Delft University of Technology
Software Engineering Research Group
Technical Report Series

Providing Rapid Feedback in Generated Modular Language Environments

Lennart C. L. Kats, Maartje de Jonge,
Emma Nilsson-Nyman, Eelco Visser

Report TUD-SERG-2009-020a

TUD-SERG-2009-020a

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:

Providing Rapid Feedback in Generated Modular Language Environments. Adding Error Recovery to Scannerless Generalized-LR Parsing. In Gary T. Leavens, editor, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2009)*, ACM, 2009.

```
@InProceedings{KJNV09,
  author = {Lennart C. L. Kats and Maartje de Jonge and Emma Nilsson-Nyman
            and Eelco Visser},
  title = {Providing Rapid Feedback in Generated Modular Language Environments.
           {Adding} Error Recovery to Scannerless Generalized-{LR} Parsing},
  booktitle = {Proceedings of the 24th ACM SIGPLAN Conference on
               Object-Oriented Programing, Systems, Languages,
               and Applications (OOPSLA 2009)},
  year = {2009},
  editor = {Gary T. Leavens},
  location = {Orlando, Florida, USA},
  address = {New York, NY, USA},
  month = {October},
  publisher = {ACM Press},
  issn =    {0362-1340},
  series = {ACM SIGPLAN Notices},
  OPTpages = {},
  OPTvolume = {},
  OPTnumber = {},
}
```

Providing Rapid Feedback in Generated Modular Language Environments

Adding Error Recovery to Scannerless Generalized-LR Parsing

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

Maartje de Jonge
Delft University of Technology
m.dejonge@tudelft.nl

Emma Nilsson-Nyman
Lund University
emma@cs.lth.se

Eelco Visser
Delft University of Technology
visser@acm.org

Abstract

Integrated development environments (IDEs) increase programmer productivity, providing rapid, interactive feedback based on the syntax and semantics of a language. A heavy burden lies on developers of new languages to provide adequate IDE support. Code generation techniques provide a viable, efficient approach to semi-automatically produce IDE plugins. Key components for the realization of plugins are the language's grammar and parser. For embedded languages and language extensions, constituent IDE plugin modules and their grammars can be combined. Unlike conventional parsing algorithms, scannerless generalized-LR parsing supports the full set of context-free grammars, which is closed under composition, and hence can parse language embeddings and extensions composed from separate grammar modules. To apply this algorithm in an interactive environment, this paper introduces a novel error recovery mechanism, which allows it to be used with files with syntax errors – common in interactive editing. Error recovery is vital for providing rapid feedback in case of syntax errors, as most IDE services depend on the parser – from syntax highlighting to semantic analysis and cross-referencing. We base our approach on the principles of island grammars, and derive permissive grammars with error recovery productions from normal SDF grammars. To cope with the added complexity of these grammars, we adapt the parser to support backtracking. We evaluate the recovery quality and performance of our approach using a set of composed languages, based on Java and Stratego.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.3.4 [*Programming Languages*]: Processors

General Terms Languages

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
Copyright © 2009 ACM 978-1-60558-734-9/09/10. . . $10.00

1. Introduction

Integrated Development Environments (IDEs) increase programmer productivity by combining a rich toolset of generic language development tools with services tailored for a specific language. These services provide a programmer with rapid, interactive feedback based on the syntactic structure and semantics of the language. High expectations with regard to IDE support place a heavy burden on the shoulders of developers of new languages. Language development environments facilitate efficient development of IDE support for new languages. Notable examples include IMP [9], the Meta-Environment [37], MontiCore [24], and openArchitectureWare [13]. In this paper we focus on Spoofax/IMP [20]. Leveraging code generation and interpretation techniques, these tools provide a viable, efficient approach to semi-automatically produce IDE plugins.

Compositional Languages and SGLR Success of a language, in part, depends on interoperability with other languages and systems. Different languages address different concerns. Language composition is a promising approach for providing integrated support for different concerns. However, compositional languages, such as language extensions and language embeddings, further increase the burden for language engineers, as they now have to provide IDE support for a combination of languages or language elements. Therefore, language development tools must offer support for extensions and combinations of languages. IDE development for compositional languages greatly benefits from code generation where several, independently defined parts can be combined. How well a tool can support language composition depends on the underlying language techniques it uses.

