MARKUS VOELTER, VOELTER@ACM.ORG

DSL ENGINEERING

1

Concrete and Abstract Syntax

TODO: Show stuff in MPS with tables and graphics

The *concrete syntax* (short: CS) of a language is what the user interacts with to create programs. It may be textual, graphical, tabular or any combination thereof. In this book we focus mostly on textual concrete syntaxes for reasons described in . We refer to other forms where appropriate.

TODO: reference to the respective place in the introduction

The *abstract syntax* (short: AS) of a language is a data structure that holds the information represented by the concrete syntax, but without any of the syntactic details. Keywords and symbols, layout (e.g., whitespace), and comments are typically not included. Note that syntactic information which doesn't end up in the AS is often preserved in some "hidden" form so the CS can be reconstructed from the combination of the AS and this hidden information — this bidirectionality simplifies creating IDE features quite a bit.

In most cases, the abstract syntax is a tree data structure. Instances that represent actual programs (i.e. sentences in the language) are hence often called an abstract syntax tree or AST. Some formalisms also support cross-references across the tree, in which case the data structure becomes a graph (with a primary containment hierarchy). It is still usually called an AST.

While the CS is the "UI" of the language to the user, the AS acts as the API to access programs by processing tools: it is used by developers of validators, transformations and code generators. The concrete syntax is not relevant in these cases.

To illustrate the dichotomy between concrete and abstract syntax, consider the following example program:

```
var x: int;
calc y: int = 1 + 2 * sqrt(x)
```

This program has a hierarchical structure: definitions of x and y at the top, inside y there's a nested expression. This structure is reflected in the corresponding abstract syntax. A possible AST is illustrated in

Fig. 1.1. The boxes represent program elements, and the names in the boxes represent language concepts that make up the abstract syntax.

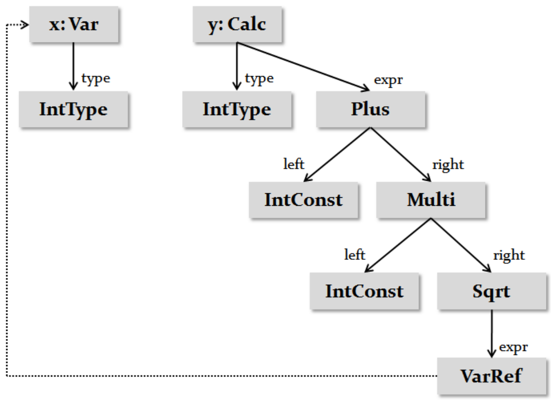


Figure 1.1: Abstract syntax tree for the above program. Boxes represent instances of language concepts, solid lines represent containment, dotted lines represent cross-references

There are two ways of defining the relationship between the CS and the AS.

AS first An existing AS is annotated with the concrete syntax. Mapping rules determine how the concrete syntax maps to existing abstract syntax structures.

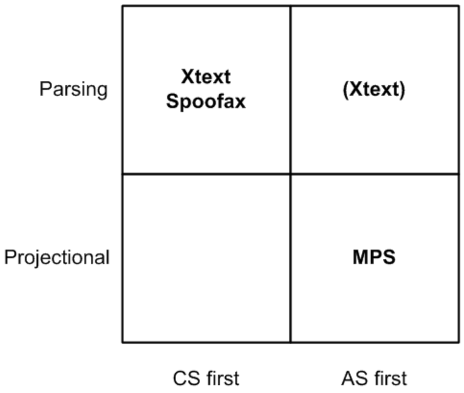
CS first From a concrete syntax definition, an abstract syntax is derived, either automatically and/or using hints in the concrete syntax specification.

Once the language is defined, there are again two ways how the abstract syntax and the concrete syntax can relate as the language is used to create programs:

Parsing In parser-based systems, the abstract syntax tree is derived from the concrete syntax of a program; a parser instantiates and populates the AS, based on the information in the program text. In this case, the (formal) definition of the CS is usually called a *grammar*.

Projection In projectional systems, the abstract syntax tree is built directly by editor actions, and the concrete syntax is rendered from the AST via projection rules.

Fig. 1 shows the typical combinations of these two dimensions. In practice, parser-based systems typically derive the AS from the CS - i.e., CS first. In projectional systems, the CS is usually annotated onto the AS data structures - i.e., AS first. In the next sections we look more closely at parsing and projection.

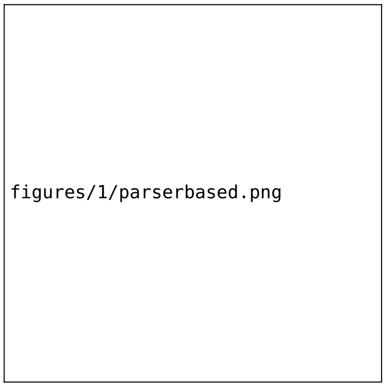


	CS first	AS first
Parsing	**Xtext** **Spoofax**	**(Xtext)**
Projectional		**MPS**

1.1 *Fundamentals of Free Text Editing and Parsing*

Most general-purpose programming environments rely on free text editing, where programmers edit programs at the character level to form (key)words and phrases.

A *parser* is used to check the program text (concrete syntax) for syntactic correctness, and create the AST by populating the AS from information extracted from the textual source. Most modern IDEs perform this task in real-time as the user edits the program. The AST is always kept in sync with the program text. Many IDE features — such as content assist, validation, navigation, refactoring support, etc. — are based on the real-time AST.



`figures/1/parserbased.png`

Figure 1.2: In parser-based systems, the user only interacts with the concrete syntax, and the AST is constructed from the information in the text.

A key characteristic of the free text editing approach is its strong separation between the concrete syntax (i.e., text) and the abstract syntax. The concrete syntax is the principal representation, used for both editing and persistence. The abstract syntax is used under the hood by the implementation of the DSL, e.g. for providing an outline view, validation, and for transformations and code generation.

Many different approaches to parser implementation exist. Each may restrict the syntactic freedom of a DSL, or constrain the way in which a particular syntax must be specified. It is important to be aware of this as not all languages can be comfortably, or even at all implemented by every parser implementation approach. You may have heard terms like context free, ambiguity, look-ahead, LL, (LA)LR, PEG. These all pertain to a certain class of parser implementation approaches. We'll provide more details on the various grammar and parser classes further on in this section.

1.1.1 *Parser Generation Technology*

In traditional compilers, parsers are often written by hand as a big program that reads a stream of characters and uses recursion to create a tree structure. Manually writing a parser in this way requires significant expertise in parsing and a large development effort. For standardized programming languages that don't change very often, and that have a large user community, this approach makes sense. It can lead to very fast parsers that also exhibit good error recovery (the ability to continue parsing after a syntax error has been found).

In contrast, language workbenches, and increasingly also traditional compilers, *generate* a parser from a grammar, a DSL for formally defining textual concrete syntax. These generated parsers may not provide the same performance or error recovery as a hand-tailored parser constructed by an expert, but they provide bounded performance guarantees that make them (usually) more than fast enough for modern machines. However, the most important argument to use parser generation is that the effort required is **much** lower than writing a custom parser from scratch. The grammar definition is also much more readable, maintainable and adaptable than the actual parser implementation (either custom-written or generated).

MV: Do we want to say something regarding our dislike of actions (procedural code) embedded in grammars? We'd like to separate these concerns. . .

EV: note that I changed lexer to scanner: lexer is more commonly used nowadays but since it's called 'scannerless parsing' it probably makes more sense to use 'scanner'

Parsing versus scanning Because of the complexity inherent in parsing, parser implementations tend to split the parsing process into a number of phases. In the majority of cases, the text input is first separated into a sequence of *tokens* (i.e., keywords, identifiers, literals, comments, whitespace, etc.) by a *scanner* (sometimes also called lexer). The *parser* then constructs the actual AST from the token sequence. This simplifies the implementation compared to directly parsing at the character level. A scanner is usually implemented using direct recognition of keywords and a set of regular expressions to recognize all other valid input as tokens.

Both the scanner and parser can be generated from grammars (see below). A well-known example of a scanner (lexer) generation tool is `lex`. Modern parsing frameworks, such as ANTLR, do their own scanner generation.

Note that the word "parser" now has more than one meaning: it can either refer to the combination of the scanner and the parser or to the post-scanner parser. Usually the former meaning is intended (both in this book as well as outside) unless scanning and parsing are discussed specifically.

