**Part III**

# DSL Implementation

dsl engineering 175

|  |  |
| --- | --- |
| This part of the book has been written together Lennart Kats and Guido Wachsmuth, who contributed the material on Spoofax, and Christian Dietrich, who helped with the language modularization in Xtext.  In this part we describe language implementation with three language workbenches, which together represent the current state of the art: Spoofax, Xtext and MPS. All of them are Open Source, so you can experiment with them. For more example language implementations using more language workbenches, |  |
| take a look at the Language Workbench Competition website10.  This part of the book does not cover a lot of design decisions or motivation for having things like constraints, type systems, transformations or generators. Conceptually these topics are introduced in Part II of the book on DSL design. This part really just looks at the "how", not the "what" or "why".  Each chapter contains examples implemented with all three tools. The ordering of the tools is different from chapter to chapter, based on the characteristics of each tool: if the example for tool A illustrates a point that is also relevant for tool B, then A is discussed before B.  The examples are not intended to serve as a full tutorial for any of these tools, but as an illustration of the concepts and ideas involved with language implementation in general. However, they should give you a solid understanding of the capabilities of each of the tools, and the class of tools they stand for. Also, if a chapter does not explain topic X for tool Y, this does *not* imply that you cannot do X with Y – it just means that Y’s approach to X is not significantly different from things that have already been discussed in the chapter. |  |

*7*

## Concrete and Abstract Syntax

*In this chapter we look at the definition of abstract and concrete syntax, and the mapping between the two in parserbased and projectional systems. We also discuss the advantages and drawbacks of these two approaches. We discuss the characteristics of typical AST definition formalisms. The meat of the chapter is made up of extensive examples for defining language structure and syntax with our three example tools.*

The *concrete syntax* (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; examples of other forms are briefly discussed Section 4.7. In this chapter we refer to other forms where appropriate.

The *abstract syntax* (AS) of a language is a data structure that holds the core information in a program, but without any of the notational details contained in the concrete syntax: keywords and symbols, layout (e.g., whitespace), and comments are typically not included in the AS. In parser-based systems the syntactic information that 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 the creation of IDE features such as quick fixes or formatters.

As we have seen in the introduction, the abstract syntax is essentially 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. Most 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 interface 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 relationship between the 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 tree. A possible AST is illustrated in Fig. 7.11.

|  |
| --- |
|  |

mar for an existing meta model.

Once the language is defined, there are again two ways in

which the abstract syntax and the concrete syntax can relate as the language is used to create programs4:

|  |  |
| --- | --- |
| *Parsing* In the parser-based approach, the abstract syntax tree |  |
| is constructed 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*5. Xtext and Spoofax use this approach. | |  |

|  |  |
| --- | --- |
| There are two ways of defining the relationship between the CS and the AS as part of language development:  *CS first* From a concrete syntax definition, an abstract syntax is derived, either automatically or using hints in the concrete syntax specification2. This is the default use for Xtext, where |  |
| Xtext derives the Ecore meta model from an Xtext grammar.  *AS first* We first define the AS. We then define the concrete syntax, referring to the AS in the definition of the concrete |  |
| syntax3. For example, in Xtext it is possible to define gram- |  |

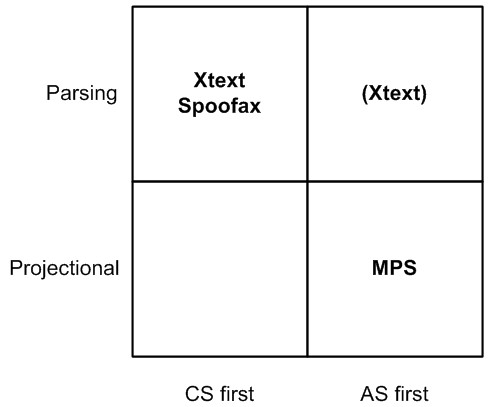
*Projection* In the projectional approach, the abstract syntax tree is built directly by editor actions, and the concrete syntax is rendered from the AST via projection rules. MPS is an example of a tool that uses projectional editing.

Fig. 7.2 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.

|  |  |
| --- | --- |
|  |  |
| *7.1 Fundamentals of Free Text Editing and Parsing*  Most programming environments rely on free text editing, where programmers edit programs at the text/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 data structures from information extracted from the textual source. Most modern IDEs perform this task in real-time as the user edits the program, and the AST is always kept in sync with the program text. Many IDE features – such as content assist, validation, navigation or refactoring support – are based on this synchronized AST.

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[[1]](#footnote-1). 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. The AS can be changed (by changing the mapping from the CS to an AS) without any effect on the CS and existing programs.

Many different approaches exist for implementing parsers.

Each may restrict the syntactic freedom of a language, or con-

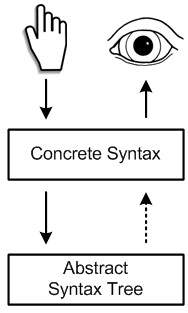


Figure 7.3: In parser-based systems, the user only interacts with the concrete syntax, and the AST is constructed from the information in the text via a parser.

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

### 7.1.1 Parser Generation Technology

In traditional compilers and IDEs (such as gcc or the Eclipse JDT), parsers are often written by hand as a big, monolothic program that reads a stream of characters and uses recursion to create a tree structure. However, manually writing a parser requires significant expertise in parsing and a significant 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 provide good error reporting and error recovery (the ability to continue parsing after a syntax error has been found).

In contrast, language workbenches, and most of today’s compilers, *generate* a parser from a grammar. A grammar is a syntax specification written in a DSL for formally defining textual concrete syntax. These generated parsers may not provide the same performance or error reporting/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. Also, they generate a *complete* parser for the *complete* grammar – developers may forget corner cases if they write the parser manually. However, the most important argument for using parser generation is that the effort of building a parser is *much* lower than manually writing a custom parser[[2]](#footnote-2). Finally, it means that the

developer who defines a language does not have to be an expert in parsing technology.

*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 or whitespace) by a *scanner* (sometimes also called lexer or tokenizer). The *parser* then constructs the actual AST from the token sequence[[3]](#footnote-3). 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**9. Modern parsing frameworks, such as |  | |
| ANTLR10, do their own scanner generation. |  | |
| A separate scanning phase has direct consequences for the overall parser implementation, when the scanner is not aware of the context of its input. An example of a typical problem that arises from this is that keywords can’t be used as identifiers even though the use of a keyword frequently 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 is also problematic when grammars 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, respectively. 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.  A recent technique to overcome these problems is *contextaware scanning*, in which the lexer relies on the state of the |  | |
| parser to determine how to interpret the next token11. With |  | |
| *scannerless parsing*, there is no separate scanner at all. Instead, the parser operates at the character level and statefully processes lexicals and keywords, avoiding the problems of contextunaware scanning illustrated above. Spoofax (or rather, the un- |  | |
| derlying parser technology SDF) uses scannerless parsing.  *Grammars* Grammars are the formal definition for concrete textual syntax. They consist of *production rules* that define how valid textual input ("sentences") look like12. Grammars are the basis for syntax definitions in text-based workbenches such as Spoofax and Xtext13. |  | |
| Fundamentally, production rules can be expressed in Backus- |  | |
| Naur Form (BNF)14, written as *S* **::=** *P*1 **...** *Pn*. This grammar |  | |
| defines a symbol *S* by a series of pattern expressions *P*1 **...** *Pn*. |  | |
| 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 (for the same symbol), or the patterns can be separated by the **|** operator to indicate a choice. An ex- |  |
| tension of BNF, called Extended BNF (EBNF)15, 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 of a pattern expression.  The following code is 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 **\*** |  |
| operators16. |  |

|  |  |
| --- | --- |
| Note how expression nesting is described using recursion in this grammar: the **Exp** rule calls itself, so sentences like **2 + 3 \* 4** are possible. This poses two practical challenges for parser generation systems: first, the precedence and associativity of the operators is not described by this grammar. Second, not all parser generators provide full support for recursion. For example, ANTLR cannot cope with left-recursive rules. We elaborate on these issues in 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 languages17. In con- |  |
| trast, *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.  Parser generation was first applied in command-line tools |  |
| such as **yacc** in the early seventies18. 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 |  |

|  |
| --- |
| **Exp** ::= **NUM**  | **Exp** "+" **Exp**  | **Exp** "\*" **Exp** |

so on. A particular parser tool supports a specific class of grammars – e.g., ANTLR supports LL(*k*) and LL(\*). In this naming scheme, 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. The bigger *k*, the more syntactic forms can be parsed19. Typi-

|  |  |
| --- | --- |
| cally, grammars for "real" DSLs tend to need only finite lookahead 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.  Supporting only a subclass of all possible context-free grammars poses restrictions on the languages that are supported by 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 differ- |  |
| ent, sometimes awkward or unintuitive way to conform to the subclass. This will be illustrated in the Xtext example, which uses ANTLR as the underlying LL(*k*) parser technology.  Parser generators can detect whether a grammar conforms to a certain subclass, reporting conflicts that relate to the implementation of the parsing algorithm20. Language developers can then attempt to manually refactor the grammar to address those errors21. As an example, consider a grammar for prop- |  |
| erty or field access, expressions of the form **customer.name** or | . |
| **"Tim".length**22: |  |

|  |
| --- |
| **Exp** ::= **ID**  | **STRING**  | **Exp** "." **ID** |

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. Leftrecursion is not supported by LL parsers such as ANTLR.