The Scannerless Generalized-LR parsing parsing algorithm (SGLR) [40] supports the modular syntax definition formalism SDF [41]. SDF is declarative yet expressive, and has been used to specify non-trivial grammars for existing languages such as Java, C, and PHP, as well as domain-specific languages and embeddings and extensions based on these languages [7]. Unlike other parsing formalisms (particularly those commonly used in IDEs), SDF is closed under composition: existing grammars can be reused and composed to form new languages. This makes it a useful parsing technique to use in language tools supporting composition of languages. Integration of SGLR into an IDE based on IMP [8] is ongoing work [20].

Parsing in IDEs Using a language development environment, the grammar is the first artifact constructed by the developer. Traditionally, IDEs have often used handtailored parsers. Doing so reduces flexibility, especially with language extensions and combinations in mind. For the efficient development of language tools it is essential that parser generators are used instead.

The parser for a language forms the foundation of all language-specific editor services. The parser performs syntactic analysis (parsing) to construct abstract syntax trees (ASTs) for user programs. These ASTs can be used for *presentational editor services*, such as syntax highlighting, code folding, and outlining. They also form the basis for *semantic analysis* of a program, allowing for editor services such as cross-referencing and checking for semantic errors.

To provide the user with rapid syntactic and semantic feedback, programs must be interactively parsed as they are edited. As the user edits a program, it is often in a syntactically invalid state. Parse error recovery techniques can diagnose and report parse errors, and can construct a valid AST for syntactically invalid programs [10]. Thus, to successfully apply a parser in an interactive setting, proper parse error recovery is of paramount importance.

The scannerless, generalized nature of SGLR is essential for parsing compositional languages, but also introduces challenges for implementing error recovery. The current SGLR implementation provides no recovery of any kind, and only reports the first unexpected character in case of failure. We have identified two main challenges. (1) *Scannerless parsing*: Where other parsers employ a separate scanner for tokenization and report errors in terms of missing (or expected) *tokens*, SGLR merely reports unexpected *characters*. (2) *Generalized parsing*: A GLR parser processes multiple branches (representing different interpretations of the input) in parallel. Syntax errors can only be detected at the point where the last branch failed, which may not be local to the actual root cause of an error. This makes it difficult to properly identify the offending substring or character.

This paper presents a novel approach to error recovery using SGLR. We base our approach on the principles of island grammars [39, 26, 27], defining new production rules for a grammar that make it more permissive of its inputs. We identified several idioms for defining such recovery rules that either discard substrings in the input or insert literals (i.e., keywords and braces) as necessary. Based on the analysis of an existing grammar, we can automatically derive a set of these rules. Using the recovery rules, parse errors of various kinds can be properly diagnosed and repaired, reporting any missing or inserted keywords and braces, addressing challenge (1). To cope with the added complexity of grammars with recovery rules, we adapt the parser implementation to apply the recovery rules in an on-demand fashion, using a backtracking algorithm. This algorithm explores an increasing, backward search space to find a minimal-cost solution for applying the set of recovery rules. This technique allows us to identify the most likely origin of an error, thus avoiding reports of spurious errors and addressing challenge (2).

We have incorporated the approach in the Spoofax/IMP IDE plugin generator [20], to obtain robust editors for composite languages that can provide feedback to the user in the presence of syntactic errors. We have evaluated the error recovery approach using a set of grammars for plain Java and for composite languages such as Stratego-Java and Java-SQL.

Contributions

- A novel approach to parse error recovery based on grammar relaxation, adding new “recovery” productions to make a grammar more permissive.
- An adaptation of the SGLR algorithm that efficiently handles the increased complexity of permissive grammars.
- A language (grammar) independent approach to error recovery using SGLR.

Outline The remainder of this paper starts with a motivating study of composite languages in Section 2. In Section 3 we discuss the requirements on error recovery. In Section 4 we discuss the notion of *island grammars*, which provide the inspiration for our error recovery approach. In Section 5 we show how the ideas of island grammars can be used to make complete language grammars into permissive grammars. In Section 6 we explain the adaptation of the SGLR algorithm to deal with the combinatorial explosion introduced by permissive grammars. Section 7 evaluates the approach while Section 8 covers related work. Finally, the paper ends with conclusions and future work in Section 9.

2. Composite Languages

Composite languages integrate elements of different language components. We distinguish two classes of composite languages: language extensions and embedded languages. Language extensions extend a base language with new, often domain-specific elements. Language embeddings combine two or more existing languages, allowing one language to be nested in the other.

Examples of language extensions include the addition of traits [11] or aspects [21] to object-oriented languages, enhancing their support for adaptation and reuse of code. Other examples include new versions of a language, introducing new features to an existing language, such as Java 1.5’s `enum` keyword for enumerated types.

