* went wrong, *where* it went wrong, and *why* it went wrong. That means it should include the file, the location in the file, a description of the message, and a preview of the text where it went wrong.

As we have three types of `BaseTraceElement` (see Chapter 10), and it may be possible there is either no base trace element (rarely), or no trace element at all, there are five specific cases of printing. While only those with a base trace element can provide useful information on where the problem is, the other cases only happen during development on WEAVE itself and are thus not a problem. See Listing 13.3 for its interface.

```
grammars/tdf.weave:5:50 error:
  Incompatible type: expected Element, got Element[]

    | elements=element* {TreeDefinitionFormalism(elements)}
                                ~~~~~
```

Listing 13.2: Example output from `TracePrinter` to the console.

```
enum MessageKind {
    Note
    Warning
    Error
}

class TracePrinter implements MessageReporter {
    report(message: String, kind: MessageKind,
           source: TraceElement[0..1])

    print_no_source(message: String, kind: MessageKind)

    print_not_base(message: String, kind: MessageKind,
                   source: TraceElement)

    print_source_location(message: String, kind: MessageKind,
                           source: TraceSourceLocation)

    print_source_range(message: String, kind: MessageKind,
                       source: TraceSourceRange)

    print_source_collection(message: String, kind: MessageKind,
                             source: TraceSourceCollection)
}
```

Listing 13.3: Class definition for the TracePrinter class, in pseudo-code.

CHAPTER 14

Tracing

In this chapter we will look at implementation notes for tracing elements as defined in Chapter 10. The structure of the implementation is almost exactly as given in Figure 10.2, with the only differences being:

- `TraceLocation` is named `TraceSourceLocation`
- `TraceRange` is named `TraceSourceRange`
- `TraceCollection` is named `TraceSourceCollection`
- `Location` is named `SourceURL`

Functionally these classes do not have much to them, only a `find_base` operation that returns themselves, as they are already a base trace element.

However, the `SourceURL` type implements a way of identifying the location of a file/-document/textual model/etc. As the name suggests it is based on URLs (Uniform Resource Locators), which use a ‘scheme’ (usually `file`), ‘network location’ (not set for local files), ‘path’ (the location of the file), and a ‘query’ and ‘fragment’ which we don’t use.

While a `SourceURL` is referenced to by `TraceSourceLocation`, it also provides the facilities to create instances of `TraceSourceLocation` and `TraceSourceRange` as a helper function, which should be the main method of creating them.

A subclass of `SourceURL` exists named `MappedSourceURL`, its purpose is to translate source locations such as encountered during parsing of fragments, which see a substring of the original document text as input. The actual translation of positions is performed by a ‘mapping function’, and these can be chained together in order to properly support nested fragments.

14.1 Source Resolvers

A concept used in the implementation but not discussed in the model overview is the ‘source resolver’. It has two purposes relating to tracing, and specifically to `SourceURL` instances:

1. returning a human-readable representation of the URL; and
2. getting the contents of the a URL for use by a `TracePrinter` (see Chapter 13), to allow it to print the source text.

A `SourceResolver` is given as an abstract class with the following implementations:

- `EmptySourceResolver`: does not resolve sources or format URLs.
- `FilesystemSourceResolver`: resolves source URLs with the ‘file’ scheme, and attempts to format the location to a relative location (relative to the current working directory).
- `BootstrapResolver`: resolves source URLs with the ‘weave-bootstrap’ scheme, a special scheme to reference the grammars and models used during bootstrapping (see Chapter 18).
- `ChainedSourceResolver`: uses several sub-resolvers to satisfy source retrieval or formatting requests. Order of use depends on the order that sub-resolvers were specified.

The goal of source resolvers is to support using WEAVE inside software such as a Language Server with the Language Server Protocol [17], which may run in an isolated container so that direct file access is impossible. A special scheme such as for example ‘language-client’ may be used to identify files made accessible by the Language Client.

We note that using URLs for sources is very powerful but also potential dangerous. We provide no implementation for retrieving sources over the network, using a non-empty network location, as it is both out of scope for this project and a bad idea in general.

In Chapter 8 we described our vision for parse trees, defining formalisms for defining the shape (TDF, see Section 8.2.1), the representation of instances (TIF, see Section 8.2.3), and the construction of trees from parts (TCF, see Section 8.2.4). Additionally in Section 8.3 we provided how these formalisms should be used in conjunction, and in Section 8.4 we gave a specification on how to parse each of them, including a definition of the TDF models for each formalism, and TCF models for each grammar production, as would be used by an actual parser using our parse tree formalisms. As such, not much needed to be left to the imagination for the actual implementation.

We will mention some of the important parts of the implementation, without going into too much detail, in the following sections. Note that the extensions to TDF and TIF in Section 11.1 are simple in nature and do not require any special consideration, they are equivalent to the `TokenType` and `TokenInstance` types.

15.1 Tree Definition

No major differences exist between the implementation and the abstract syntax as given in Section 8.2.1. The only differences are an absence of the types `NodeChild` and `NamedTypeReference`, which were only defined in the abstract syntax due to UML not supporting mapping types directly. In our implementation, these are implemented using Python dictionary objects, where the key is the name, and the value is the type.

As was noted in Section 8.2.1, the detection of cycles in in the type hierarchy is done through finding strongly connected components. Specifically, Tarjan's algorithm is used in a re-usable manner through a generic 'graph-like' type that only requires information

on vertices and their successors, without needing to know how these are stored. Using strongly connected components allows us to find loops in the type hierarchy, while also letting us know *where* these loops are, that is: which types are involved in the loop.

WEAVE further provides alternative versions of the `NodeTypeReference` classes by way of the parallel `UnresolvedReference` class and subclasses. The goal of these classes is to provide a way to reference a type by name, rather than by direct link. This may be useful in the case of parsing a grammar specification before knowing the TDF model for the grammar, and can be resolved to a proper type at a later type, or if the referenced type does not exist, can be annotated as being in error.

ModelDefinitionError Subtype	Description
<code>InheritanceLoop</code>	Indicates there is a cycle in the inheritance hierarchy
<code>DuplicateType</code>	Indicates a type name was used more than once
<code>DuplicateChild</code>	Indicates a type has two or more children with the same name
<code>ShadowedChild</code>	Indicates a type redeclares a child from a parent
<code>UnknownType</code>	Indicates a reference to an undefined node type
<code>InterfaceCannotExtend</code>	Indicates an interface parent types list containing a non-interface type
<code>ConcreteCannotExtend</code>	Indicates an interface parent types list containing a non-extendable type

Table 15.1: List of direct subtypes of `ModelDefinitionError`.

15.2 Tree Instance

In the specification for TIF, no implementation was given for `TokenInstance`, as this lies outside the scope of the formalisms themselves. For WEAVE, we used a simple implementation with two fields: the type of the token, and the value (text) of the token. See also Figure 15.1 for the abstract syntax.

TokenInstance
type: <code>String</code>
value: <code>String</code>

Figure 15.1: Abstract syntax of the implementation of the `TokenInstance` type for our implementation of TIF in WEAVE.

15.3 Tree Construction

No special differences exist between the implementation and the definition given in Sections 8.2.4 and 8.2.5. However, we note that the implementation of these operations was done as code in `Python`, rather than using a graph transformation library and execution schedule such as provided. Instead, the graph transformations and schedules were translated to `Python` manually.

In order to ensure the operations in TCF work properly, unit tests were added in order to validate them. One special case which was not accounted for in the specification is returning an empty list as node value, which would given the ‘`typeIntersection`’ algorithm in Listing 8.5 fail, as the intersection of the empty set returns the empty set. This would result in an error that the types cannot be unified, and is worked around through a special type `EmptyListType` that is assignable to any list type.

ModelConstructionError Subtype	Description
<code>IncompatibleType</code>	Returned when an expression is being assigned or returned that is not assignable to the expected type
<code>ExpectedValueSort</code>	Returned when the inferred value type of an expression is not valid to use in a context, such as attempting to use a list concatenation with non-list value types
<code>TypeNotInstantiable</code>	Indicates a type in a <code>NodeConstructOperation</code> is not instantiable, i.e. not a concrete type
<code>ValueNotFound</code>	Indicates the name referenced in a <code>ContextAccessOperation</code> is not found in the definition context
<code>TypeNotFound</code>	Indicates a referenced type does not exist in the TDF model
<code>ChildNotFound</code>	Returned when trying to access a child that does not exist in a <code>NodeAccessOperation</code>
<code>ExcessChild</code>	Indicates excess children are being assigned in a <code>NodeConstructOperation</code>
<code>MissingChild</code>	Indicates a child that is not optional is not being assigned in a <code>NodeConstructOperation</code>
<code>DuplicateChild</code>	Indicates a child being assigned to more than once in a <code>NodeConstructOperation</code>
<code>CannotUnifyTypes</code>	Returned when the ‘typeIntersection’ algorithm returns an empty set, meaning there is no common ancestry of types
<code>AmbiguousTypeIntersection</code>	Returned when the ‘typeIntersection’ algorithm returns more than one type, meaning the type of a list expression may be interpreted in more than one way

Table 15.2: List of direct subtypes of `ModelConstructionError`.

CHAPTER 16

Grammar Specification

As part of being able to define languages that can be used for multi-language parsing, we also needed to be able to describe the languages themselves. Specifically, it is necessary to have as much information regarding the parsing of a language in the same place as possible, that is:

- The terminals and how they should be matched, or whether or not they should be discarded (such as for comments and whitespace).
- The non-terminals and their productions.
- Type information on non-terminals using TDF models as annotations.
- Construction operations of the parse tree using TCF models as annotations to productions.
- The specification of where fragments can happen, and what they look like. Also includes ‘internal’ fragments or islands, such as strings or blocks surrounded by delimiters.
- Information on special processing, such as for indentation sensitive languages.

To this end we developed a separate grammar specification language, which we call the `WEAVE` grammar language, which we based initially on the `Lark` grammar language [31], but with the addition of TDF and TCF models in the grammar now very closely resembles the PEG grammar language used by `Python` [29].

Listing 16.1 shows an example grammar specification for a Causal Block Diagram language, with the accompanying TDF model given in Listing 16.2. It shows the following syntax features:

- Non-terminals are defined using a name, followed by a TDF fragment surrounded by square brackets to indicate the type.
- Non-terminal productions can contain inline token strings, and can assign ‘names’ or ‘aliases’ to each part in the production (separated by spaces).
- Non-terminal parts can have eBNF syntax elements. Shown is the Kleene star operation ‘*’. Other operators supported are the ‘+’ and ‘?’ operators as shown in Section 8.3.1, as well as the separator operator ‘o’, written using a ‘.’
- Terminals are specified using either strings (" ... ") or regular expressions (/ ... /).
- Terminals can be marked as ‘ignored’ using a concept called a *directive*, which will make the lexer output still output the token, but the parser will ignore the token when encountered. This is usually used to allow a cleaner grammar, such as allowing a optional spaces between operators without having to explicitly put them in the grammar.
- ‘Internal’ fragments are specified using the *fragment* directive, which specifies that an opening curly brace must at some point be followed by a closing curly brace.
- On line 19, the specification of an fragment embedding is given, which allows recursively embedding CBD models. See Section 16.1 for more information on this.

A complete overview of the syntax definition of WEAVE is given in Listing A.1, specified using the WEAVE grammar language itself. The accompanying TDF model is given in Listing A.2.

```

1 start[CausalBlockDiagram] :
2   | elements=element* {CausalBlockDiagram(elements)}
3
4 element[Element] :
5   | "block" name=ID ":" type=ID ";" {BlockInstance(name, type)}
6   | "constant" name=ID "=" value=NUMBER ";"
7     {ConstantBlock(name, value)}
8   | "inport" name=ID ";" {InPort(name)}
9   | "outport" name=ID ";" {OutPort(name)}
10  | "connect" from=port_ref "to" to=port_ref ";" {Link(from, to)}
11  | "composite" name=ID contents=cbd_embed
12    {CompositeDefinition(name, contents)}
13
14 port_ref[PortReference] :
15   | block=ID "." port=ID {PortReference(block, port)}
16   | block=ID {PortReference(block)}
17
18 cbd_embed[embed] :
19   | "{" @embed("cbd") "}" {embed}
20
21
22 NUMBER  : /[+-]?([0-9]+(\.[0-9]*)?|\.[0-9]+)/
23 ID      : /[\_a-zA-Z][\_a-zA-Z0-9]*/
24 WS      : /[\_t\r\n]+/
25 COMMENT : /\s*\/\[/[\^{}\\n]*/
26
27 @ignore(WS, COMMENT)
28
29 @fragment("{", "}")

```

Listing 16.1: Example WEAVE grammar specification for a Causal Block Diagram language.

```
CausalBlockDiagram(elements: Element [])

interface Element()

CompositeDefinition(name: tok, contents: embed): Element

interface Block(name: tok): Element
ConstantBlock(value: tok): Block
BlockInstance(type: tok): Block

interface Port(name: tok): Element
InPort(): Port
OutPort(): Port

Link(from: PortReference, to: PortReference): Element
PortReference(block: tok, port: tok?)
```

Listing 16.2: Example TDF model for a Causal Block Diagram language.

