Delft University of Technology
Software Engineering Research Group
Technical Report Series

Pure and Declarative Syntax Definition: Paradise Lost and Regained

Lennart C. L. Kats, Eelco Visser, Guido Wachsmuth

Report TUD-SERG-2010-019a

TUD-SERG-2010-019a

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

This paper is a pre-print of:

Lennart C. L. Kats, Eelco Visser, Guido Wachsmuth. Pure and Declarative Syntax Definition: Paradise Lost and Regained. In Proceedings of Onward! 2010. ACM, 2010.

```
@inproceedings{KWV10,
  title = {Pure and Declarative Syntax Definition: Paradise Lost and Regained},
  author = {Lennart C. L. Kats and Eelco Visser and Guido Wachsmuth},
  year = {2010},
  booktitle = {Proceedings of Onward! 2010},
  publisher = {ACM},
}
```

Pure and Declarative Syntax Definition: Paradise Lost and Regained

Lennart C. L. Kats
Delft University of Technology
l.c.l.kats@tudelft.nl

Eelco Visser
Delft University of Technology
visser@acm.org

Guido Wachsmuth
Delft University of Technology
g.h.wachsmuth@tudelft.nl

Abstract

Syntax definitions are pervasive in modern software systems, and serve as the basis for language processing tools like parsers and compilers. Mainstream parser generators pose restrictions on syntax definitions that follow from their implementation algorithm. They hamper evolution, maintainability, and compositionality of syntax definitions. The pureness and declarativity of syntax definitions is lost. We analyze how these problems arise for different aspects of syntax definitions, discuss their consequences for language engineers, and show how the pure and declarative nature of syntax definitions can be regained.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory — Syntax; D.3.4 [*Programming Languages*]: Processors — Parsing; D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Design, Languages

Prologue

In the beginning were the *words*, and the words were *trees*, and the trees were words. All words were made through *grammars*, and without grammars was not any word made that was made. Those were the days of the garden of Eden. And there where language engineers strolling through the garden. They made languages which were sets of words by making grammars full of beauty. And with these grammars, they turned words into trees and trees into words. And the trees were natural, and pure, and beautiful, as were the grammars.

Among them were software engineers who made software as the language engineers made languages. And they dwelt with them and they were one people. The language engineers were software engineers and the software engineers were language engineers. And the language engineers made *language software*. They made *recognizers* to know words, and *generators* to make words, and *parsers* to turn words into trees, and *formatters* to turn trees into words.

But the software they made was not as natural, and pure, and beautiful as the grammars they made. So they made software to make language software and began to make language software by making *syntax definitions*. And the syntax definitions were grammars and grammars were syntax definitions. With their software, they turned syntax definitions into language software. And the syntax definitions were language software and language software were syntax definitions. And the syntax definitions were natural, and pure, and beautiful, as were the grammars.

The Fall Now the serpent was more crafty than any other beast of the field. He said to the language engineers,

> *Did you actually decide not to build any parsers?*

And the language engineers said to the serpent,

> *We build parsers, but we decided not to build others than general parsers, nor shall we try it, lest we loose our syntax definitions to be natural, and pure, and beautiful.*

But the serpent said to the language engineers,

> *You will not surely loose your syntax definitions to be natural, and pure, and beautiful. For you know that when you build particular parsers your benchmarks will be improved, and your parsers will be the best, running fast and efficient.*

So when the language engineers saw that restricted parsers were good for efficiency, and that they were a delight to the benchmarks, they made software to make efficient parsers and began to make efficient parsers by making *parser definitions*. Those days, the language engineers went out from the garden of Eden. In pain they made parser definitions all the days of their life. But the parser definitions were not grammars and grammars were not parser definitions. And by the sweat of their faces they turned parser definitions into effi-

Onward! 2010, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

cient parsers. But the parser definitions were not natural, nor pure, nor beautiful, as the grammars had been before.

The Plagues Their software was full of plagues. The first plague were *grammar classes*. Only few grammars could be turned directly into parser definitions. And language engineers massaged their grammars all the days of their life to make them fit into a grammar class. And the parser definitions became unnatural, and impure, and ugly. And there was weeping and mourning.

The second plague was *disambiguation*. Their new parsers were deterministic. So the language engineers encoded precedence in parser definitions. And the parser definitions became unnatural, and impure, and ugly.

The third plague was *lexical syntax*. The new software could not handle *lexical syntax definitions*. So the language engineers made another software to turn lexical syntax definitions into *scanners*. But lexical syntax definitions were less expressive than the grammars they used before. And they were separated from parser definitions, as were scanners from parsers. And there was weeping and wailing.

The fourth plague was *tree construction*. The language engineers wanted the efficient parsers to turn words into trees, as their old parsers did. So they added code to their parser definitions. And the parser definitions became unnatural, and impure, and ugly. And those who were oblivious to the working of the efficient parsers made parsers that turn the right words into the wrong trees.

The fifth and sixth plague were *evolution* and *composition*. Once the language engineers added a new rule to their parser definitions, those tended to break. And they massaged them by the sweat of their faces to make them fit again into the grammar class. And they were not able to compose two parser definitions to a single parser definition because of grammar classes and separate scanners. And there was weeping and groaning.

The seventh plague was the *restriction to parsers*. The language engineers turned parser definitions into recognizers and into parsers. But they could not turn them into generators or formatters. That was because parser definitions were not grammars.

Dedication Many have undertaken to compile a narrative of the things that have been accomplished among us. It seemed good to us also, having followed all things closely for some time past, to write an orderly account for you that you may have certainty concerning the things you have been taught. So this is the story about the loss of the garden of Eden and about the pain and the sweat and the plagues of parser definitions. But it is also the story about the promised land and about the naturalness and the pureness and the beauty of syntax definitions. And it is the story about the stiff-necked people of language engineers which ignores the promised land and sticks to the pain and the sweat and the plagues.

```
N → D N
N → D
D → "0"
D → "1"
```

Figure 1. A generative grammar for binary numbers.

So in this paper, we show the consequences of giving up the declarativity of natural syntax definitions. We show how practical issues and trade-offs have lead to grammars plagued with restrictions. We show how upholding and protecting pure and declarative syntax definition really can make grammars full of beauty. We show how no compromise must be made: stray even slightly from the straight and narrow path and fall afoul of the maintainability and usability of declarative syntax definition. We base our story on lessons learned from 20 years of experience with SDF [27, 52], a syntax definition formalism that has withstood the temptations of impurity and stayed true to the way of declarative syntax definition.

1. The Beauty of Grammars and Trees

Grammars are a simple yet powerful formalism. Most of their beauty comes from this simplicity of power. Let us discover this beauty from different perspectives.

Words were made through grammars. Chomsky emphasizes that a linguistic theory needs to provide *finite models* for the infinite productivity of language. The oldest of such models handed down to us is the Aṣṭādhyāyī [39, 40], a model of the morphology of Sanskrit. Written in the 4th century BC by Pāṇini, an Ancient Indian Sanskrit grammarian, it includes 3,959 rules for the generation of well-formed Sanskrit words.

In the 1950's, Chomsky formalized *generative grammars* [12] as a finite set of terminal symbols, non-terminal symbols, and production rules. Terminal symbols are the elementary building blocks words in the language are constructed from. Non-terminal symbols are syntactic variables used to generate words. Production rules specify which symbols can be used in place of a non-terminal. At their left-hand side they specify a non-terminal and at the right-hand side they specify the symbols it generates. We can read these rules as rewrite rules: the left-hand side of a rule can be rewritten to its right-hand side.