Examples of language embeddings include data base query expressions integrated into an existing, general-purpose

```
public class Authentication {
  public String getPasswordHash(String user) {
    SQL stm = <| SELECT password FROM Users
                 WHERE name = ${user} |>;
    return database.query(stm);
  }
}
```

Figure 1. An extension of Java with SQL queries.

```
webdsl-action-to-java-method:
  |[ action x_action(farg*) { stat* }       ]| ->
  |[ public void x_action(param*) { bstm* } ]|
  with param* := <map(action-arg-to-java)> farg*;
       bstm*  := <statements-to-java> stat*
```

Figure 2. Program transformation using embedded object language syntax.

language such as Java. Such an embedding both increases the expressivity of the host language and facilitates static checking of queries. Figure 1 illustrates such an embedding. Using a special *quotation* construct, an SQL expression is embedded into Java. In turn, the SQL expression includes an *anti-quotation* of a Java local variable. By supporting the notion of quotations in the language, a compiler can distinguish between the static query and the variable, allowing it to safeguard against injection attacks. In contrast, when using only a basic Java API for SQL queries constructed using strings, the programmer must take care to properly filter any values provided by the user.

Language embeddings are sometimes applied in metaprogramming for quotation of their object language. Transformation languages such as Stratego [5] and ASF+SDF [38] allow fragments of a language that undergoes transformation to be embedded in the specification of rewrite rules. Figure 2 shows a Stratego rewrite rule that rewrites a fragment of code from a domain-specific language to Java. The rule uses metavariables (written in *italics*) to match "action" constructs and rewrites them to Java methods with a similar signature. SDF supports meta-variables by reserving identifier names in the context of an embedded code fragment.

Parsing Composite Languages Key to the effective realization of composite languages are modular, reusable language descriptions, which allow constituent languages to be defined independently, and then composed to form a whole.

A particularly difficult problem in composing language definitions is composition at the lexical level. Consider again Figure 2. In the embedded Java language, `void` is a reserved keyword. For the enclosing Stratego language, however, this name is a perfectly legal identifier. This difference in lexical syntax is essential for a clean and safe composition of languages. It is undesirable that the introduction of a new language embedding or extension invalidates existing, valid programs.

The difficulty in combining languages with a different lexical syntax stems from the traditional separation between scanning and parsing. The scanner recognizes words either as keyword tokens or as identifiers, regardless of the context. In the embedding of Java in Stratego this would imply that `void` becomes a reserved word in Stratego as well. Only using a carefully crafted lexical analysis for the combined language, introducing considerable complexity in the lexical states to be processed, can these differences be reconciled. Using scannerless parsing [33, 32], these issues can be elegantly addressed [6]. The *Scannerless Generalized-LR* (SGLR) parsing algorithm [40] realizes scannerless parsing by incorporating the generalized-LR parsing algorithm [35]. GLR supports the full class of context-free grammars, which is closed under composition, unlike subsets of the context-free grammars such as LL(k) or LR(k). Instead of rejecting grammars that give rise to shift/reduce and reduce/reduce conflicts in an LR parse table, the GLR algorithm interprets these conflicts to efficiently try all possible parses of a string in parallel, thus supporting grammars with ambiguities, or grammars that require more look-ahead than incorporated in the parse table. Hence, the composition of independently developed grammars does not produce a grammar that is not supported by the parser, as is frequently the case with LL or LR based parsers.

The syntax definition formalism SDF2 [41] integrates lexical syntax and context-free syntax supported by SGLR as parsing algorithm. Undesired ambiguities in SDF2 definitions can be resolved using declarative *disambiguation filters* [3]. Implicit disambiguation mechanisms such as 'longest match' are avoided. Other approaches, including PEGs [14], language inheritance in MontiCore [24], and the composite grammars of ANTLR [29], implicitly disambiguate grammars by forcing an ordering on the alternatives of a production – the first (or last) definition overrides the others. Enforcing explicit disambiguation allows undesired ambiguities to be detected, tested against in regression tests, and explicitly addressed by a developer. For non-trivial grammars, in particular composed, independently developed grammars, this characteristic is of vital importance.