A separate scanning phase has direct consequences for the overall parser implementation, because the scanner typically isn't aware of the context of any part of the input — only the parser has this awareness. An example of a typical problem that arises from this is that keywords can't be used as identifiers even though often the use of a keyword wouldn't cause ambiguity in the actual parsing. The Java language is an example of this: it uses a fixed set of keywords, such as *class* and *public*, that cannot be used as identifiers.

A context-unaware scanner can also introduce problems when languages are extended or composed. In the case of Java, this was seen with the *assert* and *enum* keywords that were introduced in Java 1.4 and Java 5. Any programs that used identifiers with those names (such as unit testing APIs) were no longer valid. For composed languages, similar problems arise as constituent languages have different sets of keywords and can define incompatible regular expressions for lexicals such as identifiers and numbers.

The term "scannerless parsing" refers to the absence of a separate scanner and in this case we don't have the problems of context-unaware scanning illustrated above, because the parser operates at a character level and statefully processes lexicals and keywords. Spoofax (or rather: the underlying parser technology) uses scannerless parsing.

MV: [symbol tables] should be discussed in the scoping and linking section?

Grammars Grammars are the formal definition for concrete textual syntax. They consist of so-called production rules which define how valid input ("sentences") looks like. They can also be used to "produce" valid input, hence their name. Grammars form the basis for syntax definitions in text-based workbenches such as Spoofax and Xtext. In these systems, the production rules are enriched with information beyond the pure grammatical structure of the language, such as the semantical relation between references and declarations.

Fundamentally, grammar production rules can be expressed in Backus-Naur Form (BNF)[1], written as

S ::= P_1 ... P_n

This grammar defines a symbol S by a series of pattern expressions P_1 ... P_n. Each pattern expression can refer to another symbol or can be a literal such as a keyword or a punctuation symbol. If there are multiple possible patterns for a symbol, these can be written as separate productions, or the patterns can be separated by the '|' operator to indicate a choice. An extension of BNF, called Extended BNF (EBNF), adds a number of convenience operators such as '?' for an optional pattern, '*' to indicate zero or more occurrences, and '+' to indicate one or more occurrences.

As an example, Fig. 1.3 shows an example of a grammar for a simple arithmetic expression language using BNF notation. Basic expressions are built up of `NUM` number literals and the + and ∗ operators.

Note how expression nesting is described using recursion in this grammar: the `Exp` rule calls itself, so sentences like `2 + 3 * 4` are allowed. This poses two practical challenges for parser generation systems: first, the precedence and associativity of the operators is not described (explicitly) by this grammar. Second, not all parser generators provide full support for recursion. We elaborate on these issues in

[1]

```
Exp ::= NUM
      | Exp "+" Exp
      | Exp "*" Exp
```

Figure 1.3: A grammar for a simple expression language. The `NUM` symbol refers to number literals.

Figure 1.4: Classes of grammars.

the remainder of the section and in the Spoofax and Xtext examples.

Grammar classes BNF can describe any grammar that maps textual sentences to trees based only on the input symbols. These are called *context-free grammars* and can be used to parse the majority of modern programming languages. In contrast, *context-sensitive grammars* are those that also depend on the context in which a partial sentence occurs, making them suitable for natural language processing but at the same time, making parsing itself a lot harder since the parser has to be aware of a lot more than just the syntax.

TODO: as opposed to what?

Parser generation was first applied in command-line tools such as `yacc` in the early seventies. As a consequence of relatively slow computers, much attention was paid to the efficiency of the generated parsers. Various algorithms were designed that could parse text in a bounded amount of time and memory. However, these time and space guarantees could only be provided for certain subclasses of the context-free grammars, described by acronyms such as LL(1), LL(*k*), LR(1), and so on, as illustrated in Fig. 1.4. A particular parser tool supports a specific class of grammars — e.g., ANTLR supports LL(*k*) and LL(*).

In the classification of Fig. 1.4, the first L stands for left-to-right scanning, and the second L in LL and the R in LR stand for leftmost and rightmost derivation. The constant *k* in LL(*k*) and LR(*k*) indicates the maximum number (of tokens or characters) the parser will look ahead to decide which production rule it can recognize. Typically, grammars for "real" DSLs tend to need only finite look-ahead and many parser tools effectively compute the optimal value for *k* automatically. A special case is LL(*), where *k* is unbounded and the parser can look ahead arbitrarily many tokens to make decisions.

TODO: Venn diagram of grammar classes: LL, LR, ...

Supporting only a subclass of all possible context-free grammars poses restrictions on the languages that are supported b a parser generator. For some languages, it is not possible to write a grammar in a certain subclass, making that particular language unparseable with a tool that only supports that particular class of grammars. For other languages, a natural context-free grammar exists, but it must be written in a different, often awkward way to conform to the subclass. This will be illustrated in the Xtext example, which uses ANTLR as the

```
Exp ::= ID
      | Exp "." ID
      | STRING
```

Figure 1.5: A grammar for property access expressions of the form `customer.name` or `"Tim".length`.

underlying LL(*k*) parser technology.

Parser generators can detect if a grammar conforms to a certain subclass, reporting conflicts that relate to the implementation of the algorithm: *shift/reduce* or *reduce/reduce* conflicts for LR parsers, and *first/first* or *first/follow* conflicts and direct or indirect *left recursion* for LL parsers. DSL developers can then attempt to manually refactor the grammar to address those errors.

As an example, consider a grammar for property or field access, expressions of the form `customer.name` or `"Tim".length`:[2]

This grammar uses left-recursion: the left-most symbol of one of the definitions of `Exp` is a call to `Exp`, i.e. it is recursive. Left-recursion is not supported by LL parsers such as ANTLR.

The grammar can be *left-factored* by changing it to a form where all left recursion is eliminated. The essence of left-factoring is that the grammar is rewritten in such a way that all production rules which are recursive consume at least one token or character before going into the recursion. Left-factoring introduces additional rules that act as intermediaries and often makes repetition explicit using the `+` and * operators. The example grammar uses recursion for repetition, which can be made explicit as follows:

```
Exp ::= ID
      | (Exp ".")+ ID
      | STRING
```

The resulting grammar is still left-recursive, but we can introduce an intermediate rule to eliminate the recursive call to `Exp`:

```
Exp ::= ID
      | (FieldPart ".")+ ID
      | STRING

FieldPart ::= ID
            | STRING
```

Unfortunately, this resulting grammar still has overlapping rules (a first/first conflict) as the `ID` symbol matches more than one rule. This conflict can be eliminated by removing the `Exp ::= ID` rule and making the `+` "one or more" repetition into a * "zero or more" repetition:

```
Exp       ::= (FieldPart ".")* ID
```

[2] Note that we use ID to indicate identifier patterns and STRING to indicate string literal patterns in these examples.

```
            | STRING

FieldPart ::= ID
            | STRING
```

This last grammar now describes the same language as the one from Fig. 1.5, but conforms to the LL(1) grammar class.

Unfortunately, it is not possible to factor all possible context-free grammars to one of the restricted classes. Valid, unambiguous grammars exist that cannot be factored to any of the restricted grammar classes. In practice, this means that some languages cannot be parsed with LL or LR parsers.

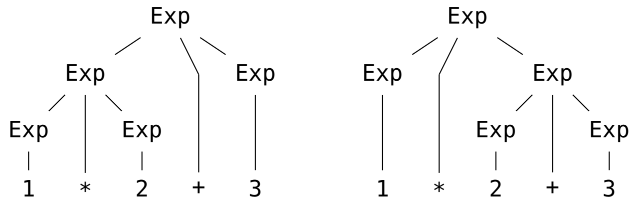
General parsers Research into parsing algorithms has produced parser generators specific to various grammar classes, but there has also been research in parsers for the full class of context-free grammars. Naive algorithms that support the full class of context-free grammars use backtracking, i.e. they allow themselves to try to match the input using another production rule even though an earlier-tried rule already seemed to match. This carries the risk of exponential execution times or non-termination and usually exhibits poor performance.

MV: I don't understand the previous sentence

There are also general parsing algorithms that can *efficiently* parse the full class. In particular, generalized LR (GLR) and Earley parsers can parse unambiguous grammars in linear time and gracefully cope with ambiguities with a cubic or $O(n^3)$ time in the worst case. Spoofax is an example of a language workbench that uses GLR parsing.

