Gradual Grammars: Syntax in Levels and Locales

Tijs van der Storm
storm@cwi.nl
Centrum Wiskunde & Informatica (CWI)
Amsterdam, Netherlands
University of Groningen
Groningen, Netherlands

Felienne Hermans
f.f.j.hermans@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

Abstract

Programming language implementations are often one-size-fits-all. Irrespective of the ethnographic background or proficiency of their users, they offer a single, canonical syntax for all language users.

Whereas professional software developers might be willing to learn a programming language all in one go, this might be a significant barrier for non-technical users, such as children who learn to program, or domain experts using domain-specific languages (DSLs).

Parser tools, however, do not offer sufficient support for graduality or internationalization, leading (worst case) to maintaining multiple parsers, for each target class of users.

In this paper we present Fabric, a grammar formalism that supports: 1) the gradual extension with (and deprecation of) syntactic constructs in consecutive levels ("vertical"), and, orthogonally, 2) the internationalization of syntax by translating keywords and shuffling sentence order ("horizontal"). This is done in such a way that downstream language processors (compilers, interpreters, type checkers etc.) are affected as little as possible.

We discuss the design of Fabric and its implementation on top of the LARK parser generator, and how Fabric can be embedded in the Rascal language workbench. A case study on the gradual programming language Hedy shows that language levels can be represented and internationalized concisely, with hardly any duplication. We evaluate the Fabric library using the Rebel2 DSL, by translating it to Dutch, and "untranslating" its concrete syntax trees, to reuse its existing compiler. Fabric thus provides a principled approach to gradual syntax definition in levels and locales.

CCS Concepts: • **Software and its engineering → Syntax**; • **Human-centered computing → Accessibility technologies**;

SLE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9919-7/22/12.
https://doi.org/10.1145/3567512.3567524

Keywords: syntax definition, internationalization, modularity

ACM Reference Format:
Tijs van der Storm and Felienne Hermans. 2022. Gradual Grammars: Syntax in Levels and Locales. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3567512.3567524

1 Introduction

Programming languages are no islands. They are embedded in cultural and ethnographic contexts to let highly diverse user communities write programs to make computers do things. Nevertheless, typical programming language implementations attempt to offer a one-size-fits-all solution. As a result, programmers are faced with an all-or-nothing choice to learn and use a programming language.

We discern two syntax-related barriers to learning software languages:

- Software languages do not typically offer support for *gradually* introducing language features to students learning the language. As a result, beginning programmers have to learn too many concepts at once, which increases cognitive load and has been known to be an impediment to learning [8, 16].
- The syntax of a language is biased towards the English speaking world in terms of keywords and word order. Research shows that non-English students struggle with using the right English keywords and symbols [11, 26]. Students with different ethnographic backgrounds will likely be helped if a language was closer to their native tongue. The same holds for non-professional, end-user developers, e.g., using domain-specific languages (DSLs).

To address these issues, we present Fabric, a modular grammar formalism that sports two novel features. First, it allows language designer to organize a grammar into *levels* to gradually disclose (and retire) language features. We call this direction of graduality "vertical" (e.g., versions). Second, Fabric employs *grammar fabrics* to customize production rules through keyword renaming and reordering elements of a production. We call this "horizontal" customization (e.g., variants). This second dimension is gradual as well, in the

sense that customization is opt-in; not all productions have to be customized.

The contributions of this paper can be summarized as follows:

- A motivation of gradual grammars, in the context of the Hedy programming language (Section 2).
- The design of Fabric, a grammar formalism supporting language levels and language internationalization (Section 3).
- An implementation of Fabric using the Rascal [20] language workbench (Section 4).
- An embedding of Fabric as a library in the Rascal metaprogramming language (Section 5)
- The evaluation of Fabric using two case-studies, Hedy [16], and Rebel2 [31] (Section 6).

We conclude the paper with a discussion of limitations and a survey of related work (Section 7)[1].

2 Background and Overview

Hedy is a programming language designed for teaching programming to children [16]. One of the novelties of Hedy is that it gradually introduces syntactic constructs to improve learning. At the time of writing, the language is split over 19 levels, where the final level corresponds to (a subset of) Python.

For instance, at level 1, students can write `print hello`, without quotes, to print something on the screen. Only later, when variables are introduced, the student is required to use quotes, to disambiguate string literals from variables. Along the same line, more advanced levels also retire or deprecate syntactic constructs because they are subsumed by more general constructs. For example, for repetition, initially the keyword `repeat` is used, which in later levels is replaced by the regular Python construct `for i in range():`.

The engineering of Hedy is a considerable task[2]. Even more so, since the present version of Hedy supports internationalized keywords for 19 languages, and currently there's ongoing work to support sentence reordering as well.

The syntactic part consists of a single reference grammar (using the LARK parser[3]), split into files corresponding to each level, amounting to around 300 LOC grammar code. LARK does not support the required modularity to make the reference grammar gradual; as a result the Hedy compiler simply concatenates the files for consecutive levels, and performs rudimentary rewriting to override and deprecate constructs.

Internationalization is implemented in a similarly ad hoc way: each locale is defined in a separate LARK file defining nonterminals for each keyword, with the respective keyword literals in the native language (around 800 LOC). Again, these files are simply prepended to the (concatenated) level grammar files. The result is a set of over 380 "total" grammars, amounting to 48,664 LOC LARK parser code.

Although the current implementation avoids having to maintain hundreds of separate grammars for each combination of level and locale, there are limitations to the current situation:

- The textual composition of various files is very ad hoc and brittle; there is no early checking of the validity of the result, and it complicates debugging.
- Internationalization through separate token definitions works for keyword translation, but does not allow for more complex transformations, such as changing sentence order, inserting/removing keywords, or swapping reading direction.
- The approach does not technically support removing of syntactic constructs, so this has to be simulated by marking productions as error-production, or leaving certain parses ambiguous, and provide custom error messages later in the language processing pipeline.

The current implementation of Hedy is based on the LARK parser framework, which is fairly advanced (general) parsing tool, with limited support for modularity. Although modularity in parser definition and grammars is well researched topic (see, e.g., [15, 20, 29]), we are not aware of any formalism with *first class* support for vertical graduality. Similarly, although textual languages with internalized keywords do exist (e.g., AppleScript, Excel), principled techniques for internationalization of syntax definitions have not been investigated by the programming language or language engineering communities.

To address the issues above, we present Fabric, an EBNF-style grammar formalism which distinguishes between two different kinds of grammars. *Reference grammars* capture the source of knowledge, defining the default syntax and defining the structure of the Abstract Syntax Tree (AST). The other kind of grammar is called *fabric grammars*, which non-invasively define internationalization of (a selection of) syntax productions as defined in the reference grammar.

Reference grammars can be organized in consecutive levels, where each subsequent level possibly adapts the previous levels, in three ways: 1) productions can be overridden, similar to a method override in object-oriented languages; 2) production removal, and 3) production deprecation.

Fabric grammars are defined relative to a reference grammar. Again, they can be organized into levels, this time defining customizations using an aspect-oriented form of production pattern[4]. Such rules match against productions in the reference grammar, and may translate, introduce, or remove keywords, and modify sentence order, as long as all AST

[1] The code and experiments supporting this paper can be found here: 10.5281/zenodo.7211893.

[2] To give an intuition, version `665c0be` of Hedy consists of 625,253 SLOC (mostly Python).

[3] https://github.com/lark-parser/lark

[4] One could say fabric grammars are defined in terms of *grammar pointcuts*.

nodes are preserved by the transformation. The customization of the reference grammar is gradual as well: language designers may choose to only reshape a subset of all productions. Consistency of a fabric grammar with respect to its reference grammar can be checked.