16.1 Embed Specifications

Specifications of language embed fragments use the same syntax as non-terminals, but use `@embed` directives in order to define them as an embedding point in the language. An embed definition can effectively consist of five different elements:

- Terminals or terminal specifications
- A `@embed` directive with parameters, specifying which language and optionally which start symbol (if supported) to use for parsing the fragment.
- A `@embed.select` directive, which gets parsed by WEAVE in order to decide which language an embed should be. Incompatible with the previously mentioned directive.
- A `@embed` directive without parameters, required with the previous directive, and specifies where the fragment should appear syntactically.
- A `@embed.options` directive that can pass options along to the parser, or be used to declare properties about the fragment.

When using a `@embed.select` directive, the `@embed` directive without parameters *must* appear after it. This is because the parser needs information on which language to parse before it can parse it.

We note that the `@embed.options` directive is unsupported in our implementation, but that the eventual goal is to allow it to specify behavior for the parser. For example, it may be necessary for indentation to be removed from a fragment, but the parser handles it incorrectly or not at all due to an improper or missing definition in the grammar. Another example would be to allow putting a name on a fragment, in order to allow referencing it from other locations.

In our current implementation, embed specifications are limited to two forms:

```
PRE_TOKEN @embed(name, rule) POST_TOKEN
```

and

```
PRE_TOKEN @embed.select MID_TOKEN @embed POST_TOKEN
```

This reflects our notes on the syntax of multi-language parsing in Section 11.2, and is used as a proof-of-concept implementation.

Note that on lines 74 and 77 of the grammar specification of WEAVE (see Listing A.1) we make use of embedding specifications, which means that the grammar language itself makes use embedded fragments, and is thus a hybrid language.

16.2 Indentation Sensitive Languages

The grammar specification exposes support specific for the Lark parsing backend in order enable processing for indentation sensitive languages that are similar to Python with regards to how these are handled. Specifically, it can output `INDENT` and `DEDENT` tokens based on whitespace, and supports suspending this process between tokens (such as between parentheses, curly braces, square brackets, etc.).

Support for this feature is enabled by way of a `@indent_aware` directive, which defines the indentation and dedentation tokens, and which token is processed in order to generate them. Additionally a `@indent_aware.suspend` directive can specify a suspension of this process.

While it is not a good idea to have parser-specific options in a grammar specification language that should be unaware of the parsing backend. This specific feature should be easy to implement in other backends as well however, as it only necessitates a step between lexing and parsing in order to support it. In the case of a scannerless parser, support would probably be best done by using a parser that supports layout sensitive parsing (see the paper by Erdweg et al. on Layout-sensitive Generalized Parsing [11]).

We note that this is new software which has not yet had much time to mature, and as such these sorts of things should not be taken as a statement of “this is how it should be done and always will be done, because it is the correct way.”

CHAPTER 17

Lark Implementation

As was mentioned a few times throughout the document, we chose to make an implementation of WEAVE using the Lark Python library [31] as a backend to handle parsing. Lark was chosen for its ability to support dynamic loading of parsers and wide range of language constructs [28], though other backends are also able to be added.

Figure 17.1 shows an overview of the different phases in going from a WEAVE grammar specification file to a parser that conforms to the `WeaveParser` interface (see Chapter 9). Note that in this process a new source text is generated from an original source text, which is then in turn handled by the Lark library. That is, WEAVE provides a front-end interface for generating parsers that can perform multi-language parsing using the Lark library.

Lark provides three main options to manipulate the parsing process:

- Setting a ‘postlex’ step, which puts an extra processor between the lexer and the parser. This processor can take tokens and discard them, create new tokens from them, or simply pass them on, but it cannot modify the lexer state.
- Setting a ‘transformer’, which runs upon successful completion of a non-terminal production, but is only supported on its LALR parser.
- Providing replacement implementations for its lexer and parser classes, giving complete control over their operations by subtyping them (or providing a complete different implementation). This is however fairly new and not well documented, and may be subject to change in the future.

Additionally, it’s possible to run a transformer on a parse tree after parsing succeeds using the same transformer type, replacing the tree in-place, but this is less efficient. Other interfaces for going over the parse tree are so called ‘visitors’ which do not change the

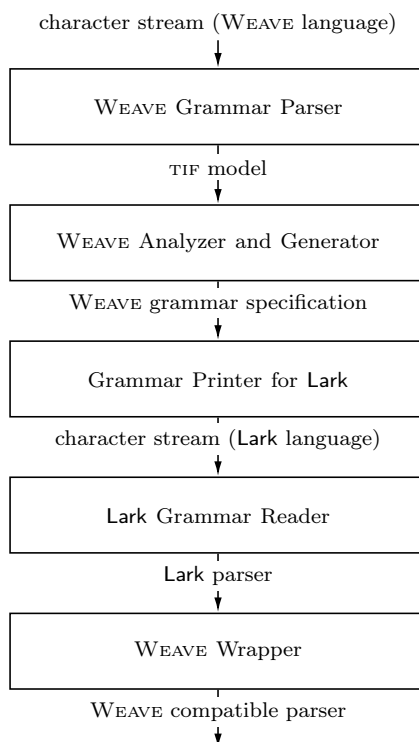


Figure 17.1: Diagram of the overall phases in generating a parser specified with the WEAVE grammar language, using the Lark backend.

tree, and ‘interpreters’ which does not change or recursively traverse the tree. Figure 17.2 provides an overview of these options as used by our implementation.

We use a postlex step on languages which are indentation sensitive, as this requires us inserting new tokens into the token stream, as well as track the current indentation level. As for the transformer in the parser, we do not use it at the moment as our tree building is done using an interpreter, which has different operating semantics, but this could be changed for more efficiency when parsing bigger models.

The addition of a ‘lexer hooks’ to the lexer in Figure 17.2 is done by providing a subtyped lexer implementation. Its goal is to work as a more powerful postlex step that can change the lexer state as well as insert or withhold tokens, which is necessary in order to handle embedded language fragments (see Section 11.2 and the next section).

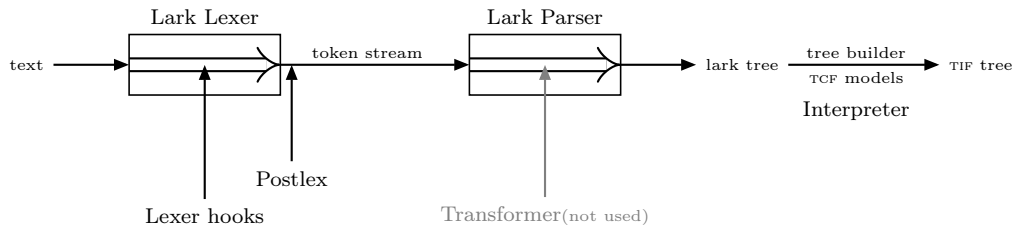


Figure 17.2: Diagram detailing the Lark parser architecture as we used it.

17.1 Multi-Language Parsing

As was explained previously in Section 11.2, it's not easy to communicate from the parser to the lexer that it should switch languages (backtracking may happen), and the lexer is where a language switch should be performed as it is the one with direct access to the source text.

Thus, in order to be able to provide a working proof of concept that allows multi-language parsing, the syntax was restricted to two specific variants, as seen in Figures 11.4 and 11.5 in Section 11.2, and more recently in Section 16.1.

To this end, for every language embed location specification we add a lexer hook for the `PRE_TOKEN`, that when encountered performs the necessary setup for switching language, such as described in a high-level overview in Figure 11.2.

Relevant operations are:

1. Detecting the language (if applicable).
2. Finding the fragment end using an island parsing algorithm.
3. Constructing a new character stream.
4. Transforming the character stream (removal of indentation, text transformations, etc.).
5. Initiating a new parse and waiting for the result.
6. Inserting the result in the token stream as a single token-like, with its value being the parse result.

We note that the transformation step is not implemented, but would potentially be nice to have.

The island parsing algorithm uses a grammar such as seen in Figure 11.8, making use of information available on the grammar of the fragment such as internal fragments and further embeds, as well as tokens that may contain the end fragment identifier. Due to the

limited possible shapes (only standalone tokens that are ignored, and tokens that occur in pairs), the implementation is done through a simple scanner algorithm that performs the same job. A future implementation may instead chose to use an actual parser for islands, that can then recursively handle nested fragments, when more generic shapes of embedding are allowed.

17.2 Dynamic Multi-Language Parsing

As we noted earlier in Section 11.5, in order to create a parser from a specification, we need to have the specification available. Generally, this happens by specifying and parsing the grammar beforehand in a separate file.

As our implementation completes parsing of a fragment before it continues parsing the rest of the document, we have the parse tree for it available. Unfortunately, this parse tree lacks any of the surrounding context, as it is supposed to be embedded in the parse tree of the parent fragment. Thus, proper semantic evaluation is currently not possible, meaning you can't for example conditionally define the grammar.

However, as a proof of context, we added a mini language that can be used to define a language, that can then be used to parse later on. See Listing A.7 for the grammar definition, and Listing A.8 for the tree model specification. See also Listing B.10 for an example where this sort of dynamic language definition is used as a proof of context.

CHAPTER 18

Bootstrapping

During the initialization of WEAVE, we perform a ‘bootstrapping’ operation of the languages that are used for the definition of languages themselves, i.e. the ‘meta’ languages. Specifically, the following languages are required in order to define a language:

- WEAVE grammar language, which is used to specify the grammar of a language.
- TDF for specifying the structure of parse trees. Also used by the grammar language for typing non-terminal rules.
- TCF for specifying construction rules of parse trees, used by the grammar language.

As a part of dogfooding our implementation, we specified the above languages using themselves. I.e. TCF is specified using a TDF model and WEAVE grammar, which in turn contains TCF models. Unfortunately, we currently have no way of storing a parsed TDF model or WEAVE grammar to disk, and as such need to have a different way of getting a basic version of these. To this end, we first attempt to build a parser for these languages using a different mechanism, and then use those parsers to load and build the effective grammars and tree models.

We start off with two models not defined using themselves: the TDF model of TDF is defined using Python code, and a Lark grammar specification for the WEAVE grammar is loaded and turned into a parser for WEAVE models. Bootstrapping then happens in several steps. We’ll use T for TDF models, G for WEAVE grammar specifications, and P for parsers. A subscript indicates the formalism, while a superscript annotates how they were made or what they are used for.

1. The base TDF model is instantiated through code, we’ll call this T_{tdf}^{code} .
2. A Lark parser is created that can parse WEAVE grammars, which we’ll call P_{weave}^{lark} .

-
3. The WEAVE, TDF and TCF grammars are parsed using P_{weave}^{lark} . We'll call these grammars G_{weave}^{lark} , G_{tdf}^{lark} and G_{tcf}^{lark} respectively.
 4. Using T_{tdf}^{code} and G_{tdf}^{lark} , we create a WEAVE parser for TDF P_{tdf}^{boot} .
 5. Using P_{tdf}^{boot} we load the TDF models for WEAVE and TCF, named T_{weave}^{boot} and T_{tcf}^{boot} respectively.
 6. We create a parser for TCF: P_{tcf}^{boot} using T_{tcf}^{code} and G_{tcf}^{lark} .
 7. We create a parser for WEAVE: P_{weave}^{boot} using T_{weave}^{boot} and G_{weave}^{lark} . This parser depends on P_{tdf}^{boot} and P_{tcf}^{boot} using multi-language parsing.
 - At this point we have TDF model and WEAVE grammar for WEAVE, TDF and TCF, as well as parsers, which we can then use to load the effective tree models, grammars, and parsers.
 8. Using P_{weave}^{boot} and P_{tdf}^{boot} we load G_{tdf} and T_{tdf} respectively, which we combine into P_{tdf} .
 9. Using P_{weave}^{boot} and P_{tcf}^{boot} we load G_{tcf} and T_{tcf} respectively, which we combine into P_{tcf} .
 10. Using P_{weave}^{boot} and P_{tdf}^{boot} we load G_{weave} and T_{weave} respectively, which we combine into P_{weave} .

See also Figure 18.1 for a schematic overview of this process.

Note that the creation of a parser from a TDF model and WEAVE grammar specification is covered in Chapter 17.