Ambiguity Grammars can be *ambiguous* meaning that at least one valid sentence in the language can be constructed in more than one (non-equivalent) way from the production rules, corresponding to multiple possible ASTs. This obviously is a problem for parser implementation as some decision has to be made on which AST is preferred.

For example, the expression language grammar given in Fig. 1.3 is ambiguous. For a string 1*2+3 there are two possible trees (corresponding to different operator precedences). The grammar does not describe which interpretation should be preferred.



Parser generators for restricted grammar classes and generalized parsers handle ambiguity differently. We discuss both approaches.

Ambiguity with grammar classes LL and LR parsers are deterministic parsers: they can only return one possible tree for a given input. This means they can't handle any grammar that has ambiguities, including the grammar in Fig. 1.3. Determining whether a grammar is ambiguous is a classical, undecidable problem. However, it is possible to detect violations of the LL or LR grammar class restrictions, in the form of conflicts. These conflicts do not always indicate ambiguities (as seen with the field access grammar of Fig. 1.5), but by resolving all conflicts (if possible) an unambiguous grammar can be formed.

Resolving grammar conflicts in the presence of associativity, precedence, and other risks of ambiguity requires carefully layering the grammar in such a way that it encodes the desired properties. To encode left-associativity and a lower priority for the addition operator we can rewrite the grammar as follows:

```
Expr ::= Expr "+" Mult
       | Mult
Mult ::= Mult "*" NUM
       | NUM
```

The resulting grammar is a valid LR grammar. Note how it puts the addition operator in the highest layer to give it the lowest priority, and how it uses left-recursion to encode left-associativity of the operators. The grammar can be left-factored to a corresponding LL grammar as follows. We will see more extensive examples of this approach in the section on Xtext ().

TODO:

```
Expr ::= Mult ("+" Mult)*
Mult ::= NUM ("*" NUM)*
```

Ambiguity with generalized parsers General parsers accept grammars regardless of recursion or ambiguity. That is, the grammar of Fig:exp-grammar-simple is readily accepted as a valid grammar. In case of an ambiguity, the generated parser simply returns all possible abstract syntax trees, e.g. a left-associative tree and a right-associative tree for the expression `1*2+3`. The different trees can be manually inspected to determine what ambiguities exist in the grammar, or the desired tree can be programmatically selected.

Language developers can use *disambiguation filters* to indicate which interpretation should be preferred. For example, left-associativity can be indicated on a per-production basis:

```
Exp ::= NUM
      | Exp "+" Exp {left}
      | Exp "*" Exp {left}
```

indicating that both operators are left-associative (using the `{left}` annotation as seen in Spoofax). Operator precedence can be indicated with relative priorities or with precedence annotations:

```
Exp ::= Exp "*" Exp {left}
>
Exp ::= Exp "+" Exp {left}
```

indicating that the multiplication operator binds stronger than the addition operator. This kind of declarative disambiguation is commonly found in GLR parsers, but typically not available in parsers that support only more limited grammar classes.

TODO:

Compositionality encoded precedence, associativity, etc. makes grammars resistent to change and composition

grammar classes and composition don't mix

can you compose the arithmetic expressions grammar with the field access grammar Fig:field-access-grammar?

and what about

```
for (Customer c : SELECT customer FROM accounts WHERE balance < 0) {
  ...
}
```

reserved keywords: for, SELECT, FROM

requires a context-sensitive scanner, or a parser that operates on characters instead of on tokens (a scannerless parser)

1.2 Fundamentals of Projectional Editing

In parser-based approaches, users use text editors to enter character sequences that represent programs. A parser then checks the program for syntactic correctness and constructs an abstract syntax tree from the character sequence. The AST contains all the semantic information expressed by the program.

In projectional editors, the process happens the other way round: as a user edits the program, the AST is modified directly. A projection engine then creates some representation of the AST with which the user interacts, and which reflects the changes. This approach is well-known from graphical editors in general, and the MVC pattern specifically. In editing a UML diagram, users don't draw pixels onto a canvas, and a "pixel parser" then creates the AST. Rather, the editor creates and instance of `uml.Class` as you drag a class from the palette to the canvas. A projection engine renders the diagram, in this case

figures/1/projectional.png

Figure 1.6: In projectional systems, the user sees the concrete syntax, but all editing gestures directly influence the AST. The AST is *not* extracted from the concrete syntax, which means the CS does not have to be parseable.

drawing a rectangle for the class. This approach can be generalized to work with any notation, including textual.

In projectional editors, every program element is stored as a node with a unique ID (UID) in the AST. References between program elements are based on actual pointers (references to UIDs). The AST is actually an ASG, an abstract syntax graph, from the start because cross-references are first-class rather than being resolved after parsing. The program is stored using a generic tree persistence mechanism, often XML.

The projectional approach can deal with arbitrary syntactic forms. Since no grammar is used, grammar classes are not relevant here. In principle, projectional editing is simpler than parsing, since there is no need to "extract" the program structure from a flat textual source. However, as we will see below, the challenge in projectional editing lies making the editing experience convenient.

1.3 Comparing Parsing and Projection

MV: The following paragraph needs to be ripped apart and the respective points need to be moved into the respective subsubsections

There's a lot to say in favor of free text editing. The linear, two-dimensional structure of text (especially when using mono-spaced fonts) is well-known and easily understood, while text editing operations are more-or-less universal. It's very convenient to have a CS that also serves as the persistence format: it makes the language essentially tool-independent and allows fairly easy integrations with other text-based tools like versioning systems, issue trackers, documentation systems, or email clients.

1.3.1 Editing Experience

In free text editing, any regular text editor will do. However, users expect powerful IDE that includes support for syntax coloring, code completion, go to definition, find references, error annotations, refactoring and the like. Xtext and Spoofax provide such IDE support. However, you can always go back to any text editor to edit the programs.

In projectional editing, this is different. since a normal text editor is obviously not sufficient as you would be editing some textual representation of the AST. A specialized editor has to be supplied. Like in free text editing, it has to provide the IDE support features mentioned above. MPS provides those. Howevre, there is another challenge: for textual-looking notations, it is important that the editor tries and makes the editing experience as text-like as possible, i.e. the keyboard actions we have gotten used to from free-text editing should work as far as possible. MPS does a decent job here, using, among others, the following strategies:

- Every language concept that is legal at a given program location is available in the code completion menu. In naive implementations, users have to select the language concept (based on its name) and instantiate it. This is inconvenient. In MPS, languages can instead define aliases for language concepts, allowing users to "just type" the alias, after which the concept is immediately instantiated.

- So-called side transforms make sure that expressions can be entered conveniently. Consider a local variable declaration `int a = 2;`. If this should be changed to `int a = 2+3;` the 2 in the init expression needs to be replaced by an instance of the binary + operator, with the 2 in the left slot and the 3 in the right. Instead of removing the 2 and inserting a +, users can simply type + on the right side of the 2; the system performs the tree refactoring that moves the + to the root of the subtree, puts the 2 in the left slot, and then puts the cursor into the right slot, to accept the second argument. This means that expressions (or anything else) can be entered linearly, as expected.

- Delete actions are used to similar effect when elements are deleted. Deleting the 3 in `2+3` first keeps the plus, with an empty right slot. Deleting the + then removes the + and puts the 2 at the root of the subtree.

- Wrappers support instantiation of concepts that are actually children of the concepts allowed in a given location. Consider again a local variable declaration `int a;`. The concept represented could be LocalVariableDeclaration, a subtype of Statement, to make it legal in method bodies (for example). However, users simply want to start typing `int`, i.e. selecting the content of the type field of the LocalVariableDeclaration. A wrapper can be used to support entering types where LocalVariableDeclarations are expected. Once a type is selected, the wrapper implementation creates a LocalVariableDeclaration and puts the type into its type field, and moves the cursor into the name slot.

- Smart references achieve a similar effect for references (as opposed to children). Consider pressing Ctrl-Space after the + in `2+3`. Assume further, that a couple of local variables are in scope and that these can be used instead of the 3. These should be available in the code completion menu. However, technically, a VariableReference has to be instantiated, whose variable slot then is made to point to any of the variables in scope. This is tedious. Smart references trigger special editor behavior: if in a given context a VariableReference is allowed, the editor *first* evaluates its scope to find the possible targets and then puts them into the code completion menu. If a user selects one, then the VariableReference is created, and the selected element is put into its variable slot. This makes the reference object effectively invisible in the editor.