An implementation of Fabric would flatten a reference grammar for a certain level l into a single context-free grammar, ensuring that the consecutive adapations of levels $1..l$ are observed. A fabric grammar is *stitched* onto a reference grammar's levels before flattening to obtain an internationalized grammar.

3 Fabric by Example

3.1 Reference Grammars

QL is a simple DSL for defining interactive questionnaire forms [10], consisting of a number of (possibly conditional) questions to be filled out by the user. Next to ordinary questions, a QL form may contain computed "questions", which derive their value from an expression computing a value from other questions. QL forms are executed as an interactive graphical user interface, for instance as an HTML form in the browser.

This DSL is an example of a language that could be used by non-professional, end-user programmers, to create simple, interactive applications. Hence, for teaching purposes, it makes sense to gradually introduce the language to novices. Furthermore, one can imagine that in this small domain, some English biased idioms could be foreign to users in different countries, so that internationalization is desired.

Below we gradually introduce Fabric using QL as the example language. The first level is defined as follows (omitting the definition of all-caps tokens for brevity):

```
grammar QL
level 1
  form = form: "form" ID question*
  question = question: "ask" STRING "into" ID  ":" type
  type = bool: "boolean"
```

In this language level the only kinds of forms that are allowed consist of questionnaires with boolean questions. Each non-terminal has a single alternative with labels identifying the production.

Level 2 introduces conditional questions and basic boolean expressions:

```
level 2
  question = ifThen: "if" "(" or_expr ")" question
  or_expr = or: and_expr "||" or_expr | and: and_expr
  and_expr = and: primary "&&" and_expr | prim: primary
  primary = ref: ID | boolean: bool
  bool = true: "true" | false: "false"
```

The `question` nonterminal is extended with another alternative defining the syntax of conditional questions. Other than that, this level introduces new syntax for expressions, consisting of boolean literals, logical connectives and question references.

The third level adapts level 2 to require grouping of multiple questions below an if-then condition.

```
level 3
  question
    = @override ifThen: "if" "(" or_expr ")" "{" question* "}"
    | @error ifThenError: "if" "(" or_expr ")" question
```

The `ifThen` construct of level 2 is overridden using the `@override` annotation to replace the previous construct. The original production, however, is demoted to an error production, so that if users still use the old notation, an informative error message can be produced.

Next to the override and error annotations, Fabric supports directives at the scope of a level to remove or deprecate productions. For instance, **`remove`** $l_1, ..., l_n$ indicates that all productions labeled $l_1, ..., l_n$ should be removed of any previous levels at this point. Analogously, **`deprecate`** $l_1, ..., l_n$ entails marking productions labeled $l_1, ..., l_n$ to be deprecated since the current level.

Excluding `@override`, this leads to a partitioning of productions in a gradual grammar, where each class indicates a different (implicit) message to the user:

- **Normal**: it's part of the language and fully supported.
- **Error**: it's not supported, but we expect this error and we want a nice error message
- **Deprecate**: it's supported, but be aware, it will be removed in the future.
- **Remove**: it's not supported, and you'll get a normal parse error.

One may wonder whether it would make sense to never remove productions, and just deprecate or convert to an error production. From a usability point of view, this may indeed be sufficient. However, depending on the underlying parsing technology used to implement Fabric, actually removing syntax might be necessary to avoid technical parsing problems, such as unwanted ambiguity or shift/reduce conflicts[5].

3.2 Fabric Grammars

The gradual grammar introduced above should be considered the reference grammar: it captures the syntax of QL in what we consider the canonical syntax that (implicitly) defines the structure of the AST. The second kind of gradual grammars supported by Fabric allow language designers (or translators) to specify the aforementioned customizations per level.

Consider the following grammar fabric that translates level 1 of QL to Dutch:

[5] In fact, the embedding of Fabric as a Rascal library (Section 5), employs precisely this strategy, since Rascal's parser is general and supports ambiguous parse forest that can be filtered after parsing.

```
grammar QL_NL modifies "QL.fabric"
level 1
  form = form: "formulier" _ _
  type = bool: "waarheidswaarde"
  question = question: "vraag" _2 "met" _1 ":" _3
```

The header of the module specifies that this grammar is dependent on (or relative to) a reference grammar, in this case contained in the file `QL.fabric`. The first level of this grammar defines a number of production patterns that are matched to the reference grammar based on the production labels. The symbols of a production use placeholders (indicated by "_") to match sub-trees of interest in the reference grammar.

For instance, the first rule has a single production with the pattern `"formulier" _ _` which matches the production `"form" ID question*` shown above. The placeholders correspond to `ID` and `question*`, respectively. The keyword `"form"` is translated to the Dutch equivalent `"formulier"`. The second production is another example of a basic translation pattern, where the keyword `boolean` is translated to the Dutch equivalent, `waarheidswaarde`.

The third pattern, however, goes a step further. In this case the placeholders are suffixed with a 1-based position of symbols in the original production, `"ask" STRING "into" ID ":" type`. So in this case, the pattern does not simply translate the keywords `ask` to `vraag` and `into` to `met`, but also swaps the order of the symbols `STRING` and `ID`.

Realistically speaking, the example is slightly contrived: Dutch does not require the reordering in this case, but for languages such as Arabic and Japanese changing sentence order can be important to make a language more natural to use for native speakers.

Grammar fabrics are also leveled. Here's the translation of level 2:

```
level 2
  bool = true: "waar" | false: "onwaar"
  question = ifThen: "als" _  "dan" ":" _
```

The boolean values are translated to their Dutch equivalent, as before. More interestingly, however, the translation of the if-then construct not only translates the keyword **`if`** but also inserts an extra keyword `dan` and the interpunction `":"` in between the condition and the body of the conditional, and removes the parentheses. So the production patterns in grammar fabrics can liberally replace, introduce, and/or remove keywords from a syntax production to adapt a grammar to a certain locale.

Note, however, that the language designer customizes the reference grammar at their own risk: there is no guarantee that the resulting grammar (after applying the patterned rules) will be well-formed for the underlying parse technology. As a pathological example: removing all keywords most likely will lead to a highly ambiguous grammar. Note further that fabric grammars are gradual as well: anything not reshaped is left as-is, as per the reference grammar.

```
data Grammar
  = grammar(str name, list[Level] levels, str base="");

data Level
  = level(int n, list[str] rem, list[str] depr
          , list[Rule] rules);

data Rule
  = rule(str nt, list[Prod] prods);

data Prod(bool error=false, bool override=false
          , int deprecatedAt=-1, int level=-1)
  = prod(str label, list[Symbol] symbols);

data Symbol
  = nonterminal(str name) | literal(str lit)
  | placeholder(int pos=-1) | ...;
```

Figure 1. Abstract syntax of Fabric grammars (simplified).

Fabric furthermore supports imports, layout definitions (whitespace and comments), an extended set of regular symbols, and token definitions. These features are by all means useful or even necessary for defining realistic languages, however, they are immaterial to the contributions of this paper.

4 Fabric Implementation

4.1 Flattening

Figure 1 shows the (slightly simplified) abstract syntax of Fabric in the algebraic data type notation of Rascal. A grammar has a name, contains a number of levels, and (optionally) modifies a base grammar[6]. Note that the Grammar type captures both reference grammars and fabric grammars.

A level is indexed with a number, defines a list of removed and deprecated production labels, and contains a list of rules. Rules are identified by a nonterminal and contain a list of production alternatives. Productions have a mandatory label and a sequence of symbols. Optionally, they can be tagged to be an error production, an overriding production or marked as deprecated at a certain level (`deprecatedAt`). The final keyword parameter `level` indicates the level this production originated from, and is used to track the level of a production after flattening.