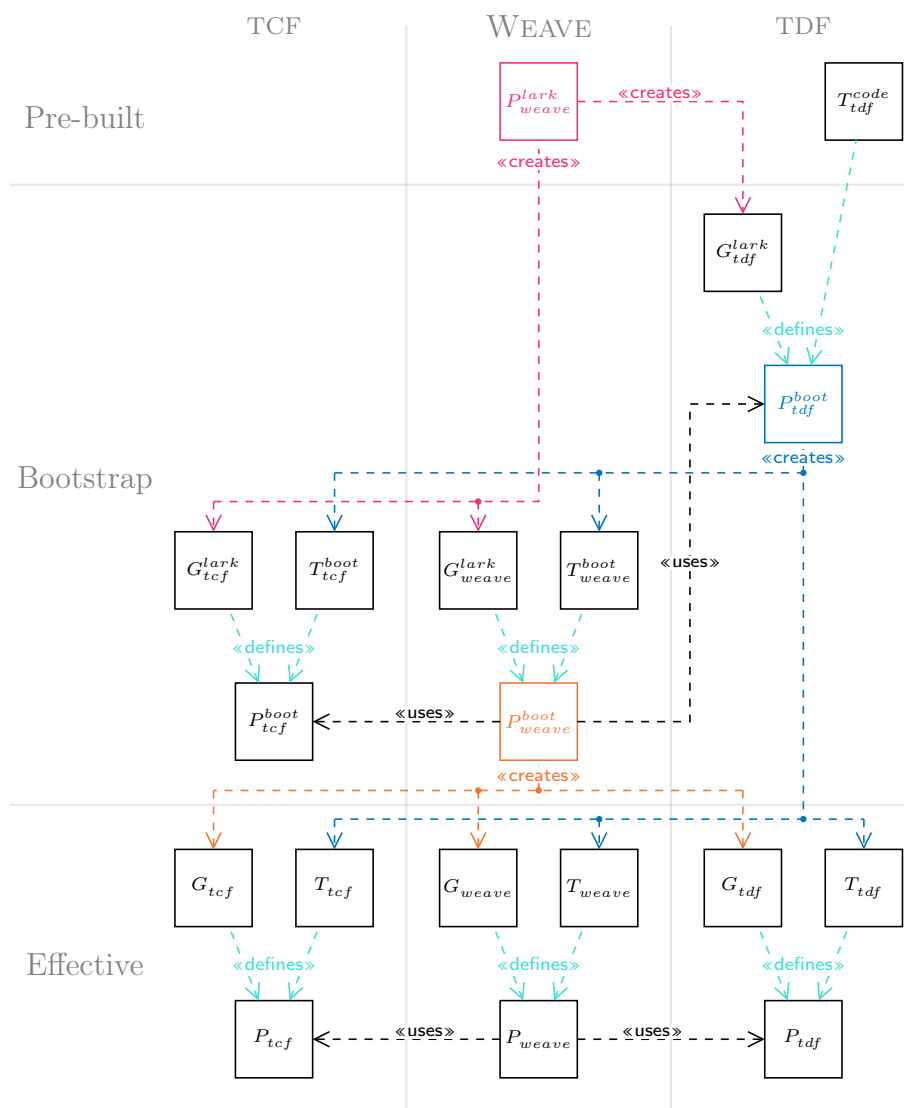


Figure 18.1: Schematic overview of the bootstrapping process.

Part IV

End

We introduced a reusable and extendable set of formalisms for the definition and construction of parse trees that are defined in a language-agnostic way: the Tree Definition Formalism (TDF) for defining their shape, the Tree Instance Formalism (TIF) for instances of parse trees, and the Tree Construction Formalism (TCF) for building instances of parse trees.

We also introduced concepts and procedures for the handling of hybrid textual models using multi-language parsing through the use of multiple distinct parsers, and a way to handle the switching between parsers during parsing.

As an example implementation we produced the WEAVE framework, that uses the Python Lark library to handle parsing, while extending this library in order to add the functionality for multi-language parsing, and a proof-of-concept implementation for dynamic multi-language parsing. This framework also includes a formalism for defining grammars that parse hybrid languages, which we named the WEAVE grammar language.

19.1 Future Work

As mentioned in the introduction, WEAVE is only a first step into having a language workbench or IDE for the development of domain specific languages. Further work will involve using WEAVE as a library for the parsing of hybrid models, which can then also be used as a Language Server for the Language Server Protocol [17].

Further work can also focus on refining the TDF and TCF formalisms, such as introducing the Liskov substitution principle, and using type unification instead of type inference

on expressions. Additionally, instructions for the handling of types inside eBNF groupings could be developed, and a DSL for converting a TIF model into abstract syntax instances.

With regards to parsing, a system for pre-processing fragment text may also be beneficial. This would be run after the fragment text has been extracted, but before it is sent to the fragment parser. Use cases for this could be the removal of indentation for fragments that are indentation sensitive, other sorts of string transformations that may be necessary to make an embedded model parse (in case the language wasn't explicitly designed to support being embedded).

Furthermore, relaxing the syntax for language fragments or finding a different mechanism for initiating a language switch can use further research, as these are only at the proof-of-concept stage.

Other possible future additions to WEAVE is more parsing backends, such as using PLY or ANTLR4, and a formalism for augmenting existing grammars such as extending the grammar like in MontiCore [15], [30], or redeclaring or replacing parts of the grammar like in Modelica [19].

Bibliography

- [1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley series in computer science / World student series edition). Addison-Wesley, 1986, ISBN: 0-201-10088-6. [Online]. Available: <https://www.worldcat.org/oclc/12285707>.
- [2] P. Anderson, “The performance penalty of XML for program intermediate representations”, in *5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005), 30 September - 1 October 2005, Budapest, Hungary*, IEEE Computer Society, 2005, pp. 193–202. DOI: 10.1109/SCAM.2005.25. [Online]. Available: <https://doi.org/10.1109/SCAM.2005.25>.
- [3] P. Boullier, “Dynamic grammars and semantic analysis”, INRIA, Research Report RR-2322, 1994, Projet CHLOE. [Online]. Available: <https://hal.inria.fr/inria-00074352>.
- [4] M. van den Brand, H. de Jong, P. Klint, and P. Olivier, “Efficient annotated terms”, *Softw. Pract. Exp.*, vol. 30, no. 3, pp. 259–291, 2000. DOI: 10.1002/(SICI)1097-024X(200003)30:3<259::AID-SPE298>3.0.CO;2-Y. [Online]. Available: [https://doi.org/10.1002/\(SICI\)1097-024X\(200003\)30:3%5C%3C259::AID-SPE298%5C%3E3.0.CO;2-Y](https://doi.org/10.1002/(SICI)1097-024X(200003)30:3%5C%3C259::AID-SPE298%5C%3E3.0.CO;2-Y).
- [5] M. Bunge, “A general black box theory”, *Philosophy of Science*, vol. 30, no. 4, pp. 346–358, 1963.
- [6] 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>.

- [7] 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>.
- [8] 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).
- [9] B. Combemale, R. France, J.-M. Jézéquel, B. Rumpe, J. Steel, and D. Vojtisek, *Engineering modeling languages: Turning domain knowledge into tools*. CRC Press, 2016.
- [10] A. van Deursen and T. Kuipers, “Building documentation generators”, in *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999*, IEEE Computer Society, 1999, pp. 40–49. DOI: 10.1109/ICSM.1999.792497. [Online]. Available: <https://doi.org/10.1109/ICSM.1999.792497>.
- [11] 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.
- [12] A. M. Fard, A. Deldari, and H. Deldari, “Quick grammar type recognition: Concepts and techniques”, *CoRTA’2007*, p. 51, 2007.
- [13] B. Ford, “Parsing expression grammars: A recognition-based syntactic foundation”, in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, N. D. Jones and X. Leroy, Eds., ACM, 2004, pp. 111–122. DOI: 10.1145/964001.964011. [Online]. Available: <https://doi.org/10.1145/964001.964011>.
- [14] R. Heinrich, F. Durán, C. L. Talcott, and S. Zschaler, Eds., *Composing Model-Based Analysis Tools*. Springer, 2021, ISBN: 978-3-030-81914-9. DOI: 10.1007/978-3-030-81915-6. [Online]. Available: <https://doi.org/10.1007/978-3-030-81915-6>.
- [15] H. Krahn, B. Rumpe, and S. Völkel, “Monticore: A framework for compositional development of domain specific languages”, *Int. J. Softw. Tools Technol. Transf.*, vol. 12, no. 5, pp. 353–372, 2010. DOI: 10.1007/s10009-010-0142-1. [Online]. Available: <https://doi.org/10.1007/s10009-010-0142-1>.

- [16] E. Mamas and K. Kontogiannis, “Towards portable source code representations using XML”, in *Proceedings of the Seventh Working Conference on Reverse Engineering, WCRE’00, Brisbane, Australia, November 23-25, 2000*, IEEE Computer Society, 2000, p. 172. DOI: 10.1109/WCRE.2000.891464. [Online]. Available: <https://doi.org/10.1109/WCRE.2000.891464>.
- [17] Microsoft. “Official page for Language Server Protocol”, [Online]. Available: <https://microsoft.github.io/language-server-protocol/> (visited on 08/04/2021).
- [18] D. Mikulin, M. Vijayasundaram, and L. Wong, “Incremental linking on HP-UX”, in *Proceedings of the First Workshop on Industrial Experiences with Systems Software, WIESS 2000, October 22, 2000, San Diego, CA, USA*, D. S. Milojicic, Ed., USENIX, 2000, pp. 47–56. [Online]. Available: <http://www.usenix.org/events/wiess2000/mikulin.html>.
- [19] Modelica Association, *Modelica© - A Unified Object-Oriented Language for Systems Modeling*. 2017. [Online]. Available: <https://modelica.org/documents/ModelicaSpec34.pdf>.
- [20] L. Moonen, “Generating robust parsers using island grammars”, in *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE’01, Stuttgart, Germany, October 2-5, 2001*, E. Burd, P. Aiken, and R. Koschke, Eds., IEEE Computer Society, 2001, p. 13. DOI: 10.1109/WCRE.2001.957806. [Online]. Available: <https://doi.org/10.1109/WCRE.2001.957806>.
- [21] S. Mustafiz, B. Barroca, C. Gomes, and H. Vangheluwe, “Towards modular language design using language fragments: The hybrid systems case study”, in *Information Technology: New Generations*, Springer, 2016, pp. 785–797.
- [22] S. Mustafiz, C. Gomes, B. Barroca, and H. Vangheluwe, “Modular design of hybrid languages by explicit modeling of semantic adaptation”, in *Proceedings of the Symposium on Theory of Modeling & Simulation, TMS/DEVS 2016, part of the 2016 Spring Simulation Multiconference, SpringSim ’16, Pasadena, CA, USA, April 3-6, 2016*, F. Barros, H. Prähofer, X. Hu, and J. Denil, Eds., ACM, 2016, p. 29. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2975418>.
- [23] Object Management Group, *Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM)*. 2011. [Online]. Available: <https://www.omg.org/spec/ASTM/1.0/PDF>.
- [24] Object Management Group, *Object Constraint Language*. 2014. [Online]. Available: <https://www.omg.org/spec/OCL/2.4/PDF>.
- [25] Object Management Group, *OMG© Unified Modeling Language© (OMG UML©)*. 2017. [Online]. Available: <https://www.omg.org/spec/UML/2.5.1/PDF>.

- [26] R. Paredis, J. Denil, and H. Vangheluwe, “Specifying and executing the combination of timed finite state automata and causal-block diagrams by mapping onto devs”, in *Winter Simulation Conference, WSC 2021, Phoenix, AZ, USA, December 12-15, 2021*, IEEE, 2021, pp. 1–12. DOI: 10.1109/WSC52266.2021.9715387. [Online]. Available: <https://doi.org/10.1109/WSC52266.2021.9715387>.
- [27] L. Presser and J. R. White, “Linkers and loaders”, *ACM Comput. Surv.*, vol. 4, no. 3, pp. 149–167, 1972. DOI: 10.1145/356603.356605. [Online]. Available: <https://doi.org/10.1145/356603.356605>.
- [28] M. Pyl and H. Vangheluwe, “Comparison of parsing and editing tools for multi-language parsing”, Research Project, University of Antwerp, 2021.
- [29] G. van Rossum, P. Galindo, and L. Nikolaou, “New PEG parser for CPython”, PEP 617, 2020. [Online]. Available: <https://peps.python.org/pep-0617/>.
- [30] B. Rumpe, K. Hölldobler, and O. Kautz, *MontiCore Language Workbench and Library Handbook*. 2021. [Online]. Available: <https://se-rwth.de/publications/MontiCore-Language-Workbench-and-Library-Handbook-Edition-2021.pdf>.
- [31] 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).
- [32] 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).
- [33] H. Stachowiak, *Allgemeine modelltheorie*. Springer, 1973.
- [34] N. Synytsky, J. R. Cordy, and T. R. Dean, “Robust multilingual parsing using island grammars”, in *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative Research, October 6-9, 2003, Toronto, Ontario, Canada*, D. A. Stewart, Ed., IBM, 2003, pp. 266–278. [Online]. Available: <https://dl.acm.org/citation.cfm?id=961364>.
- [35] E. Syriani and H. Vangheluwe, “A modular timed graph transformation language for simulation-based design”, *Softw. Syst. Model.*, vol. 12, no. 2, pp. 387–414, 2013. DOI: 10.1007/s10270-011-0205-0. [Online]. Available: <https://doi.org/10.1007/s10270-011-0205-0>.
- [36] 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).
- [37] The Unicode Consortium, *The Unicode[®] Standard - Version 13.0 - Core Specification*. 2020.