- Smart delimiters are used to simplify inputting list-like data that is separated with a specific separator symbol, such as parameter lists. Once a parameter is entered, users can press comma, i.e. the list delimiter, to instantiate the next element.

Notice that, except for having to get used to the somewhat different way of editing programs, all other problems can be remedied with good tool support. Traditionally, this tool support has not always existed or been sufficient, and projectional editors have gotten a bit of a bad rep because of that. In case of MPS this tool support is available, and hence, MPS provides a productive and pleasant working environment.

1.3.2 Language Modularity

As we have seen in , language modularization and composition is an important building block in working with DSLs. Parser-based and projectional editors come with different trade-offs in this respect. Notice that is this section we only consider syntax issues — semantics has been covered in .

TODO:

TODO: the respective section in DSL design

In parser-based systems it depends on the grammar class supported by the tool whether composition can be supported well. The problem is that combining two or more independently developed grammars into one may become ambiguous, for example, because the same character sequence is defined as two different tokens. The resulting grammar cannot be parsed and has to be disambiguated manually, typically by invasively changing the resulting grammar. This of course breaks modularity and hence is not an option. Parsers that support the full set of context-free grammars, such as ANTLR, and hence Xtext, have this problem.

Parsers that support the full class of context-free grammars, such as the GLR parser used as part of Spoofax, do not have this problem. While a grammar may become ambiguous in the sense that a program may be parseable in more than one way, this can be resolved by declaratively specifying which alternative should be used. This specification can be made *without* invasively changing the resulting grammar, retaining modularity.

In projectional editors, language modularity and composition is not a problem at all. There is no grammar, no parsing, no grammar classes, and hence no problem with composed grammars becoming ambiguous. Any combination of languages will be syntactically valid. If a composed language would be ambiguous, the user has to make a disambiguating decision as the program is entered. For example, in MPS, if in a given location two language concepts are available under the same alias, just typing the alias won't bind, and the user has to manually decide by picking one alternative from the code completion menu.

1.3.3 Notational Freedom

Parser-based systems process linear sequences of character symbols. Traditionally, the character symbols were taken from the ASCII character set, resulting in textual programs being made up from "plain text". With the advent of unicode, a much wider variety of characters is available while still sticking to the linear sequence of character symbols approach. For example, the Fortress programming language makes use of this: greek letters and a wide variety of different bracket styles can be used in the programs. However, character layout is always ignored. For example it is not possible to use parsers to handle tabular notations, fraction bars or even graphics.

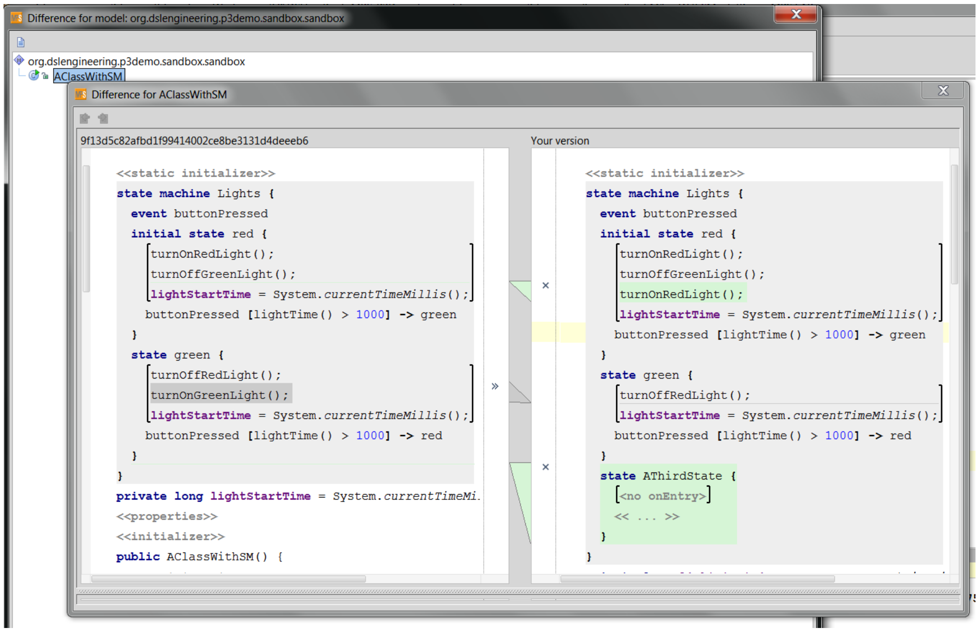
In projectional editing, this limitation does not exist. A projectional editor never has to extract the AST from the concrete syntax; editing gestures directly influence the AST, and the concrete syntax is rendered from the AST. This mechanism is basically like a graphical editor. Notations other than text can be used. For example, MPS supports tables as well as simple diagrams. Since these non-textual notations are handled the same way as the textual ones (possibly with other input gestures), they can be mixed easily: tables can be embedded into textual source, and textual languages can be used within table cells. Textual notations can also be used inside boxes or as connection labels in diagrams.

1.3.4 Language Evolution

Evolution of languages can be a bit more complex, since, if language concepts are not present anymore in a new revision of the language, existing instances of these concepts will break. MPS solves this problem with deprecation and "automatic" migration scripts. We will discuss this in more detail in the chapter on language evolution.

1.3.5 Infrastructure Integration

Special support also needs to be provided for infrastructure integration. Since the CS is not pure text, a generic persistence format needs to be devised. Therefore, special tools need to be provided for diff and merge. MPS provides integration with the usual VCS systems and handles diff and merge in the tool, using the concrete, projected syntax. Note that since every program element has a UUID, *move* can be distinguished from *delete* and *create*, providing more useable semantics for diff and merge. Fig. 1.3.5 shows an example of an MPS diff.


Difference for model: org.dslengineering.p3demo.sandbox.sandbox
org.dslengineering.p3demo.sandbox.sandbox
AClassWithSM
Difference for AClassWithSM
9f13d5c82afbd1f99414002ce8be3131d4deeeb6
Your version

```
<<static initializer>>
state machine Lights {
  event buttonPressed
  initial state red {
    [turnOnRedLight();
    turnOffGreenLight();
    lightStartTime = System.currentTimeMillis();]
    buttonPressed [lightTime() > 1000] -> green
  }
  state green {
    [turnOffRedLight();
    turnOnGreenLight();
    lightStartTime = System.currentTimeMillis();]
    buttonPressed [lightTime() > 1000] -> red
  }
}
private long lightStartTime = System.currentTimeMi.
<<properties>>
<<initializer>>
public AClassWithSM() {
```

```
<<static initializer>>
state machine Lights {
  event buttonPressed
  initial state red {
    [turnOnRedLight();
    turnOffGreenLight();
    turnOnRedLight();
    lightStartTime = System.currentTimeMillis();]
    buttonPressed [lightTime() > 1000] -> green
  }
  state green {
    [turnOffRedLight();
    lightStartTime = System.currentTimeMillis();]
    buttonPressed [lightTime() > 1000] -> red
  }
  state AThirdState {
    [<no onEntry>]
    << ... >>
  }
}
```

1.3.6 Other

In parser-based systems, the complete AST has to be reconstructable from the CS. This implies that there can be no information in the tree that is *not* obtained from parsing the text. This is different in projectional editors. For example, the textual notation could only project a subset of the information in the tree. The same information can be projected with different projections, each possibly tailored to a different stakeholder, and showing a different subset from the overall data. Since the tree uses a generic persistence mechanism, it can hold data that has not been planned for in the original language definition. All kinds of meta data (documentation, presence conditions, requirements traces) can be stored, and projected if required. MPS supports so-called annotations, where additional data can be added to model elements of existing languages and projecting that data inside the original projection, all without changing the original language specification.

1.4 Characteristics of AST formalisms

Most AST formalisms, aka Meta Meta Models, are ways to represent trees or graphs. Usually, such an AST formalism is "meta circular" in the sense that it can describe itself.

1.4.1 EMF Ecore

The Eclipse Modeling Framework (EMF) is at the core of all of Eclipse's modeling related activities. Its core component is the Ecore meta meta model, a version of the EMOF standard. The central concepts are: EClass, EAttribute, EReference and EObject, the latter providing an in-memory/run-time representation of EClass instances. EReferences can be containing or not - each EObject can be contained by at most one EReference instance. Fig. 1.4.1 shows a class diagram of Ecore.