SDF has been used to define various composite languages, often based on mainstream languages such as C/C++ [42], PHP [4], and Java [7, 19]. The example grammar shown in Figure 3 extends Java with embedded SQL queries. It imports both the Java and SQL grammars, adding only productions that integrate the two. In SDF, grammar productions take the form $p_1 \ldots p_n$ `->` s and specify that a sequence of strings matching symbols p_1 to p_n matches the symbol s. The productions in this particular grammar specify two productions to embed SQL into Java expressions and two productions to embed Java into SQL. The productions are annotated with the `{cons(`*name*`)}` annotation, which indicates the constructor name used to label these elements when an abstract syntax tree is constructed.

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

Figure 3. A grammar extending Java with SQL queries; adapted from [4].

To avoid name collisions between the Expr symbol in the SQL grammar and the Expr symbol in the Java grammar, SDF uses *parametrization* of symbol names: the [Java] parameter in the JavaMix import indicates that all Java symbols should be referenced using the [[Java]] postfix.

3. Interactive Parsing and Error Recovery

For all the merits of SDF and the SGLR parser, in the form of modularity, declarative disambiguation, and compositionality, one may wonder why these technologies have not (yet) seen more widespread use. In part, this is due to a lack of publicity. There are also a number of more fundamental issues that have hindered their adoption. Recently, Bravenboer et al [6] analyzed some of these issues. First, the syntax of productions in SDF may be awkward and unappealing to developers accustomed to BNF-style rules. Perhaps reversing the order of the production pattern and the symbol would appeal to a broader audience. Second, [6] identifies error handling as an open issue. Third, a lack of tool support for analyzing ambiguities is identified as an open issue. Last and perhaps a more practical issue has been that the tools were only implemented in C, targeting the Unix/Linux platform.

The main obstacles for employing SDF and SGLR in an interactive environment arguably are the second issue, a lack of error handling; and the last issue, lacking cross-platform support. This paper addresses the second issue, adding both error diagnosis and error recovery to the SGLR parser. This makes it possible to parse syntactically incorrect files and use the result in the different syntactic and semantic editor services of an IDE. We base our implementation on JSGLR, a Java implementation of SGLR [18]. With its development, although the implementation is not quite mature, the last issue has also been effectively resolved. The remaining issues impact only developers of SDF grammars. Based on our own experience, and looking at the large set of grammars already developed using SDF, we trust that these issues are not a significant obstacle, and that they will be resolved in the future.

Parse error handling encompasses two concerns: error reporting and error recovery. Error reporting, by itself, has an important role in giving feedback to the user. An error handling technique should accurately report all syntactic errors without introducing spurious errors. This requires accurate diagnosis of errors. A faulty correction may leave the parser in a state that will cause spurious errors to be reported later. Furthermore, when an error is misdiagnosed, the error message issued for it tends to be misleading. An error message should indicate the exact location of the error and provide a suggestion for correction. Good error messages reflect the intention of the programmer.

```
module ExtractCalls
exports
  context-free start-symbols
    Module
  context-free syntax
    Chunk*    -> Module {cons("Module")}
    WATER     -> Chunk  {cons("WATER")}
    "CALL" Id -> Chunk  {cons("Call")}
  lexical syntax
    [\ \t\n]       -> LAYOUT
    ~[\ \t\n]+     -> WATER {avoid}
    [A-Z][A-Z0-9]* -> Id
  lexical restrictions
    WATER -/- [A-Za-z0-9]
```

Figure 4. An island grammar for extracting calls from a legacy application; adapted from [26].

Furthermore, recovery from parse errors allows the parser to continue the analysis of the source code after the detection of an error. The resulting parse tree of the analysis is a parse tree representing the corrected input. This parse tree allows further analysis of the source code at the syntactical and semantic level even for programs that are not in a syntactically valid state as the programmer is editing them.

4. Island Grammars

Island grammars [39, 26, 27] combine grammar production rules for the precise analysis of parts of a program and selected language constructs with general rules for skipping over the remainder of an input. Island grammars are commonly applied for reverse engineering of legacy applications, for which no formal grammar may be available, or for which many (vendor-specific) dialects exist [26]. Using an island grammar, a parser can skip over any uninteresting bits of a file ("water"), including syntactic errors or constructs found only in specific language dialects. A small set of declarative context-free production rules specifies only the interesting bits (the "islands") that are parsed 'properly'. Island grammars were originally developed using SDF [39, 26]. The integration of lexical and context-free productions of SDF allows island grammars to be written in a single, declarative specification that includes both lexical syntax for the definition of water and context-free productions for the islands. Although SGLR did not support error recovery per se, a parser using an island grammar behaves similar to one that implements a noise-skipping algorithm. It can skip over any form of noise in an input file. However, using an island grammar, this logic is entirely encapsulated in the grammar definition itself.