- [38] Y. Van Tendeloo, B. Barroca, S. Van Mierlo, and H. Vangheluwe, “Modelverse specification”, *University of Antwerp, Tech. Rep*, 2016.
- [39] E. Vergnaud. “The prompto platform”. <http://prompto.org/>, Accessed on 2021-02-25., [Online]. Available: <http://prompto.org/> (visited on 02/25/2021).
- [40] E. Visser, *Scannerless generalized-LR parsing*. Universiteit van Amsterdam. Programming Research Group, 1997.
- [41] T. A. Wagner and S. L. Graham, “Efficient and flexible incremental parsing”, *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 5, pp. 980–1013, 1998. DOI: 10.1145/293677.293678. [Online]. Available: <https://doi.org/10.1145/293677.293678>.
- [42] M. Woerister. “Incremental Compilation | Rust Blog”. (Sep. 8, 2016), [Online]. Available: <https://blog.rust-lang.org/2016/09/08/incremental.html> (visited on 07/28/2021).
- [43] V. Zaytsev, “BNF was here: What have we done about the unnecessary diversity of notation for syntactic definitions”, in *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, S. Ossowski and P. Lecca, Eds., ACM, 2012, pp. 1910–1915. DOI: 10.1145/2245276.2232090. [Online]. Available: <https://doi.org/10.1145/2245276.2232090>.

Appendices

APPENDIX A

Grammars and Models

```
1 start[WeaveGrammar] :
2   | _NL* elements=element* {WeaveGrammar(elements)}
3
4 element[Element] :
5   | element=directive_element _NL* {element}
6   | element=terminal _NL*      {element}
7   | element=non_terminal _NL*  {element}
8
9 terminal[TerminalSpecification] :
10  | name=NAME ":" pattern=literal _NL {TerminalSpecification(name, pattern)}
11
12 literal[tok] :
13   | REGEXP {REGEXP}
14   | STRING {STRING}
15
16 non_terminal[NonTerminalSpecification] :
17   | name=NAME type=tdf_specification ":" _NL alternatives=root_alternatives {
18     NonTerminalSpecification(name, type, alternatives)}
19
20 root_alternatives[Alternative[]] :
21   | alternatives=root_alternative+ {alternatives}
22
23 root_alternative[Alternative] :
24   | "|" parts=expr* builder=tdcf_specification _NL {Alternative(parts, builder)}
25 // Expressions
```

```

26 expr[Expression] :
27   | expr=atom_expr      {expr}
28   | expr=opt_expr      {expr}
29   | expr=star_expr     {expr}
30   | expr=plus_expr     {expr}
31   | expr=directive_expr {expr}
32
33 atom_expr[Expression] :
34   | alias=NAME "=" base=atom {AtomExpression(alias, base)}
35   | base=atom                {AtomExpression(base)}
36
37 opt_expr[Expression] :
38   | alias=NAME "=" base=atom "?" {OptionalExpression(alias, base)}
39   | base=atom "?"                {OptionalExpression(base)}
40
41 star_expr[Expression] :
42   | alias=NAME "=" base=atom "*" {RepeatStarExpression(alias, base)}
43   | base=atom "*" {RepeatStarExpression(base)}
44   | alias=NAME "=" separator=atom "." base=atom "*" {RepeatDotStarExpression(
45     alias, base, separator)}
46   | separator=atom "." base=atom "*" {RepeatDotStarExpression(base, separator)}
47
48 plus_expr[Expression] :
49   | alias=NAME "=" base=atom "+" {RepeatPlusExpression(alias, base)}
50   | base=atom "+" {RepeatPlusExpression(base)}
51   | alias=NAME "=" separator=atom "." base=atom "+" {RepeatDotPlusExpression(
52     alias, base, separator)}
53   | separator=atom "." base=atom "+" {RepeatDotPlusExpression(base, separator)}
54
55 atom[Atom] :
56   | name=NAME      {NameAtom(name)}
57   | pattern=REGEXP {LiteralAtom(pattern)}
58   | pattern=STRING {LiteralAtom(pattern)}
59
60 directive_expr[DirectiveExpression] :
61   | directive {DirectiveExpression(directive)}
62
63 // Embedded elements
64 tdf_specification[embed] :
65   | "[" @embed("tdf", "type_reference") "]" {embed}
66
67 tcf_specification[embed] :
68   | "{" @embed("tcf") "}" {embed}

```

```

69
70 // Directives
71 directive_element[RootDirective] :
72   | directive _NL {RootDirective(directive)}
73
74 directive[Directive] :
75   | "@" name=NAME {DirectiveBase(name)}
76   | base=directive "." derived=NAME {DirectiveDerived(base, derived)}
77   | base=directive "(" params=",".directive_param* ")" {DirectiveParams(base,
      params)}
78
79 directive_param[tok] :
80   | NAME {NAME}
81   | STRING {STRING}
82
83
84 // Terminals
85 NAME      : /[_a-zA-Z][_a-zA-Z0-9]*/
86 STRING    : /"(\\"|\\\\\\|[\^"\n])*?"/i?/
87 REGEXP    : /\(?!\/)(\\\/|\\\\\\|[\^\/])*?\/[imslux]*/
88 _NL       : /(\r?\n)+\s*/
89 WS        : /[_\t]+/
90 COMMENT   : /((?!\\n)\s)*\/\/[\^\\n]*/
91
92 @ignore(WS, COMMENT)
93
94 @fragment("(", ")")
95 @fragment("[", "]")
96 @fragment("{", "}")

```

Listing A.1: WEAVE grammar specification, in WEAVE.

```

1 WeaveGrammar(elements: Element[])
2
3 interface Element()
4
5 TerminalSpecification(name: tok, pattern: tok): Element
6
7 NonTerminalSpecification(name: tok, type: embed, alternatives:
  Alternative[]): Element
8
9 Alternative(parts: Expression[], builder: embed)
10
11 // Expressions
12 interface Expression()
13
14 OptionalExpression(alias: tok?, base: Atom): Expression
15
16 RepeatStarExpression(alias: tok?, base: Atom): Expression
17 RepeatDotStarExpression(alias: tok?, base: Atom, separator: Atom):
  Expression
18
19 RepeatPlusExpression(alias: tok?, base: Atom): Expression
20 RepeatDotPlusExpression(alias: tok?, base: Atom, separator: Atom):
  Expression
21
22 AtomExpression(alias: tok?, base: Atom): Expression
23
24 // Atoms
25 interface Atom()
26 NameAtom(name: tok): Atom
27 LiteralAtom(pattern: tok): Atom
28
29 RootDirective(directive: Directive): Element
30
31 // Directives
32 interface Directive()
33 DirectiveExpression(directive: Directive): Expression
34
35 DirectiveBase(name: tok): Directive
36 DirectiveDerived(base: Directive, derived: tok): Directive
37 DirectiveParams(base: Directive, params: tok[]): Directive

```

Listing A.2: TDF model for WEAVE.

```

1 start[TreeDefinitionFormalism] :
2   | elements=element* {TreeDefinitionFormalism(elements)}
3
4 element[Element] :
5   | interface_definition {interface_definition}
6   | concrete_definition {concrete_definition}
7
8 interface_definition[InterfaceDefinition] :
9   | "interface" name=ID "(" children=",".node_child* ")" parents {
10      InterfaceDefinition(name, children, super_types=parents)}
11
12 concrete_definition[ConcreteDefinition] :
13   | "node"? name=ID "(" children=",".node_child* ")" parents {ConcreteDefinition
14      (name, children, super_types=parents)}
15
16 node_child[NodeChild] :
17   | name=ID ":" type=type_reference {NodeChild(name, type)}
18
19 type_reference[TypeReference] :
20   | type=ID {BaseTypeReference(type)}
21   | type=ID "?" {OptionalTypeReference(type)}
22   | type=ID "[" "]" {ListTypeReference(type)}
23
24 parents[tok[]] :
25   | {}
26   | ":" parents=",".ID+ {parents}
27
28 ID      : /[_a-zA-Z][_a-zA-Z0-9]*/
29 WS      : /[_\t\r\n]*/
30 COMMENT : /\s*\//[^\\n]*/
31
32 @ignore(WS, COMMENT)
33 @fragment("[", "]")
34 @fragment("(", ")")

```

Listing A.3: TDF grammar specification, in WEAVE.

```

1 TreeDefinitionFormalism(elements: Element[])
2
3 interface Element()
4
5 interface NodeDefinition(name: tok): Element
6 InterfaceDefinition(children: NodeChild[], super_types: tok[]):
    NodeDefinition
7 ConcreteDefinition(children: NodeChild[], super_types: tok[]):
    NodeDefinition
8
9 NodeChild(name: tok, type: TypeReference)
10
11 interface TypeReference()
12 BaseTypeReference(type: tok): TypeReference
13 OptionalTypeReference(type: tok): TypeReference
14 ListTypeReference(type: tok): TypeReference

```

Listing A.4: TDF model for TDF.

```

1 start[Operation] :
2   | operation {operation}
3
4 operation[Operation] :
5   | operation=context_access {operation}
6   | operation=node_access {operation}
7   | operation=node_construct {operation}
8   | operation=list_construct {operation}
9   | operation=list_concat {operation}
10
11 context_access[ContextAccess] :
12   | name=ID {ContextAccess(name)}
13
14 node_access[NodeAccess] :
15   | base=node_access_base "." name=ID {NodeAccess(base, name)}
16
17 node_access_base[Operation] :
18   | base=context_access {base}
19   | base=node_access {base}
20   | base=node_construct {base}
21
22 node_construct[NodeConstruct] :
23   | id=ID "(" children=",".node_construct_child* ")" {NodeConstruct(type=id,
    children)}
24
25 node_construct_child[NodeConstructChild] :

```

```

26 | name=ID "=" value=operation {NodeConstructChild(name, value)}
27 | name=ID {NodeConstructChild(name, value=ContextAccess(name))}
28
29 list_construct[ListConstruct] :
30 | "[" elements=",".list_construct_element* "]" {ListConstruct(elements)}
31
32 list_construct_element[Operation] :
33 | element=context_access {element}
34 | element=node_access {element}
35 | element=node_construct {element}
36
37 list_concat[ListConcat] :
38 | head=list_concat_element ".." tail="..".list_concat_element+ {ListConcat(
    lists=[head] .. tail)}
39
40 list_concat_element[Operation] :
41 | element=context_access {element}
42 | element=node_access {element}
43 | element=list_construct {element}
44
45
46 ID      : /[_a-zA-Z][_a-zA-Z0-9]*/
47 WS      : /[_\t]*/
48 COMMENT : /\s*\//[^\\n]*/
49
50 @ignore(WS, COMMENT)
51
52 @fragment("[", "]")
53 @fragment("(", ")")

```

Listing A.5: TCF grammar specification, in WEAVE.

```

1 interface Operation()
2 node ContextAccess(name: tok): Operation
3 node NodeAccess(base: Operation, name: tok): Operation
4 node NodeConstruct(type: tok, children: NodeConstructChild[]):
5     Operation
6 node ListConstruct(elements: Operation[]): Operation
7 node ListConcat(lists: Operation[]): Operation
8
9 node NodeConstructChild(name: tok, value: Operation)

```

Listing A.6: TDF model for TCF.

```

1 start[LanguageDefinition] :

```



```

2   | _NL* name backend tree_model grammar {LanguageDefinition(name, backend,
      tree_model, grammar)}
3
4   name[tok] :
5     | "name" "=" name=ID _NL+ {name}
6
7   backend[tok] :
8     | "backend" "=" backend=ID _NL+ {backend}
9
10  tree_model[embed] :
11    | "tree" "model" "=" language_embed _NL+ {language_embed}
12
13  grammar[embed] :
14    | "grammar" "=" language_embed _NL+ {language_embed}
15
16
17  // Embedded elements
18  language_embed[embed] :
19    | ">" @embed.select "{" @embed "}" {embed}
20
21
22  ID      : /[a-zA-Z_0-9]+(:[a-zA-Z_0-9]+)?/
23  _NL     : /(\r?\n)+\s*/
24  WS      : /[\t]+/
25  COMMENT : /((?!\\n)\\s)*\\\/[^\n]*/
26
27  @ignore(WS, COMMENT)
28
29  @fragment("{", "}") // Should not be necessary

```

Listing A.7: Grammar specification for the `language_define` mini-language, used for dynamic language specifications. Written in WEAVE.

```

1  TreeDefinitionFormalism(elements: Element[])
2
3  interface Element()
4
5  interface NodeDefinition(name: tok): Element
6  InterfaceDefinition(children: NodeChild[], super_types: tok[]):
      NodeDefinition
7  ConcreteDefinition(children: NodeChild[], super_types: tok[]):
      NodeDefinition
8
9  NodeChild(name: tok, type: TypeReference)
10

```

```
11 interface TypeReference()  
12 BaseTypeReference(type: tok): TypeReference  
13 OptionalTypeReference(type: tok): TypeReference  
14 ListTypeReference(type: tok): TypeReference
```

Listing A.8: TDF model for the language_define language.