Finally, symbols capture the well-known symbols from EBNF-like grammar formalisms, such as nonterminals, literals, etc. Additionally, Fabric symbols include a placeholder kind (optionally with a position) as used in grammar fabrics; their role is further discussed in Section 4.2.

[6]This is an example of Rascal's keyword parameters: optional parameters to a constructor declared with a default value, either at the constructor level, or at the type level so that all constructors get them.

```
Grammar flatten(int n, Grammar g) {
  Level current = g.levels[0];
  for (Level l ←g.levels[1..n])
    current = adapt(l, current);
  return grammar(g.name, [current]);
}

Level adapt(Level l, Level g) =
  sort(merge(delOv(l, markDep(l, delRem(l, g))), l.rules));
```

Figure 2. Flattenening a Fabric grammar for level n.

```
Prod stitch(Prod base, Prod fabric) {
  astKids = [ s | Symbol s ←base.symbols
                , !(s is literal) ];
  Symbol lookup(Symbol s) =
    s is placeholder ? astKids[s.pos - 1] : s;
  base.symbols =
    [ lookup(s) | Symbol s ←fabric.symbols ];
  return base;
}
```

Figure 3. Stitching productions.

The main operation on Fabric grammars is to obtain a flattened grammar for a certain level n. Figure 2 shows the pseudo-code of this algorithm in Rascal notation. The first function receives the level n and a value of the type `Grammar` (cf. Listing 1). It then constructs a new grammar with a single level that consists of applying the `adapt` function on the accumulated level, starting at the first level up till the level at position $n-1$ (recall that levels are 1-based).

The `adapt` function transforms the accumulated level `g` according to the directives and annotations defined in level `l` using five auxiliar functions, which are explained as follows (innermost first):

1. **delRem**: remove the productions marked using the `remove` directive in level l_i.
2. **markDep**: set the `deprecatedAt` field of deprecated productions to i.
3. **delOv**: remove the original productions with the same label as productions annotated with `@override` in l_i.
4. **merge**: combine the rules of l_i with the current set of rules, merging sets of alternatives for shared nonterminals.
5. **sort**: order the sequence of alternatives of each rule so that the productions with `error=true` come last[7].

The `flatten` function is called for every level in a Fabric grammar to obtain flattened grammar values corresponding to each level. This flattened grammar is then compiled to the formalism of the back-end parser generator, in this case LARK.

4.2 Stitching Fabrics

To define the semantics of stitching a fabric onto a reference grammar to obtain an internationalized grammar, we will focus on stitching two matched productions, as shown in Figure 3. Without loss of generality, we will assume that all bare _ placeholders have been normalized to placeholders with 1-based indices. Furthermore, we gloss over the fact that nested sequences, such as `(expr "and" expr)?`, may contain keywords; this can be supported by a simple recursive fix of the code shown in Figure 3.

The function `stitch` takes a production from the reference grammar (`base`), and a matched fabric production with placeholders. The first line extracts the symbols from the the base production that represent AST nodes, because this is how numbered placeholders refer to elements of a production. The auxiliary function `lookup` performs the shuffling according to the fabric symbol, if the symbol is a placeholder, otherwise the symbol (e.g., a translated literal) remains unchanged. Finally, the symbol list of the base production is updated, so that it now conforms to the fabric production, but with all placeholders substituted. Note that by updating the base production, we preserve meta-data about `@error`, `@override` etc.

The top level stitching algorithm iterates over a flattened reference grammar, and for each production it encounters, it applies the fabrics (if any) defined in the levels lower or equal than the production's origin. It does this from the highest level down to the lowest, to ensure that later fabric customizations (i.e. in higher levels) are applied first. The result is a flattened, internationalized grammar.

4.3 Implementation

Fabric has been implemented in Rascal, according to the semantics described above, but additionally dealing with certain corner cases, and providing dynamic error checking. Based on Rascal's language workbench features, Fabric comes with basic IDE support in VSCode, which provides folding, outlining, syntax highlighting, and compilation of both reference and fabric grammars. A partial screenshot of the QL reference grammar is shown in Figure 4.

Flattened Fabric grammars are compiled to LARK grammars. In the case of shuffled productions, the corresponding LARK productions are annotated with labels encoding the reordering that has occurred. This allows AST construction code to undo the shuffling, before constructing the ASTs. As a result, existing backend processors (compilers, type checkers, etc.) can still be reused.

[7] Technically, this is an implementation detail of LARK seeping through: LARK tries out productions in the order they are specified in the grammar, and error productions should be tried as a last resort only. Ordered evaluation of alternatives is, however, a common strategy of (non-general) top-down parsing algorithms.

```
Compile
module QL

level 1
    STRING = string: /"[^"]*"/
    ID = id: /[a-zA-Z][a-zA-Z0-9]*/
    form = form: "form" ID question*
    question = question: "ask" STRING "into" ID  ":" type
    type = bool: "boolean"

level 2
    bool = true: "true" | false: "false"
    question = ifThen: "if" "(" or_expr ")" question
    or_expr = or: and_expr "||" or_expr | and: and_expr
    and_expr = and: primary "&&" and_expr | prim: primary
    primary = ref: ID | boolean: bool

level 3
    question
        = @override ifThen: "if" "(" expr ")" "{" question* "}"
        | @error ifThenError: "if" "(" expr ")" question

```

Figure 4. Partial screenshot of the Fabric VSCode IDE.

5 Fabric as a Rascal library

5.1 Embedding Gradual Grammars

In this section we show how the concepts of Fabric can be simulated in the Rascal language workbench, using Rascal's modularity and disambiguation features. Although embedding typically forces us to give up control with respect to certain aspects of a language (i.e. we lose full control over what grammars we generate or how certain meta-language features are realized), what we gain in return is better integration with the other metaprogramming of the Rascal language workbench, used to define type checkers, compilers, and IDE support.

There is, however, another complication we need to tackle: language processors implemented using Rascal are often defined using concrete syntax matching and construction. For instance, a compiler for QL could have a case matching on QL's conditional using a pattern like this:

```
(Question)`if (<Expr cond>) {<Question* body>}`
```

This concrete pattern matches against *concrete* syntax trees, as produced by parsers derived from the actual surface syntax grammar. As a result, such patterns will not match against internationalized versions of the QL grammar, because the keyword "if" might have been translated. Needless to say, the situation is worse if subnodes have been shuffled by a fabric grammar. We will address this problem in Section 5.2.

5.1.1 Embedding Reference Grammars. Figure 5 shows the Rascal version of the QL Fabric grammar. It shows three Rascal modules containing grammars, where the lower modules extend the modules above. Rascal module extension is transitive, so module `QL_3` includes the grammar rules of both level 1 and level 2. The first two modules are simply modules the way one would define and modularize a syntax

```
module QL_1
syntax Form = form: "form" Id Question*;
syntax Question =
  question: "ask" String "into" Id ":" Type;
syntax Type = boolean: "boolean";
```

```
module QL_2 extend QL_1;
syntax Question = ifThen: "if" "(" OrExpr ")" Question;
syntax OrExpr = or: AndExpr "||" OrExpr | and: AndExpr;
syntax AndExpr = and: Primary "&&" AndExpr
    | prim: Primary;
syntax Primary = ref: Id  \ Keywords
    | boolean: Bool;
syntax Bool = tru: "true" | fal: "false";
keyword Keywords = "true" | "false";
```

```
module QL_3 extend QL_2;
  syntax Question
    = @override=3 ifThen: "if" "(" OrExpr ")"
        "{" Question* "}"
    | @error ifThenErr: "if" "(" OrExpr ")" Question;
```

Figure 5. Gradual grammar embedding in Rascal.

in Rascal. The productions in the third one, however, are annotated, to indicate that a production overrides another one or represents an error production. To obtain a parser for a certain level, it suffices to import the corresponding module, and use Rascal's built-in parser generator.