The Ecore file is at the heart. From that, all kinds of other aspects are derived, specifically, a generic tree editor, and a generated Java API for accessing the AST of models and programs. This also forms the basis for Xtext's processing.

EMF has grown to be a fairly large ecology within the Eclipse community and numerous projects use EMF as a basis for model manipulation and persistence, with Ecore for meta model definition.

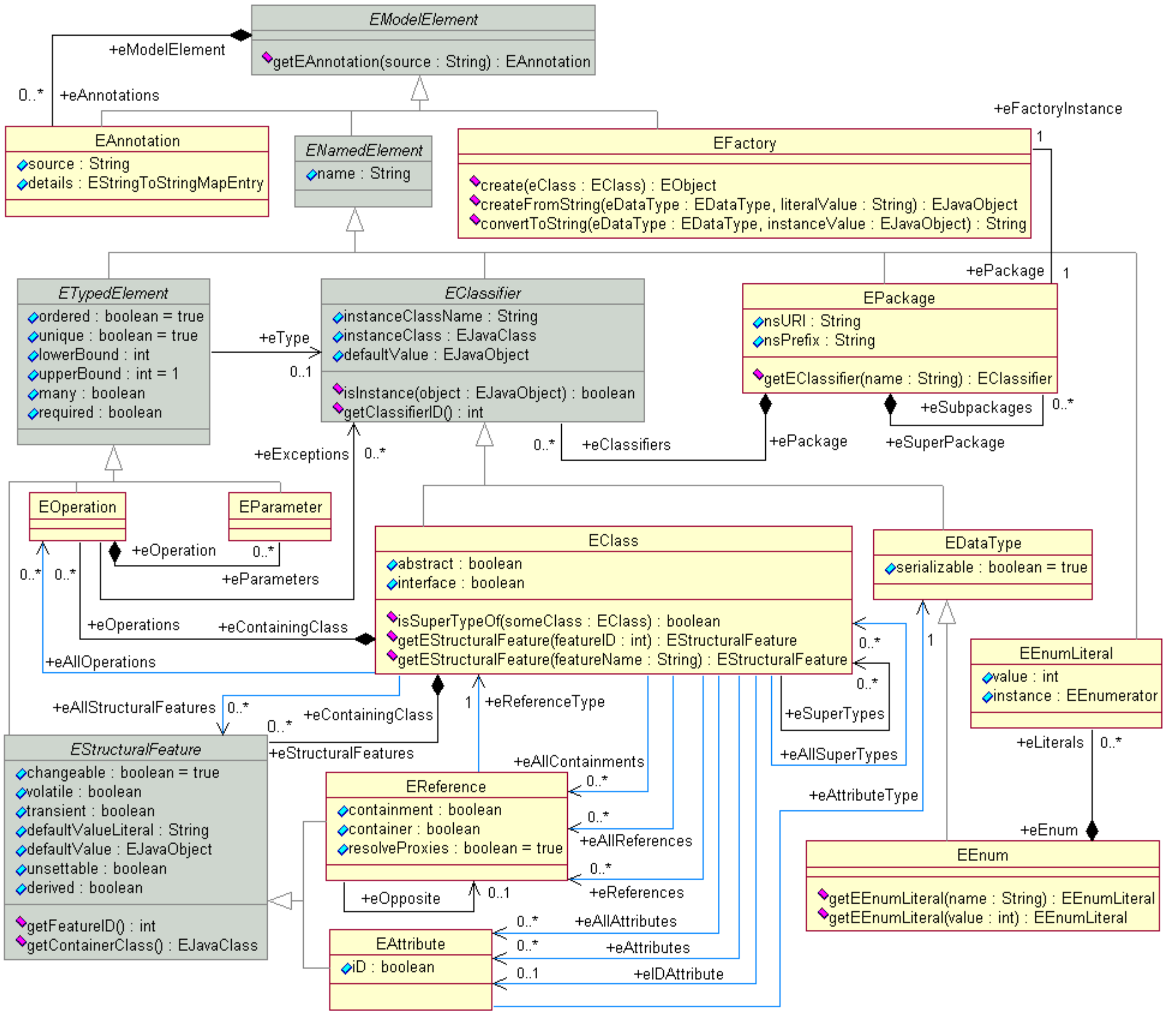
1.4.2 Spoofax' ATerm

Spoofax uses the ATerm format to represent abstract syntax. ATerms are a generic tree structure representation format that can be serialized textually similar to XML or JSON. ATerms consist of the following elements:

- Strings, e.g. `"Mr. White"`
- Numbers, e.g. `15`
- Lists, e.g. `[1,2,3]`
- Constructor applications, e.g. `Order(5, 15, "Mr. White")`

In addition to these basic elements, ATerms can also be annotated with additional information, e.g. `Order(5{ProductName("Apples")}, 15, "Mr. White")`. These annotations are often used to represent references to other parts of a model.

The textual notation of ATerms can be used for exchanging data between tools and as a notation for model transformations or code generation rules. In memory, ATerms can be stored in a tools-specific fashion (i.e., simple Java objects in the case of Spoofax). The generic structure and serializability of ATerms also allows them to be converted to other data formats. For example, the aterm2xml/xml2aterm tools can convert between ATerms and XML.



1.4.3 *MPS' Structure Definition*

Every program element in MPS is a node. A node has a structure definition and projection rules for rendering. This is also true for language definitions. Nodes are instances of Concepts, which corresponds to EMF's EClass. Like EClasses, concepts are meta circular, i.e. there is a concept that defines the properties of concepts:

```
concept ConceptDeclaration extends AbstractConceptDeclaration
                           implements INamedConcept
                                      IStructureDeprecatable

  instance can be root: false

  properties:
    helpURL : string
    rootable : boolean

  children:
    InterfaceConceptReference  implements                 0..n
    LinkDeclaration            linkDeclaration            0..n
    PropertyDeclaration        propertyDeclaration        0..n
    ConceptProperty            conceptProperty            0..n
    ConceptLink                conceptLink                0..n
    ConceptPropertyDeclaration conceptPropertyDeclaration 0..n
    ConceptLinkDeclaration     conceptLinkDeclaration     0..n

  references:
    ConceptDeclaration         extends                   0..1

  concept properties:
    alias = concept
```

A concept may extend a single other concept and implement any number of interfaces. It can declare references and child collections. It may also have a number of primitive-type properties as well as a couple of "static" features (those stating with "concept"). In addition, concepts can have behavior methods.

1.5 Xtext Example

Cooling programs represent the behavioral aspect of the refrigerator language. Here is a trivial program that can be used to illustrate some of the features of the language.

TODO: We assume an introduction of the case studies somewhere before this point

```
cooling program EngineProgram0 {

    var v: int
    event e

    init { set v = 1 }

    start:
       on e { state s2 }

    state s2:
        entry { set v = 0 }

}
```

The program declares a variable `v` and an event `e`. When the program starts up, the init section is executed, setting `v` to 1. The system then (automatically) transitions into the start state. There it waits until it receives the `e` event. It then transition to the `s2` state, where it uses an entry action to set `v` back to 0. More complex programs include checks of changes of measurements (`compartment1->currentTemp`) and commands to the hardware (`do compartment1->coolOn`), as shown in the next snippet:

```
start:
    check ( compartment1->currentTemp > maxTemp ) {
        do compartment1->coolOn
        state initialCooling
    }
    check ( compartment1->currentTemp <= maxTemp ) {
        state normalCooling
    }

state initialCooling:
    check ( compartment1->currentTemp < maxTemp ) {
        state normalCooling
    }
```

Grammar basics In Xtext, the language is specified using a grammar which is a collection of *parser rules*. These rules specify (by default) the concrete syntax of a program element, as well as its mapping to the AS. Xtext generates an Ecore meta model to describe the AS. Here is the definition of the CoolingProgram rule:

```
CoolingProgram:
    "cooling" "program" name=ID "{"
      (events+=CustomEvent |
        variables+=Variable)*
      (initBlock=InitBlock)?
      (states+=State)*
    "}";
```

Rules begin with the name (CoolingProgram), a colon, and then the rule body. The body defines the syntactic structure of the language concept defined by the rule. In our case, we expect the keywords *cooling* and *program*, followed by an ID. ID is a *terminal rule* that is defined in the parent grammar from which we inherit. It is defined as an unbounded sequence of lowercase and uppercase characters, digits, and the underscore, although it may not start with a digit. This terminal rule is defined as follows, again using a BNF-like syntax:

```
terminal ID: (’a’..’z’|’A’..’Z’|’_’) (’a’..’z’|’A’..’Z’|’_’|’0’..’9’)*;
```

In pure grammar languages, one would write `"cooling" "program" ID"{"`, specifying that after the two keywords we expect an ID as defined above. However, in Xtext grammars don’t just express the concrete syntax - they also determine the mapping to the AS. We have encountered two such mappings so far. The first one is implicit (although it can be made explicit as well): the name of the rule will be the name of the generated meta class. So we will get a meta class CoolingProgram. The second mapping we have encountered in `name=ID`. It specifies that the meta class get a property name that holds the contents of the ID in the program text. Since nothing else is specified in the ID terminal rule, the type of this property defaults to EString, Ecore’s implementation of a string data type.