APPENDIX B

Example Models

```
1 start[Graph] :
2   | elements=element* {Graph(elements)}
3
4 element[Element] :
5   | name=ID "=" model_embed {Vertex(name, value=model_embed)}
6   | "connect" from=ID "to" to=ID {Edge(from, to)}
7
8 model_embed[embed] :
9   | ">" @embed.select "{" @embed "}" {embed}
10
11
12 ID      : /[_a-zA-Z][_a-zA-Z0-9]*/
13 WS      : /[_\t\r\n]*/
14 COMMENT : /\s*\//[^\\n]*/
15
16 @ignore(WS, COMMENT)
17
18 @fragment("{", "}")
```

Listing B.1: WEAVE grammar specification for the example graph (Directed Graph) language.

```

1 Graph(elements: Element [])
2
3 interface Element()
4
5 Vertex(name: tok, value: embed): Element
6
7 Edge(from: tok, to: tok): Element

```

Listing B.2: TDF model for the example graph (Directed Graph) language.

```

1 start[TimedFiniteStateAutomaton] :
2   | elements=element* {TimedFiniteStateAutomaton(elements)}
3
4 element[Element] :
5   | initial="initial"? final="final"? "state" name=ID state_type? state_entry?
6     state_exit? ";" {State(name, initial, final, state_type, enter_action=
7     state_entry, exit_action=state_exit)}
8   | "transition" "from" from=ID "to" to=ID trigger=transition_trigger? action=
9     transition_run? ";" {Transition(from, to, trigger, action)}
10  | name=ID "=" model_embed {EmbeddedDefinition(name, value=model_embed)}
11
12 state_type[tok] :
13   | ":" type=ID {type}
14
15 state_entry[embed] :
16   | "on" "entry" statements_embed {statements_embed}
17
18 state_exit[embed] :
19   | "on" "exit" statements_embed {statements_embed}
20
21 transition_trigger[Trigger] :
22   | "on" event=ID {EventTrigger(event)}
23   | "after" time=NUMBER {TimeoutTrigger(time)}
24   | "when" condition=condition_embed {ConditionTrigger(condition)}
25
26 transition_run[embed] :
27   | "do" statements_embed {statements_embed}
28
29 // Embeds
30 condition_embed[embed] :
31   | "[" @embed("alc", "expression") "]" {embed}
32
33 statements_embed[embed] :
34   | "{" @embed("alc", "block") "}" {embed}

```

```

33
34 model_embed[embed] :
35   | ">" @embed.select "{" @embed "}" {embed}
36
37
38 // Tokens
39 NUMBER  : /[+-]?([0-9]+(\.[0-9]*)?|\.[0-9]+)/
40 ID      : /[_a-zA-Z][_a-zA-Z0-9]*/
41 WS      : /[\t\r\n]+/
42 COMMENT : /\s*\/\[/[^\n]*/
43
44 @ignore(WS, COMMENT)
45
46 @fragment("[", "]")
47 @fragment("{", "}")

```

Listing B.3: WEAVE grammar specification for the example `tfsa` (Timed Finite State Automaton) language.

```

1 TimedFiniteStateAutomaton(elements: Element[])
2
3 interface Element()
4
5 State(name: tok, initial: tok?, final: tok?, state_type: tok?,
6       enter_action: embed?, exit_action: embed?): Element
7
8 Transition(from: tok, to: tok, trigger: Trigger?, action: embed?):
9   Element
10
11 interface Trigger()
12 EventTrigger(event: tok): Trigger
13 TimeoutTrigger(time: tok): Trigger
14 ConditionTrigger(condition: embed): Trigger
15
16 EmbeddedDefinition(name: tok, value: embed): Element

```

Listing B.4: TDF model for the example `tfsa` (Timed Finite State Automaton) language.

```

1 start[CausalBlockDiagram] :
2   | elements=element* {CausalBlockDiagram(elements)}
3
4 element[Element] :
5   | "block" name=ID ":" type=ID ";" {BlockInstance(name, type)}
6   | "constant" name=ID "=" value=NUMBER ";" {ConstantBlock(name, value)}
7   | "inport" name=ID ";" {InPort(name)}
8   | "outport" name=ID ";" {OutPort(name)}
9   | "connect" from=port_ref "to" to=port_ref ";" {Link(from, to)}
10  | "composite" name=ID contents=cbd_embed {CompositeDefinition(name, contents)}
11  | "tfssa" name=ID embed=tfssa_embed {EmbedBlockDefinition(name, embed)}
12
13 port_ref[PortReference] :
14  | block=ID "." port=ID {PortReference(block, port)}
15  | block=ID {PortReference(block)}
16
17 cbd_embed[embed] :
18  | "{" @embed("cbd:tfssa") "}" {embed}
19
20 tfssa_embed[embed] :
21  | "(" @embed("tfssa:alc") ")" {embed}
22
23
24 NUMBER : /[+-]?([0-9]+(\.[0-9]*)?|\.[0-9]+)/
25 ID      : /[_a-zA-Z][_a-zA-Z0-9]*/
26 WS      : /[ \t\r\n]+/
27 COMMENT : /\s*\//[^ \n]*/
28
29 @ignore(WS, COMMENT)

```

Listing B.5: WEAVE grammar specification for the example cbd (Causal Block Diagrams) language.

```

1 CausalBlockDiagram(elements: Element[])
2
3 interface Element()
4
5 CompositeDefinition(name: tok, contents: embed): Element
6 EmbedBlockDefinition(name: tok, embed: embed): Element
7
8 interface Block(name: tok): Element
9 ParameterBlock(): Block
10 ConstantBlock(value: tok): Block
11 BlockInstance(type: tok): Block
12

```

```

13 interface Port(name: tok): Element
14   InPort(): Port
15   OutPort(): Port
16
17   Link(from: PortReference, to: PortReference): Element
18   PortReference(block: tok, port: tok?)

```

Listing B.6: TDF model for the example cbd (Causal Block Diagrams) language.

```

1 start[Unit] :
2   | elements=element* {Unit(elements)}
3
4 element[Element] :
5   | include _NL* {include}
6   | definition _NL* {definition}
7   | function _NL* {function}
8
9 include[Include] :
10  | "include" from=STRVALUE _NL {Include(from)}
11
12 function[Function] :
13  | pre=function_prefix ":" block {FunctionDef(type=pre.type, name=pre.name,
14    params=pre.params, block)}
15  | pre=function_prefix "=" "?" referenced=ANYTHING _NL {FunctionRef(type=pre.
16    type, name=pre.name, params=pre.params, referenced)}
17
18 function_prefix[FunctionPrefix] :
19  | type=function_type "mutable"? "function" name=ID "(" params=",".
20    function_parameter* ")" {FunctionPrefix(type, name, params)}
21
22 function_type[tok] :
23  | type_spec {type_spec}
24  | type="Void" {type}
25
26 function_parameter[FunctionParameter] :
27  | name=ID ":" type_spec {FunctionParameter(name, type=type_spec)}
28
29 type_spec[tok] :
30  | type="Integer" {type}
31  | type="Float" {type}
32  | type="Boolean" {type}
33  | type="String" {type}
34  | type="Type" {type}
35  | type="Action" {type}
36  | type="Element" {type}

```

```

34
35 block[Block] :
36   | _NL INDENT statements DEDEDENT {Block(statements)}
37
38 statements[Statement[]] :
39   | statements=statement+ {statements}
40
41 statement[Statement] :
42   | definition _NL* {definition}
43   | assignment _NL+ {assignment}
44   | return _NL+ {return}
45   | function_call _NL+ {function_call}
46   | if_statement _NL* {if_statement}
47   | while_statement _NL* {while_statement}
48   | continue _NL+ {continue}
49   | break _NL+ {break}
50
51 definition[Definition] :
52   | type=type_spec name=ID _NL {Definition(type, name)}
53   | type=type_spec name=ID "=" value=atomvalue _NL {Definition(type, name, value
54   )}
55
56 assignment[Assignment] :
57   | lhs=lvalue "=" rhs=expression {Assignment(lhs, rhs)}
58
59 return[Return] :
60   | "return" expression? "!" {Return(expression)}
61
62 function_call[FunctionCall] :
63   | function=rvalue "(" params=",".expression* ")" {FunctionCall(function,
64   params)}
65
66 if_statement[IfElse] :
67   | "if" expression ":" block elif=elif_statement {IfElse(if_block=IfBlock(
68   expression, block), alt_blocks=elif.elifs, else_block=elif.else)}
69   | "if" expression ":" block else=else_statement? {IfElse(if_block=IfBlock(
70   expression, block), alt_blocks=[], else_block=else)}
71
72 elif_statement[IfElseTail] :
73   | "elif" expression ":" block tail=elif_statement {IfElseTail(elifs=[IfBlock(
74   expression, block)] .. tail.elifs, else=tail.else)}
75   | "elif" expression ":" block else=else_statement? {IfElseTail(elifs=[IfBlock(
76   expression, block)], else)}
77
78 else_statement[Block] :

```



```

73   | "else" ":" block {block}
74
75 while_statement[While] :
76   | "while" expression ":" block {While(expression, block)}
77
78 continue[Continue] :
79   | "continue" "!" {Continue()}
80
81 break[Break] :
82   | "break" "!" {Break()}
83
84 lvalue[LValue] :
85   | name=ID {Name(name)}
86
87 rvalue[RValue] :
88   | base=rvalue "[" expression "]" {Index(base, expression)}
89   | name=ID {Name(name)}
90
91 expression[Expression] :
92   | binary_operation {binary_operation}
93
94 binary_operation[Expression] :
95   | disjunction {disjunction}
96
97 disjunction[Expression] :
98   | disjunction "or" conjunction {Disjunction(lhs=disjunction, rhs=conjunction)}
99   | conjunction {conjunction}
100
101 conjunction[Expression] :
102   | conjunction "and" comparison {Conjunction(lhs=conjunction, rhs=comparison)}
103   | comparison {comparison}
104
105 comparison[Expression] :
106   | comparison "==" relation {Equals(lhs=comparison, rhs=relation)}
107   | comparison "!=" relation {NotEquals(lhs=comparison, rhs=relation)}
108   | relation {relation}
109
110 relation[Expression] :
111   | relation "<" sum {LessThan(lhs=relation, rhs=sum)}
112   | relation ">" sum {GreaterThan(lhs=relation, rhs=sum)}
113   | relation "<=" sum {LessThanEquals(lhs=relation, rhs=sum)}
114   | relation ">=" sum {GreaterThanEquals(lhs=relation, rhs=sum)}
115   | sum {sum}
116
117 sum[Expression] :

```

```

118 | sum "+" term {Plus(lhs=sum, rhs=term)}
119 | sum "-" term {Minus(lhs=sum, rhs=term)}
120 | term {term}
121
122 term[Expression] :
123 | term "*" factor {Times(lhs=term, rhs=factor)}
124 | term "/" factor {Divide(lhs=term, rhs=factor)}
125 | factor {factor}
126
127 factor[Expression] :
128 | "not" primary {LogicalNot(expression=primary)}
129 | "-" primary {InvertSign(expression=primary)}
130 | "+" primary {KeepSign(expression=primary)}
131 | primary {primary}
132
133 primary[Expression] :
134 | "(" expression ")" {expression}
135 | rvalue {rvalue}
136 | function_call {function_call}
137 | atomvalue {atomvalue}
138
139 atomvalue[AtomValue] :
140 | string {string}
141 | value=DEC_NUMBER {Integer(value)}
142 | value=FLOAT_NUMBER {Float(value)}
143 | value="True" {Boolean(value)}
144 | value="False" {Boolean(value)}
145 | value=type_spec {TypeSpec(value)}
146 | action_name {action_name}
147 | deref {deref}
148
149 string[String] :
150 | value=STRVALUE {String(value)}
151 | value=LONG_STRVALUE {String(value)}
152
153 // IF_NODE | WHILE_NODE | ASSIGN_NODE | CALL_NODE | BREAK_NODE | CONTINUE_NODE |
    RETURN_NODE | RESOLVE_NODE
154 // | ACCESS_NODE | CONSTANT_NODE | GLOBAL_NODE | DECLARE_NODE | INPUT_NODE |
    OUTPUT_NODE | NONE_NODE
155 action_name[ActionName] :
156 | value="!if" {ActionName(value)}
157 | value="!while" {ActionName(value)}
158 | value="!assign" {ActionName(value)}
159 | value="!call" {ActionName(value)}
160 | value="!break" {ActionName(value)}