Rascal module extension is monotone: it is impossible to remove any definition from extended modules, and this holds as well for grammar productions. So how to deal with production removal? We post-pone this decision till after parsing. Since Rascal' parser algorithm is *general*, it supports arbitrary context-free grammars, including ambiguous ones. If a grammar is ambiguous, the parser produces a parse forest containing all derivations. Such a parse forest is like a parse tree, except it contains ambiguity nodes, consisting of sets of alternative derivations. Nevertheless, a parse forest can be inspected, analyzed, and transformed, just like parse trees. The Fabric embedding exploits the grammar annotations embedded in such parse forests to simulate the behavior of Fabric after parsing.

The process proceeds as follows: after parsing a program according to a certain level, we obtain a parse forest, which will be filtered as follows:

- If there is an ambiguity node, and the node contains one ore more trees with an override annotation, keep the tree with the highest override number, and filter the others.

```
module QL_NL_fabric
start syntax Form_NL = form: "formulier" X "{" X "}";
syntax Question_NL
  = question: "vraag" X_2 "met" X_1 ":" X_3
  | ifThen: "als" X "dan" ":" X  () !>> "anders"
  | ifThenElse: "als" X "dan" ":" X "anders" X;
syntax Bool_NL = tru: "waar" | fal: "onwaar";
keyword Keywords_NL = "waar" | "onwaar" ;
syntax Type_NL = boolean: "waarheidswaarde"
```

Figure 6. A QL fabric grammar in Rascal.

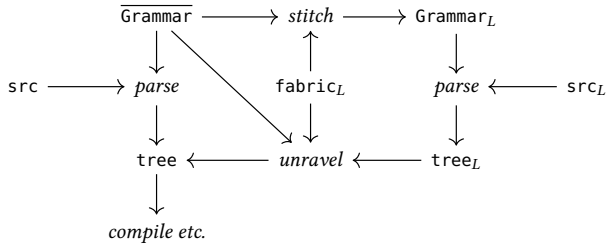

Figure 7. Grammar stitching and parse tree unraveling.

- In the remaining ambiguity nodes, remove the trees annotated with `@remove`, `@error`, and `@deprecate`, in that order, but at least retain one tree.
- For every tree with a `@deprecate` annotation, generate a warning.
- For every tree with an `@error` annotation, generate an error.
- For every tree with a `@remove` annotation, generate a parse error.

The end result is a parse forest and a possibly empty set of error/warning messages. If the parse forest is still ambiguous, it is an ambiguity inherent to the language, unrelated to Fabric.

5.1.2 Embedding Fabric Grammars. Similar to the leveled reference grammars discussed above, fabric grammars are embedded as Rascal syntax definitions as well. An example QL fabric is shown in Figure 6. The placeholders are encoded using a (configurable) dedicated nonterminal, in this case `X`. All nonterminals are suffixed with a locale-identifying string (e.g., `_NL`) to avoid name clashes. Fabric productions are matched to productions in the reference grammar using the production labels (e.g., `ifThen`). Note that keyword reservation, expressed using the **`keyword`** directive, is translated like an ordinary production.

The fabric grammars themselves are never used to parse any program, but function as recipes to transform the reference grammar. In Rascal this is achieved using type reflection: the `#` operator allows any Rascal type to be converted to a value representation. For instance, the expression **`#bool`** produces a value representing the **`bool`** type.

Since every nonterminal in a context-free grammar defines a proper type of parse trees of a certain shape, we can use these reflective capabilities to reify grammars into values, which can be dynamically analyzed, transformed, or constructed. In this case this amounts to reifying both the reference grammar and the fabric grammar, and then to stitch (see Section 4.2) the fabric grammar onto the reference grammar. The result is then pretty printed to a Rascal module which can be used to parse programs in the specific locale.

There is still one problem left: the parse trees from the locale-grammar possibly have a different shape from the parse trees that a backend processor (e.g., a compiler) expects in its concrete syntax patterns. In a sense, the effect of stitching a fabric grammar onto a reference grammar has to be undone, but this time on parse trees. We call this process *parse tree unraveling*.

5.2 Parse Tree Unraveling

Parse tree unraveling consists of undoing the effect of fabric grammar stitching on the parse trees resulting from parsing using the stitched grammar.

An overview of the approach is shown in Figure 7. The left of the figure shows the normal situation where a reference grammar is used to parse some source code, which produces a tree that can be compiled. The middle shows how both the reference grammar and the fabric grammar are used by both stitch and unravel. In the case of stitch, this produces an internationalized grammar; in the case of unravel, parse trees obtained using the internationalized grammar are converted back to parse trees that can be compiled. As a result, all backend language processor are reused.

The essence of parse tree unraveling is to reuse the reified fabric grammar as a recipe to transform the parse trees to a shape *as if* it resulted from a parse using the reference grammar. However, there is one challenge that needs to be addressed: since stitching allows the addition and/or removal of tokens at various places in a production, the number and placement of layout nodes (whitespace and comments) in between symbols has to be adjusted when going from internationalized parse tree to a parse tree as dictated by the reference grammar.

This problem is illustrated in Figure 8 using the Dutch version of QL. The bottom row shows the arrangement of the children of a parse tree representing the Dutch conditional. The layout positions are indicated with ␣ at positions 1, 3, 5, and 7. The top row shows the desired (reference) arrangement of children. Note that, incidentally, there is an equal number of layout positions, but their positions are not aligned with the bottom row.

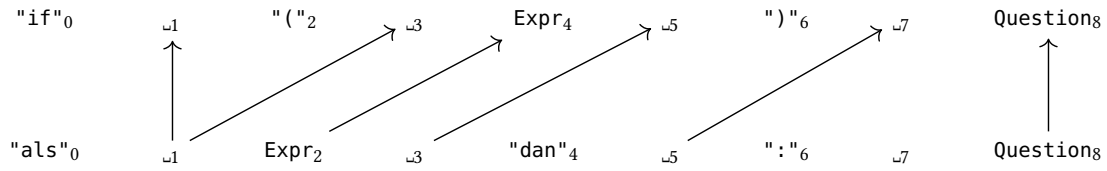

Figure 8. Aligning production patterns (no reordering)

The arrows indicate the way the unravel algorithm heuristically assigns actual layout from the Dutch parse tree into the constructed reference parse tree. In the meantime, the original keywords are simply replicated from the reference grammar, because they are constant. Note further that layout at position 1 is used twice, because the heuristic assumes that the layout after $\texttt{Expr}_2$ should come after $\texttt{Expr}_4$ at the top row. Finally, a consequence of this heuristic is that layout at position 7 is never replicated.

What about reordering of AST nodes? After a matching has been established like in Figure 8, the indices on the placeholders are used to shuffle the AST nodes into the positions dictated by the reference grammar. This only applies to AST nodes, so $\texttt{Expr}_2$ and $\texttt{Question}_8$ would be eligible for shuffling, if the fabric grammar's placeholders would specify that. The algorithm is similar to the stitching process of Figure 3, but in reverse, and modulo the layout nodes.

The full algorithm in (simplified) Rascal code can be found in Appendix A. The above explanation details a specific heuristic to deal with the layout problem. Others are possible too. For instance, we could duplicate layout nodes, if the internationalized tree has fewer layout nodes than the reference tree; in the current algorithm we put a dummy layout if there are not enough layout nodes available, e.g., due to keyword removal in the fabric. Alternatively, one could merge multiple layout nodes into a single one, if the internationalized tree has a surplus. In any case, which heuristic works best, and for what purpose, requires further research.

