PARSING in a BROAD SENSE

Vadim Zaytsev, Universiteit van Amsterdam

Anya Helene Bagge, Universitetet i Bergen
Compilers
Principles, Techniques, and Tools

Компиляторы
Принципы, технологии, инструменты
3.1 Two classes of parsing methods

A parsing method constructs the syntax tree for a given sequence of tokens. Constructing the syntax tree means that a tree of nodes must be created and that these nodes must be labeled with grammar symbols, in such a way that:

- leaf nodes are labeled with terminals and inner nodes are labeled with non-terminals;
- the top node is labeled with the start symbol of the grammar;
- the children of an inner node labeled $N$ correspond to the members of an alternative of $N$, in the same order as they occur in that alternative;
- the terminals labeling the leaf nodes correspond to the sequence of tokens, in the same order as they occur in the input.

Left-to-right parsing starts with the first few tokens of the input and a syntax tree, which initially consists of the top node only. The top node is labeled with the start symbol.

The parsing methods can be distinguished by the order in which they construct the nodes in the syntax tree: the top-down method constructs them in pre-order, the bottom-up methods in post-order. A short introduction to the terms “pre-order” and “post-order” can be found below. The top-down method starts at the top and constructs the tree downwards to match the tokens in the input; the bottom-up methods combine the tokens in the input into parts of the tree to finally construct the top node. The two methods do quite different things when they construct a node. We will first explain both methods in outline to show the similarities and then in enough detail to design a parser generator.

Note that there are three different notions involved here:

- visiting a node, which means doing something with the node that is significant to the algorithm in whose service the traversal is performed;
- traversing a node, which means visiting that node and traversing its subtrees in some order; and
- traversing a tree, which means traversing its top node, which will then recursively traverse the entire tree. “Visiting” belongs to the algorithm; “traversing” in both meanings belongs to the control mechanism. This separates two concerns and is the source of the usefulness of the tree traversal concept. In everyday speech these terms are often confused, though.

3.1.1 Principles of top-down parsing

A top-down parser begins by constructing the top node of the tree, which it knows to be labeled with the start symbol. It now constructs the nodes in the syntax tree in pre-order, which means that the top of a subtree is constructed before any of its lower nodes are.

When the top-down parser constructs a node, the label of the node itself is already known, say $N$; this is true for the top node and we will see that it is true for all other nodes as well. Using information from the input, the parser then determines the...
A parsing method constructs the syntax tree for a given sequence of tokens. Constructing the syntax tree means that a tree of nodes must be created and that these nodes must be labeled with grammar symbols, in such a way that:

- leaf nodes are labeled with terminals and inner nodes are labeled with non-terminals;
- the top node is labeled with the start symbol of the grammar;
- the children of an inner node labeled \( N \) correspond to the members of an alternative of \( N \), in the same order as they occur in that alternative;
- the terminals labeling the leaf nodes correspond to the sequence of tokens, in the same order as they occur in the input.

Left-to-right parsing starts with the first few tokens of the input and a syntax tree, which initially consists of the top node only. The top node is labeled with the start symbol.

The parsing methods can be distinguished by the order in which they construct the nodes in the syntax tree: the top-down method constructs them in pre-order, the bottom-up methods in post-order. A short introduction to the terms “pre-order” and “post-order” can be found below. The top-down method starts at the top and con-
to DSMs, where relatively frequent changes in the problem domain as well as in the implementation target domain (e.g., due to external technical or strategic decisions) must be reflected in the respective languages. This is to maintain the high coupling between domain and language. The first problem is the need for rapid development techniques for DSMs, as they are created and modified frequently during the life cycle of the system they are used for. The second, and far greater problem is that possibly large numbers of modelling artefacts, such as instance models or transformation models developed become invalid and unusable when a related DSM is modified/evolved. Early adopters of MDE and DSM dealt with language evolution issues manually [41]. However, this approach, as well as an ad hoc approach to any language change, is tedious and error-prone [49]. The reason for this is that syntax of languages such as UML [27] and BPMN [54], which have evolved considerably over the last few years, easily comprise several hundreds of elements. Also, the semantic differences resulting from this evolution, either intended or intentional, can be subtle. Hence, dealing with evolution requires in-depth knowledge of the language as a whole. Without a proper scientific foundation, as well as methods, techniques and tools to support evolution, MDE in general and DSM in particular, cannot live up to its promise of ten-fold productivity increase [19]. This becomes apparent when projects span longer periods of time [44]. Since the problem of modelling language evolution was first identified by Sprinkle and Karsai [47], the general problem has only grown in importance, yet still remains largely unsolved. The importance of modelling language evolution is further evidenced by the attention it receives in the research community. The evolution of modelling languages is one of the 11 topics for paper submission at MODELS 2010 (ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems), and workshops such as ME 2010 (International Workshop on Models and Evolution) are devoted largely to the topic. Current state-of-the-art contributions in this field are focused on (semi-)automatic model differencing [6] and on the co-evolution of instance models [16].