```

```

161 | value="!continue" {ActionName(value)}
162 | value="!return" {ActionName(value)}
163 | value="!resolve" {ActionName(value)}
164 | value="!access" {ActionName(value)}
165 | value="!constant" {ActionName(value)}
166 | value="!global" {ActionName(value)}
167 | value="!declare" {ActionName(value)}
168 | value="!input" {ActionName(value)}
169 | value="!output" {ActionName(value)}
170 | value="!none" {ActionName(value)}
171
172 deref[Dereference] :
173 | "?" name=ANYTHING? {Dereference(name)}
174
175
176 // Terminals
177 ID : /[_a-zA-Z][_a-zA-Z0-9.]*/
178 ANYTHING : /[_a-zA-Z0-9.]*/
179 _NL : /(\r?\n\s*|\n)/
180 WS : /[_\t]*/
181 COMMENT : /((?!\\n)\\s)*\\n/
182 DEC_NUMBER : /[+-]?([0-9]+|\\.[0-9]+|\\.[0-9]+[eE][+-]?[0-9]+)/
183 FLOAT_NUMBER : /[+-]?((\\d+\\.\\d*|\\.\\d+)([eE][+-]?\\d+)?)|\\d+[eE][+-]?\\d+/
184 STRVALUE : /u?r?("(?!").*(?<\\)\\\\\\\\)*?"|'(?!')'.*(?<\\)\\\\\\\\)*?'')/
185 LONG_STRVALUE : /(?s)u?r?("".*(?<\\)\\\\\\\\)*?"|'''.*(?<\\)\\\\\\\\)*?'')/
186
187 @ignore(WS, COMMENT)
188
189 @fragment("(", ")")
190 @fragment("[", "]")
191
192 @indent_aware(INDENT, DEDENT, _NL)
193 @indent_aware.suspend("(", ")")
194 @indent_aware.suspend("[", "]")

```

Listing B.7: WEAVE grammar specification for the example `alc` (ActionLanguage) language.

```

1 Unit(elements: Element[])
2
3 interface Element()
4
5 Include(from: tok): Element
6
7 interface Function(type: tok, name: tok, params: FunctionParameter

```

```

    []): Element
8  FunctionParameter(name: tok, type: tok)
9  FunctionDef(block: Block): Function
10 FunctionRef(referenced: tok): Function
11 FunctionPrefix(type: tok, name: tok, params: FunctionParameter[])
12
13 Block(statements: Statement[]): Element
14
15 interface Statement()
16
17 Definition(type: tok, name: tok, value: AtomValue?): Statement,
    Element
18
19 Assignment(lhs: LValue, rhs: Expression): Statement
20
21 Return(expression: Expression?): Statement
22
23 FunctionCall(function: RValue, params: Expression[]): Statement,
    Expression
24
25 ElseIf(if_block: IfBlock, alt_blocks: IfBlock[], else_block: Block
   ?): Statement
26 IfBlock(expression: Expression, block: Block)
27 ElseIfTail(elifs: IfBlock[], else: Block?)
28
29 While(expression: Expression, block: Block): Statement
30
31 Continue(): Statement
32 Break(): Statement
33
34 interface Expression()
35 BinaryExpression(lhs: Expression, rhs: Expression): Expression
36 Disjunction(): BinaryExpression
37 Conjunction(): BinaryExpression
38 Equals(): BinaryExpression
39 NotEquals(): BinaryExpression
40 LessThan(): BinaryExpression
41 GreaterThan(): BinaryExpression
42 LessThanEquals(): BinaryExpression
43 GreaterThanEquals(): BinaryExpression
44 Plus(): BinaryExpression
45 Minus(): BinaryExpression
46 Times(): BinaryExpression
47 Divide(): BinaryExpression
48 LogicalNot(expression: Expression): Expression

```

```

49 InvertSign(expression: Expression): Expression
50 KeepSign(expression: Expression): Expression
51
52 interface AtomValue(): Expression
53 String(value: tok): AtomValue
54 Integer(value: tok): AtomValue
55 Float(value: tok): AtomValue
56 Boolean(value: tok): AtomValue
57 TypeSpec(value: tok): AtomValue
58 ActionName(value: tok): AtomValue
59 Dereference(name: tok?): AtomValue
60
61 interface LValue()
62 interface RValue(): Expression
63
64 Name(name: tok): LValue, RValue
65 Index(base: RValue, expression: Expression): RValue

```

Listing B.8: TDF model for the example alc (ActionLanguage) language.

```

1 // Root language: graph language
2
3 // Selection of the block non-terminal
4 values = >alc#block{
5   Float h = 10.0
6   Float v = 0.0
7   Float EPS = 0.01 // Epsilon
8 }
9
10 automaton = >tfsa{
11   // Define a state behaviour
12   // Select "cbd" language here explicitly, transition actions are
13   implicit
14   FallGravity = >cbd{
15     // CBD fragment, operationally unaware of outer level
16     parameter h_0;
17     parameter v_0;
18     constant g = 9.81;
19
20     block v: Integrator;
21     block h: Integrator;
22     block neg_g: Negator;
23
24     connect g      to neg_g;

```

```

25     // dv/dt = -g
26     connect neg_g to v;
27     connect v_0   to v.IC;
28
29     // dh/dt = v
30     connect v     to h;
31     connect h_0   to h.IC;
32 }
33
34 initial state start;
35
36 state fallGravity: FallGravity; // Uses FallGravity
37
38 final state stopped;
39
40 transition from start to fallGravity do {
41     // ActionLanguage fragment
42     // Initialize fallGravity internal CBD
43     target.v_0 = parent.values.v
44     target.h_0 = parent.values.h
45 };
46
47 transition from fallGravity to fallGravity
48     when [fallGravity.h < 0]
49     do {
50         // ActionLanguage fragment
51         target.v_0 = -0.8 * from.v
52         target.h_0 = from.h
53     };
54
55 // When the input v_0 is less than EPS, stop
56 transition from fallGravity to stopped
57     when [abs(fallGravity.v_0) - EPS < 0]
58     do {
59         // ActionLanguage fragment
60         // Return values back to upper level
61         parent.values.v = 0
62         parent.values.h = from.h
63     };
64 }

```

Listing B.9: Example complex model combining a graph, CBD, ActionLanguage and TFSA formalism.

```

1 language = >language_define{

```

```

2   name = tfsa
3   backend = lark
4   tree model = >tdf{
5       TimedFiniteStateAutomaton(elements: Element[])
6
7       interface Element()
8
9       State(name: tok): Element
10
11      Transition(from: tok, to: tok, trigger: Trigger?): Element
12
13      interface Trigger()
14      EventTrigger(event: tok): Trigger
15      TimeoutTrigger(time: tok): Trigger
16  }
17  grammar = >weave{
18      start[TimedFiniteStateAutomaton] :
19          | elements=element* {TimedFiniteStateAutomaton(elements)}
20
21      element[Element] :
22          | "state" name=ID ";" {State(name)}
23          | "transition" "from" from=ID "to" to=ID trigger=
24              transition_trigger? ";" {Transition(from, to, trigger)}
25
26      transition_trigger[Trigger] :
27          | "on" event=ID {EventTrigger(event)}
28          | "after" time=NUMBER {TimeoutTrigger(time)}
29
30      NUMBER    : /[+-]?([0-9]+(\.[0-9]*)?|\.[0-9]+)/
31      ID        : /[_a-zA-Z][_a-zA-Z0-9]*/
32      WS        : /[\t\r\n]+/
33      COMMENT   : /\s*\#[^\n]*/
34
35      @ignore(WS, COMMENT)
36  }
37 }
38
39 some_tfisa = >tfisa{
40     state neutral;
41     state down;
42     state up;
43     state emergency;
44
45     transition from initial to neutral;

```

```
46 transition from neutral to down on d;  
47 transition from down to neutral on n;  
48 transition from neutral to up on u;  
49 transition from up to neutral on n;  
50 transition from up to emergency on e;  
51 transition from emergency to neutral after 1;  
52 }
```

Listing B.10: Example model that parses using dynamic multi-language parsing.

APPENDIX C

Parsing Tools Comparison

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 choose 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.

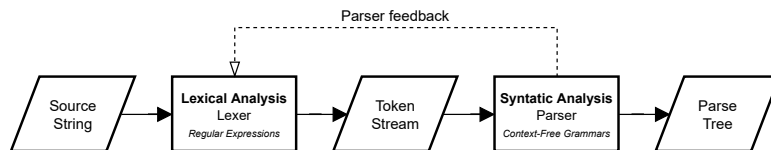


Figure 1: Overview of a parser architecture as it is usually implemented, with a lexing step (scanner/tokenizer) before the actual parsing. Usually there exists a mechanism for the parser to interact with the lexer, for example to change its state.

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.

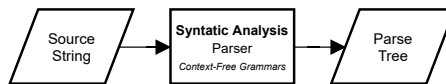


Figure 2: Overview of a parser architecture without lexing step.

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.

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

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

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.

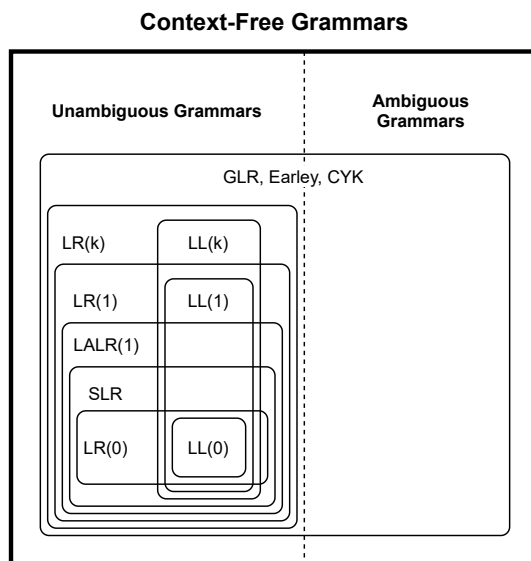


Figure 3: Hierarchical representation of Context-Free Grammar classes, adapted from [31].

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].

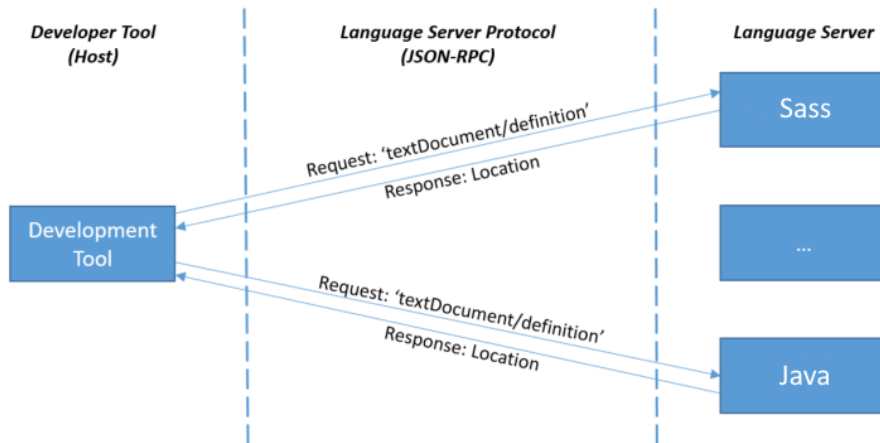


Figure 4: Illustration of the Language Server Protocol when a user using a development tool acting as the language client is working with multiple languages. The client starts multiple language servers that each process requests for their respective languages [71].

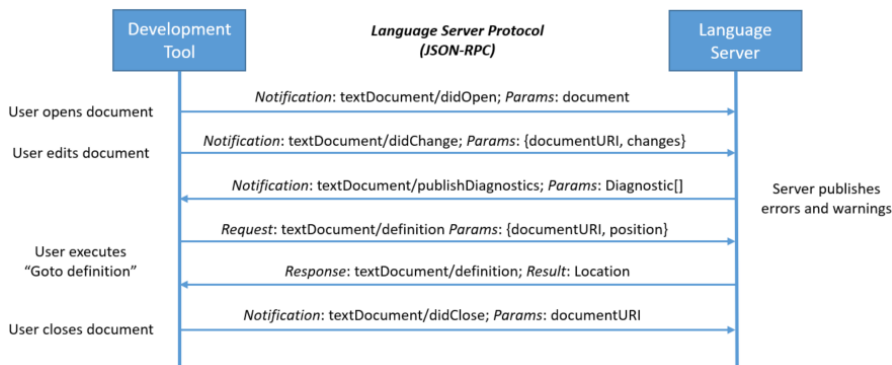


Figure 5: Illustration of the Language Server Protocol showing the communication between a language client and a single language server [71].

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.