Consider the grammar given in Figure 1. It uses the terminal symbols `0` and `1` and the non-terminal symbols `D` and `N` to generate strings of binary numbers. For example, to generate the word `1 0`, we can start with symbol `N` and apply the rule `N → D N`, giving

```
D N
```

then `N → D` can be applied, giving

```
D D
```

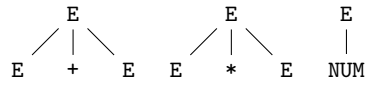
then `D → "1"` can be applied, giving

```
E → E "+" E
E → E "*" E
E → NUM
```

(a) In productive form.

```
E "+" E → E
E "*" E → E
NUM     → E
```

(b) In reductive form.



(c) As tree construction rules.

Figure 2. A grammar for arithmetic expressions

```
1 D
```

and finally `D → "0"` gives

```
1 0
```

Note that the same word can often be generated in multiple different ways. For example, a sequence of production rule applications of the form `N`; `D N`; `1 N`; `1 D`; `1 0` also generates the word `1 0`.

It is common practice in linguistics to discuss the structure of words (morphology) and the structure of sentences (syntax) separately. This practice found its way to computer science as well. Generative grammars are useful in both cases: in the same way that we can generate words from a vocabulary of letters (i.e., an alphabet), we can generate sentences from a vocabulary of words. Computer scientists are interested in both cases, but the stronger emphasis is on sentences made out of words. As such, we focus on sentences from now on and return to words made out of letters later. As an example of a generative grammar that focuses on sentences rather than words, consider Figure 2(a). It uses non-terminal symbol `E` and terminals `+` and `*`. The `NUM` symbol represents decimal numbers. At the sentence level, `NUM` is considered a terminal symbol, and the individual letters it is constructed from are ignored.

They made languages by making grammars. Grammars as proposed by Chomsky provide an intuitive, natural means to *describe* languages. A grammar is a perfect language description if it generates all the sentences of the language — and only these. Conversely, grammars can also be employed to *prescribe* new languages. In this case, a sentence is part of the language if it can be generated with the grammar. Otherwise, it is not. The grammar is the only truth.

So how can we determine if a sentence complies to a grammar, i.e. if it can be generated by a grammar? A naive approach would be to generate sentences until we find the one in question. A better approach is to consider grammars from a different perspective, no longer viewing them as a generative device, but as an oracle that determines if a sentence complies to it. To illustrate this view, Figure 2(b) shows our expression grammar again, now with the left-hand and right-hand sides rules switched. This emphasizes an alternative reading of the grammar rules as reduction rules: the symbols from the left-hand reduce to the non-terminal symbol of the right-hand side. Grammars in practical applications are written in one of the two forms depending on conventions and the grammar engineering software used. In the remainder of this paper we show grammars in the reductive form.

Relying on the reductive reading of a grammar, we can try to rewrite a sentence to a single non-terminal symbol. If this works, the sentence is part of the language, otherwise it is not. We start with a sentence and repeatedly interpret the reduction rules as rewrite rules:

```
3 * 4 + 5
```

given this sentence, we can apply `NUM → E` successively:

```
E * 4 + 5
E * E + 5
E * E + E
```

given the last sentence, we can apply `E "*" E → E`:

```
E + E
```

and finally reducing that to

```
E
```

Again note that multiple different ways can be used to reduce this sentence to a non-terminal symbol. The result is always `E`.

Generation and reduction are symmetric processes. If we can generate a sentence from a non-terminal symbol by applying a sequence of production rules, we can reduce this sentence to the non-terminal symbol by applying the same rules in reversed order — and vice versa.

They turned words into trees and trees into words. Often, we are not only interested in recognizing whether sentences are part of the language, but also in the grammatical structure of sentences. This structure can naturally be represented using trees. Even in primary school where we learned to recognize word classes like nouns and adjectives as well as higher-level concepts like clauses and phrases, ultimately trees were the basis for analyzing sentences in natural language. Each node of a tree corresponds to a syntactic construct, made up by the constructs represented by its child nodes. The leaves of the tree correspond to the terminals of the grammar, and the words in a sentence.

Like grammars, trees are simple yet powerful. We can combine these two beautiful formalisms by reading grammar production rules as tree construction rules. Figure 2(c) shows the grammar we know already from Figures 2(a) and 2(b). In this notation, production rules are represented as trees of depth 1. The leaves of each tree indicate the symbols that match the tree construction rule. Before, these symbols were rewritten simply to the non-terminal symbol at the root of

the tree. Now, they are rewritten to the tree representing the production rule.

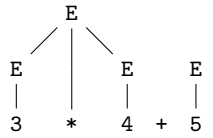
Let us see how this works for the sentence

```
3 * 4 + 5
```

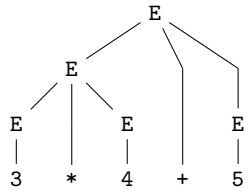
again, we apply the last production rule successively:

```
E                E    E          E    E    E
|                |    |          |    |    |
3 * 4 + 5;       3 * 4 + 5;     3 * 4 + 5
```

given the last tree, we can apply the multiplication rule:



and finally the addition rule:



Turning trees back into words is conceptually very simple. We just need to collect the terminal symbols at the leaves of a tree from left to right. This gives us the sentence represented by the tree structure. Things will become more complicated when we take layout into account, something we will discuss later.

They made software to make language software. Grammars can be employed in a variety of different ways. So far we discussed how grammars can be used to generate, recognize, and build trees from sentences. These and more tasks are automated by language software: *generators* produce sentences, *recognizers* decide wether a sentence is part of a language or not, *parsers* turn words into trees, and *formatters* turn trees into words. These components are essential for building compilers, interpreters, documentation generators, and other tools.

Since the formalisation of grammars by Chomsky, computer scientists came up with generic tools to turn grammars into language software. They made *syntax definitions* that encode a grammar for use by these tools. Syntax definitions can only be applied for the complete spectrum of language software if they maintain the virtue of *declarativity* that grammars provide. Grammars are declarative as they only describe *what* language software should do, not necessarily *how* to do it. This description does not have to be specifically written or altered with any one of the applications in mind.

Tools to construct language software have to address many practical issues, and, as we discuss in the following sections[1], there are many potential pitfalls where declarativity is lost due to imprudent design decisions. Many tool vendors are tempted to leave the righteous path of declarativity and incorporate various facets of the implementation of language software into syntax definitions. Unfortunately, by deviating from the path, the declarativity of grammars is lost. This loss always comes at a price, incurring limitations in the way syntax definitions can be used, or requiring additional effort in the specification of syntax definitions. An early example of the loss of declarativity is the Aṣṭādhyāyī: Though it is known as one of the most consolidated descriptions of human knowledge, the description is highly algorithmic and technical. Due to the focus on brevity, its structure is quite unintuitive, reminiscent of contemporary machine code.

2. Parser Generation

Parsers can be implemented by hand or generated using a parser generator. Handwritten parsers tend to use left-to-right scanning of the input, constructing a leftmost derivation of a parse tree. In contrast, generated parsers can use a variety of efficient algorithms that may not be easy to write by hand.

Grammar subclasses Most parsing algorithms work only for a subclass of the set of all context-free grammars, such as LL(1), LL(k), LR(1), LR(k), LALR(1), LALR(k), etc. We recall that the first L in these abbreviations stands for left-to-right scanning. Instead of keeping the whole sentence in memory, efficient parsers process them word by word. The constant k indicates the number of words of lookahead that is available. The second L in LL and the R in LR stand for a leftmost respectively rightmost derivation.