The left-recursion can be removed by *left-factoring* the grammar, i.e. 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 recursive production rules 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. Our example grammar from above uses recursion for repetition, which can be made explicit as follows:

|  |
| --- |
| **Exp** ::= **ID**  | **STRING**  | **Exp** ("." **ID**)+ |

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

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

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

|  |
| --- |
| **Exp** ::= **FieldPart** ("." **ID**)\*  **FieldPart** ::= **ID**  | **STRING** |

|  |  |
| --- | --- |
| be mapped 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. A naive approach to avoid the restrictions of LL or LR parsers may be to add backtracking, so that if any input doesn’t match a particular production, the parser can go back and try a different production. Unfortunately, this approach risks exponential execution times or non-termination and usually exhibits poor performance.  There are also general parsing algorithms that can *efficiently* |  |
| parse the full class. In particular, generalized LR (GLR) parsers24 and Earley parsers25 can parse in linear time *O*(*n*) in the com- | **\_** |
| mon case. In the case of ambiguities, the time required can increase, but in the worst case they are bounded by cubic *O*(*n*3) time. In practice, most programming languages have few or no |  |

This last grammar describes the same language as the original grammar shown above, but conforms to the LL(1) grammar class23. In the general case, not all context-free grammars can

ambiguities, ensuring good performance with a GLR parser. 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 rules26, corresponding to multiple possible ASTs. This obvi- 26 This also means that this sentence can ously is a problem for parser implementation, as some decision be parsed in more than one way.

has to be made on which AST is preferred. Consider again the expression language introduced above.

|  |
| --- |
| **Exp** ::= **NUM**  | **Exp** "+" **Exp**  | **Exp** "\*" **Exp** |

This grammar is ambiguous, since for a string **1 \* 2 + 3** there are two possible trees (corresponding to different operator precedences).

Exp Exp

Exp

Exp

Exp

Exp

Exp

Exp

Exp

Exp

1 \* 2 + 3 1 \* 2 + 3

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 below.

*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 a grammar that has ambiguities, including our simple expression grammar. Determining whether a grammar is ambiguous is a classic 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 discussed above), but by resolving all conflicts (if possible) an unambiguous grammar can be obtained.

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 **+** 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 **+** operator in the highest layer to give it the lowest priority27, 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 follows28: |  |

**Expr** ::= **Mult** ("+" **Mult**)\*

(Section 7.5). **Mult** ::= **NUM** ("\*" **NUM**)\*

*Ambiguity with Generalized Parsers* Generalized parsers accept grammars regardless of recursion or ambiguity. So our expression grammar is readily accepted as a valid grammar. In the 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. A way of programmatically selecting one alternative is *disambiguation filters*. For example, leftassociativity can be indicated on a per-production basis:

|  |
| --- |
| **Exp** ::= **NUM**  | **Exp** "+" **Exp** {left}  | **Exp** "\*" **Exp** {left} |

This indicates that both operators are left-associative (using the **{left}** annotation from Spoofax). Operator precedence can be indicated with relative priorities or with precedence annotations:

**Exp** ::= **Exp** "\*" **Exp** {left}

>

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

|  |  |
| --- | --- |
| The **>** indicates that the **\*** operator binds stronger than the **+** operator. This kind of declarative disambiguation is commonly found in GLR parsers, but typically is not available in parsers |  |
| that support only more limited grammar classes29. |  |
| *Grammar Evolution and Composition* Grammars evolve as languages change and new features are added. These features can be added by adding single, new productions, or by composing the grammar with an existing grammar. Composition of grammars is an efficient way of reusing grammars and quickly |  |

constructing or extending new grammars. As a basic example of grammar composition, consider once again our simple grammar for arithmetic expressions:

|  |
| --- |
| **Expr** ::= **NUM**  | **Expr** "\*" **Expr**  | **Expr** "+" **Expr** |

Once more operators are added and the proper associativities and precedences are specified, such a grammar forms an excellent unit for reuse30. As an example, suppose we want to

compose this grammar with the grammar for field access expressions31:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Expr** ::= **ID**  | **STRING**  | **Expr** "." **ID** |  |  |  | | --- | --- | |  |  | | In the ideal case, composing two such grammars should be trivial – just copy them into the same grammar definition file. However, reality is often less than ideal. There are a number of challenges that arise in practice, related to ambiguity and to grammar class restrictions32. |  |   31 |

used as guard conditions in state machines, for pre- and postconditions in interface definitions, or to specify

* Composing arbitrary grammars risks introducing ambiguities that did not exist in either of the two constituent grammars. In the case of the arithmetic expressions and field access grammars, care must specifically be taken to indicate the precedence order of all operators with respect to all others. With a general parser, new priority rules can be added without changing the two imported grammars. When an LL or LR parser is used, it is often necessary to change one or both of the composed grammars to eliminate any conflicts. This is because in a general parser, the precedences are *declarative* (additional preference specification can simply be added at the end), whereas in LL or LR parsers the precedence information is encoded in the grammar structure (and hence invasive changes to this structure may be required).
* We have shown how grammars can be massaged with techniques such as left-factoring in order to conform to a certain grammar class. Likewise, any precedence order or associativity can be encoded by massaging the grammar to take a certain form. Unfortunately, all this massaging makes grammars very resistant to change and composition: after two grammars are composed together, the result is often no

32 See Laurence Tratt’s article *Parsing – the solved problem that isn’t.* at **tratt.net/laurie/ tech\_articles/articles/parsing \_the\_solved\_problem\_that\_isnt**.

longer LL or LR, and another manual factorization step is required.

• Another challenge is in composing scanners. When two grammars that depend on a different lexical syntax are composed, conflicts can arise. For example, consider what happens when we compose the grammar of Java with the grammar of SQL:

**for** (Customer c : **SELECT** customer **FROM** accounts **WHERE** balance < 0) {

...

}

The SQL grammar reserves keywords such as **SELECT**, even though they are not reserved in Java. Such a language change could break compatibility with existing Java programs which happen to use a variable named **SELECT**. A common programmatic approach to solve this problem is the introduction of easy-to-recognize boundaries, which trigger switches between different parsers. In general, this problem can only be avoided completely by a scannerless parser, which considers the lexical syntax in the context in which it appears; traditional parsers perform a separate scanning stage in which no context is considered.

### 7.2 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 AST 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 model-view-controller (MVC) pattern specifically. When users edit a UML diagram, they don’t draw pixels onto a canvas, and a "pixel parser" then creates the AST. Rather, the editor creates an instance of **uml.Class** as you drag a class from the palette to the canvas. A projection engine renders the diagram, in this case drawing a rectangle for the class. Projectional editors generalize this approach to work with any notation, including textual.

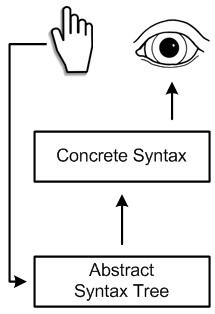


Figure 7.4: In projectional systems, the user sees the concrete syntax, but all editing gestures directly influence the AST. The AST is *not* extracted from the concrete syntax, which means the CS does not have to be parseable.