The accurate backporting of the layout is not an essential aspect of the approach, because the unraveled parse trees are not for human consumption, at least not if the reason is to reuse backends. If unraveling is used for the actual translation of locale-specific programs back to the reference format, one would like to provide a reasonable layout. Still, for debugging concerns of the language engineer, it might be useful to have a layout at least somewhat close to the internationalized original program.

A pretty printer could also be used, but this requires unparsing to text, and parsing again, which is inefficient, and furthermore breaks down when identifiers were used that are reserved keywords in the reference grammar. It would also have been possible to simply insert a default layout (e.g., a single space) everywhere, but this would lead to unraveled programs that are completely unrecognizable to the original author.

5.2.1 Unraveling to Abstract Syntax.

Although it is common and idiomatic in Rascal to define language processors in terms of concrete syntax trees, it is possible to define a language's abstract syntax as an algebraic data type, and convert parse trees to ASTs over such a data type. This process is sometimes called "implode", and Rascal comes with a built-in function that performs implode generically, mapping production labels in the parse tree to constructor names in the algebraic data type.

To accomodate internationalization we have reimplemented implode to take reordering of AST children into account. This means that for any internationalized parse tree, a single, canonical abstract syntax type can be used. Since the conversion to abstract syntax trees typically involves eliminating syntactic elements like keywords and whitespace, the problem of layout reconstruction and "untranslating" keywords does not apply here.

6 Evaluation

6.1 Case Study: Hedy′

To evaluate the expressiveness of Fabric, we reimplemented the first 10 levels of a refactored version of Hedy [16], called Hedy′, together with a fabric grammar reproducing the Dutch translation.

We call this version Hedy′, because the actual implementation has organically evolved, contains a lot of tricky corner cases, and displays a strong dependency on LARK's parsing algorithm. The Fabric formalism is designed to be independent of specific parsing technology (as much as possible); replicating the actual Hedy implementation would entail too much tailoring for this specific case, which defeats the point of being a generic tool. Nevertheless, this case study shows that the graduality and internationalization required by Hedy can be concisely represented in Fabric. We delegate refactoring the actual implementation of Hedy to future work.

Table 1 shows a metrics-based summary of the Hedy′ Fabric grammars. As can be seen, most of the features of Fabric are used extensively, with the exception of `@deprecate`. The Dutch translation requires a mere 57 source lines of code (SLOC). Both the reference grammar and the Dutch

Table 1. Hedy′ case study metrics.

#levels	10	Reference grammar	151 SLOC
#overrides	29	Dutch fabric	57 SLOC
#removes	4	Base LARK	503 SLOC
#error	16	Dutch LARK	597 SLOC
#deprecate	0		

fabric grammar generate a total of 503 SLOC and 597 SLOC of LARK code, respectively. These results are in the same order of magnitude as the actual implementation of Hedy. Yet, using Fabric, the leveled structure of Hedy′ is made explicit and analyzable. Furthermore, the Dutch translation, supports reordering of AST arguments, which is currently unsupported by Hedy.

6.2 Case Study: Rebel2

To exercise the embedding of Fabric in the Rascal language workbench, we have performed a second case study on the DSL Rebel2 [31]. Rebel2 is a DSL for state-based specification of financial products, and performing subsequent lightweight model checking operations. Since the target audience of Rebel2 consists of domain experts in finance, it makes sense to offer the DSL close to the users' native language (in this example we use Dutch).

Figure 9 shows a simple Rebel2 specification of a counter system, using Dutch keywords. Such specifications can be parsed using the grammars obtained from stitching the Dutch Rebel2 fabric onto the reference grammar (which required no changes at all). The resulting parse trees are then unraveled to reference parse trees, which can be compiled, type checked, or unparsed to text (if so desired). Appendix C shows the unraveled parse tree of Figure 9. Note that in this case all layout is perfectly preserved.

The back translation of keywords in parse trees poses the problem of keyword reservation. Whereas fabric grammars specify how to reserve keywords in the internationalized grammars, it may happen that users of that locale use identifiers that are actually reserved in the reference grammar. For instance, in Dutch Rebel2 the token "waar" (true) is reserved, and cannot be used as an identifier. However, a Dutch user could use the token "true" as an identifier. When the Dutch parse tree is unraveled, the reference parse tree contains an identifier that could never have resulted from a parse using the reference grammar.

Keyword reservation is a disambiguation construct. As a result, backend language processors generally do not care whether identifiers like "true" occur in the parse tree, since they will still be distinguished from actual keywords through their syntactic category. The problem thus only surfaces if the unraveled parse tree is rendered to text and if one would then try to parse again using the reference grammar, which would cause a parse error. If such rendering to text and

```
module Counter
spec Counter
  i: Integer;
  start gebeurtenis create() post: this.i' = 0;
  gebeurtenis inc() post: this.i' = this.i + 1;
  gebeurtenis dec() post: this.i' = this.i - 1;
  toestanden:
    (*) -> active: create;
    active -> active: inc,dec;

config Simple = c: Counter is uninitialized;

stel vast Eventually3Later1 = uiteindelijk
  bestaat c:Counter | c.i = 3 && uiteindelijk c.i = 1;
doe Eventually3Later1 van Simple in maximaal 7 stappen;
```

Figure 9. Snippet of Dutch Rebel2.

parsing again would be required, e.g., in the case unravel would literally be used to translate programs between locales, a possible solution would be to have the language support escaped keywords as identifiers. Unravel would then detect whether an identifier is reserved in the target grammar, and insert escape characters accordingly. In that case the Dutch identifier "true" would end up as the English identifier, say, "\true".

6.3 Performance Overhead of Unravel

Since the embedding of Fabric in Rascal postpones unraveling to run time, we have performed a small experiment to assess the performance cost of both unravel, and implode. This exercise was performed on 2019 MacBook Pro 2.6 GHz, 6-Core Intel Core i7, with 16 GB of memory. Using random sentence generation, we've synthesized 1000 Dutch QL programs, up to an AST depth of 12, and measured the time of unravel and implode. The results are shown in Figure 10.

From the plot it is clear there is a linear correlation between size of the program and time of unravel and implode, as expected. Somewhat surprising, however, implode is slower that unravel, possibly because implode creates a new tree structure, whereas unravel reuses as much as possible of the original, internationalized parse tree, and only creates new literals, and shuffles subtrees.

Although all randomly synthesized programs incur an unravel/implode overhead that is still below one third of a second, the programs are arguably small ($\leq$ 2500 characters). For large programs, therefore, it could incur a noticeable delay when compiling internationalized programs. However, both in the setting of education (where programs are typically very small), and in the setting of DSLs (which are designed to make programs smaller), the performance overhead of unravel/implode might be acceptable. Nevertheless,

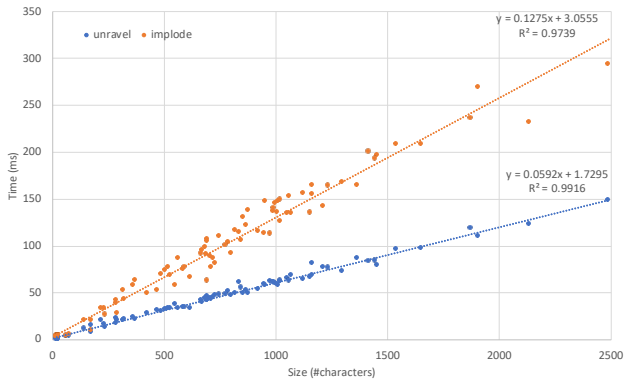

Figure 10. Performance of unravel and implode.

further research is needed to assess the overhead of unraveling/imploding realistic programs, with actual users.