To prospective users, grammar class restrictions can seem quite arbitrary. For instance, LL grammars do not support recursion in the left-most symbol of production patterns [36]. This means that the expression grammar of Figure 2 is not supported. From an implementation point of view, the restrictions make sense for the algorithms used for these parsers. Left recursion in an LL-style recursive descent parser would lead to non-termination. However, from a usability point of view, they reveal a leaky abstraction: the implementation directs and restricts the way in which grammars can be written.

LL parsers In 1968, Lewis and Stearns [36] described a class of grammars that could be efficiently parsed using a simple top-down algorithm that constructs a left-most derivation. Variations of the algorithm are still in popular use today, notably in the ANTLR parser generator [41]. The distinguishing feature of the class of grammars supported by these LL parsers is that they do not support recursion in the left-most symbol of productions. While they do not support

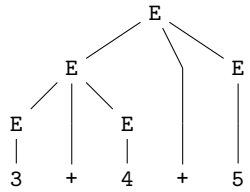
[1] Examples include handling ambiguous grammars, supporting layout and comments that may interleave syntactic constructs, constructing trees with a particular API, and ensuring acceptable performance.

```
Term ("+" Term)* → E
Fact ("*" Fact)* → Term
NUM              → Fact
```

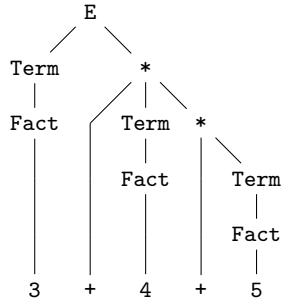
Figure 3. A left-factored expression grammar.

the natural grammar from Figure 2, language engineers can manually *left factor* the grammar, eliminating all left recursion, to allow it to be parsed using an LL parser. The resulting grammar[2] in Figure 3 no longer corresponds to the natural grammar and becomes harder to maintain. Consider for example what happens when new operators such as minus or modulo need to be introduced. Other forms of left recursion may be indirect, which makes it harder to pinpoint and solve the problem.

As the grammar loses its naturalness, so do the trees constructed by the parser. Consider the sentence `3 + 4 + 5`. Using the natural grammar, we can construct the following tree[3]:



This tree perfectly captures the left-associativity of the + operator. In contrast, the only possible tree according to the left-factored grammar from Figure 3 looks like this:



Here, the pattern `("+" Term)*` introduces right-associative structure for each repetition. By eliminating left recursion from the grammar, the tree structure does not capture the associativity of the operators. The tree no longer incorporates logical subtrees for the subexpressions `(3 + 4)` and `(3 + 4) + 5`. This complicates the work of compilers and other language processing tools that operate on the tree. Additional work is required to construct trees that have a more natural form, a topic we revisit in Section 5.

[2] Note that we employ the Kleene star (*) operator in this grammar to define repetitive syntactic structures more concisely.

[3] Note that our natural grammar can also be used to construct a right-associative tree for this input sentence. We discuss how to select the desired tree in Section 3.

LR parsers LR(k) parsers, introduced by Knuth in 1965 [35], can handle a strictly larger set of grammars than LL(k) parsers and are able to cope with left recursion. Unlike LL parsers, they can provisionally match multiple productions with the same left-hand side at the same time. Like LL parsers, they rely on a strictly deterministic algorithm. For every symbol that is consumed, the parser must decide to either consume more input (shift) or to apply a particular production rule (reduce). A state in which the parser cannot decide whether to shift or reduce is said to have a *shift/reduce conflict*; if it cannot decide which production to apply it has a *reduce/reduce conflict*. In order to employ LR parser generators, users have to eliminate such conflicts by factorization.

LALR parsing is a common variation of LR parsing that slightly restricts the set of supported grammars in order to allow for more efficient implementation [17]. Well-known LALR(1) parser generators are YACC and the very similar Bison[4]. A recent case study by Malloy et al. [37] gives an interesting insight into the typical development process of a Bison-based LALR parser for the C# language. The case study started with a grammar from the C# language definition, and described how over the course of 19 revisions it was painstakingly factored to eliminate all 40 shift/reduce and 617 reduce/reduce conflicts that were reported for it. According to Malloy et al., this is not a surprising number of conflicts for a grammar not specifically designed for the targeted grammar class. In their efforts to resolve all conflicts, they encountered various limitations of the parser: for example, using only 1 symbol for lookahead, their parser had problems distinguishing array types of the form `int[][]` and the form `[,]`. Another notable problem they ran into is the use of context-sensitive keywords: for example, the `add` keyword used for properties is not reserved in C#. In general, they had to make a large number of small changes for aspects of the original, natural grammar that were not supported by Bison.

Lookahead Using lookahead increases the recognition power of a parser. LL parsers are particularly dependent on lookahead [42]. Programmers can manually left-factor common prefixes of competing productions in order to decrease the required lookahead. However, at best this process leads to unnatural grammars since grammars using LL(k) or LR(k) with $k > 1$ are often more natural than with $k = 1$ [42]. At worst, it may be insufficient as there are languages that are LL(k) but not LL($k - 1$) [21]. While traditionally the importance of the lower memory consumption and performance overhead of LL(1) and LR(1) was emphasized, modern parser generators such as ANTLR can automatically select an appropriate lookahead k for a given grammar.

[4] In fact, on many systems `yacc` is an alias of `bison`.

Consequences of restricted grammar classes Based on parsing algorithms such as LL and LR, which only support a subset of all context-free grammars, language engineers are forced to focus on the accidental complexity of the inner workings of parser implementation. Instead of focusing on language design, they have to get absorbed into the idiosyncrasies of parsing algorithms. Factorization and "massaging" of syntax definitions leads to specifications that have little correspondence to the high-level declarative description of a natural grammar for the language. It leads to a loss of *obliviousness*: a property used in the aspect-oriented programming community [20] that says that software engineers should be able to reason about something while being unconcerned or even unaware about the implementation details. Without obliviousness, language engineers are forced to write *parser definitions* instead of declarative syntax definitions, describing the parsing process instead of the language while destroying declarativity of natural grammars. *Paradise lost* indeed.

Parser definitions are a step back from natural syntax definitions. To maintain the declarativity of natural grammars and to ensure obliviousness in language design, syntax definitions should be freed from the shackles of restricted grammar classes. Only parser generators that support the *full class* of context-free grammars should be used.

Generalized parsers In order to keep to the straight and narrow path of declarative syntax definition, generalized parsing algorithms that extend or substitute LL and LR must be used. Parsing algorithms that can handle the full class of context-free grammars instead of being restricted to a particular subclass. Unfortunately, a naive implementation of such a parsing algorithm using backtracking risks exponential execution time.

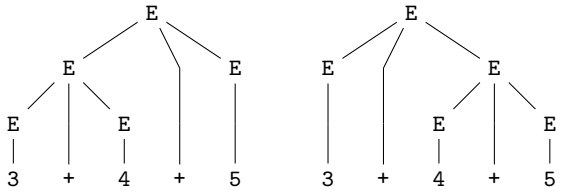
The CYK algorithm, independently proposed by Cocke, Younger, and Kasami [13, 30, 54] in the 1960s, was historically the first generalized parsing algorithm that operated in polynomial (cubic) time. The algorithm required grammars to be written in or converted to Chomsky normal form. Another algorithm introduced in the late 1960s was Earley's algorithm [18], which did not require this normal form. In 1985, Tomita designed a generalized form of LR parsing [48], of which many variations and improvements have been developed since [29, 43, 49].

Of these, Tomita's generalized LR (GLR) has the attractive property that it runs in linear time for unambiguous grammars and gracefully copes with ambiguities. In practice, most programming languages have few or no ambiguities, ensuring good performance with a GLR parser. A recent variation of GLR is generalized LL [46], which acts much like a recursive descent parser but uses GLR machinery to handle ambiguities. It achieves linear execution time for LL grammars, and cubic execution time in the worst case.