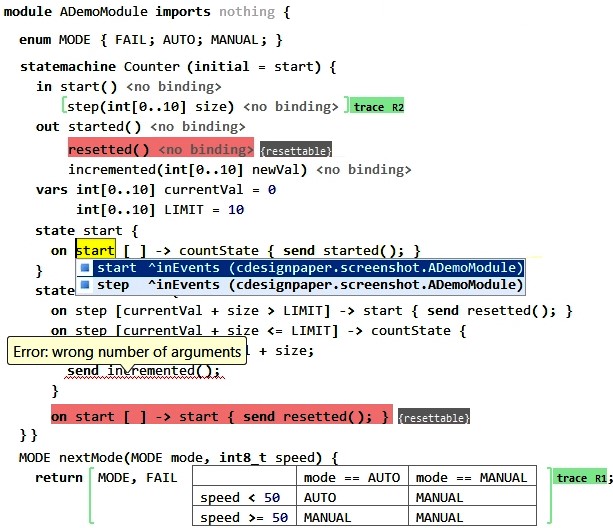
This explicit instantiation of AST objects happens by picking the respective concept from the code completion menu using a character sequence defined by the respective concept (typically the "leading keyword" of the respective program construct, or the name of a referenced variable). If at any given program location two concepts can be instantiated using *the same character sequence*, then the projectional editor prompts the user to decide33. Once a concept is instantiated, it is stored in the AST as

|  |  |
| --- | --- |
| a node with a unique ID (UID). References between program elements are pointers to this UID, and the projected syntax that represents the reference can be arbitrary. The AST is actually an abstract syntax graph *from the start* because cross-references |  |
| are first-class rather than being resolved after parsing34. The |  |
| program is stored using a generic tree persistence mechanism, often XML35. |  |
| Defining a projectional editor, instead of defining a grammar, involves the definition of projection rules that map language concepts to a notation. It also involves the definition of event handlers that modify the AST based on a user’s editing gestures. The way to define the projection rules and the event handlers is specific to the particular tool used.  The projectional approach can deal with arbitrary syntactic forms including traditional text, symbols (as in mathematics), tables or graphics. Since no grammar is used, grammar classes are not relevant here. In principle, projectional editing is simpler in principle 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 convenient36. Mod- |  |
| ern projectional editors, and in particular MPS, do a good job in meeting this challenge. |  |

### 7.3 Comparing Parsing and Projection

#### 7.3.1 Editing Experience

In free text editing, any regular text editor will do. However, users expect a 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 IDE support that is essentially similar to what a modern IDE provides for mainstream languages (e.g. Eclipse for Java)[[4]](#footnote-4). However, you can always go back to any text editor to edit the programs.



|  |  |
| --- | --- |
| In projectional editing, this is different: a normal text editor is obviously not sufficient; a specialized editor has to be supplied to perform the projection (an example is shown in Fig. 7.5). As in free text editing, it has to provide the IDE support features mentioned above. MPS provides those. However, there is another challenge: for textual-looking notations, it is important that the editor tries to make the editing experience as text-like as possible, i.e. the keyboard actions we have become used to from free-text editing should work as far as possible. MPS does a decent job here, using, among others, the |  |
| following strategies38: |  |
| • 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 instantiated39. |  |

* *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

the leading keyword (e.g. **if** for an **IfStatement**), users can "just type" the code.

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 manually inserting a **+**, users can simply type **+** on the right side of the **2**. The system performs the tree restructuring 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, so the user can enter the second argument. This means that expressions (or anything else) can be entered linearly, as expected. For this to work, operator precedence has to be specified, and the tree has to be constructed taking these precedences into account. Precedence is typically specified by a number associated with each operator, and whenever a side transformation is used to build an expression, the tree is automatically reshuffled to make sure that those operators with a higher precedence number are further down in the tree.

* 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 at a given location. Consider again a local variable declaration **int a;**. The respective concept could be **LocalVariableDeclaration**, a subconcept 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 **Type**s where **LocalVariableDeclaration**s are expected. Once a **Type** is selected, the wrapper implementation creates a **LocalVariableDeclaration**, puts the **Type** into its **type** field, and moves the cursor into the **name** slot. Summing up, this means that a local variable declaration **int a;** can be entered by starting to type the **int** type, as expected.
* 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 is then 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, then puts those targets 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 terms of the editing experience.

* Smart delimiters are used to simplify inputting list-like data, where elements are separated with a specific separator symbol. An example is argument lists in functions: once a parameter is entered, users can press comma, i.e. the list delimiter, to instantiate the next element.

Except for having to get used to the somewhat different way of editing programs, the strategies mentioned above (plus some others) result in a reasonably good editing experience. Traditionally, projectional editors have *not* used these or similar strategies, and projectional editors have acquired a bit of a bad reputation because of that. In the case of MPS this tool support is available, and hence MPS provides a productive and pleasant working environment.

#### 7.3.2 Language Modularity

As we have seen in Section 4.6, 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.

In parser-based systems the extent to which language composition can be supported depends on the supported grammar class. As we have said above, the problem is that the result of 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 composite grammar. This of course breaks modularity and hence is not an option. Parsers that do not support the full set of context-free grammars, such as ANTLR, and hence Xtext, have this problem. Parsers that do support the full set of context-free grammars, such as the GLR parser used as part of Spoofax, are better off. 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 externally, *without* invasively changing either the composed or the component grammars, retaining modularity.

|  |  |
| --- | --- |
| ing, no grammar classes, and hence no problem with composed grammars becoming ambiguous. Any combination of languages will be syntactically valid. In cases where a composed language would be ambiguous in a GLR-based system, the user has to make a disambiguating decision *as the program is entered*. For example, in MPS, if at a given location two language concepts are available with 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.  *7.3.3 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 characters approach. For example, the Fortress programming language41 makes use of this: Greek letters and a wide variety of different bracket styles can be used in Fortress programs. However, character layout is always ignored. For example it is not possible to use parsers to handle tabular notations, fraction bars or even graphics42. |  |

In projectional editors, language modularity and composition is not a problem at all40. There is no grammar, no pars-

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 and notations other than text can be used easily. For example, MPS supports tables, fraction bars and "big math" symbols[[5]](#footnote-5). Since these non-textual notations are handled in the same way as the textual ones (possibly with other input gestures), they can be mixed easily[[6]](#footnote-6): tables can be embedded into textual source, and textual languages can be used within table cells (see Fig. 7.6).

#### 7.3.4 Language Evolution

If the language changes, existing instance models temporarily become outdated, in the sense that they were developed for the

uses this approach.

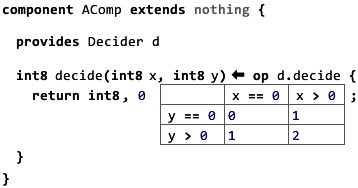


Figure 7.6: A table embedded in an otherwise textual program

old version of the language. If the new language is not backward compatible, these existing models have to be migrated to conform to the updated language.

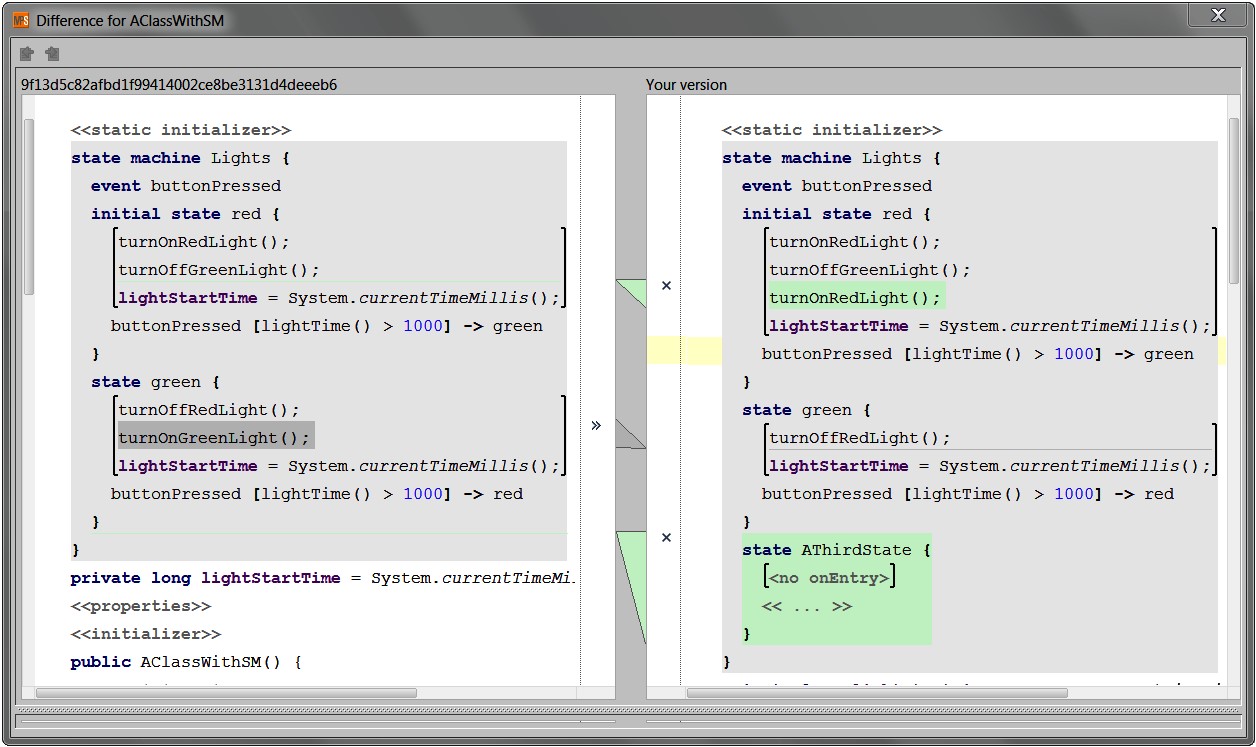
Since projectional editors store the models as structured data in which each program node points to the language concept it is an instance of, the tools have to take special care that such "incompatible" models can still be opened and then migrated, manually or by a script, to the new version of the language. MPS supports this feature, and it is also possible to distribute migration scripts with (updated) languages to run the migration automatically45.