The remainder of the paper is organised as follows: Section 2 is a short introduction to modelling languages. Section 3 discusses related work. Section 4 introduces an example that will be used to illustrate our approach throughout the paper. Section 5 presents the possible kinds of evolution. Section 6 introduces a way to tackle evolution of modelling languages by deconstructing the problem into primitives. Section 7 presents a framework and algorithm for the evolution of modelling artefacts when languages evolve. Section 8 concludes the paper and describes future work.

2. Modelling languages

To allow for a precise discussion of language evolution, we briefly introduce fundamental modelling language concepts. This introduction which we elaborate in [16] is based on foundations laid by Harel and Rümper [15] and Kühtre [21]. The two main aspects of a model are its syntax (how it is represented) and its semantics (what it means).

Firstly, the syntax comprises concrete syntax and abstract syntax. The concrete syntax describes how the model is represented (e.g., in 2D vector graphics or in textual form), which can be used for model input as well as for model visualisation. The abstract syntax contains the “essence” of the model (e.g., as a typed Abstract Syntax Graph (ASG)—when models are represented as graphs).

A single abstract syntax may be represented by multiple concrete syntaxes. There exists a mapping between a concrete syntax and its abstract syntax, called the parsing function. There is also a mapping in the opposite direction, called the rendering function. These are the concrete mapping functions. Mappings are usually implemented, or can at least be represented, as model transformations. The abstract syntax and concrete syntax of a model are related by a surjective homomorphic function that translates a concrete syntax graph into an abstract syntax graph.

Secondly, the semantics of a model are defined by a complete, total and unique semantic mapping function which maps every abstract syntax model into a single element in a semantic domain, such as Ordinary Differential Equations, Petri nets [48], or a set of behaviour traces. These are domains with well-known and precise semantics. For convenience, semantic mapping is usually performed on abstract syntax, rather than on concrete syntax directly. More explicitly, the abstract syntax can be used as a basis for semantic anchoring [4].

A meta-model is a finite model that explicitly describes the abstract syntax and static semantics, which are statically checkable, of a language. Dynamic semantics are not covered by the meta-model. The abstract syntax of a model can be represented as a graph, where the nodes are elements of the language and the edges are relations between these elements, and also elements of the language. Instance models of the language are said to conform to the meta-model of the language. In [21], Kühtre refers to this relation as linguistic instance of. The description of the abstract syntax is typically specified in a modelling language such as EKU Class Diagrams [34]. Static semantics can be described in a constraint language such as the Object Constraint Language (OCL) [36]. Often, but not necessarily, the concrete syntax mapping is directly attached to a meta-model, where every element of the concrete syntax can be explicitly traced back to its corresponding element of the abstract syntax.

When I choose the different kinds of relations involving a model, relations are visualised by arrows, “conforms to” relationships are dotted arrows. The abstract syntax model mm conforms to a meta-model $\text{MM}_{\text{mm}}$, the explicit model of the language $\text{lang}$. There is a rendering function $\mathfrak{r}$, between mm and a concrete syntax $\text{sc}(\text{mm})$ model. The inverse of $\mathfrak{r}$ is a parsing function $\mathfrak{p}$, so that $\mathfrak{r}(\text{mm}) = \text{sc}(\text{mm})$. The index highlights the fact that multiple concrete representations may be used, $\text{sc}(\text{mm})$ conforms to a meta-model $\text{MM}_{\text{sc}}$, the explicit model of the concrete syntax language (such as the set of all 2D vector graphics drawings). Semantics are described by the semantic mapping function $\mathfrak{m}$, and maps in $\text{mm}$ to a model $\text{mm}$ in the semantic domain. This semantic domain is a different modelling language with its own syntax on semantics. Similar to $\mathfrak{m}$ conforming to $\text{MM}_{\text{mm}}$, $\mathfrak{m}$ conforms to $\text{MM}_{\text{sem}}$. Additionally, transformations $\mathfrak{T}$, may be defined for in.
deconstructing the problem into primitives. Section 7 presents a framework and algorithm for the evolution of modelling artefacts when languages evolve. Section 8 concludes the paper and describes future work.

2. Modelling languages

To allow for a precise discussion of language evolution, we briefly introduce fundamental modelling language concepts. This introduction which we elaborated in [10] is based on foundations laid by Harel and Rumpe [13] and Kühne [21]. The two main aspects of a model are its syntax (how it is represented) and its semantics (what it means).

Firstly, the syntax comprises concrete syntax and abstract syntax. The concrete syntax describes how the model is represented (e.g., in 2D vector graphics or in textual form), which can be used for model input as well as for model visualisation. The abstract syntax contains the “essence” of the model (e.g., as a typed Abstract Syntax Graph (ASG)—when models are represented as graphs).