7 Discussion and Related Work

7.1 Discussion

Fabric is designed to be independent of specific parsing technology. However, in the current implementation, specifics of the back-end seep through. For instance, LARK tries alternatives of a syntax rule in order, leading to a preferential order of productions. Fabric makes no assumptions about production ordering, so the actual result of parsing using LARK might incur surprises.

Although Fabric supports declarative internationalization through its fabric grammars, backend processors are not oblivious to shuffling of AST nodes. Language engineers have to detect and interpret the shuffling annotations embedded in the LARK grammars to construct the canonical ASTs that are expected by the language's compiler or interpreter.

The embedding of Fabric in Rascal does not suffer from this problem. However, the obliviousness comes at a performance cost at compile time. Whether this is a relevant performance penalty for the language user is up for debate, and probably depends on language pragmatics, e.g. whether Fabric is used for teaching purposes or for actual internationalization of DSLs.

One additional benefit of the Rascal embedding is that it provides a seamless way to implement the actual translation of programs from a locale to the reference locale. Due to heuristically re-inserting of layout, unraveling an internationalized parse tree into a reference parse tree, we obtain reasonable, human-readable programs, modulo some layout artifacts of the chosen heuristic. The other way round, one would expect it to be possible to translate a program conforming to the reference grammar to any other locale, by taking the generated grammar from the fabric as the (temporary) reference grammar. Unfortunately, this requires a "reference" fabric for each generated localized grammar, which surely does not scale. Further research is needed to extend the Fabric approach to be more symmetric.

There has been recent work on generating Scratch-like [23] block-based environments from context-free grammars [24, 36]. The Kogi tool interprets the productions of a grammar as templates for constructing a toolbox of jigsaw-like pieces which can be then be pieced together to construct programs. In combination with Fabric, language engineers can now construct internationalized block-based environments at practically zero effort.

In essence, Fabric is a modular grammar formalism, which is at odds with parsing algorithms that do not support the full class of context-free grammars. In the current incarnation, Fabric is implemented on top of LARK and embedded in Rascal, both of which are backed by general parsing algorithms. This does not necessarily mean Fabric cannot be realized on top of, say, a variant of Yacc [19], or ANTLR4 [25], since the basic composition mechanism is still merging of sets of production rules. However, there is the risk that the resulting composition is not well-defined, or, at minimum, might not behave as intuitively expected.

7.2 Related Work

In essence, Fabric is modular syntax definition formalism. Modularity in language engineering is a research topic that has been extensively researched. Modular language development systems such as JastAdd [9], Silver [41], Lisa [27], Rascal [1, 20], Asf+Sdf [15, 33], Spoofax [37, 40], Neverlang [32], Monticore [21], GEMOC studio/Melange [3, 7], Xtext [12], MPS [38, 39], Ensō [34], Naked Object Algebras [13, 18], all support varying degrees of modularity and syntax composition.

Many of these approaches employ underlying class-based metamodels to define abstract syntax. As a result, their grammar formalisms (with some variations) follow the mapping pattern where nonterminals correspond to (abstract) base classes, and productions to concrete subclasses. Some form of inheritance is supported, where nonterminals can be extended with additional productions, which in turn correspond to additional subclasses. Others, e.g., Rascal, Asf+Sdf, Spoofax, are based on algebraic data types or signature, which in a similar way only support addition of productions/constructors.

The modularity features offered by these systems could be leveraged to realize gradual grammars. Nevertheless, this kind of modularity is always *monotonic*: language components or language modules can be extended in a strictly increasing fashion, i.e., the only operation is *adding* new constructs, not removing them.

As far as we know, Fabric is novel by supporting *non-monotonic* graduality as well. This comes in two forms. First, production overriding (replacing a syntactic construct with a different syntax) effectively support eliminating existing syntax. Second, the explicit removal can be used to strictly make

language smaller. These features are essential for languages like Hedy, which start by offering extremely simplified notations, which have to be removed later on to avoid parsing conflicts and/or ambiguity.

The second novelty of Fabric is the use of grammar fabrics for internationalization. Although one could argue that a sufficiently modular language development system could leverage its modularity constructs to realize translation of keywords and/or reordering productions, this would require duplicating large parts of the grammar, and does not trivially solve the problem of reusing existing back-ends. We believe that a dedicated, aspect-like fabric formalism is more lightweight, is mostly oblivious to, e.g., names of nonterminals and (labeled) production arguments, and is easy to check for conformance with respet to a base grammar.

Gradual disclosure of language features in levels for the sake of programming education was pioneered in the PLT Scheme/Racket ecosystem [5]. The DrRacket environment[8] offers menu options to enable certain teaching languages, corresponding to a student's proficiency. Unlike Hedy, however, Racket's levels are primarily semantic. For instance, the "Beginning Student" language is a small version of Racket, tailored to novice computer science students, whereas the "Intermediate Student" language adds local bindings and higher-order functions. Racket and Hedy differ in both granularity of graduality and target audience. First of all, the former is oriented towards computer science students, whereas the latter is aimed at children from 8 year on. Second, Hedy is syntactically more fine-grained (at statement level), and non-monotonic: certain syntax becomes invalid in later levels.

More recently, others have also described languages that are extended over the duration of a course, for example Cazzola and Olivares [2] describe a language which gradually builds up to JavaScript, in which students were provided with different JavaScript variants, where each variant focused on another language feature, e.g., loops, recursion, exception handling, object orientation. Vega *et al.* describe their Java-based system Cupi2, in which students solve increasingly more complicated problems, with partly generated programs [35].

Organizing syntax definitions into consecutive levels can be seen as a form of language product lines [6, 22], where each level is a variant of some base language. This has applications outside of programming education. For instance, language restriction is a well-known concept to make languages less expressive, e.g., for the purpose of security or safety. One example of this is Misra C, a version of C used in automotive software development that disallows certain features of the full C programming language [14].

Another way of looking at the vertical aspect of gradual grammars is from the perspective of language evolution. It has been argued that languages should be growable, gradually adding features according to the demands of a certain domain [30]. Industrial languages typically prioritize backwards compatibility over feature removal, but gradual grammars could be used to make language versioning first-class in the language engineering process.

There are number of languages which feature internationalization or localization of their syntax. Next to Hedy, notable examples include Microsoft Excel[9], AppleScript [4], and Scratch [23]. All three target end-user developers and/or novice programmers. While these languages are very successful, they have not lead to reusable, generic principles or techniques to engineer languages for internationalization.

Most relevant to our work is AppleScript, since it is an end-user programming language and has a textual syntax. The implementation approach, however, is different from the Fabric approach we present in this paper. AppleScript programs are stored in a locale-independent format, and they are syntactically skinned depending on in which locale the program is edited or viewed. This means that the rendering to text has to perform pretty printing, thereby possibly losing the layout of the original program. Both Fabric, and its embedding in Rascal, retain the original layout of the programmer, which is desirable in most situations.

Another difference is that AppleScript was designed to look like a natural language, an experiment that, according to Cook [4], has failed. Our approach is independent of such goals. As the prime example of an internationalized programming language today, Hedy definitely is not designed to look like a natural language, but nonetheless benefits from the fact that its keywords can be translated to different languages.