|  |  |
| --- | --- |
| Most textual IDEs do not come with explicit support for evolving programs as languages change. However, since a model is essentially a sequence of characters, it can *always* be opened in the editor. The program may not be parseable, but users can always update the program manually, or with global search and replace using regular expressions. More complex migrations may require explicit support via transformations on the AST.  *7.3.5 Infrastructure Integration*  Today’s software development infrastructure is typically textoriented. Many tools used for diff and merge, or tools like **grep** and regular expressions, are geared towards textual storage. Programs written with parser-based textual DSLs (stored as plain text) integrate automatically and nicely with these tools.  In projectional IDEs, special support needs to be provided for infrastructure integration. Since the CS is not pure text, a generic persistence format is used, typically based on XML. While XML is technically text as well, it is not practical to perform diff, merge and the like on the level of the XML. Therefore, special tools need to be provided for diff and merge. MPS provides integration with the usual version control systems and handles diff and merge in the IDE, using the con- |  |
| crete, projected syntax46. Fig. 7.7 shows an example of an MPS |  |
| diff. However, it clearly is a drawback of projectional editing (and the associated abstract syntax-based storage) that many well-known text utilities don’t work[[7]](#footnote-7). |  |

Also, copy and paste with textual environments may be a challenge. MPS, for example, supports pasting a projected program that has a textual-looking syntax into a text editor. However, for the way back (from a textual environment to the pro-

jectional editor), there is no automatic support. However, special support for specific languages can be provided via *paste handlers*. Such a paste handler is available for Java, for example: when a user pastes Java text into a Java program in MPS, a parser is executed that builds the respective MPS tree48.

|  |  |
| --- | --- |
|  |  |
| *7.3.6 Tool Lock-in*  In the worst case, textual programs can be edited with any text editor. Unless you are prepared to edit XML, programs expressed with a projectional editor *always* require that editor to edit programs. As soon as you take IDE support into account though, both approaches lock users into a particular tool. Also, there is essentially no standard for exchanging language def- |  |
| initions between the various language workbenches49. So the |  |

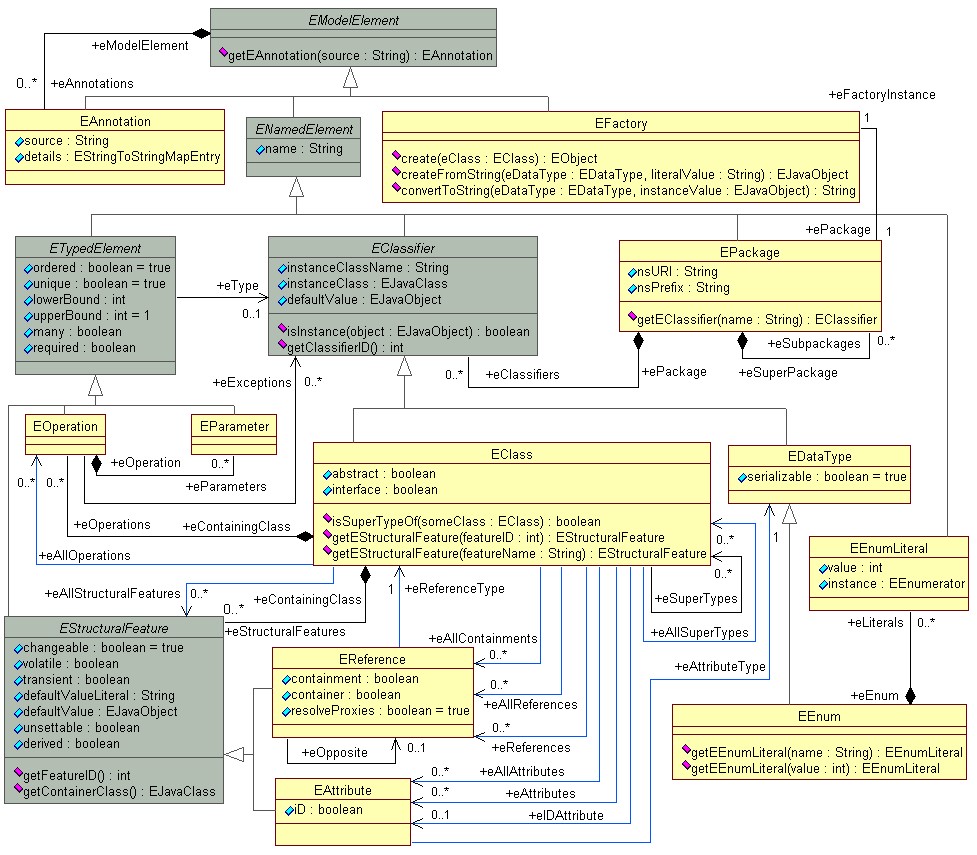
.

effort of implementing a language is always lost if the tool must be changed.

#### 7.3.7 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 of 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 required50. |  |
| *7.4 Characteristics of AST Formalisms*  Most AST formalisms, aka meta meta models51, are ways to |  |
| represent trees or graphs. Usually, such an AST formalism is "meta circular" in the sense that it can describe itself.  This section is a brief overview over the three AST formalisms relevant to Xtext, Spoofax and MPS. We will illustrate them in more detail in the respective tool example sections.  *7.4.1 EMF Ecore* |  |
| The Eclipse Modeling Framework52 (EMF) is at the core of all Eclipse Modeling tools. It provides a wide variety of services and tools for persisting, editing and processing models and abstract syntax definitions. EMF has grown to be a fairly large ecosystem within the Eclipse community and numerous projects use EMF as their basis.  Its core component is Ecore, a variant of the EMOF stan- |  |
| dard53. Ecore acts as EMF’s meta meta model. Xtext uses Ecore |  |
| as the foundation for the AS: from a grammar definition, Xtext derives the AS as an instance of Ecore. Ecore’s central concepts are: **EClass** (representing AS elements or language concepts), **EAttribute** (representing primitive properties of **EClass**es), **EReference** (representing associations between **EClass**es) and **EObject** (representing instances of **EClass**es, i.e. AST nodes). **EReferences** can have containment semantics or not and each **EObject** can be contained by at most one **EReference** instance. Fig. 7.8 shows a class diagram of Ecore.  When working with EMF, the Ecore file plays a central role. From it, all kinds of other aspects are derived; specifically, a generic tree editor and a generated Java API for accessing an AST. It also forms the basis for Xtext’s model processing: The  Ecore file is derived from the grammar, and the parser, when |  |



#### 7.4.2 Spoofax’ ATerm

Spoofax uses the ATerm format to represent abstract syntax. ATerm provides a generic tree structure representation format that can be serialized textually similar to XML or JSON. Each tree node is called an ATerm, or simply a *term*. Terms consist of the following elements: Strings (**"Mr. White"**), Numbers (**15**), Lists (**[1,2,3]**) and constructor applications (**Order(5, 15, "Mr. White")** for labeled tree nodes with a fixed number of children.

The structure of valid ATerms is specified by an algebraic *signature*. Signatures are typically generated from the concrete-

|  |  |  |  |
| --- | --- | --- | --- |
| syntax definition, but can also be specified manually. A signature introduces one or more algebraic *sorts*, i.e. collections | | |  |
| of terms. The sorts **String**, **Int**, and **List**54 are predefined. | | |  |
| User-defined sorts are inhabited by declaring term *constructors* and *injections*. A constructor has a name and zero or more subterms. It is declared by stating its name, the list of sorts of its direct subterms, and the sort of the constructed term. Constructor names may be overloaded. Injections are declared as nameless constructors. The following example shows a signature for expressions: | | |  |
| **signature**  **sorts**  Exp  **constructors**  Plus : Exp \* Exp -> Exp  Times: Exp \* Exp -> Exp  : Int -> Exp |

The signature declares sort **Exp** with its constructors **Plus** and **Times**, which both require two expressions as direct subterms. Basic expressions are integers, as declared by the injection rule

**: Int -> Exp**.

Compared to XML or JSON, perhaps the most significant distinction is that ATerms rely on the order of subterms rather than on labels. For example, a product may be modeled in JSON as follows:

|  |
| --- |
| {  "product": {  "itemnumber": 5,  "quantity": 15,  "customer": "Mr. White"  }  } |

Note how this specification includes the actual data describing the particular product (the model), but also a description of each of the elements (the meta model). With XML, a product would be modeled in a similar fashion. An equivalent of the JSON above written in ATerm format would be the following:

Order([ItemNumber(5), Quantity(15), Customer("Mr.\ White")])

However, this representation contains a lot of redundant information that also exists in the grammar. Instead, such a product can be written as **Order(5, 15, "Mr. White")**. This more concise notation tends to make it slightly more convenient to use in handwritten transformations.