A single abstract syntax may be represented by multiple concrete syntaxes. There exists a mapping between a concrete syntax and its abstract syntax, called the parsing function. There is also a mapping in the opposite direction, called the rendering function. These are the concrete mapping functions. Mappings are usually implemented, or can at least be represented, as model transformations. The abstract syntax and concrete syntax of a model are related by a surjective homomorphic function that translates a concrete syntax graph into an abstract syntax graph.

Secondly, the semantics of a model are defined by a complete, total and unique semantic mapping function which maps every abstract syntax model onto a single element in a semantic domain, such as Ordinary Differential Equations, Petri nets [39], or a set of behaviour traces. These are domains with well-known and precise semantics. For convenience, semantic mapping is usually performed on abstract syntax, rather than on concrete syntax directly. More explicitly, the abstract syntax can be used as a basis for semantic anchoring [4].

A meta-model is a finite model that explicitly describes the abstract syntax and static semantics, which are statically checkable, of a language. Dynamic semantics are not covered by the meta-model. The abstract syntax of a model can be represented as a graph, where the nodes are elements of the language and the edges are relations between these elements, and also elements of the language. Instance models of the language are said to conform to the meta-model of the language. In [21], Kühne refers to this relation as linguistic instance of. The description of the abstract syntax is typically specified in a modelling language such as UML Class Diagrams [34]. Static semantics can be described in a constraint language such as the Object Constraint Language (OCL) [36]). Often, but not necessarily, the concrete syntax mapping is directly attached to a meta-model, where every element of the concrete syntax can be explicitly traced back to its corresponding element of the abstract syntax.

Fig. 1 shows the different kinds of relations involving a model m. Relations are visualised by arrows, “conforms to”-
Technological Space Megamodel
Bidirectional Transformations...
in the paper...
in the paper...
Parsening

- recognising structure
  - text → tree
  - parse tree → AST
  - forest disambiguation
  - tokens → graph

Unparsing

- representing structure
  - model → picture
  - ASG → text
  - (re)formatting
  - serialisation
Separate tokens
Separate tokens

Detailed parse tree
Separate tokens

Detailed parse tree
Parse tree without layout
Parse tree without layout
Abstract syntax tree
Abstract syntax tree
Abstract syntax graph
typed tokens
lexical model
Examples
Lifting & Lowering

cf. Zaytsev/Bagge, Modelling Parsing and Unparsing, Parsing@SLE, 2014.
Ast depicted on kinds of artefacts, which can be considered to fit into one of twelve categories, Di

3.1 Problem description

3 (Un)parsing case study

defined mappings \[ ... \] line definitions; solid lines denote universally

Fig. 1. cf. Zaytsev, Case Studies in Bidirectionalisation

Abstract

erent approaches and phases of software language processing feature di

It lacks certain details specific to

is the most suitable for automated program analysis and assigning semantics.

into a di

but it is not apparent until the language instance is processed and turned

used to for the last six decades. There is some structure in such a program,

transfer and maintain and familiar to what mainstream programmers are

— an abstract syntax tree, a conceptual representation of a program which

Textual

TTk

Lex

Ast

Di

Cst

Dia

For

Ptr'

Ptr'

Gra

Dra

Pic

DFP, 2014.

References

is required for the general case of model synchronisation \[ ... \] of the update strategy in the case of more than two instances (like in

error-correcting strategies. However, it is yet unclear how to prove termination

instead of maintainers:

two repositories referenced above. It mostly consists of documented Prolog \[ ... \] bidirectionalisation in the future.

Two practical cases of bidirectionalisation have been described in this abstract.

4 Conclusion

http://github.com/grammarware/bx-parsing

and is publicly available from a dedicated repository:

of it would feature iterative parsing, for which an old instance of

need to be stored as well.

of it would feature iterative parsing, for which an old instance of

Annotations for the fragments that are new. A truly e

spond to any fragments of the updated code, and recalculates all inferred

reasons that we consider in this paper.

loops are examples of transformations we consider in this paper.

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row

Column

Row
Renarration

To 

Ptr

For

Cst

TTk

Lex

Ast

Dra

Dia

Pic

Gra

Dra

Dia

Ast

Lex

parser

programming

(Java,C++,…)

tokenizer

lexer

Str

For

Ptr

Conclusion

• Contributions:
  • BX framework classification/extension
  • Megamodel for parsing (in a broad sense)
    • Explain yourself
    • Understand others
    • Find new ground

• More:
  • Usage: TFP’14, Parsing@SLE’14, GEMOC’14, EduSymp’14,…
  • Reach us at: @grammarware, @anyahelene, #models14,…

• Questions?

• Photos are self-made; from PEXELS (CC0); by Eelco Visser (Jean Bézivin).