Parser generators based on generalized parsing algorithms have a number of major advantages. They can construct a working parser for *any* context-free grammar, and do not require massaging, ensuring that the syntax definition reflects the natural, intended structure of the language, ensuring maintainability and preservation of obliviousness. Furthermore, the full class of context-free grammars is closed under composition, allowing for modular grammars and reuse, a topic we revisit in Section 6.

3. Ambiguity Handling

Some grammars allow us to turn the same words into different trees. Such grammars are called *ambiguous*. The expression grammar from Figure 2 is ambiguous. For instance, for the sentence `3 + 4 + 5`, there are two possible trees:



In this section we discuss different ways of handling ambiguities and their effects on the declarativity of syntax definitions. We first revisit deterministic parsing algorithms such as LL and LR, where ambiguities result in conflicts. Next, we discuss parsing expression grammars, a class of grammars that by definition does not allow ambiguous grammars. Finally, we discuss how ambiguity is *embraced* by generalized parsing algorithms. We then show how to amend an ambiguous grammar with disambiguation rules in a way that preserves the naturalness and the declarativity of syntax definitions.

Ambiguity as conflicts LL and LR parsers are deterministic parsers: they can only return one parse tree for a given string. This means that they cannot handle ambiguous grammars. Detecting whether a context-free grammar is ambiguous is a classical undecidable problem in formal language theory [10, 24]. However, based on the restrictions of the LL and LR grammar classes, conflicts can give an indication of ambiguity. Tools can statically tell if there is a conflict. Absence of such conflicts indicates an unambiguous grammar.

The restrictions posed by LL and LR parsers are far from a panacea for ambiguity handling. For one thing, they are restricted to only a subset of the unambiguous grammars. Grammars in these classes also do not enjoy good closure properties: often, many new, non-local conflicts are introduced when modifying the grammar. This can make modifications tedious as a grammar evolves. Few reported conflicts actually indicate ambiguities. Recall the over 600 conflicts encountered in the case study by Malloy et al. [37]. Each had to be resolved by hand. In general, to address all conflicts can require significant changes to a parser definition to the point that the original structure is lost and the definition becomes harder to maintain.

```
E    "+" Term → E
Term          → E
Term "*" Fact → Term
Fact          → Term
NUM           → Fact
```

Figure 4. Encoding of operator precedence in grammar productions.

One way to resolve ambiguities in grammars is by encoding precedence and associativity directly in the productions (Figure 4). Precedences can be encoded by introducing an extra level of indirection, and associativity can be enforced by restricting the production patterns. Unfortunately, by encoding precedences and operator associativity directly in the grammar, we lose the declarativity and conciseness of the natural grammar in Figure 2.

Ambiguity ignored A recent addition to the spectrum of parser generators is that of packrat parsers [22]. Like LL parsers, these belong to the family of recursive descent parsers, but they implement a form of backtracking by means of memoization, allowing them to consider multiple candidate productions rather than just one with LL.

Packrat parsers are based on parsing expression grammars (PEGs) [23], and are an adaptation of the TS formalism originally conceived in the 1970s [3]. Parsing expression grammars are a distinct class of grammars: they cannot be used to express all context-free grammars (CFGs), but are also not a strict subset of this class. PEGs are based on greedy matching of repetitions and the idea of a strict ordering between all productions: the first alternative that matches "wins." As such, they use an ordered choice operator (`/`) instead of the unordered choice operator (`|`) of BNF. By virtue of disallowing any kind of non-deterministic choice, ambiguity is ignored and effectively *defined away* in PEGs.

Grammars with disjoint production patterns describe the same language in both PEGs and CFGs. For non-disjoint patterns, this is not the case: in CFGs, $ab|a \rightarrow A$ and $a|ab \rightarrow A$, as well as $a \rightarrow A$ $ab \rightarrow A$, describe the language $\{ab, a\}$. However, for PEGs, $ab/a \rightarrow A$ describes $\{ab, a\}$, whereas $a/ab \rightarrow A$ describes $\{a\}$. Detection of disjointness of productions is undecidable [23].

As an example, consider the `if` statement in a context-free grammar:

```
"if" E "then" E          → E
"if" E "then" E "else" E → E
```

which has the notorious "dangling else" ambiguity: for a nested `if` statement, it is not clear whether the else clause belongs to the inner or the outer `if`. In PEGs, production ordering forces one alternative to be selected:

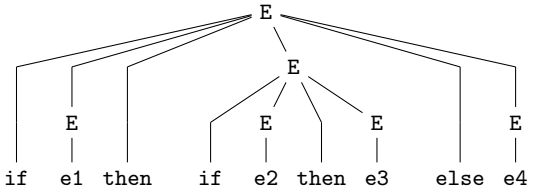
```
"if" E "then" E "else" E /
"if" E "then" E          → E
```

causing the outer `if` to "win." Programmers may not always be aware of the effects of such orderings, especially for larger, modular grammars with injection productions. Incorrect orderings can cause subtle errors, where a tree different from the intended tree is selected. No conflicts are reported for such grammars, and the alternative trees are never shown. Errors in orderings can also cause parse errors at runtime, as we show next. Consider the following grammar:

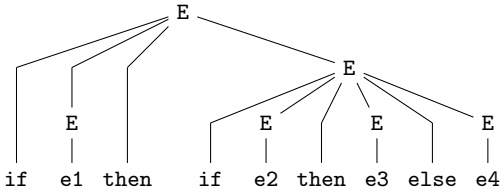
```
"if" E "then" E          /
"if" E "then" E "else" E → E
```

Intuitively, this variation would now cause the *inner* `if` to "win." However, because the prefix of a statement of the form `if e1 then e2 else e3` matches the first production, the second is never considered. A parse error is then reported at the `else` keyword, which can be confusing to programmers that are oblivious to the subtle semantics of PEGs that are essential for efficient packrat parsing.

Ambiguity embraced Generalized parsers take a fundamentally different approach to ambiguity handling. They can handle the full class of context-free grammars, including those that are ambiguous. As such, they can return all possible derivations for an input: a *parse forest* rather than a parse tree. This parse forest can be used to visualize ambiguities that may not be obvious from mere inspection of a grammar. As an example, for a statement `if e1 then if e2 then e3 else e4`, there are two possible interpretations:



and



For each ambiguity in an input sentence, the parse forest branches at that point, combining all possible trees into a single, larger forest, as shown in Figure 5. To some, the notion of parse forests can seem intimidating, as something that adds to the complexity of language engineering. We argue that parse forests *reduce* the complexity, by showing ambiguities in terms of the grammar, rather than in terms of the implementing algorithm. Language engineers can remain oblivious about the implementation of a parser, and only deal with grammars and trees. Using a textual or visual representation of the parse forest, language engineers can simply inspect any ambiguity and decide which is the intended tree.

Disambiguation rules As early as 1975, Aho and Johnson recognized [1] that the most natural grammar of a language is often not accepted by the parser generators that are

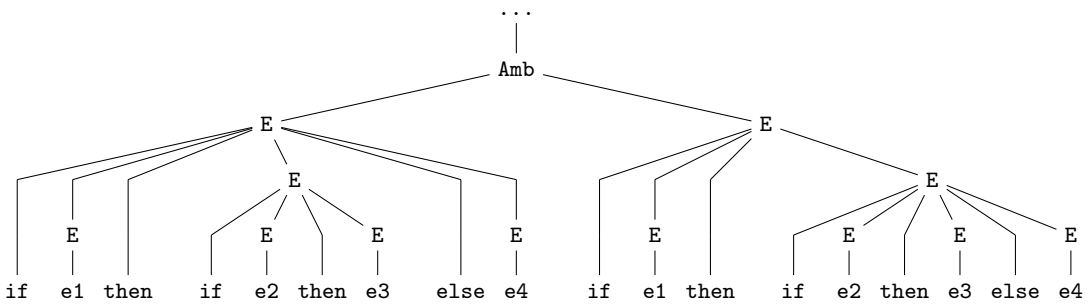


Figure 5. A parse forest for the "dangling else" ambiguity. Note that parse forests can be efficiently represented in memory by sharing identical subtrees.

used in practice, since the grammar does not fall in the subclass of context-free grammars for which the generator can efficiently produce a parser. They proposed to define languages using an ambiguous grammar instead – indeed, embracing ambiguity – supplemented with a set of *disambiguation rules* that are consulted to resolve parsing conflicts. This approach was first implemented in the YACC [28] parser generator.