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 tool-specific way (i.e. simple Java objects in the case of Spoofax)55.

In addition to the basic elements above, ATerms support annotations to add additional information to terms. These are similar to attributes in XML. For example, it is possible to annotate a product number with its product name:

Order(5{ProductName("Apples")}, 15, "Mr. White")

|  |  |
| --- | --- |
| Spoofax also uses annotations to add information about references to other parts of a model to an abstract syntax tree. While ATerms only form trees, the annotations are used to represent the graph-like references.  *7.4.3 MPS’ Structure Definition*  In MPS, programs are trees/graphs of *nodes*. A node is an instance of a *concept* which defines the structure, syntax, type |  |
| system and semantics of its instance nodes56. Like **EClass**es57, |  |
| concepts are meta circular, i.e. there is a concept that defines the properties of concepts: |  |

|  |
| --- |
| **concept** ConceptDeclaration **extends** AbstractConceptDeclaration **implements** INamedConcept  **instance can be root**: **false**  **properties**:  helpURL : **string** rootable : **boolean**  **children**:  InterfaceConceptReference implementsInterfaces 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 extendsConcept 0..1 |

|  |  |
| --- | --- |
| A concept may extend a single other concept and implement |  |
| any number of interfaces58. It can declare references (non- |  |
| containing) and children (containing). It may also have a number of primitive-type properties as well as a couple of "static" features. In addition, concepts can have behavior methods. |  |

While the MPS structure definition is proprietary to MPS and does not implement any accepted industry standard, it is conceptually very close to Ecore[[8]](#footnote-8).

### 7.5 Xtext Example

Cooling programs60 represent the behavioral aspect of the re-

frigerator descriptions. Here is a trivial program that can be used to illustrate some of the features of the language. The program is basically a state machine.

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **cooling program** HelloWorld **uses** stdlib {  **var** v: **int event** e  **init** { **set** v = 1 }  **start**:  **on** e { **state** s }  **state** s:  **entry** { **set** v = 0 }  } |   The program declares a variable **v** and an event **e**. When the program starts, 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 transitions to the state **s**, where it uses an entry action to set **v** back to **0**. More complex programs include checks of changes of properties of hardware elements (**aCompartment->currentTemp**) and commands to the hardware (**set aCompartment->isCooling = true**), as shown in the next snippet:   |  | | --- | | **start**:  **check** ( aCompartment->currentTemp > maxTemp ) { **set** aCompartment->isCooling = **true state** initialCooling  }  **check** ( aCompartment->currentTemp <= maxTemp ) { **state** normalCooling }  **state** initialCooling:  **check** ( aCompartment->currentTemp < maxTemp ) { **state** normalCooling  } | |

section refer back to the case studies introduced at the beginning of the book in Section 1.11.

EBNF-like notation, a collection of productions that are typically called *parser rules*. These rules specify the concrete syntax of a program element, as well as its mapping to the AS. From the grammar, Xtext generate the abstract syntax represented in Ecore[[9]](#footnote-9). 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** in the example above), 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 (not shown). **ID** 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:

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

In pure grammar languages, one would typically write the following:

"cooling" "program" **ID** "\{ ..."}

This expresses the fact that after the two keywords we expect an **ID**. However, 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: the name of the rule will be the name of the derived meta class62. So we will get a meta class **CoolingProgram**. The

second mapping we have encountered is **name=ID**. It specifies that the meta class gets a property **name** that holds the contents of the **ID** from the parsed program text. Since nothing else is specified in the **ID** terminal rule, the type of this property defaults to **EString**, Ecore’s version 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 **\*** specifies unbounded multiplicity), an optional init block (optionality is specified by the **?**) 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 **CoolingProgram** 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 will be typed to be a *list* of **State**s. In the case of the optional **init** block, the meta class will have an **initBlock** property, typed as **InitBlock** (whose parser rule we don’t show 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 the fact that whatever is inside the parentheses can occur any |  |
| number of times63. Inside the parentheses we expect either |  |
| a **CustomEvent** *or* a **Variable**, which is expressed with the **|**. 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. |  |

(events+=CustomEvent)\*

(variables+=Variable)\*

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

|  |
| --- |
| State:  BackgroundState | StartState | CustomState; |

|  |
| --- |
| Statement:  Statement | AssignmentStatement | PerformAsyncStatement | ChangeStateStatement | AssertStatement;  ChangeStateStatement:  "state" targetState=[State]; |

The vertical bar operator is used here to express syntactic alternatives. This is translated to inheritance in the meta model. The definition of **CustomState** is shown in the following code snippet:

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

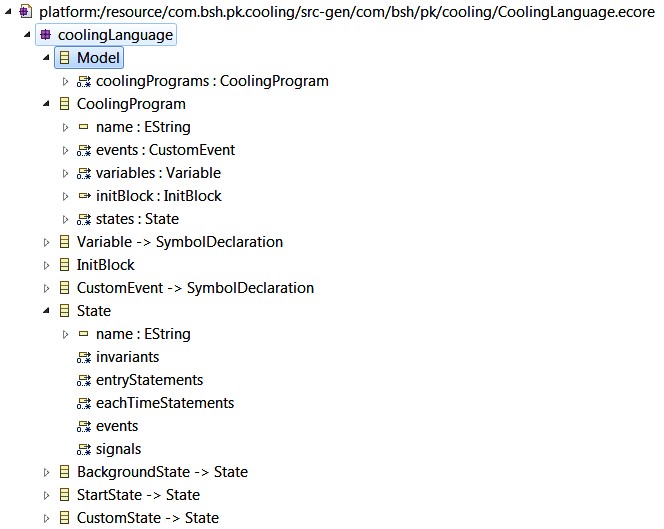
**StartState** and **BackgroundState**, the other two subtypes of **State**, share some properties. Consequently, Xtext’s AS derivation algorithm pulls them up into the abstract **State** meta class. This way they can be accessed polymorphically. Fig. 7.9 shows the resulting meta model using EMF’s tree view.

*References* Let us now look at statements and expressions. **State**s have entry and exit actions which are procedural statements that are executed when a state is entered and left, respectively. The **set v = 1** in the example program above is an example. **Statement** itself is **abstract** and has the various kinds of statements as subtypes/alternatives:

The **ChangeStateStatement** is used to transition into another state. It uses the keyword **state** followed by a reference to the target state. Notice how Xtext uses square brackets to express the fact that the **targetState** property points to an *existing* state, as opposed to containing a new one (which would be written as **targetState=State**); i.e. the square brackets express non-containing cross-references.

|  |
| --- |
| AssignmentStatement:  "set" left=Expr "=" right=Expr; |

This is another example of where the Xtext grammar language goes beyond classical grammar languages, where one would write **"state" targetStateName=ID;**. Writing it in this way only specifies that we expect an **ID** after the **state** keyword. The fact that we call it **target- StateName** communicates *to the programmer* that we expect this text string to correspond to the name of a state – a later phase in model processing *resolves* the name to an actual state reference. Typically, the code to resolve the reference has to be written manually, because there is no way for the tool to derive from the grammar automatically the fact that this **ID** is actually a reference to a **State**. In Xtext, the **targetState=[State]** notation makes this explicit, so the resolution of the reference can be automatic. This approach also has the advantage that the resulting meta class types the **targetState** property to **State** (and not just to a string), which makes processing the models much easier.



.

Note that the cross-reference definition only specifies the target type (**State**) of the cross-reference, but not the concrete syntax of the reference itself. By default, the **ID** terminal is used for the reference syntax, i.e. a simple (identifier-like) text string is expected. However, this can be overridden by specifying the concrete syntax terminal behind a vertical bar in the reference64. In the following piece of code, the **targetState** reference uses the **QID** terminal as the reference syntax.

|  |
| --- |
| ChangeStateStatement:  "state" targetState=[State|QID];  QID: **ID** ("." **ID**)\*; |

|  |
| --- |
| 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: ({TrueLiteral} "true") |  ({FalseLiteral} "false") |  ({ParenExpr} "(" expr=Expr ")") |  ({NumberLiteral} value=DECIMAL\_NUMBER) |  ({SymbolRef} symbol=[SymbolDeclaration|QID]); |

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

*Expressions* The **AssignmentStatement** shown earlier is one of the statements that uses expressions. We repeat it here:

|  |
| --- |
| AssignmentStatement:  "set" left=Expr "=" right=Expr; |

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).

To understand the above definition, we first have to explain in more detail how 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 lazily, upon the first assignment to one of its properties. If no assignment is performed at all, no object is created. For example, in the grammar rule **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";**65.

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 of another AST object.