Fabric stitching resembles advice weaving, such as offered by, for instance AspectJ [17], or could be seen as a domain-specific form of delta-oriented programming [28]. The production patterns with placeholders can be seen as simple pointcuts specifying join points in the grammar, or deltas, modifying some base structure. Unlike the general pointcuts of AspectJ, however, matching is one-to-one, through the unique production labels. Since stitched productions preserve the AST symbols via the placeholders, one could say such a pattern realizes a simple form of "around" advice. Along similar lines, fabric grammars resemble Cascading Style Sheets (CSS), which are also used to style or "skin" a base document using pointcut like selectors.

8 Conclusion

Much of software language engineering is based on the assumption that languages have a single, canonical syntax. This syntax, however, can be too intimidating to process all at once by end-users, students, or children learning the language. Furthermore, most programming language syntax is biased towards the western, English speaking world,

[8] https://racket-lang.org/

[9] https://www.microsoft.com/en-us/microsoft-365/excel

which can be another barrier to learning. In this paper we have presented Fabric, a grammar formalism that allows language engineers to organize syntax definitions into levels, to gradually introduce language features. Furthermore, each of those levels can be internationalized by keyword translation and sentence reordering directives, defined in fabric grammars.

Fabric has been implemented in Rascal [20] as a standalone grammar formalism which is compiled to LARK grammars. An embedding of Fabric as a library in Rascal allows DSLs developed in the Rascal language workbench to be seamlessly gradualized, as well as internationalized. We have evaluated Fabric using two case studies: an idealized version of Hedy [16], a gradual programming language designed for education, and Rebel2 [31], a DSL for financial product modeling. Both case studies demonstrate the potential of the approach.

Future directions for research include: extending the internationalization features of Fabric to include adaptation of reading direction, and customization towards using non-Latin punctuation, such as the Arabic comma, and investigating how internationalization could be realized in visual languages. Finally, we would like to refactor the current Hedy implementation to use Fabric, to make the development and maintenance of its gradual, internationalized syntax easier and more reliable.

A Unravel algorithm

The top level unravel function is shown below. It receives two reified types, `ref`, and `fabric`, representing the reference grammar and the fabric grammar, respectively. The third parameter, `pt`, is the parse tree, that resulted from parsing using a localized grammar. The final parameter indicates the locale.

The function traverse the parse tree, and rewrites every encountered node for which the fabric grammar contains a matching production pattern. The `appl` constructor is the internal representation of Rascal parse trees, and is used to destructure the visited node. The expression after the ⇒ is unraveled using the helper function `unravelKids`, shown below.

```
Tree unravel(type[&T<:Tree] ref, type[&U<:Tree] fabric,
                Tree pt, str locale) {
  // bottom-up traversal of the parse tree
  return visit (pt) {
    // match a parse tree over the localized grammar
    case appl(prod(label(str l, sort(str nt)), _, _),
              list[Tree] kids) ⇒
    // & rewrite it to a  tree over the ref grammar
    appl(bp, unravelKids(bp.symbols, fp.symbols, kids))
      when
        // a mathing prod bp exists in ref-grammar
        /bp:prod(label(l, sort(nt)), _, _)
              := ref.definitions,
        // and locale prod fp exists in the fabric
        /fp:prod(label(l, sort(/<nt>_<locale>/)), _, _)
              := fabric.definitions
  }
}
```

Rascal code to unravel children (`kids`) of a parse tree obtained from a stitched grammar into parse tree children conforming to a reference grammar production (`ref`); `fab` represents the fabric production with placeholders used to stitch the grammar.

```
list[Tree] unravelKids(list[Symbol] ref,
  list[Symbol] fab,  list[Tree] kids) {
  int cur = 0; // child index in kids
  Tree nextLayout() { // find the next layout node
    while (cur < size(fab))
      if (isLit(fab[cur])) cur += 1; // skip lits
      else if (isLayout(fab[cur])) return kids[cur];
    return dummyLayout(); // if no more layout available
  }

  Tree nextAST() { // find the next AST node
    while (cur<size(fab), isLit(fab[cur])
           || isLayout(fab[cur]))
      cur += 1;
    return kids[cur];
  }

  // map maintaining reordering of AST child nodes
  map[int,int] shuffle = (i: 0 | i ←[0..size(ref)]);
  int idx = 0; // index in the future (ref) production
  list[Tree] newKids = [];

  // reconstruct kids according to ref
  while (idx < size(ref)) {
    if (isLit(ref[idx]))
      newKids += makeLitTree(ref[idx]);
    else if (isLayout(ref[idx]))
      newKids += nextLayout();
    else {
      newKids += nextAST();
      shuffle[idx] = placeholderPos(fab[cur]);
      cur += 1;
    }
    idx += 1;
  }

  list[Tree] shuffled = newKids;
  for (int i ←[0..size(newKids)])
    if (shuffle[i] > 0)
      shuffled[i] = astAt(newKids, shuffle[i] - 1);

  return shuffled;
}
```

B Dutch Hedy' fabric grammar

```
module HedyNL

modifies "hedy-nice.gradgram"
locale nl

level 1
command
  = ask: "vraag" _
  | turn: "draai" _
  | forward: "vooruit" _
  | print: "print" _ "uit"

level 2
command
  = sleep: "slaap" _
  | ask: _ "is" "vraag" _
  | color: "kleur" _
  | error_ask_dep_2: "vraag" _

color
  = black: "zwart" | blue: "blauw" | brown: "bruin"
  | gray: "grijs" | green: "groen" | orange: "oranje"
  | pink: "roze" | purple: "paars" | red: "rood"
  | white: "wit" | yellow: "geel"

level 3
command = add: "voeg" _ "aan" _ "toe"
  | remove: "verwijder" _ "uit" _

list_access = _ "op" (_ | "willekeurig")

level 4
command = error_ask_no_quotes: _ "is" "vraag" _

level 5
command_with_ifs
  = ifs: "als" _ "dan" _
  | ifelse: "als" _ "dan" _ " anders" _
  | list_access_var: _ "is" _ "op" _

level 6

level 7
command
  = repeat: "herhaal" _ "keer" _
  | error_repeat_no_command: "herhaal" _ "keer"
  | error_repeat_no_print: "herhaal" _ "keer" _
  | error_repeat_no_times: "herhaal" _ _

level 8
command
  = ifs: "als" _ "dan" "\n" _ "\n" _
  | ifelse: "als" _ "dan" "\n" _ "\n" _ "\n"
            "anders" "\n" _ "\n" _
  | repeat: "herhaal" _  "keer" _ "\n" _

level 9
level 10
command
  = "voor" _ "in" _ "\n" _ "\n" _
```

C Unraveled Rebel2 specification

The unraveled Rebel2 specification of Figure 9:

```
module Counter
spec Counter
  i: Integer;
  init event create() post: this.i' = 0;
  event inc() post: this.i' = this.i + 1;
  event dec() post: this.i' = this.i - 1;
  states:
    (*) -> active: create;
    active -> active: inc,dec;

config Simple = c: Counter is uninitialized;

assert Eventually3Later1 = eventually
  exists c:Counter | c.i = 3 && eventually c.i = 1;
run Eventually3Later1 from Simple in max 7 steps;

assert EventuallyAlwaysHigherThan3 = eventually
  always exists c:Counter | c.i > 3;
check EventuallyAlwaysHigherThan3 from Simple
  in max 6 steps with infinite trace;
```

References

[1] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen J. Vinju. 2015. Modular language implementation in Rascal - experience report. *Sci. Comput. Program.* 114 (2015), 7–19. https://doi.org/10.1016/j.scico.2015.11.003

[2] Walter Cazzola and Diego Mathias Olivares. 2016. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing* 4, 3 (July 2016), 404–415. https://doi.org/10.1109/TETC.2015.2446192

[3] Benoît Combemale, Olivier Barais, and Andreas Wortmann. 2017. Language Engineering with the GEMOC Studio. In *ICSA Workshop'17*. IEEE Computer Society, 189–191. https://doi.org/10.1109/ICSAW.2017.61

[4] William R. Cook. 2007. AppleScript. In *HOPL III* (San Diego, California). 1–1–1–21. https://doi.org/10.1145/1238844.1238845

[5] Marcus Crestani and Michael Sperber. 2010. Experience report: growing programming languages for beginning students. *ACM Sigplan Notices* 45, 9 (2010), 229–234.