The rest of the definition of a cooling program is enclosed in curly braces. It contains three elements: first the program contains a collection of events and variables (the asterisk specifies unbounded multiplicity), an optional init block (optionality is specified by the question mark) and a list of states. Let us inspect each of these in more detail.

The expression `(states+=State)*` specifies that there can be any number of State instances in the program. The meta class gets a property `states`, it is of type State (the meta class derived from the State rule). Since we use the += operator, the `states` property is a list as well. In case of the optional init block, the meta class will have an `initBlock` property, typed as InitBlock (whose parser rule we don't show and discuss here), with a multiplicity of 0..1. Events and variables are more interesting, since the vertical bar operator is used within the parentheses. The asterisk expresses that whatever is inside the parentheses can occur any number of times - note that the use of a * usually goes hand in hand with that of a +=. Inside the parentheses we expect either a CustomEvent or a Variable. Variables are assigned to the variables collection, events are assigned to the events collection. This notation means that we can mix events and variables in any order.

The following alternative notation would first expect all events, and then all variables. Im the end, it depends on your end users which notation is more suitable.

```
      (events+=CustomEvent)*
      (variables+=Variable)*
```

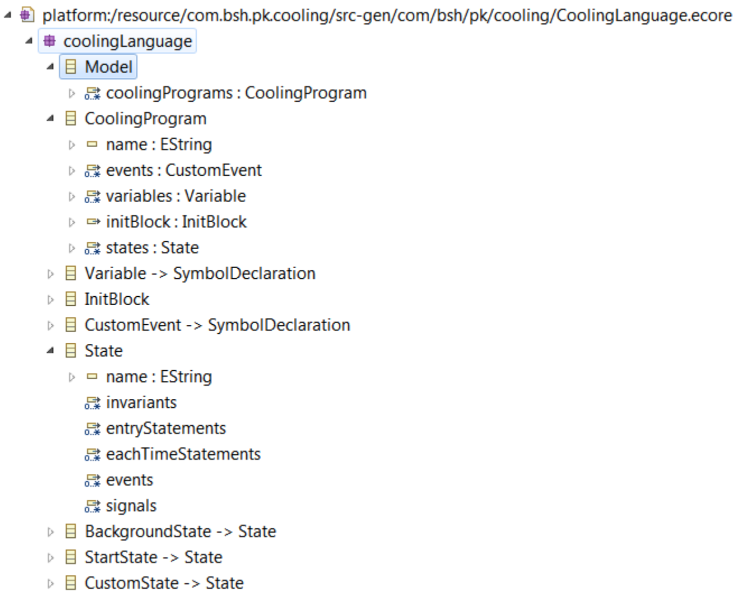
The definition of State is different, since State is intended to be an abstract meta class with several subtypes.

```
State:
    BackgroundState | StartState | CustomState;
```

The vertical bar operator is used here to express alternatives regarding the syntax. This is translated to inheritance in the meta model. The definition of custom state is shown in the following code snippet. It uses the same grammar language features as explained above.

```
CustomState:
    "state" name=ID ":"
        (invariants+=Invariant)*
        ("entry" "{"
            (entryStatements+=Statement)*
        "}")?
        ("eachTime" "{"
            (eachTimeStatements+=Statement)*
        "}")?
        (events+=EventHandler | signals+=SignalHandler)*;
```

StartState and BackgroundState, the other two subtypes of State, share some properties. Consequently, Xtexts AS derivation algorithm pulls them up into the abstract State meta class so they can be accessed polymorphically. Fig. 1.5 shows the resulting meta model using EMF's tree view.



```
platform:/resource/com.bsh.pk.cooling/src-gen/com/bsh/pk/cooling/CoolingLanguage.ecore
  coolingLanguage
    Model
      coolingPrograms : CoolingProgram
    CoolingProgram
      name : EString
      events : CustomEvent
      variables : Variable
      initBlock : InitBlock
      states : State
    Variable -> SymbolDeclaration
    InitBlock
    CustomEvent -> SymbolDeclaration
    State
      name : EString
      invariants
      entryStatements
      eachTimeStatements
      events
      signals
    BackgroundState -> State
    StartState -> State
    CustomState -> State
```

References Let us now look at statements and expressions. States have entry and exit statements, procedural code that is executed when a state is entered and left respectively. The set v = 1 is an example. Statement itself is of course abstract and has all the various kinds of statements as subtypes/alternatives (in the actual language there are many more):

EV: maybe this should first show the grammar rule without the cross-reference, and then explain that's not how you normally do it in Xtext, but that you should make it a cross-reference instead. I suppose you could then even show it with targetState=[State|QID] and say that targetState=[State] is a short-hand in case it's an ID. that way maybe we can maintain a grammar perspective in this section

```
Statement:
    IfStatement | AssignmentStatement | PerformAsyncStatement |
    ChangeStateStatement | AssertStatement;

ChangeStateStatement:
    "state" targetState=[State];

AssignmentStatement:
    "set" left=Expr "=" right=Expr;
```

The ChangeStateStatement is used to transition into another state of the overall state-based program. It uses the keyword *state* and then a reference to the actual target state in the form of its name. Notice how Xtext uses brackets to express the fact that the targetState property points to an existing State as opposed to containing a new one (written as targetState=State), i.e.: a non-containing cross-reference. In traditional grammars, the rule would be written as "state" targetStateName=ID;

and the language implementation would have to provide functionality to follow the `targetState` link separately.

Note that the cross-reference definition only specifies the target type of the cross-reference, but not the CS itself: by default, the ID terminal is used for the syntax. Initially, this ID ends up in a proxy URI which is resolved lazily as soon as we try and access the property. The Xtext framework takes care of most of this out-of-the-box, but you can exert a lot of control on its behavior by means of a custom scope provider.

There are two more details to explain. In case the actual terminal or datatype rule used to represent the reference is not an ID, but, for example, a name with dots in it (qualified name), this can be specified as part of the reference. Notice how in this case the vertical bar does not represent an alternative, it is merely used as a separator between the target type and the terminal used to represent the reference.

```
ChangeStateStatement:
    "state" targetState=[State|QID];

QID: ID ("." ID)*;
```

The other remaining detail is scoping. During the linking phase, where the text of ID (or QID) is used to find the target object, several objects with the same name might exist, or some target elements are not visible based on visibility rules of the language. To control the possible reference targets, scoping functions are used. These will be explained in the next section.

TODO: MB: I still have to reconcile the duplicate explanation of scoping in this paragraph. . .