Now we know enough about AST construction to understand how expressions are encoded and parsed. In the expression grammar above, 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:

* The **AssignmentStatement** refers to the **Expr** rule in its **left** and **right** properties, so we "enter" the expression tree at the level of **Expr** (which is the root of the expression hierarchy).
* The **Expr** rule just calls the **ComparisonLevel** rule, which calls **AdditionLevel**, and so on. No objects are created at this point, since no assignment to any property is performed.
* The parser "dives down" until it finds something that matches the first symbol in the parsed text: the **2**. This occurs on **AtomicLevel** when it matches the **DECIMAL\_NUMBER** terminal. At this point it creates an instance of the **NumberLiteral** meta class and assigns the number **2** to the **value** property. It also sets the **current** object to point to the just-created **NumberLiteral**, since this is now the AST object created most recently.
* The **AtomicLevel** rule ends, and the stack is unwound. We’re back at **PrefixOpLevel**, in the second branch. Since nothing

else is specified after the call to **AtomicLevel**, we unwind once more.

* We’re now back at the **MultiplicationLevel**. The rule is not finished yet and we try to match an **\*** and a **/**. The match on **\*** succeeds. At this point the *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** created before) to its **left** property. Then it makes the newly created **Multi** the new **current**. At this point we have a subtree with the **\*** at the root, and the **NumberLiteral** in the **left** property.
* The rule hasn’t ended yet. We dive down to **PrefixOpLevel** and **AtomicLevel** once more, matching the **3** in the same way as the **2** before. The **NumberLiteral** for **3** is assigned to the **right** property as we unwind the stack.
* At this point we unwind the stack further, and since no more text is present, no more objects are created. The tree structure has been constructed as 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**, so it has to end up "higher" in the tree). Once we’re at **4 +**, 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 precedence 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 (see the **ParenExpr** in the **AtomicLevel**): inside those, we can again embed an **Expr**, i.e. we jump back to the lowest precedence level66.

Once you understand the basic approach, it is easy to add new expressions with a precedence similar to another one (just add it as an alternative to the respective **Level** rule) or to introduce a new precedence level (just interject a new **Level** rule between two existing ones)[[10]](#footnote-10).

### 7.6 Spoofax Example

Mobl’s68 data modeling language provides entities, properties

and functions. To illustrate the language, below are two data type definitions related to a shopping list app. It supports lists of items that can be favorited, checked, and so on, and which are associated with some **Store**.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **module** shopping  **entity** Item {  name : String checked : Bool favorite : Bool onlist : Bool order : Num store : Store  } | | In Mobl, most files start with a module header, which can be followed by a list of entity type definitions. In turn, each entity can have one or more property or function definitions (shown in the next example snippet). Modules group entities. Inside a module, one can only access entities from the same module or from imported modules.  *Grammar Basics* In Spoofax, the syntax of languages is de- | | |  | | scribed using SDF69. SDF is short for Syntax Definition Formalism, and is a modular and flexible syntax definition formalism | | |  | | that is supported by the SGLR70 parser generator. It can gener- | | |  | | ate efficient, Java-based scannerless and general parsers, allowing Spoofax to support the full class of context-free grammars, grammar composition, and modular syntax definitions. An example of a production written in SDF is: | | |  |   "**module**" ID Entity\* -> Start {"Module"}   |  |  | | --- | --- | | The pattern on the left-hand side of the arrow is matched by the |  | | symbol **Start** on the right-hand side71. After the right-hand |  | |

tions for mobile devices. It is based on HTML 5 and is closely related to WebDSL, which has been introduced earlier (Section 1.11).

side, SDF productions may specify annotations using curly brackets. Most productions specify a quoted *constructor label* that is used for the abstract syntax. This particular production creates a tree node with the constructor **Module** and two children that represent the **ID** and the list of **Entities** respectively. As discussed earlier, Spoofax represents abstract syntax trees as ATerms. Thus, the tree node will be represented as

**Module(..., [...])**. In contrast to Xtext, the children are not named; instead, they are identified via the position in the child collection (the **ID** is first, the **Entity** list is second). Spoofax generates the following signature from the production above:

order for productions as the grammars we’ve discussed so far, switching the left-hand and right-hand side.

|  |
| --- |
| The left-hand side of an SDF production is the pattern it matches against. SDF supports symbols, literals and character classes in this pattern. Symbols are references to other productions, such as **ID**. Literals are quoted strings such as **"module"** that must appear in the input literally. Character classes specify a range of characters expected in the input, e.g. **[A-Za-z]** specifies that an alphabetic character is expected. We discuss character classes in more detail below.  The basic elements of SDF productions can be combined using operators. The **A\*** operator shown above specifies that zero or more occurrences of **A** are expected. **A+** specifies that one or more are expected. **A?** specifies that zero or one are expected. **{A B}\*** specifies that zero or more **A** symbols, separated by **B** symbols, are expected. As an example, **{ID ","}\*** is a commaseparated list of identifiers. **{A B}+** specifies one or more **A** symbols separated by **B** symbols.  Fig. 7.10 shows an SDF grammar for a subset of Mobl’s entities and functions syntax. The productions in this grammar should have few surprises, but it is interesting to note how SDF groups a grammar in different sections. First, the **context-free start symbols** section indicates the start symbol of the grammar. Then, the **context-free syntax** section lists the context-free syntax productions, forming the main part of the grammar. Terminals are defined in the **lexical syntax** section.  *Lexical Syntax* As Spoofax uses a scannerless parser, all lexical syntax can be customized in the SDF grammar[[11]](#footnote-11). Most |

|  |
| --- |
| **signature sorts**  Start **constructors**  Module: ID List(Entity) -> Start |

lexical syntax is specified using character classes such as **[0-9]**. Each character class is enclosed in square brackets, and can consist of ranges of characters (**c**1**-c**2), letters and digits (e.g. **x** or **4**), non-alphabetic literal characters (e.g., **\_**), and escapes (e.g., \**n**). A complement of a character class can be obtained using the ∼ operator, e.g. ∼**[A-Za-z]** matches all non-alphabetic characters. For whitespace and comments a special terminal

**LAYOUT** can be used.

SDF implicitly inserts **LAYOUT** between all symbols in contextfree productions. This behavior is the key distinguishing fea-

|  |
| --- |
| **module** MoblEntities **context**-**free start symbols**  Module **context**-**free syntax**  "**module**" ID Decl\* -> Module {"Module"}  "import" ID -> Decl {"Import"}  "entity" ID "{" EntityBodyDecl\* "}" -> Decl {"Entity"}  ID ":" Type -> EntityBodyDecl {"Property"}  "function" ID "(" {Param ","}\* ")" ":" ID "{" Statement\* "}"  -> EntityBodyDecl {"Function"}  ID ":" Type -> Param {"Param"}  ID -> Type {"EntityType"}  "var" ID "=" Expr ";" -> Statement {"Declare"} "return" Exp ";" -> Statement {"Return"}  Exp "." ID "(" Exp ")" -> Exp {"MethodCall"}  Exp "." ID -> Exp {"FieldAccess"}  Exp "+" Exp -> Exp {"Plus"}  Exp "\*" Exp -> Exp {"Mul"}  ID -> Exp {"Var"}  INT -> Exp {"Int"}  **lexical syntax**  [A-Za-z][A-Za-z0-9]\* -> ID  [0-9]+ -> INT  [\ \t\n] -> LAYOUT |

ture between context-free and lexical productions: lexical symbols such as identifiers and integer literals cannot be interleaved with layout. The second distinguishing feature is that lexical syntax productions usually do not have a constructor label in the abstract syntax, as they form terminals in the abstract syntax trees (i.e. they don’t own any child nodes).

*Abstract Syntax* To produce abstract syntax trees, Spoofax uses the ATerm format, described in Section 7.4.2. SDF combines the specification of concrete and abstract syntax, primarily through the specification of constructor labels. Spoofax allows users to view the abstract syntax of any input file. As an example, the following is the textual representation of an abridged abstract syntax term for the shopping module shown at the beginning of this section:

|  |
| --- |
| Module(  "shopping", [ Entity(  "Item",  [Property("name", EntityType("String")), Property("checked",  EntityType("Bool")), ...] ) ]  ]) |

Note how this term uses the constructor labels of the syntax above: **Module**, **Entity** and **Property**. The children of each node correspond to the symbols referenced in the production: the **Module** production first referenced **ID** symbol for the module name and then included a list of **Decl** symbols (lists are in square brackets).

In addition to constructor labels, productions that specify parentheses can use the special **bracket** annotation:

"(" Exp ")" -> Exp {bracket}

The **bracket** annotation specifies that there should not be a separate tree node in the abstract syntax for the production. This means that an expression **1 + (2)** would produce **Plus ("1","2")** in the AST, and not **Plus("1",Parens("2"))**.