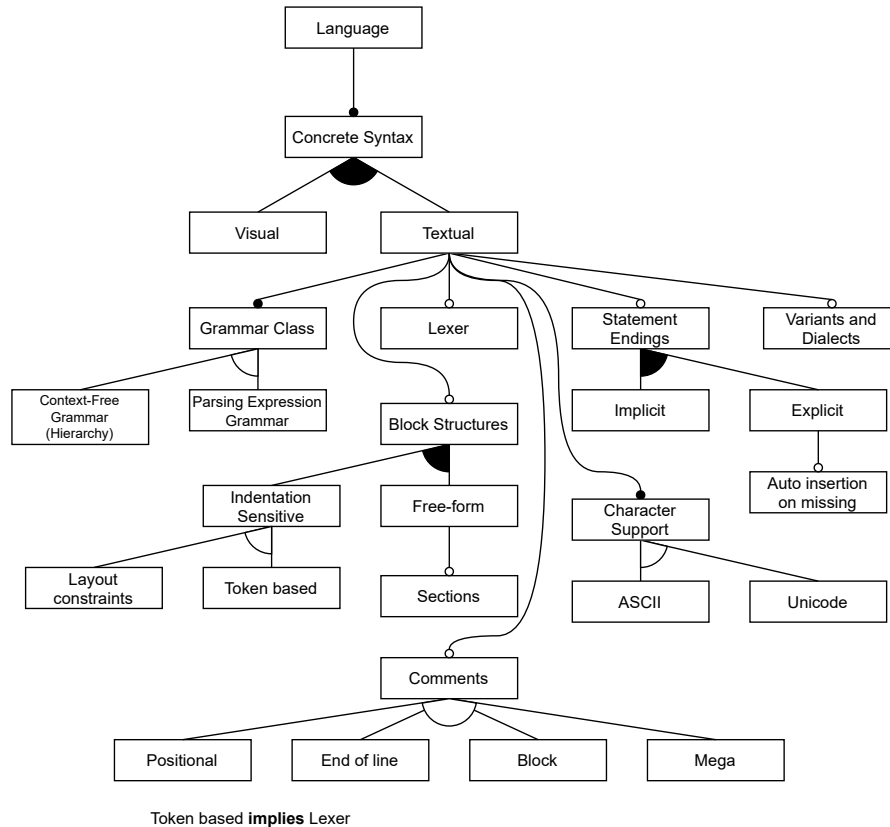


Figure 6: Visual representation of the variability model which we will use in our comparison in Section 6.

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.

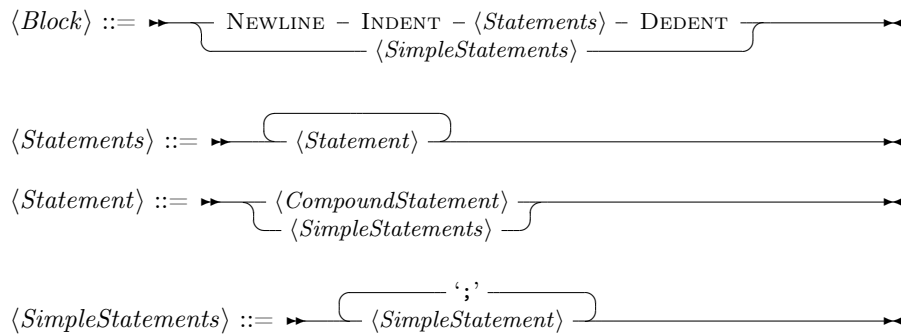
In the following sections we describe two major and one minor forms of syntax to indicate the start and end of a block.

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.



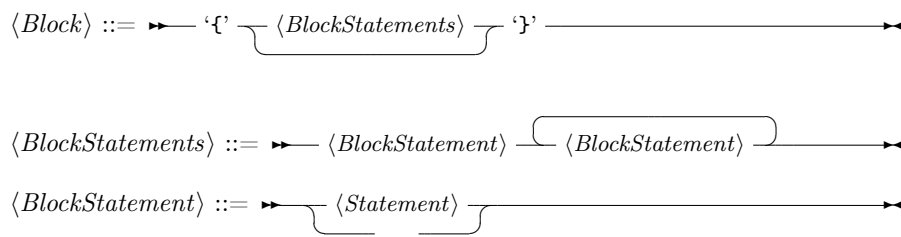
Listing 12: Block definitions for Python 3 grammars. INDENT and DEDENT are tokens generated by the lexer based on the indentation level of the source document [91].

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.



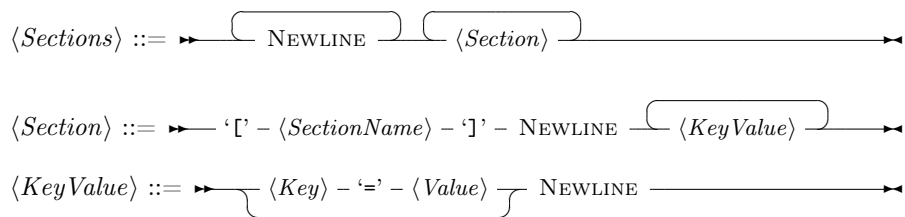
Listing 13: Block statement grammar specification for Java 16 [41] (chapter 14).

3.3.3 Section based

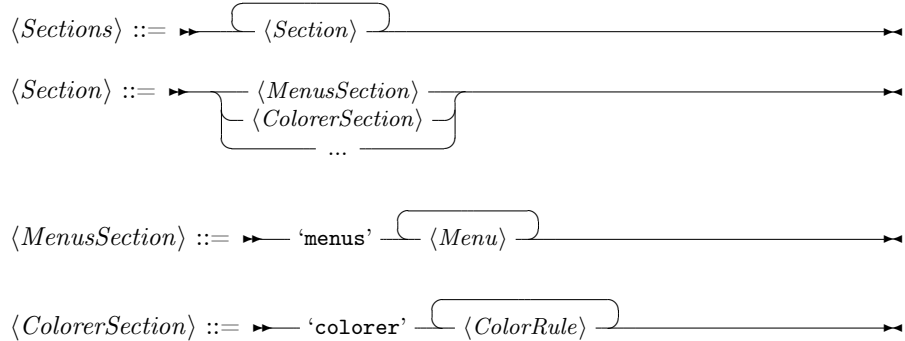
We view section based block structures as a variation to free-form languages. They have a single or series of symbols which indicate the start of a block, but have no specific symbol(s) to end a block, as they instead end implicitly.

Examples of these sorts of languages are INI (see Listing 14) configuration files, and the meta-languages used by Spoofox 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.

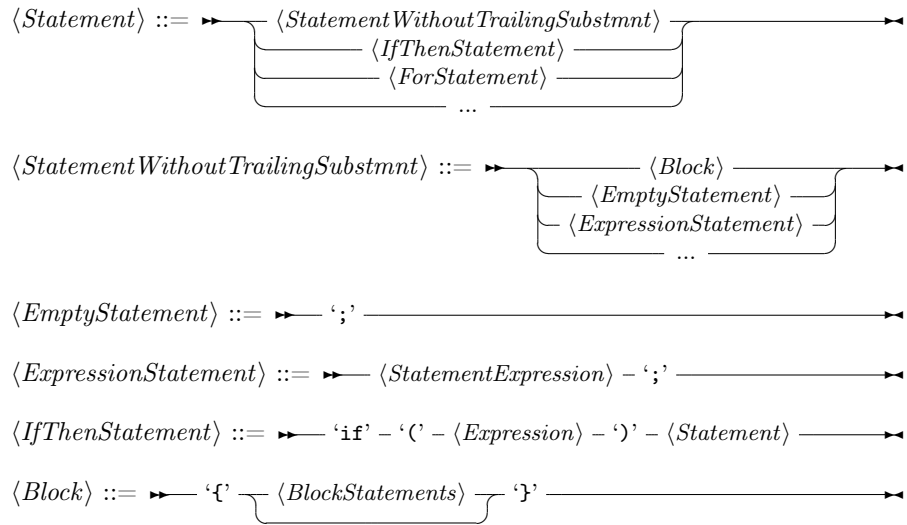


Listing 15: Part of the section based language ESV from the Spoofox Language Workbench [66].

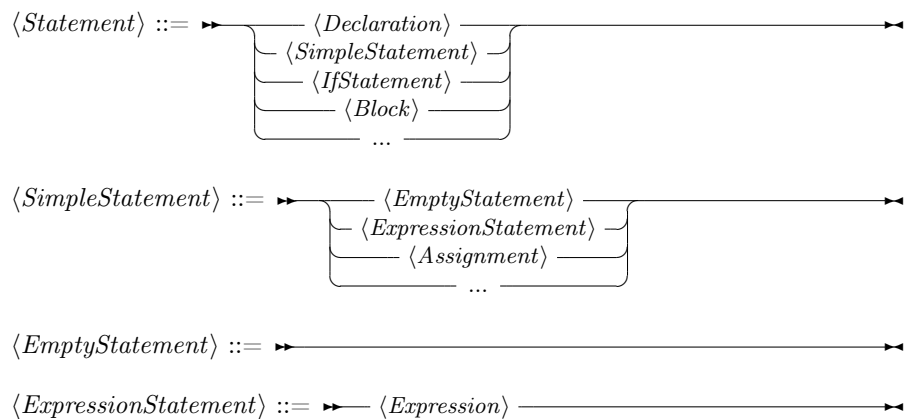
3.4 Statement Endings

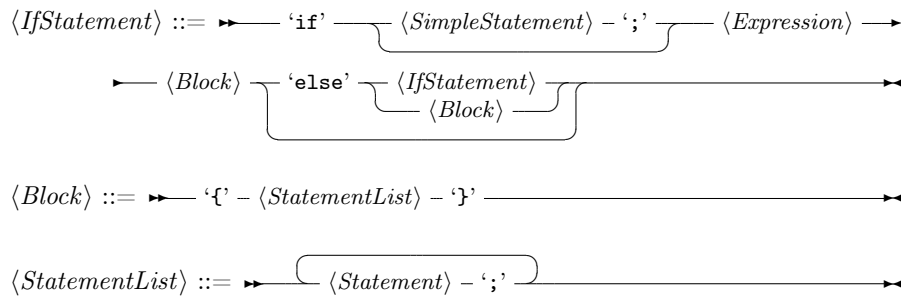
Most programming languages have a concept of statements, which denote a sequence of operations to be performed on the current program state. In a lot of those languages such as C++ [42], Java [41] (see Listing 16), Go [90] (see

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 Spoofox

Spoofox 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].

Spoofox 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 **Extended Backus-Naur Form** 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/jsclr/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 **L**eft-to-**r**ight **L**eftmost 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.vscod-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 and 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).

Spooifax

Spooifax is split into multiple git repositories: the main runtime repository ('spooifax'), IDE specific repositories ('spooifax-eclipse' and 'spooifax-intellij'), and core language development language repositories.

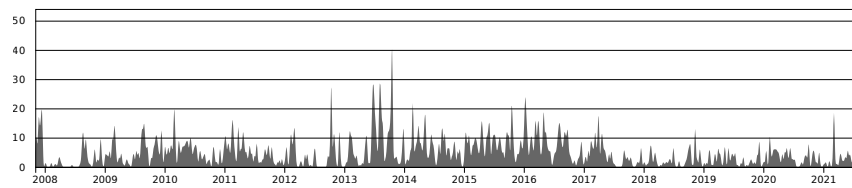


Figure 7: Contributions graph for the `metaborg/spooifax` repository on GitHub. See <https://github.com/metaborg/spooifax>.

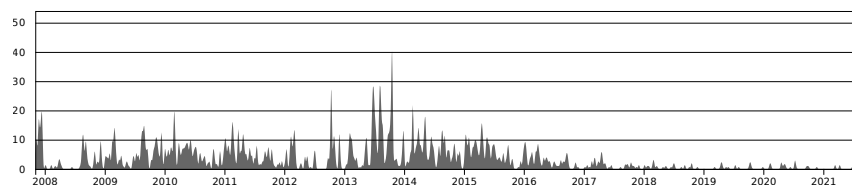


Figure 8: Contributions graph for the `metaborg/spooifax-eclipse` repository on GitHub. See <https://github.com/metaborg/spooifax-eclipse>.

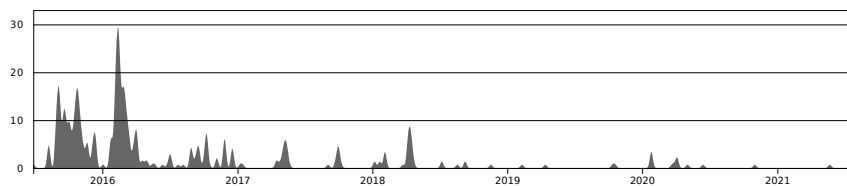


Figure 9: Contributions graph for the `metaborg/spoofax-intellij` repository on GitHub. See <https://github.com/metaborg/spoofax-intellij>.

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 Spoofox, 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 Spoofox 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 Spoofox is being actively developed by its developers.

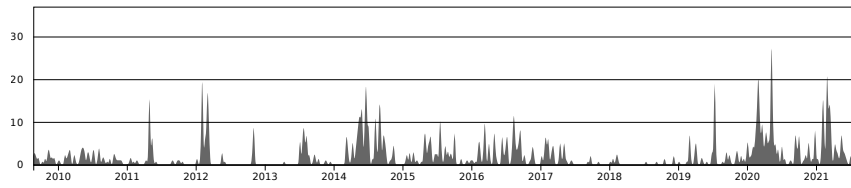


Figure 10: Contributions graph for the metaborg/spoofax-sdf repository on GitHub. See <https://github.com/metaborg/spoofax-sdf>.