Expressions The AssignmentStatement is one of the statements that uses expressions. This leads us to discussing the implementation of expressions in Xtext. Let us look at the definition of expressions as a whole. The following snippet is a subset of the actual definition of expressions (we have omitted some additional expressions that don't add anything to the description here).

```
Expr:
    ComparisonLevel;

ComparisonLevel returns Expression:
    AdditionLevel ((({Equals.left=current} "==") |
                   ({LogicalAnd.left=current} "&&") |
                   ({Smaller.left=current} "<"))
                                        right=AdditionLevel)?;

AdditionLevel returns Expression:
    MultiplicationLevel ((({Plus.left=current} "+") |
                         ({Minus.left=current} "-")) right=MultiplicationLevel)?;

MultiplicationLevel returns Expression:
    PrefixOpLevel ((({Multi.left=current} "*") |
                   ({Div.left=current} "/")) right=PrefixOpLevel)?;
```

```
PrefixOpLevel returns Expression:
    ({NotExpression} "!" "(" expr=Expr ")") |
    AtomicLevel;

AtomicLevel returns Expression:
    ({TrueExpr} "true") |
    ({FalseExpr} "false") |
    ({ParenExpr} "(" expr=Expr ")") |
    ({NumberLiteral} value=DECIMAL_NUMBER) |
    ({SymbolRef} symbol=[SymbolDeclaration|QID]);
```

We first have to explain in more detail how the AST construction works in Xtext. Obviously, as the text is parsed, meta classes are instantiated and the AST is assembled. However, instantiation of the respective meta class happens only when the first assignment to one of its properties is performed. If no assignment is performed at all, no object is created. For example in case of `TrueLiteral: "true";` no instance of TrueLiteral will ever be created, because there is nothing to assign. In this case, an action can be used to force instantiation: `TrueLiteral: {TrueLiteral} "true";` Notice that the action can instantiate meta classes other than those that are derived from the rule name. Unless otherwise specified, an assignment such as `name=ID` is always interpreted as an assignment on the object that has been created most recently. The *current* keyword can be used to access that object in case it itself needs to be assigned to a property.

Now we know enough to understand how expressions are encoded and parsed. For the rules with the Level suffix, no meta classes are created, because (as Xtext is able to find out statically) they are never instantiated. They merely act as a way to encode precedence. To understand this, let's consider how `2*3` is parsed:

- We start (by definition) with the Expr rule. The Expr rule just calls the ComparisonLevel rule. Note that a rule call can have two effects: it returns an object which represents the parsed result or null in case no object was instantiated, or it internally throws an exception indicating that the input couldn't be matched to the called rule. The returned object is available in the calling rule using the *current* keyword.

- The parser now "dives down" until it matches the 2. This occurs on AtomicLevel, as it matches the `DECIMAL\_NUMBER` terminal. At this point it creates an instance of NumberLiteral and assigns the actual number (2) to the value property. It also sets the *current* object to point to the just created NumberLiteral.

- The AtomicLevel rule ends, and the stack is unwound. We're back at PrefixOpLevel, in the second branch. Since nothing else is specified, we unwind once more.

- We're now back at the MultiplicationLevel. The rule we have on the stack is not finished yet and we try to match a * and a /. The match on * succeeds. At this point the so-called assignment action on the left side of the * kicks in (`{Multi.left=current}`). This action creates an instance of Multi, and assigns the current (the NumberLiteral) to its left property. Then it assigns the newly created Multi to be the current object. At this point we have a subtree with the * at the root, and the NumerLiteral 2 in the left property.
- The rule hasn't ended yet. We dive down to PrefixOpLevel once more, matching the 3 in the same way as the two before. The NumerLiteral for 3 is assigned to the right property.
- At this point we unwind the stack further, and since no more text is present, no more objects are created. The tree structure is as we had expected.

If we'd parsed `4 + 2*3` the + would have matched before the *, because it is "mentioned earlier" in the grammar, it is in a lower-precedence group, the AdditionLevel. Once we're a the 4 + tree, we'd go down again to match the 2. As we unwind the stack after matching the 2 we'd match the *, creating a Multi again. The current, at this point, would be the 2, so it would be put onto the left side of the *, making the * the current. Unwinding further, that * would be put onto the right side of the +, building the tree just as we'd expect.

Notice how a rule at a given level only always delegates to rules at higher precedence levels. So higher precedence rules always end up further down in the tree. If we want to change this, we can use parentheses: inside those, we can again embed an Expr, i.e. we jump back to the lowest precedence level.

TODO: Explain why Xtext does it this way (left recursion, etc.) and refer back to general parsing discussion.

To use Xtext with expressions it is not necessary to completely understand the mechanism, since expressions are always structured in the way shown above. The code shown here can be used as a template. Adding new operators or new levels can be done easily.

Polymorphic References In the cooling language, expressions also include references to other entities, such as configuration parameters, variables and hardware elements such as compressors and fans (defined in a different model, not shown above). All of these referencable elements extend the SymbolDeclaration meta class. This means that all of them can be referenced by the single SymbolRef construct.

```
AtomicLevel returns Expression:
    ...
    ({SymbolRef} symbol=[SymbolDeclaration|QID]);
```

The problem with this situation is that the reference itself does not encode the kind of thing that is referenced. By looking at the reference alone we only know that we reference some kind of symbol. This makes writing code that processes the model cumbersome, since the target of a SymbolRef has to be taken into account when deciding how to treat (translate, validate) a symbol reference. A more natural design of the language would use different reference constructs for the different referencable elements. In this case, the reference itself is specific to the referenced meta class, making processing much easier.

```
AtomicLevel returns Expression:
    ...
    ({VariableRef} var=[Variable]);
    ({ParameterRef} param=[Parameter]);
    ({HardwareBuildingBlockRef} hbb=[HardwareBuildingBlock]);
```

However, this is not possible with Xtext, since the parser cannot distinguish the three cases syntactically. In all three cases, the reference itself is just an ID. Only during the linking phase could the system check which kind of element is actually referenced, but this is too late for the parser, which needs an unambiguous grammar. The grammar could be disambiguated by using a different syntax for each element:

```
AtomicLevel returns Expression:
    ...
    ({VariableRef} "v:" var=[Variable]);
    ({ParameterRef} "p:" param=[Parameter]);
    ({HardwareBuildingBlockRef} "bb:" hbb=[HardwareBuildingBlock]);
```

While this approach will technically work, it would lead to an awkward syntax and is hence typically not used. The only remaining alternative is to make all referencable elements extend SymbolDeclaration and use a single reference.

TODO: Explain how to go from AS to CS

1.6 Spoofax Example

Mobl's data modeling language provides entities and entity functions. To illustrate the language, here is an example of two data type definitions related to a shopping list app:

```
module shopping

entity Item {
  name     : String
  checked  : Bool
  favorite : Bool
  onlist   : Bool
  order    : Num
  store    : Store
}

entity Store {
  name  : String
  open  : Time
  close : Time

  function isOpen() {
    return open <= now.getTime() && close <= now().getTime();
  }
}
```

In mobl, most files starts with a module header, which can be followed by a list of entity type definitions. Our example *shopping* module defines two entities for items on a shopping list and stores for those items. Entities are persistent data types that are stored in a database and can be retrieved using mobl's querying API.

Syntax:

```
"entity" QId ":" Type "{" EntityBodyDecl* "}"  -> Definition {cons("Entity")}
"entity" QId "{" EntityBodyDecl* "}"  -> Definition {cons("EntityNoSuper")}
ID ":" Type "(" {Anno ","}* ")" -> EntityBodyDecl {cons("Property")}
ID ":" Type                     -> EntityBodyDecl {cons("PropertyNoAnnos")}
FunctionDef                               -> EntityBodyDecl
ID                              -> Anno {cons("SimpleAnno")}
"inverse" ":" ID                -> Anno {cons("InverseAnno")}


"function" QId "(" {FArg ","}* ")" ":" Type "{" Statement* "}" -> FunctionDef {cons("Function")}
"return" Exp ";"                            -> Statement
{cons("Return")}

Exp "." ID "(" { Exp  ")" -> Exp {cons("MethodCall")}
Exp "." ID                -> Exp {cons("FieldAccess")}
ID                        -> Exp {cons("Var")}
```

1.7 MPS Example

We start by defining a simple language for state machines. Core concepts include StateMachine, State, Transition and Trigger. The state machine can be embedded in C code as we will see later. The language supports the definition of state machines as shown in the following piece of code:

```
statemachine linefollower {
  event initialized;
  event bumped;
  event blocked;
  event unblocked;
  initial state initializing {
    initialized [true] -> running
  }
  state paused {
    entry int16 i = 1;
    unblocked [true] -> running
  }
  state running {
    blocked [true] -> paused
    bumped [true] -> crash
  }
  state crash {
    <<transitions>>
  }
}
```

TODO: Add UML diagram of the language. Generate via MPS.

Concept Definition MPS is projectional so we start with the definition of the AS. In MPS, AS elements are called concepts. The code below shows the definition of the concept Statemachine. It contains a collection of States and a collection of Events. The alias is defined as "statemachine", so typing this word inside C modules instantiates a state machines. Statemachine also implements a couple of interfaces; IHasIdentifierName contributes a property name, IModuleContent makes the state machine embeddable in C Modules — the module owns a collection of IModuleContents, just like the Statemachine contains States and Events.