*Precedence and Associativity* SDF provides special support for specifying the associativity and precedence of operators or other syntactic constructs. As an example, let us consider the production of the **Plus** operator. So far, it has been defined as:

Exp "+" Exp -> Exp {"Plus"}

Based on this operator, a parser can be generated that can parse an expression such as **1 + 2** to a term **Plus("1", "2")**. However, the production does not specify if an expression **1 + 2 + 3** should be parsed to a term **Plus("1", Plus("2", "3"))** or **Plus(Plus("1", "2"), "3")**. If you try the grammar in Spoofax, it will show *both* interpretations using the special **amb** constructor:

|  |
| --- |
| amb([  Plus("1", Plus("2", "3")),  Plus(Plus("1", "2"), "3") ]) |
| The **amb** node indicates an *ambiguity* and it contains all possi- | | |  |
| ble interpretations73. Ambiguities can be resolved by adding | | |  |
| annotations to the grammar that describe the intended interpretation. For the **Plus** operator, we can resolve the ambiguity by specifying that it is left-associative, using the **left** annotation: | | |  |

Exp "+" Exp -> Exp {"Plus", **left**}

In a similar fashion, SDF supports the definition of the precedence order of operators. For this, the productions can be placed into the **context-free priorities** section:

**context**-**free priorities**

|  |
| --- |
| Exp "\*" Exp -> Exp {"Mul", **left**}  >  Exp "+" Exp -> Exp {"Plus", **left**} |

This example specifies that the **Mul** operator has a higher priority than the **Plus** operator, resolving the ambiguity that arises for an expression such as **1 + 2 \* 3**.

*Reserved Keywords and Production Preference* Parsers generated with SDF do not use a scanner, but include processing of lexical syntax in the parser. Since scanners operate without any context information, they will simply recognize any token that corresponds to a keyword in the grammar as a reserved keyword, *irrespective of its location in the program*. In SDF, it is also possible to use keywords that are not reserved, or keywords that are only reserved in a certain context. As an example, the following is a legal entity in Mobl:

**entity entity** {}

Since our grammar did not specify that **entity** is a reserved word, it can be used as a normal **ID** identifier. However, there are cases in which it is useful to reserve keywords, for example to prevent ambiguities. Consider what would happen if we added new productions for predefined type literals:

"Bool" -> Type {"BoolType"}

"Num" -> Type {"NumType"}

"String" -> Type {"StringType"}

If we were now to parse a type **String**, it would be ambiguous: it matches the **StringType** production above, but it also matches the **EntityType** production, as **String** is a legal entity identifier[[12]](#footnote-12). Keywords can be reserved in SDF by using a

production that rejects a specific interpretation:

"String" -> ID {**reject**}

This expresses that **String** can never be interpreted as an identifier. Alternatively, we can say that we prefer one interpretation over the other:

"String" -> Type {"StringType", **prefer**}

This means that this production is to be preferred if there are any other interpretations. However, since these interpretations cannot always be foreseen as grammars are extended, it is considered good practice to use the more specific **reject** approach instead[[13]](#footnote-13).

**lexical restrictions**

ID -/- [A-Za-z0-9]

|  |  |
| --- | --- |
| *Longest Match* Most scanners apply a *longest match* policy |  |
| for scanning tokens76. For most languages, this is the expected |  |
| behavior, but in some cases longest match is not what users expect. SDF instead allows the grammar to specify the intended behavior. In Spoofax, the default is specified in the *Common* syntax module using a **lexical restrictions** section: |  |

This section restricts the grammar by specifying that any **ID** cannot be directly followed by a character that matches **[A-Z a-z0-9]**. Effectively, it enforces a longest match policy for the **ID** symbol. SDF also allows the use of lexical restrictions for keywords. By default it does not enforce longest match, which means it allows the following definition of a Mobl entity:

entityMoblEntity {}

As there is no longest match, the parser can recognize the **entity** keyword even if it is not followed by a space. To avoid this behavior, we can specify a longest match policy for the **entity** keyword:

|  |
| --- |
| **lexical restrictions**  "entity" -/- [A-Za-z0-9] |
| *Name Bindings* So far we have discussed purely syntax specification in SDF. Spoofax also allows the specification of name binding rules, which specify semantic relations between productions. We discuss how these relations are specified in Chapter 8.  *7.7 MPS Example*  We start by defining a simple language for state machines, roughly similar to the one used in the state machine exten- |  |
| sion77 to mbeddr C. Its core concepts include **StateMachine**, |  |
| **State**, **Transition** and **Trigger**. The language supports the definition of state machines, as shown in the following piece of code: |  |
| **module** LineFollowerStatemachine {  **statemachine** LineFollower { **events** unblocked()  blocked() bumped() initialized()  **states** (**initial** = initializing) { |
| **state** initializing {  **on** initialized [ ] -> running { }  } **state** paused {  **on** unblocked [ ] -> running { }  } **state** running {  **on** blocked [ ] -> paused { } **on** bumped [ ] -> crashed { }  } **state** crashed {  }  }  }  } |

*Concept Definition* MPS is projectional, so we start with the definition of the AS. The code below shows the definition of the concept **Statemachine**. It contains a collection of **State**s and a collection of **InEvent**s. It also contains a reference to one of the states to mark it as the **initial** state. The **alias** is defined as **statemachine**, so typing this string inside a C module instantiates a state machine (it picks the **Statemachine** concept from the code completion menu). State machines also implement a couple of interfaces: **IIdentifierNamedElement** contributes a property **name**, **IModuleContent** makes the state

machine embeddable in C **Module**s78.

|  |
| --- |
| **concept** Statemachine **extends** BaseConcept **implements** IModuleContent  ILocalVarScopeProvider  IIdentifierNamedElement  **children**:  State states 0..n  InEvent inEvents 0..n  **references**:  State initial 1  **concept properties**: alias = statemachine |

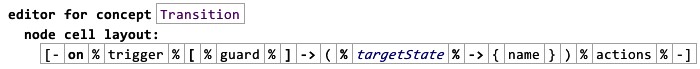
A **State** (not shown) contains two **StatementLists** as **entryActions** and **exitActions**. **StatementList** is a concept defined by the **com. mbeddr.core.statements** language. To make it available visible, our statemachine language extends **com.mbeddr.core.statements**. Finally, a **State** contains a collection of **Transition**s.

|  |
| --- |
| **concept** Transition **children**:  Trigger trigger 1 Expression guard 1  StatementList actions 1  **references**:  State target 1  **concept properties**:  alias = on |

***Transition****s contain a* ***Trigger****, a guard condition, transition actions and a reference to the target state. The trigger is an abstract concept; various specializations are possible: the default implementation is the* ***EventTrigger****, which references an* ***Event****79. The guard condition is an* ***Expression****, a concept reused from* ***com.mbeddr.core.expressions****80. The tar-*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| get state is a reference, i.e. we point to an existing state instead of owning a new one. **action** is another **StatementList** that can contain arbitrary C statements used as the transition actions.   |  |  | | --- | --- | | *Editor Definition* Editors, i.e. the projection rules, are made of cells. When defining editors, various cell types are arranged so that the resulting syntax has the desired structure. Fig. 7.11 shows the editor definition for the **State** concept. It uses an **indent** collection of cells with various style attributes to arrange the **state** keyword and name, the entry actions, the transitions and the exit actions in a vertical list. Entry and exit actions are shown only if the respective **StatementList** is not empty (a condition is attached to the respective cells, marked |  | | by the **?** in front of the cell). An intention81 is used to add a |  | | new statement and hence make the respective list visible. |  |   . |

It arranges the keyword **on**, the trigger, the guard condition, target state and the actions in a horizontal list of cells, the guard surrounded by brackets, and an arrow (**->**) in front of the target state. The editor for the **actionsStatementList** comes with its own set of curly braces.



The **%targetState% -> {name}** part expresses the fact that in order to render the target state, the target state’s **name** attribute should be shown. We could use any text string to refer to the target state82.

|  |
| --- |
| **on** |

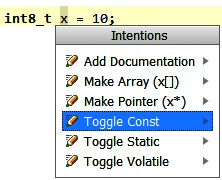
|  |  |
| --- | --- |
| Note how we use both as the leading keyword for a transition and as the alias. This way, if a user types the **on** alias to instantiate a transition, it feels as if they typed the leading keyword of a transition (as in a regular text editor).  If a language extension defined a new concept **SpecialTransition**, they could use another alias to uniquely identify this concept in the code completion menu. The user decides which alias to type depending on whether they want to instantiate a **Transition** or a **SpecialTransition**. Alternatively, the **SpecialTransition** could use *the same alias* **on**. In this case, if the user types **on**, the code completion menu pops open and the user has to decide which of the two concepts to instanti- |  |
| ate83. As we have discussed above, this means that there is |  |
| never an ambiguity that cannot be handled – as long as the user is willing and able to make the decision of which concept should be instantiated. A third option would transform a **Transition** into a **SpecialTransition** on demand, for exam- |  |

82 We could even use the symbol **X** to render *all* target state references. The reference would still work, because the underlying data structure uses the target’s unique ID to establish the reference. It does not matter what we

ple if the user executes a specific extension, or types a specific string on the right side of a **Transition**.