Unfortunately, most of the work on disambiguation rules has been guided by parser implementation algorithms [5]. Disambiguation rules were simply a way to control (shift/reduce) actions of the parser. This resulted in an efficient implementation, but also resulted in very subtle, sometimes unexpected semantics. To use them, language engineers still needed to understand the underlying implementation. Once again, obliviousness was lost. Therefore, in practice, many language specifications tend to encode precedence rules directly in grammars instead, to ensure the semantics are clear and well-understood [5].

Disambiguation rules are only effective if they are designed from the ground up to be declarative and natural of form. Any premature optimization should be avoided and obliviousness must be maintained. To avoid tie-in to parsing algorithms, they can be implemented as a generic filtering mechanism that simply picks the intended tree from a parse forest. Then, and only then, should optimizations be considered for a particular algorithm. For the disambiguation rules we describe here, it turns out most can be encoded in a generated GLR parse table, eliminating the overhead of post-parse filtering [50].

Declarative disambiguation rules can take different forms [34, 44, 45], such as follow restrictions (a form of longest match), exclusion/reject rules, priority and associativity rules, and preference attributes for selecting a default among several alternatives. As an example, the "dangling else" ambiguity can be resolved in SDF by placing a preference attribute on either production [34]:

```
"if" E "then" E          → E {prefer}
"if" E "then" E "else" E → E
```

Similarly, we can disambiguate the grammar of Figure 2 using the following priority rule and associativity annotations:

```
E "*" E → E {left} >
E "+" E → E {left}
```

which specifies that the multiplication rule has priority over the addition rule, and that the operators should be treated as left-associative.

By *embracing* ambiguity, any undesired ambiguities that are not captured by ambiguity rules can be addressed through other means, such as non-context-free disambiguation strategies. For example, a fall-back disambiguation strategy can be to simply select the first alternative (much like the strategy of packrat parsers). Another strategy can be to employ semantic information for disambiguation, as seen in [4] for the C language.

Testing Standard software engineering practices are essential for creating robust parsers [33]. Version management and automated testing are particularly important. Grammar testing has an important role in all three approaches to ambiguity handling: testing against regressions as a grammar is factored to fit a certain class (as described for C# in [37]), testing against inconsistencies because of possible ordering errors in PEGs, and direct testing against ambiguities with generalized parsers.

Only using generalized parsers can ambiguities in failing tests be visualized and explicitly resolved. They are also the only class of parsers that maintain the declarativity of natural grammars when handling ambiguities, by separating the concern of disambiguation into separate rules and annotations.

4. From Terminals to Lexical Syntax

So far we have focused on parsing sentences, in the form of sequences of terminals. For practical applications, we also need to consider the individual characters associated with each terminal. A lexical syntax definition describes the structure of terminals.

```
[\ \t\n]          → LAYOUT
"//" ~[\n]* [\n]  → LAYOUT
```

Figure 6. Lexical syntax for whitespace and comments.

Lexical syntax is often specified using regular grammars. As such, regular expressions form the basis of lexical syntax specifications. For example

```
[0-9]+ -> NUM
```

specifies the lexical production for integer numbers.

Lexical syntax is also used to recognize whitespace and comments. These are not considered by the productions of the context-free syntax. As an example, consider an expression with a C++-style comment:

```
3 * // 4 +
5
```

When parsed using the context-free syntax of Figure 2, only the terminals `3*5` are considered. Whitespace and comments are only relevant for the lexical syntax of the language. Figure 6 defines the lexical syntax of these constructs.

Scanners and parsers Character-level grammars often require arbitrary length lookahead to parse, as non-terminals can be separated by arbitrary length layout and comments. This means that conventional LL(1), LR(1) or even LR(k) parsers cannot cope with grammars that incorporate a lexical syntax definition [49]. Instead, conventional parser implementations separate the processing of lexical and context-free syntax into a separate scanner and parser. The scanner (or "lexer") then uses regular grammars to tokenize the input, and the parser operates on these tokens.

Separating lexical and context-free analysis poses a number of restrictions on syntax definitions. First, it requires two different syntax definition languages, with different expressive power (e.g., Lex and YACC). Using only regular grammars, constructs such as nested comments cannot be expressed. Another restriction is that a separate scanner cannot consider the context in which a token occurs. For example, consider a sentence `"array [1..10] of integer"` in Pascal, a language that to some degree was designed to be easy to parse. The range `1..10` can be tokenized either as the real `1.` followed by the real `.10`, or as an integer `1` followed by the range operator `..` and another integer. Similarly, the modern C# language has several non-reserved keywords that can be used as an identifier in some contexts. Only based on the syntactic context could a scanner decide which to select. One solution is to introduce lexical state, letting the scanner and parser interact (as done with the infamous "lexer hack" for parsing C), but this complicates the implementation and is usually detrimental to the declarativity of syntax definitions.

Scannerless parsers A scannerless parser [44, 45] eliminates the need for a separate scanner to tokenize the input. Instead, it parses directly at the level of characters. Scannerless parsers use a single syntax definition for both lexical and context-free syntax, and can be used to parse languages with a sophisticated lexical syntax such as C [4] and AspectJ [9], without resorting to hacks. Scannerless parsers can only be implemented using generalized parsing algorithms. Scannerless GLR (SGLR) [49, 51] combines scannerless parsing with GLR parsing, and is used for parsing SDF. Other scannerless parsers include the parser of TXL [14] (based on recursive descent backtrack parsing), and various implementations of packrat parsing [22].

5. From Parse Trees to Abstract Syntax Trees

So far, we have considered parse trees to represent tree structures. Abstract syntax trees abstract over these trees, hiding details such as whitespace, grouping parentheses, and keywords. Only elements with semantic value are maintained. For example, for the "if" construct of Section 3, a tree node can be constructed with only the condition and the two branches as its children, ignoring any layout, comments, and literals such as `if` which are part of the parse tree. Abstract syntax trees are useful for subsequent processing of the input, such as semantic analysis and transformations.

Semantic actions One way of constructing an abstract syntax tree is by use of *semantic actions*. Semantic actions are functions that are called once a production successfully matches (or sometimes before that point). They can be used to construct an abstract syntax tree by calling tree construction functions. They can encode properties of the abstract syntax tree that are not encoded by the grammar. For example, they can use functions to construct left-associative data structures for the `+` operator from the left-factored grammar of Figure 3.

Liberal use of semantic actions can constrain the implementation of parsing algorithms. Semantic actions are bound to a particular implementation language (e.g., C in YACC), reducing the portability of grammars. Semantic actions can include side effects, which means that they can become dependent on a particular evaluation order. Users oblivious to the way such an algorithm is implemented can be caught by surprise when an implementation does not evaluate the functions in the order they would expect. Any changes to the parsing algorithm – such as optimization, generalization, or adding error recovery – has the potential to break existing grammars that depend on semantic actions. Semantic actions also tie a grammar to syntax recognition, whereas a declarative grammar can also be used for other purposes such as pretty printing.