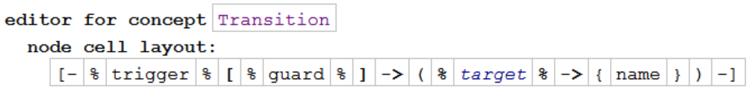
```
concept Statemachine extends MedBase
                     implements IHasIdentifierName
                                IModuleContent
  properties:
    << ... >>

  children:
    State states 0..n specializes: <none>
    Event events 0..n specializes: <none>

  references:
    << ... >>

  concept properties:
    alias = statemachine
```

The State contains collections of EntryActions and ExitActions. These both extend Action, which in turn owns a StatementList to include C code. StatementList is a concept defined by the C core language. To make that visible, our statemachine language extends C core. A state contains a boolean attribute initial, to mark the initial state. Finally, a State contains a collection of Transitions.

```
concept Transition extends MedBase
                   implements <none>
  properties:
    << ... >>

  children:
    Trigger trigger 1 specializes: <none>
    Expression guard 1 specializes: <none>

  references:
    State target 1 specializes: <none>

  concept properties:
    alias = transition
```

Transitions contain a Trigger, a guard condition and the target state. The trigger is an abstract concept; various specializations are possible, the default implementation is the EventTrigger, which references an event. The guard condition is of type Expression; another concept inherited from the C core language. A typesystem rule will be defined later to constrain this expression to be boolean. The target state is a reference, i.e. we point to an existing state instead of owning it.

Editor Definition Editors are defined via projection rules. Editors are made of cells. When defining editors, various cell types are arranged so that the resulting syntax has the desired structure. Fig. 1.7 shows the definition of the editor for a transition. It arranges the trigger, guard and target state in a horizontal list of cells, the guard surrounded by brackets, and an arrow (->) in front of the target state.

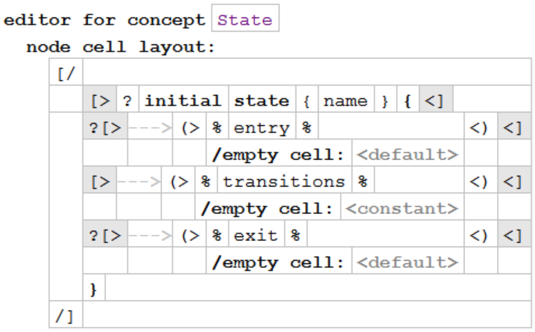


```
editor for concept Transition
  node cell layout:
    [- % trigger % [ % guard % ] -> ( % target % -> { name } ) -]
```

TODO: Add a table projection to show off non textual notations

Fig. 1.7 shows the editor definition for the State. It uses a vertical list to arrange several horizontal elements. The first line contains the keyword initial, the keyword state and the name property of the state. The question mark in front of the initial label denotes this cell as being optional; an expression determines whether it is shown or not. In this case, the expression checks the initial property of the current state, and shows the keyword only if is true. A quick fix is used to toggle the property.

The rest of the editor arranges the entry actions, transitions and exit actions in a vertical arrangement. Each of the collections is indented (—->). The entry and exit actions are only shown if the collections are not empty (another optional cell). A quick fix is used to create the first entry and exit actions, subsequent ones can be created by pressing enter at the end of an existing action.



```
editor for concept State
  node cell layout:
    [/
      [> ? initial state { name } { <]
      ?[> ---> (> % entry %           <) <]
                 /empty cell: <default>
      [> ---> (> % transitions %      <) <]
                /empty cell: <constant>
      ?[> ---> (> % exit %            <) <]
                 /empty cell: <default>
      }
    /]
```

Intentions Since the use of quick fixes (called intentions in MPS) is an essential part of editing with MPS, we show how they are created in this section. The following piece of code shows the definition on the one that triggers the initial properties. Intentions specify for which language concept they apply, a text that describes the intention in the intentions menu, an optional boolean expression that determines whether the intention is currently applicable, and the actual code that performs the quickfix. Note how this is simply a model-to-model transformation expressed with MPS APIs.

```
intention makeInitialState for concept State {

  description(editorContext, node)->string {
    "Statemachine: Make Initial";
  }

  <isApplicable = true>

  execute(editorContext, node)->void {
    foreach s in node.ancestor<concept = Statemachine>.states {
      s.initial = false;
    }
    node.initial = true;
  }
}
```

Expressions Since we inherit the guard expression structure and syntax from the C core language, we don't have to define expressions. It is nonetheless interesting to look at its implementation.

Expressions are arranged into a hierarchy starting with the abstract concept Expression. All other kinds of expressions extend Expression, directly or indirectly. For example, the PlusExpression extends the BinaryExpression which in turn extends Expression. BinaryExpressions have left and right child Expressions. This way, arbitrarily complex expressions can be built. Representing expressions as trees is a standard approach; in that sense, the abstract syntax of MPS expressions is not very interesing. The editors are also trivial — in case of the plus expression, they are a horizontal list of: editor for left argument, the plus symbol, and the editor for the right argument.

As we have explained in the general discussion about projectional editing, MPS supports linear input of hierarchical expressions using so-called side transforms. The code below shows the right side transformation for expressions that transforms an arbitrary expression into a PlusExpression but putting the PlusExpression "on top" of the current node. Using the alias of expressions and the inheritance hierarchy, it is possible to factor all side transformations for all binary operations into one single action declaration, resulting in much less implementation effort.

```
side transform actions makeArithmeticExpression

  right transformed node: Expression tag: default_

  actions :
    add custom items  (output concept: PlusExpression)
      simple item
        matching text
          +
        description text
          <default>
        icon
          <default>
        type
          <default>
        do transform
          (operationContext, scope, model, sourceNode, pattern)->node< > {
            node<PlusExpression> expr = new node<PlusExpression>();
            sourceNode.replace with(expr);
            expr.left = sourceNode;
            expr.right.set new(<default>);
            return expr.right;
          }
```

In the AST, operator precedence is encoded by where in an expression tree a given expression is located. For example, in `2+3*4` the + is higher up in the tree than the *, encoding that the * has higher precedence. This is natural since processing of ASTs is typically depth-first, so the * is evaluated before the +. However, as we enter an expression linearly using the side transformations actions, we have to make sure that we don't accidentally put a + "below" a *. There are two ways to deal with this:

- The first alternative allows lower-precedence operators below higher-precedence operators in the tree, but the editor automatically renders parentheses to make sure visual appearance is in line with the AST.

- The other alternative reshuffles the tree automatically whenever an expression is edited, putting higher-precedence operators further down in the tree. Parenthesis can still be used explicitly.

In both cases the system needs information about the precedence level of operators or expressions. This information can be stored in a concept property (comparable to a static member in a Java class).

Polymorphic References We have explained above how references work in principle: they are actual pointers to the references element (the necessary scopes are explained in the next section). In the section on Xtext () we have seen how from a given location only one kind of reference for any given syntactic form can be implemented. Consider the following example, where we refer to a local variable (a) and an event parameter (timestamp) from within expressions:

EV: even more so than the previous bit on references, this part doesn't fit here and should be moved to the section on Scoping and Linking

TODO: make sure we explain that first

```
int a;
int b;

statemachine linefollower {
  event initialized(int timestamp);
  initial state initializing {
    initialized [now() - timestamp > 1000 && a > 3] -> running
  }
}
```

Both references to local variables and to event parameters use the same syntactic form: simply a name. In Xtext, this has to be implemented with a single reference (typically called SymbolReference) that can reference to any kind of Symbol. LocalVariableDefintions and EventParameters would both extend Symbol, and scopes would make sure both kinds are visible from within guard expressions. The problem with this approach is that the reference itself contains no type information about what it references, it is simply a SymbolReferenece. Processing code has to inspect the type of the symbol to find out what a SymbolReference actually means.

In projectional editors this done differently. In the example above there is a LocalVariableReference and an EventParameterReference. Both have an editor that simply renders the name of the referenced element. Entering the reference happens by typing the name of the referenced element (cf the concept of smart references introduced above). In the (rare) case where there's a LocalVariable and a EventParameter, the user has to make an explicit decision, at the time of entry (the name won't bind, and the CC menu requires a choice). It is important to understand that, although the names are similar, the tool still knows which one refers to a LocalVariable and which one refers to a EventParameterReference. Upon selection, the correct reference objects are created.