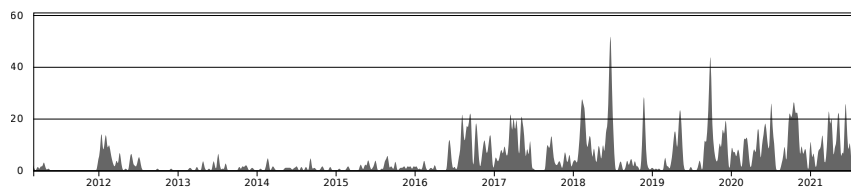


Figure 11: Contributions graph for the metaborg/spoofax-nabl repository on GitHub. See <https://github.com/metaborg/spoofax-nabl>.

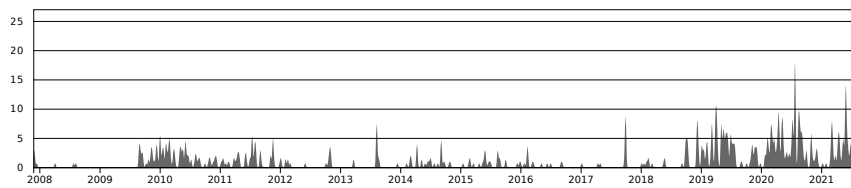


Figure 12: Contributions graph for the metaborg/spoofax-stratego repository on GitHub. See <https://github.com/metaborg/spoofax-stratego>.

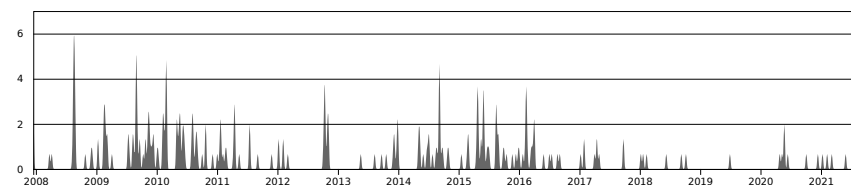


Figure 13: Contributions graph for the metaborg/spoofax-esv repository on GitHub. See <https://github.com/metaborg/spoofax-esv>.

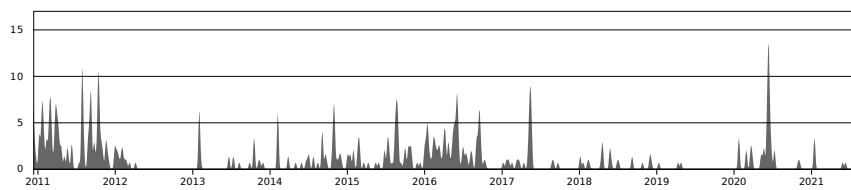


Figure 14: Contributions graph for the metaborg/spoofax-spt repository on GitHub. See <https://github.com/metaborg/spoofax-spt>.

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.

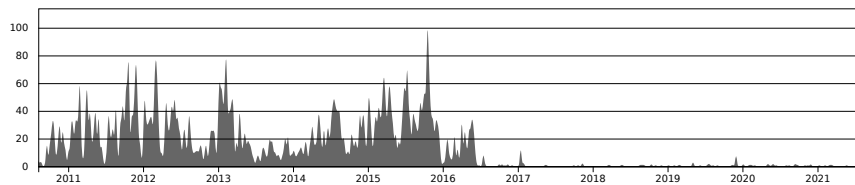


Figure 15: Contributions graph for the `eclipse/xtext` repository on GitHub. See <https://github.com/eclipse/xtext>.

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.

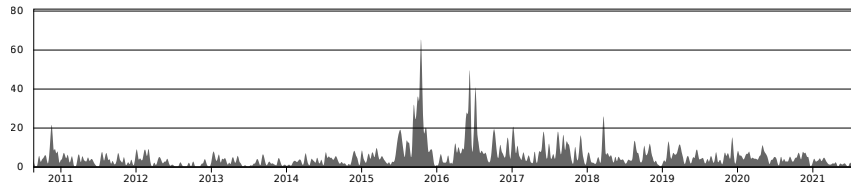


Figure 16: Contributions graph for the `eclipse/xtext-core` repository on GitHub. See <https://github.com/eclipse/xtext-core>.

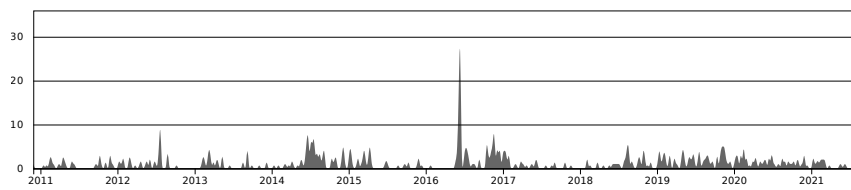


Figure 17: Contributions graph for the `eclipse/xtext-lib` repository on GitHub. See <https://github.com/eclipse/xtext-lib>.

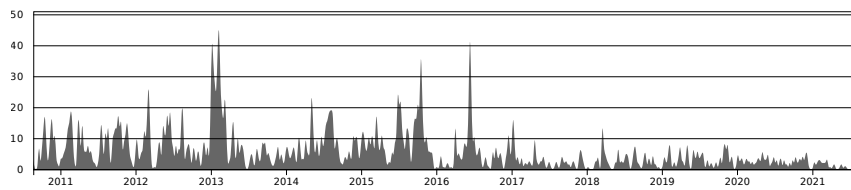


Figure 18: Contributions graph for the `eclipse/xtext-extras` repository on GitHub. See <https://github.com/eclipse/xtext-extras>.

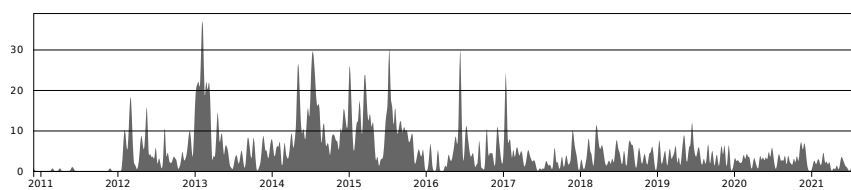


Figure 19: Contributions graph for the `eclipse/xtext-xtend` repository on GitHub. See <https://github.com/eclipse/xtext-xtend>.

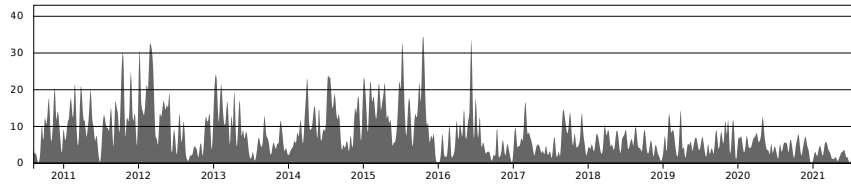


Figure 20: Contributions graph for the `eclipse/xtext-eclipse` repository on GitHub. See <https://github.com/eclipse/xtext-eclipse>.

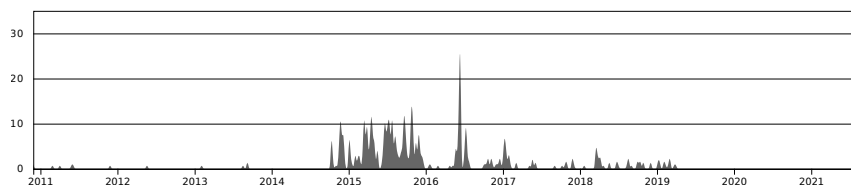


Figure 21: Contributions graph for the `eclipse/xtext-idea` repository on GitHub. See <https://github.com/eclipse/xtext-idea>.

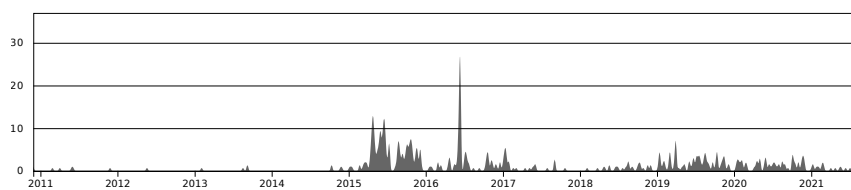


Figure 22: Contributions graph for the `eclipse/xtext-web` repository on GitHub. See <https://github.com/eclipse/xtext-web>.

JetBrains MPS

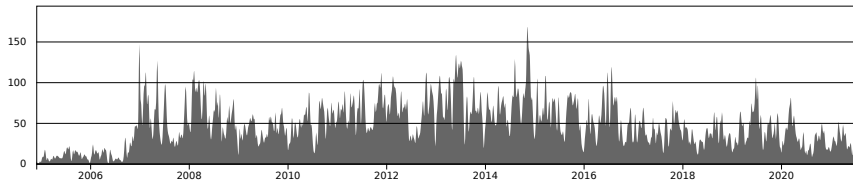


Figure 23: Contributions graph for the `JetBrains/MPS` repository on GitHub. See <https://github.com/JetBrains/MPS>.

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

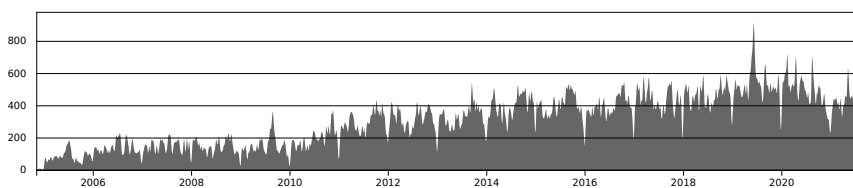


Figure 24: Contributions graph for the `JetBrains/intellij-community` repository on GitHub. See <https://github.com/JetBrains/intellij-community>.

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.

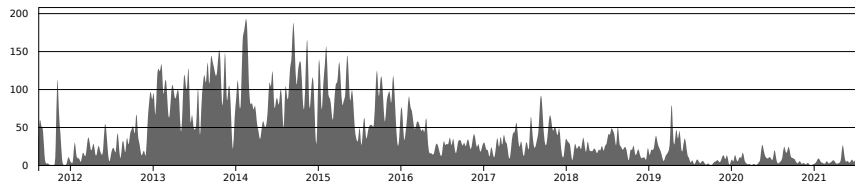


Figure 25: Contributions graph for the `atom/atom` repository on GitHub. See <https://github.com/atom/atom>.

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

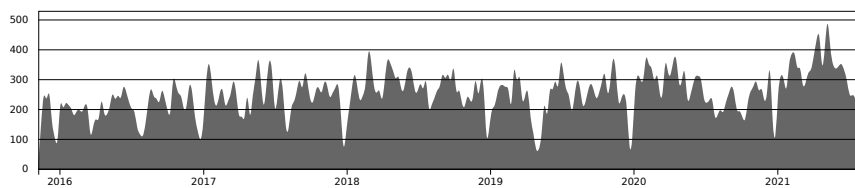


Figure 26: Contributions graph for the `microsoft/vscode` repository on GitHub. See <https://github.com/microsoft/vscode>.

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

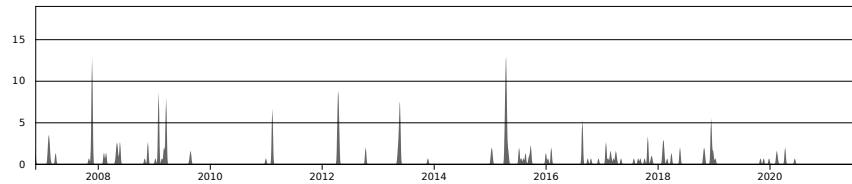


Figure 27: Contributions graph for the `dabeaz/ply` repository on GitHub. See <https://github.com/dabeaz/ply>.

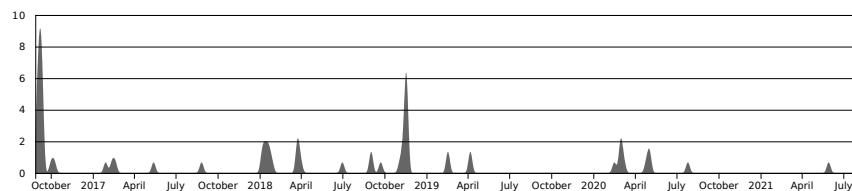


Figure 28: Contributions graph for the `dabeaz/sly` repository on GitHub. See <https://github.com/dabeaz/sly>.

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.

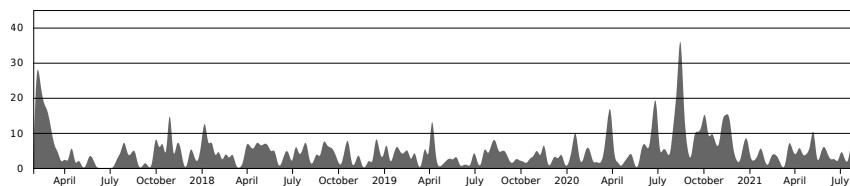


Figure 29: Contributions graph for the `lark-parser/lark` repository on GitHub. See <https://github.com/lark-parser/lark>.

6.4 Documentation

Spoofox

Extensive documentation for Spoofox 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

Spoofox

Based on the official Spoofox documentation [66] the Spoofox 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. Bänder, “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/xttext-idea: xttext-idea,” [Online]. Available: <https://github.com/eclipse/xttext-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. Hedlin, 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. "Spooifax on Yellowgrass.org," [Online]. Available: <https://yellowgrass.org/project/Spooifax> (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.