[6] Juan de Lara and Esther Guerra. 2020. Multi-level Model Product Lines - Open and Closed Variability for Modelling Language Families. In *FASE'20 (LNCS)*, Heike Wehrheim and Jordi Cabot (Eds.), Vol. 12076. Springer, 161–181. https://doi.org/10.1007/978-3-030-45234-6_8

[7] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: a meta-language for modular and reusable development of DSLs. In *SLE'15*, Richard F. Paige, Davide Di Ruscio, and Markus Völter (Eds.). ACM, 25–36. https://doi.org/10.1145/2814251.2814252

[8] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the syntax barrier for novices. ACM, 208–212. https://doi.org/10.1145/1999747.1999807

[9] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd system - modular extensible compiler construction. *Sci. Comput. Program.* 69, 1-3 (2007), 14–26. https://doi.org/10.1016/j.scico.2007.02.003

[10] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007 Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[11] Alejandro Espinal, Camilo Vieira, and Valeria Guerrero-Bequis. 2022. Student ability and difficulties with transfer from a block-based programming language into other programming languages: a case study in Colombia. *Computer Science Education* 0, 0 (2022), 1–33. https://doi.org/10.1080/08993408.2022.2079867

[12] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: implement your language faster than the quick and dirty way. In *OOPSLA'10 Companion*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 307–309. https://doi.org/10.1145/1869542.1869625

[13] Maria Gouseti, Chiel Peters, and Tijs van der Storm. 2014. Extensible language implementation with object algebras (short paper). In *GPCE'14*. ACM, 25–28. https://doi.org/10.1145/2658761.2658765

[14] Les Hatton. 2004. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology* 46, 7 (2004), 465–472.

[15] Jan Heering, Paul Robert Hendrik Hendriks, Paul Klint, and Jan Rekers. 1989. The syntax definition formalism SDF—reference manual—. *ACM Sigplan Notices* 24, 11 (1989), 43–75.

[16] Felienne Hermans. 2020. Hedy: A Gradual Language for Programming Education. In *ICER'20*. Association for Computing Machinery, New York, NY, USA, 259–270. https://doi.org/10.1145/3372782.3406262

[17] Erik Hilsdale and Jim Hugunin. 2004. Advice weaving in AspectJ. In *AOSD'04*. 26–35.

[18] Pablo Inostroza and Tijs van der Storm. 2017. Modular interpreters with implicit context propagation. *Comput. Lang. Syst. Struct.* 48 (2017), 39–67. https://doi.org/10.1016/j.cl.2016.08.001

[19] Stephen C Johnson et al. 1975. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ.

[20] Paul Klint, Tijs van der Storm, and Vinju Jurgen. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM'09*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

[21] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: a framework for compositional development of domain specific languages. *Int. J. Softw. Tools Technol. Transf.* 12, 5 (2010), 353–372. https://doi.org/10.1007/s10009-010-0142-1

[22] Thomas Kühn, Walter Cazzola, Nicola Pirritano Giampietro, and Massimiliano Poggi. 2019. Piggyback IDE support for language product lines. In *SPLC'19*. ACM, 22:1–22:12. https://doi.org/10.1145/3336294.3336301

[23] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 16 (nov 2010), 15 pages. https://doi.org/10.1145/1868358.1868363

[24] Mauricio Verano Merino, Tom Beckmann, Tijs van der Storm, Robert Hirschfeld, and Jurgen J. Vinju. 2021. Getting grammars into shape for block-based editors. In *SLE'21*. ACM, 83–98. https://doi.org/10.1145/3486608.3486908

[25] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: the power of dynamic analysis. *ACM SIGPLAN Notices* 49, 10 (2014), 579–598.

[26] Yizhou Qian, Peilin Yan, and Mingke Zhou. 2019. Using Data to Understand Difficulties of Learning to Program: A Study with Chinese Middle School Students. In *CompEd'19* (Chengdu,Sichuan, China). New York, NY, USA, 185–191. https://doi.org/10.1145/3300115.3309521

[27] Damijan Rebernak, Marjan Mernik, Pedro Rangel Henriques, Daniela Carneiro da Cruz, and Maria João Varanda Pereira. 2006. Specifying Languages Using Aspect-oriented Approach: AspectLISA. *J. Comput. Inf. Technol.* 14, 4 (2006), 343–350. https://doi.org/10.2498/cit.2006.04.11

[28] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *SPLC'10 (LNCS)*, Vol. 6287. Springer, 77–91. https://doi.org/10.1007/978-3-642-15579-6_6

[29] August Schwerdfeger and Eric Van Wyk. 2009. Verifiable composition of deterministic grammars. In *PLDI'09*, Michael Hind and Amer Diwan (Eds.). ACM, 199–210. https://doi.org/10.1145/1542476.1542499

[30] Guy L. Steele. 2006. A Growable Language. In *Companion to OOPSLA'06* (Portland, Oregon, USA). Association for Computing Machinery, New York, NY, USA, 505. https://doi.org/10.1145/1176617.1176621

[31] Jouke Stoel, Tijs van der Storm, and Jurgen J. Vinju. 2021. Modeling with Mocking. In *ICST'21*. IEEE, 59–70. https://doi.org/10.1109/ICST49551.2021.00018

[32] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Comput. Lang. Syst. Struct.* 43 (2015), 1–40. https://doi.org/10.1016/j.cl.2015.02.001

[33] Mark van den Brand, Arie van Deursen, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. 2001. The Asf+Sdf Meta-Environment: a Component-Based Language Development Environment. *Electron. Notes Theor. Comput. Sci.* 44, 2 (2001), 3–8. https://doi.org/10.1016/S1571-0661(04)80917-4

[34] Tijs van der Storm, William R. Cook, and Alex Loh. 2014. The design and implementation of Object Grammars. *Sci. Comput. Program.* 96 (2014), 460–487. https://doi.org/10.1016/j.scico.2014.02.023

[35] Carlos Vega, Camilo Jiménez, and Jorge Villalobos. 2013. A scalable and incremental project-based learning approach for CS1/CS2 courses. *Education and Information Technologies* 18, 2 (June 2013), 309–329. https://doi.org/10.1007/s10639-012-9242-8

[36] Mauricio Verano Merino and Tijs van der Storm. 2020. Block-Based Syntax from Context-Free Grammars. In *SLE'20* (Virtual, USA) *(SLE 2020)*. ACM, New York, NY, USA, 283–295. https://doi.org/10.1145/3426425.3426948

[37] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *Onward! 2014*. ACM, 95–111. https://doi.org/10.1145/2661136.2661149

[38] Markus Voelter and Vaclav Pech. 2012. Language modularity with the MPS language workbench. In *ICSE'12*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 1449–1450. https://doi.org/10.1109/ICSE.2012.6227070

[39] Markus Voelter, Jos Warmer, and Bernd Kolb. 2015. Projecting a Modular Future. *IEEE Softw.* 32, 5 (2015), 46–52. https://doi.org/10.1109/MS.2014.103

[40] Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE Softw.* 31, 5 (2014), 35–43. https://doi.org/10.1109/MS.2014.100

[41] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Sci. Comput. Program.* 75, 1-2 (2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004