While syntax tree construction is a common reason for parser generators to support semantic actions, another use case is disambiguation. Predicate functions can be used to resolve conflict states in a parser algorithm. However, compared to declarative disambiguation rules, they have the same disadvantages as other semantic actions. Rather than

```
module Expressions
exports context-free syntax
  E "+" E                   → E {cons("Plus")}
  E "*" E                   → E {cons("Mul")}
  NUM                       → E {cons("Num")}
  "(" E ")"                 → E {bracket}
  "if" E "then" E           → E {cons("If"),prefer}
  "if" E "then" E "else" E  → E {cons("IfElse")}

context-free priorities
  E "*" E → E {left} >
  E "+" E → E {left}

lexical syntax
  [0-9]+            → NUM
  [\ \t\n]          → LAYOUT
  "//" ~[\n]* [\n]  → LAYOUT
```

Figure 7. The full SDF grammar for the expression language.

"polluting" declarative grammars with semantic actions, we argue that declarative disambiguation rules should be used to address this concern for context-free languages. For other, non-context-free languages such as C++, semantic postprocessing can be used, separating the syntax definition from the semantics.

Declarative tree construction An alternative to full-fledged semantic actions is to add small, declarative annotations with information for constructing abstract syntax trees. In SDF, productions can be annotated with a $\{\texttt{cons}(n)\}$ annotation to map a production to a node constructor n. Productions for grouping parentheses can use the $\{\texttt{bracket}\}$ annotation, and lexical productions and injections do not need to be annotated. Figure 7 shows the full SDF grammar for our expression language as given so far, including priorities, lexical syntax, and constructor annotations. It also introduces a new production for parenthesized expressions.

Based on the constructors in the language, an abstract syntax tree of the following form can be created for SDF grammars:

```
t ::= "..."      // terminals
    | c(t1,...,tn) // constructor applications
    | [t1,...,tn]  // lists of terms
```

By including only terminals, constructor applications, and lists, only those elements that have semantic value are included in the tree. As an example, an expression `3*4+5` in abstract syntax has the form

```
Plus(Mul("3","4"),"5")
```

and an expression `3*(4+5)` becomes

```
Mul("3",Plus("4","5"))
```

Declarative tree construction is most effective for grammars with a natural structure. The usefulness of these semi-automatically constructed trees decreases as grammars are massaged to fit a certain grammar class (Figure 3) or are changed to encode precedence and associativity (Figure 4).

Declarative tree construction does not dictate a particular implementation technology. For SGLR, a simple tree walker is used that creates abstract syntax tree nodes that are efficiently stored using the ATerm library [6]. Other implementations can use their own tree walker or factory class (when using the Java version of SGLR [47]) that transforms the parse tree to an abstract syntax tree using the annotations on the productions.

6. Language Evolution and Composition

Languages evolve over time [19], based on new domain insights, new applications, and new technological developments. Syntax definitions should be flexible and allow for change and reuse.

A typical example of language evolution is language extension. Languages can be extended to meet new requirements. For instance, we may want to add comparison expressions to our expression language:

```
E "==" E → E
E "<"  E → E
E ">"  E → E
```

When using a conventional parser generator, actually integrating these constructs to the parser definition requires language engineers to factorize the productions to fit the grammar class. The new expressions also have to be integrated into the existing sequence of encoded priorities, and have to be restricted to encode associativity.

Unfortunately, when extending a grammar G_1, even a fully factorized and massaged grammar extension G_2 can cause conflicts when added together: $G_1 \cup G_2$ may have additional overlapping left-hand sides and other conflicts. These problems arise as subclasses of the context-free grammars such as LL(k) or LR(k) are not closed under composition. More often than not, compositions need to be massaged to conform to the class again. In some cases, it may not even be possible to express the combined language using the grammar subclass. These problems make parser definitions rather fragile artifacts.

Separate scanners Using a separate scanner and parser increases problems with extensibility and compositionality of parsers. Tokens introduced by language extensions are added to the global set of tokens for the grammar. This can lead to conflicts. These conflicts are particularly prevalent for larger extensions or combinations of languages. For example, extending the Java language with (AspectJ) aspects is simply not possible with a stateless scanner [9].

Separate scanners do not always lead to full-blown breaking problems. Introducing new tokens can also simply mean that they are reserved from being used as an identifier. Consider for example the `enum` keyword introduced in Java 5. Before it was introduced, it could be used as an identifier. In fact, as it expresses a useful programming concept, its oc-

```
public boolean authenticate(String user, String pw) {
    SQL stm = <| SELECT id FROM Users
                 WHERE name     = ${user}
                 AND   password = ${pw} |>;
    return executeQuery(stm).size() != 0;
}
```

Figure 8. An extension of Java with SQL queries, adapted from [8].

```
module Java-SQL
imports Java SQL
exports context-free syntax
  "<|" Query "|>" → Expr    {cons("ToSQL")}
  "${" Expr  "}"  → SqlExpr {cons("FromSQL")}
```

Figure 9. Syntax of Java with embedded SQL queries, adapted from [8].

currence was not rare in Java programs. Still, because Java is parsed using a conventional, stateless scanner, the language designers had to concede and reserve keywords, breaking backward compatibility. Using a scannerless parser instead, they could simply use these keywords in the declarative syntax definition without it implicitly causing such harmful side effects. Rather, any uses of `enum` as identifier could be deprecated and phased out as desired.

Using disambiguation rules, keywords can be explicitly marked reserved when needed:

```
"if" | "then" | "else" -> ID {reject}
```

Composite and embedded languages Declarative syntax definitions can be reused by grouping reusable productions into modules. Based on modules, we can even build complete, reusable language components. Using declarative syntax definitions, we can combine independent languages, such as Java and SQL. Consider Figure 8, which shows such an embedding. In this Java method, SQL is embedded using `<|` ... `|>` brackets. In turn, Java expressions are embedded in the SQL query using `${` ... `}` brackets. Figure 9 defines the syntax for this composite language by importing the definitions of Java and SQL and adding additional productions for the embedding constructs.

Unfortunately, when combining entire modules of languages, the composition problems of conventional parser generators are greatly amplified. The entire, combined language must then be factorized to fit into a grammar subclass. Likewise, a stateless scanner for the grammar of Figure 9 would run into problems with the added `<|` ... `|>` brackets, which in standard Java would be recognized as comparison signs and pipes. A stateless scanner would also reserve all keywords in both languages, including keywords such as `SELECT` in Java or `this` in SQL. Another challenge to the definition of a composite scanner is that identifiers in both languages have subtly different definitions, but only one can be supported by a combined scanner.

7. Error Recovery

To use a language in an interactive development environment (IDE), a parser with error recovery is essential. Using error recovery, parsers can create a (partial) abstract syntax tree for incomplete or erroneous inputs. Such inputs are common when programmers are actively editing a file. Using the (partial) tree, IDEs can still provide editor services such as the error markers and content completion, even when programs are not in a syntactically valid state.

LL, LR, and GLR parsers have the *valid prefix property*, which means that the prefix string parsed at any point can always form a syntactically valid program. When a syntax error is encountered, the state of the parser at the point of failure can be used to recover from the error. A common approach to error recovery is to insert or delete symbols at or near the point of failure, in order to return the parser in a syntactically correct state and continue parsing. Sometimes, the parse failure is caused by a mistake in the prefix – maybe an extra closing bracket that closed a method prematurely – which means that symbols have to be inserted or deleted in the prefix.