*Intentions* Intentions are MPS’ term for what is otherwise known as a Quick Fix: a little menu can be displayed on a program element that contains a set of actions that change the underlying program element (see Fig. 7.13). The intentions menu is opened via **Alt-Enter**. In MPS, intentions play an important role in the editor. In many languages, certain changes to the program can *only* be made via an intention[[14]](#footnote-14). Using the intentions menu in a projectional editor is idiomatic. For example, in the previous section we mentioned that we use them to add an entry action to a **State**. Here is the intention code:

|  |
| --- |
| **intention** addEntryActions **for concept** State { **available in child nodes** : **true**  **description**(editorContext, node)->**string** { "Add Entry Action";  }  **isApplicable**(editorContext, node)->**boolean** { node.entryAction.isNull;  }  **execute**(editorContext, node)->**void** { node.entryAction.**set new**(<default>); editorContext.selectWRTFocusPolicy(node.entryAction); |



|  |
| --- |
| }  } |

|  |
| --- |
| **side transform actions** makeArithmeticExpression **right transformed node**: Expression **tag**: default\_  **actions** :  **add custom items** (**output concept**: PlusExpression) **simple item** |

An intention is defined for a specific language concept (**State** in the example). It can then be invoked by pressing **Alt-Enter** on any instance of this concept. Optionally it is possible to also make it available in child nodes. For example, if you are in the guard expression of an transition, an intention for **State** with **available in child nodes** set to **true** will be available as well. The intention implementation also specifies an expression used as the title in the menu and an applicability condition. In the example the intention is only applicable if the corresponding state does not yet have any entry action (because in that case you can just type in additional statements). Finally, the **execute** section contains procedural code that performs the respective change on the model. In this case we simply create a new instance of **StatementList** in the **entryAction** child. We also set the cursor into this new **StatementList**85.

|  |  |
| --- | --- |
|  |  |
| *Expressions* Since we inherit the expression structure and syntax from the C core language, we don’t have to define expressions ourselves to be able to use them in guards. It is nonetheless interesting to look at their implementation in the language **com.mbeddr.core.expressions**.  Expressions are arranged into a hierarchy starting with the abstract concept **Expression**. All other kinds of expressions extend **Expression**, directly or indirectly. For example, **PlusExpression** extends **BinaryExpression**, which in turn extends  **Expression**. **BinaryExpressions** have **left** and **right** child **Expressions**. This way, arbitrarily complex expressions can be |  |
| built86. The editors are also straightforward – in the case of the |  |
| **+** expression, they are a horizontal list of: editor for **left** argument, the **+** symbol, and the editor for the **right** argument.  As we have explained in the general discussion about projectional editing (Section 7.2), MPS supports linear input of hierar- |  |

chical expressions using *side transforms*. The code below shows the right side transformation for expressions that transforms an arbitrary expression into a **PlusExpression** by putting the **PlusExpression** "on top" of the current node[[15]](#footnote-15).

|  |
| --- |
| **matching text**  + **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**;  } |

The fact that you can enter expressions linearly leads to a problem not unlike the one found in grammars regarding operator precedence. If you enter **2 + 3 \* 4** by typing these characters sequentially, there are two ways in which the tree could look, depending on whether **+** or **\*** binds more tightly88.

|  |  |
| --- | --- |
| This method scans through an expression tree and checks for cases in which a binary expression with a higher precedence is an ancestor of a binary expression with a lower precedence value. If it finds one, it rearranges the tree to resolve the prob- |  |
| lem89. |  |

To deal with this problem, we proceed as follows: each subconcept of **BinaryExpression** has a numerical value associated with it that expresses its precedence. The higher the number, the higher the precedence (i.e. the lower in the tree). The action code shown above is changed to include a call to a helper function that rearranges the tree according to the precedence values.

|  |
| --- |
| **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>);  // rearranges tree to handle precedence PrecedenceHelper.rearrange(expr); **return** expr.right;  } |

*Context Restrictions* MPS makes strong use of polymorphism. If a language concept defines a child relationship to another concept **C**, then any subtype of **C** can also be used in this child relationship. For example, a function has a **body** which is typed to **StatementList**, which contains a list of **Statement**s. So every subtype of **Statement** can be used inside a function body. In general, this is the desired behavior, but in some cases, it is not. Consider test cases. Here is a simple example:

|  |
| --- |
| **module** UnitTestDemo **imports** nothing {  **test case** testMultiply {  **assert** (0) times2(21) == 42; } |

|  |
| --- |
| int8 times2(int8 a) {  **return** 2 \* a;  }  } |

Test cases reside in a separate language **com.mbeddr.core.unittest**. The language defines the **TestCase** concept, as well as the **assert** statement. **AssertStatement** extends **Statement**, so by default, an **assert** can be used wherever a **Statement** is expected, once the **com.mbeddr.core.unittest** is used in a program. However, this is not what we want: **assert** statements should be restricted to be used inside a **UnitTest**90. To

support such a use case, MPS supports a set of constraints.

|  |
| --- |
| **concept constraints** AssertStatement {  **can be child**  (operationContext, scope, parentNode, link, childConcept)->**boolean** { parentNode.ancestor<TestCase, +>.isNotNull; }  } |

Here is the implementation for **AssertStatement**:

|  |
| --- |
| provides the **table** cell for this. For example, the editor for the decision table is shown in Fig. 7.14 (and an example table is shown in Fig. 14.7). |

|  |  |
| --- | --- |
| This constraint checks that a **TestCase** is among the ancestors of a to-be-inserted **AssertStatement**. The constraint is checked *before* the new **AssertStatement** is inserted and *prevents* inser- |  |
| tion if not under a **TestCase**91. |  |
| *Tables and Graphics* The MPS projectional editor associates projection rules with language concepts. A projection rule consists of cells. Each cell represents a primitive rendering element. For example, a **constant** cell contains a constant text that is rendered as-is in the programs. A property cell renders a property (for example, the **name**). Collections cells arrange other cells in some predefined or configurable layout. Among others, MPS has vertical and horizontal collections. To render concepts as a table, a suitable kind of cell is required: MPS |  |

However, this is only half of the story. The real definition of the table contents happens via a table model implementation inside the **table** cell. The inspector for the **table** cell contains a function that has to return a **TableModel**, an interface that determines the structure of the table[[16]](#footnote-16). Here is the code used

in the decision table:

(node, editorContext)->TableModel { **return new** XYCTableModel(node, **link**/DecTab : xExpr/, **link**/DecTab : yExpr/,

|  |
| --- |
| **link**/DecTab : cExpr/, editorContext);  } |

The **XYCTableModel** class is a utility class that ships with MPS for tables whose contents are represented by a concept that has three child collections, one for the contents of the row headers, one for the contents of the column headers and one for the remaining cells. We pass in the **node** that represents the table as well as the three child collections (and the **editorcontext**). If none of the existing utility classes is suitable, you have to implement the **TableModel** interface yourself93. Here is the

definition of the interface:

|  |
| --- |
| **public interface** TableModel **extends** <none> { **int** getColumnCount(); **int** getRowCount(); **void** deleteRow(**int** rowNumber); node<> getValueAt(**int** row, **int** column); **void** createElement(**int** row, **int** column);  NodeSubstituteInfo getSubstituteInfo(**int** row, **int** column); **void** insertRow(**int** rowNumber); **void** deleteColumn(**int** columnNumber); **void** insertColumn(**int** columnNumber); **int** getMaxColumnWidth(**int** columnNumber); } |

|  |  |
| --- | --- |
| Note how the **getValueAt** method returns a **node<>**. The editor then renders the editor for that node into the respective table cell, supporting nesting of arbitrary other editors into tables.  A similar approach will be used for graphical notations. New kinds of cells (for example, **rectangle** and **line**) may |  |
| be required94. The fundamentally interesting characteristic of |  |

projectional editors is that completely different styles of notations can be supported, as long as the necessary primitive cell types are available. The approach to editor definition remains unchanged. Because all the different notations are based on the same paradigm, the combination of different notational styles is straightforward.

1. . [↑](#footnote-ref-1)
2. . [↑](#footnote-ref-2)
3. . [↑](#footnote-ref-3)
4. . [↑](#footnote-ref-4)
5. . [↑](#footnote-ref-5)
6. . [↑](#footnote-ref-6)
7. . [↑](#footnote-ref-7)
8. . [↑](#footnote-ref-8)
9. The entity that contains the meta classes is actually called an **EPackage**. [↑](#footnote-ref-9)
10. . [↑](#footnote-ref-10)
11. r [↑](#footnote-ref-11)
12. *e* [↑](#footnote-ref-12)
13. . [↑](#footnote-ref-13)
14. . [↑](#footnote-ref-14)
15. . [↑](#footnote-ref-15)
16. . [↑](#footnote-ref-16)