Using a declarative syntax definition is essential for flexibility in the implementation of error recovery. For example, a backtracking algorithm may be added to a parser, which inserts and deletes symbols at a point earlier in the input. If syntax definitions use semantic actions that control the behavior of the parser or manipulate the global state, such an approach is not viable.

Automated error recovery can often be improved with help of the language engineer. Some parser generators use semantic actions as an imperative "exception handling" mechanism in parser definitions. While very flexible, this ties syntax definitions that use them to a particular platform and implementation algorithm. Worse yet, is also destroys obliviousness.

A declarative approach to describe error recovery strategies is by using error recovery rules [2, 26]. Using automated analysis of a grammar it is even possible to derive such rules [11, 16, 31]. Derivation of rules is based on recognizing typical programming language patterns, such as scope structures or string literals, for which recovery rules can be formulated. For such an analysis to work, it is essential that a syntax definition uses a declarative formalism, free from semantic actions that influence its meaning. The grammar should also have a natural structure that makes it easier to use heuristics to derive recovery rules.

8. Beyond Parsers

So far we focused on grammars as a way to declaratively construct parsers. Indeed, as a software artifact, grammars are primarily used to construct parsers. However, as a declarative formalism, grammars can also be used for other applications.

Complementary to parsing, grammars can also be used for *unparsing* or pretty printing of trees. Using a formatting language such as the Box language [7], parse trees and abstract syntax trees can be formatted and printed according to a set of pretty printing rules. Using a declarative syntax definition, such rules can be automatically derived [15]. Automatically derived rules can be adapted by hand or composed with handwritten rules for improved results.

Similar to unparsing, *sentence generation* is a technique for constructing sentences using just the grammar of a language, following all paths or a selection of paths from a nonterminal symbol. One use case of generated sentences is automated testing, as done with the DGL tool [38]. Generated tests can ensure coverage of language features in processing tools such as code generators. Sentence generation can also be used for detecting ambiguities in grammars [24, 25].

Languages, tools, and frameworks for language processing tools generally operate on the tree structure defined by a syntax definition. Using a declarative syntax definition, it is possible to automatically generate statically typed classes, data structures, and type signatures for use in different tools. This may not be feasible if the grammar is encoded in a parser definition for a particular parser generator. Some meta-programming languages and syntax-aware template engines can also embed the concrete syntax of a language, to allow for concise specifications of transformation and code generation [53].

In IDEs we can distinguish syntactic and semantic editor services. Syntactic editor services are based purely on the syntax of a language, and are closely tied to its definition, whereas semantic services typically use a description of the semantics of a language defined using a meta-programming language or framework. Examples of syntactic editor services are syntax highlighting, the outline view, code folding, and syntactic code completion. These services can be declaratively described using annotations in the grammar or using separate descriptor languages. Based on heuristic analysis of the structure of a grammar, default specifications can be automatically derived [32]. Similar to derived error recovery and pretty printing rules, this works best for a natural grammar. Derived specifications can be adapted and composed with handwritten rules.

9. Discussion

In the preceding sections, we have shown how pure and declarative syntax definitions have significant advantages over parser definitions. Generalized parsing algorithms can be used to parse such syntax definitions, supporting the full class of context-free grammars and preserving obliviousness about the parser implementation. They can smoothly handle ambiguity by building parse forests instead of trees, while declarative disambiguation rules can be specified separately rather than requiring the syntax to be changed. Scannerless parsing algorithms allow for the seamless integration of lexical and context-free syntax. Declarative annotations on productions can be used to construct abstract syntax trees without polluting syntax definitions with imperative semantic actions that rely on specific parser implementations. Modular syntax definitions ease language evolution and composition. Pure and declarative syntax definitions also allow error recovery rules to be derived or to be specified as separate rules. Finally, unlike parser definitions, declarative syntax definitions are not restricted to parser generation but can be used to generate a variety of software artifacts, such as pretty-printers, sentence generators, and IDE services.

Despite their advantages, techniques that support declarative syntax definition without restriction have not yet become mainstream. Most current parser generators still use restricted grammar classes, separate scanners, encoded precedence and associativity, and semantic actions. Only a few parser generators allow truly declarative syntax definitions. SGLR for the modular syntax definition SDF is one of these, and may be the most used scannerless GLR implementation.

Adoption The rather unfortunate state of the practice is that general parsing technology and declarative syntax definitions have seen a lack of adoption by both users and developers of parser generators.

In part, practical issues can be blamed for lack of adoption by prospective users. Bravenboer et al. [9] recently analyzed some of these issues. First, the implementation of SGLR was in C, targeting the Unix/Linux platform. With the development of JSGLR [47], this issue has recently been addressed. A second issue they listed was the lack of good syntax error handling. This issue has also been recently addressed, supporting error recovery and integrating SGLR into the Eclipse platform [16, 31, 47]. A third issue they listed was the lack of tool support for analyzing ambiguities, which has unfortunately not yet been addressed for scannerless GLR. Finally, they mentioned that the syntax of productions in SDF may be awkward and unappealing to developers used to BNF-style rules. This is certainly something to consider for a potential successor to SDF.

Tool developers have focused for the most part on LL and LR parser generators. These algorithms are reasonably simple to implement, and do no require the investment in time and effort that general algorithms such as (S)GLR or Earley do. This allows tool developers to focus on peripheral issues, such as supporting different languages, integrating into meta-programming frameworks, and providing tool support. While these are all important areas – indeed, SGLR adoption may have suffered from a lack of attention in this area – only adopting general parsing algorithms can truly change the way these tools are used.

Education Even LL, LR, and LALR are not simple algorithms that can be easily explained to the lay programmer. Implementations of these algorithms, generated by parser generators, can be quite large to the point of being intimidating. Still, to eliminate left recursion, reduce lookahead,

and resolve shift/reduce and reduce/reduce conflicts, a certain level of understanding of these algorithms is *compulsory*. Not all language engineers fully understand – or even aspire to understand – what the real meaning of these issues is. More often than not parser implementation is a process of trial-and-error: users simply torture the parser definition until it confesses.

The requirement of understanding the parsing process for working with parser generators has lead to the misunderstanding that simpler algorithms are easier to use. Writing a parser in recursive descent style is still doable, and even straightforward. Then why not use an LL parser generator that automates this? Why would one use a LALR parser generator that parses programs in bottomup order based on some unfathomable compressed parse table? Let alone a more complicated algorithm that can handle larger grammar classes?

Current computer science curricula tend to focus on the implementation of parsing algorithms. Students implement their own LL(1) or LR(1) parser and are familiarized with tools such as ANTLR and YACC that give an order of magnitude in productivity gain over manual implementations. At the same time, they reinforce the line of thinking that programmers need to understand the inner workings of parsers to work with grammars. Instead, we feel courses should focus on language engineering topics as described in [33] and grammar design involving modularity, composition, and ambiguity.

Epilogue

Since the days language engineers had left the garden of Eden, they carried heavy burdens. When they built parsers they suffered from the plagues of parser definitions. They were slaves to parser generators and their lives were bitter with turning grammars into parser definitions and parser definitions into parsers. And the people of language engineers groaned because of their slavery and cried out for help.

The promised land The advent of *scannerless, generalized parsing* delivered language engineers out of the slavery to parser generators. It brought them up out of that to a good and broad land, a land flowing with milk and honey.

In this land, there are no grammar classes. Grammars are syntax definitions and syntax definitions are grammars. And syntax definitions are parser definitions and parser definitions are syntax definitions. And the syntax definitions are natural, and pure, and beautiful.

In this land, there is declarative disambiguation. And the syntax definitions stay natural, and pure, and beautiful.

In this land, there are no separate scanners. Lexical and context-free syntax definitions are one definition. And scanners and parsers one tool.

In this land, there is declarative tree construction. The parsers turn the right words into the right trees. And the trees are natural, and pure, and beautiful, as are the syntax definitions.

In this land, there is language evolution and composition without pain. Language engineers can add new rules to their syntax definitions. And the syntax definitions stay natural, and pure, and beautiful. And they can compose two syntax definitions to a single syntax definition because there are neither grammar classes nor separate scanners.

In this land, there is no restriction to parsers. Syntax definitions are parser definitions and parser definitions are syntax definitions. And syntax definitions are grammars and grammars are syntax definitions. And language engineers turn syntax definitions into recognizers, and into generators, and into parsers, and into formatters.

Exodus But the language engineers did not follow, because of their broken spirit and harsh slavery. Only few went out of the house of slavery, into the promised land. Those few made new software to make parsers and began to make parsers by making syntax definitions again. And the syntax definitions were grammars again and grammars were syntax definitions again. And the syntax definitions were natural, and pure, and beautiful again, as were the grammars.

The other language engineers feared greatly. And the people of language engineers cried out,

> *Leave us alone that we may make parser definitions. For it would have been better for us to serve the parser generators.*

But here is our answer,

> *Fear not! Stand firm! See the naturalness, and the pureness, and the beauty of declarative syntax definitions, which will work for you. For the parser definitions that you see today, you shall never see again.*
>
> *Go out to the promised land! Make new software to make parsers and begin to make parsers by making syntax definitions again. Let the syntax definitions be grammars again and grammars be syntax definitions again. And the syntax definitions will be natural, and pure, and beautiful again, as will the grammars.*

Acknowledgements We thank Glenn Vanderburg and the anonymous referees for providing valuable feedback to improve the presentation of this paper. This research was supported by NWO/JACQUARD projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

[1] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic parsing of ambiguous grammars. *Commun. ACM*, 18(8):441–452, 1975.

[2] A. V. Aho and T. G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.*, 1(4):305–312, 1972.

[3] A. Birman and J. D. Ullman. Parsing algorithms with backtrack. In *Conference Record of 1970 Eleventh Annual Symposium on Switching and Automata Theory, 28-30 October 1970, Santa Monica, California, USA*, pages 153–174. IEEE, 1970.

[4] A. Borghi, V. David, and A. Demaille. C-Transformers: a framework to write C program transformations. *ACM Crossroads*, 12(3):3, 2005.

[5] E. Bouwers, M. Bravenboer, and E. Visser. Grammar engineering support for precedence rule recovery and compatibility checking. In A. Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 85–101, Braga, Portugal, March 2008. Elsevier.

[6] M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[7] M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *TOSEM*, 5(1):1–41, January 1996.

[8] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. In C. Consel and J. L. Lawall, editors, *Generative Programming and Component Engineering, 6th International Conference, GPCE 2007*, pages 3–12, Salzburg, Austria, 2007. ACM.

[9] M. Bravenboer, Éric Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 209–228. ACM, 2006.

[10] D. G. Cantor. On the ambiguity problem of backus systems. *Journal of the ACM*, 9(4):477–479, 1962.

[11] P. Charles. *A practical method for constructing efficient LALR(K) parsers with automatic error recovery*. PhD thesis, New York University, 1991.

[12] N. Chomsky. *Syntactic Structures*. Mouton de Gruyter, 1957.

[13] J. Cocke. *Programming languages and their compilers: Preliminary notes*. Courant Institute of Mathematical Sciences, New York University, 1969.

[14] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow. TXL: a rapid prototyping system for programming language dialects. In *Conf. on Comp. Languages*, pages 280–285. IEEE, 1988.

[15] M. de Jonge. A pretty-printer for every occasion. In *The International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia, 2000.

[16] M. de Jonge, E. Nilsson-Nyman, L. C. L. Kats, and E. Visser. Natural and flexible error recovery for generated parsers. In M. van den Brand, D. Gasevic, and J. Gray, editors, *Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2010.

[17] F. L. DeRemer. *Practical translation for LR(k) languages*. PhD thesis, MIT, Cambridge, MA, USA, September 1969.

[18] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[19] J.-M. Favre. Languages evolve too! Changing the software time scale. In *8th International Workshop on Principles of Software Evolution (IWPSE 2005), 5-7 September 2005, Lisbon, Portugal*, pages 33–44. IEEE Computer Society, 2005.

[20] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced separation of Concerns*, 2000.

[21] C. Fischer and R. LeBlanc. *Crafting a compiler*. Benjamin/Cummings Menlo Park, California, USA, 1988.

[22] B. Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional Programming (ICFP 2002)*, pages 36–47, 2002.

[23] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 111–122. ACM, 2004.

[24] S. Ginsburg and J. Ullian. Ambiguity in context free languages. *J. ACM*, 13(1):62–89, 1966.

[25] S. Gorn. Detection of generative ambiguities in context-free mechanical languages. *J. ACM*, 10(2):196–208, 1963.

[26] S. L. Graham, C. B. Haley, and W. N. Joy. Practical LR error recovery. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction, Denver, Colorado, USA, August 6-10, 1979*, pages 168–175. ACM, 1979.

[27] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[28] S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J., 1975.

[29] A. Johnstone, E. Scott, and G. Economopoulos. Evaluating GLR parsing algorithms. *Science of Computer Programming*, 61(3):228–244, 2006.

[30] T. Kasami. An efficient recognition and syntax analysis algorithm for context free languages. Science Report AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass., USA, 1965.

[31] L. C. L. Kats, M. de Jonge, E. Nilsson-Nyman, and E. Visser. Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In S. Arora and G. T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA.*, pages 445–464, 2009.

[32] L. C. L. Kats and E. Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In M. Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, 2010. (to appear).

[33] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[34] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy, October 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

[35] D. E. Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[36] P. M. Lewis II and R. E. Stearns. Syntax-directed transduction. *Journal of the ACM*, 15(3):465–488, 1968.

[37] B. A. Malloy, J. F. Power, and J. T. Waldron. Applying software engineering techniques to parser design: the development of a C# parser. In *SAICSIT '02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 75–82, Port Elizabeth, Republic of South Africa, 2002. South African Institute for Computer Scientists and Information Technologists.

[38] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.

[39] Pāṇini. *Aṣṭādhyāyī*. Otto Böhtlingk, editor, König, 1839-1840.

[40] Pāṇini. *Aṣṭādhyāyī*. Benares, 1896. Translated by Śrīśa Chandra Vasu.

[41] T. J. Parr. ANTLR Parser Generator. `http://www.antlr.org/`.

[42] T. J. Parr and R. W. Quong. LL and LR translators need k>1 lookahead. *SIGPLAN Not.*, 31(2):27–34, 1996.

[43] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, January 1992.

[44] D. Salomon and G. Cormack. The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical Report 95/06, Department of Computer Science, University of Manitoba, Winnipeg, Canada, 1995.

[45] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Not.*, 24(7):170–178, 1989.

[46] E. Scott and A. Johnstone. GLL parsing. In *Workshop on Language Descriptions, Tools and Applications (LDTA'09)*, 2009.

[47] The Spoofax project. `http://www.spoofax.org/`.

[48] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*, volume 14. Kluwer Academic Publishers, 1988.

[49] M. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 2002.

[50] E. Visser. A case study in optimizing parsing schemata by disambiguation filters. In *IWPT*, pages 210–224, 1997.

[51] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[52] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[53] E. Visser. Meta-programming with concrete object syntax. In D. S. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2002.

[54] D. Younger. Recognition and parsing of context-free languages in time n^3. *Information and control*, 10(2):189–208, 1967.

TUD-SERG-2010-019a
ISSN 1872-5392