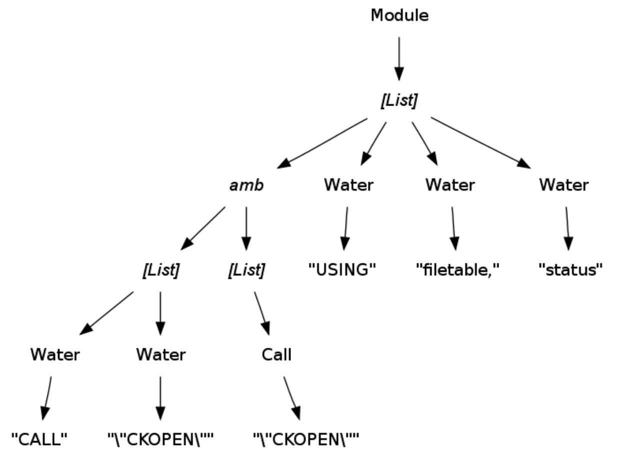


Figure 5. The unfiltered abstract syntax tree for a COBOL statement, constructed using the ExtractCalls grammar.

As an example, consider Figure 4, which shows an SDF specification of an island grammar that extracts call statements from COBOL programs. Any other statements in the program are skipped and parsed as water. The first context-free production of the grammar defines the `Module` symbol, which is the start symbol of the grammar. A `Module` is a series of chunks. Each `Chunk`, in turn, is parsed either as a patch of `WATER` or as an island, in the form of a `CALL` construct. The lexical productions define patterns for layout, water, and identifiers. The layout rule, using the special `LAYOUT` symbol, specifies the kind of layout (i.e., whitespace) used in the language. Layout is ignored by the context-free syntax rules, since their patterns are automatically interleaved with optional layout. The `WATER` symbol is defined as the inverse of the layout pattern, using the ~ negation operator. Together, they can match any given character stream. The `{avoid}` annotation on the `WATER` rule specifies a disambiguation filter for these productions, indicating that the production is to be avoided: at all times, a non-water `Chunk` is to be preferred. In the lexical restrictions section, we specify a follow restriction for the `WATER` symbol. This rule ensures that water is always greedily matched, and never followed by any other water character. Consider the following COBOL statement:

```
CALL "CKOPEN" USING filetable, status
```

Given our island grammar, the SGLR parser can construct a parse tree – or rather a parse *forest* – that includes all possible interpretations of this text.[1] The parse tree includes the complete character stream, all productions used, and their annotations. In this paper, we focus on abstract syntax trees (derived from the parse trees) where only the `{cons(`*name*`)}` constructor labels appear in the tree. Figure 5 shows the complete, ambiguous abstract syntax tree for our example input program. Note in particular the *amb* node, which indicates an ambiguity in the tree: `CALL "CKOPEN"` in our example can be parsed either as a proper `Call` statement or as `WATER`. Since the latter has an `{avoid}` annotation in its definition, a disambiguation filter can be applied to resolve the ambiguity [3]. (Normally, these filters are applied automatically during or after parsing.)

```
module Java-15
exports
  lexical syntax
    [\ \t\12\r\n]           -> LAYOUT
    "\""  StringPart*  "\"" -> StringLiteral
    "/*"  CommentPart* "*/" -> Comment
    Comment                 -> LAYOUT
    ...
  context-free syntax
    "if" "(" Expr ")" Stm -> Stm {cons("If")}
    "if" "(" Expr ")" Stm "else" Stm
                   -> Stm {avoid, cons("IfElse")}
    ...
```

Figure 6. Part of the standard Java grammar in SDF; adapted from [6].

5. Permissive Grammars

As we have observed in the previous section, there are many similarities between a parser using an island grammar and a noise-skipping parser. In the former case, the water productions of the grammar are used to “fall back” in case an input sentence cannot be parsed, in the latter case, the parser algorithm is adapted to do so. This observation suggests that the basic principle behind island grammars may be adapted for use in recovery for complete, well-defined grammars. In contrast, the technique of island grammars is targeted only towards partial grammar definitions.

In the remainder of this section, we illustrate how the notion of productions for defining “water” can be used in regular grammars, and how these principles can be further applied to achieve alternative forms of recovery from syntax errors. Without loss of generality, we focus many of our examples on the familiar Java language. Figure 6 shows a part of the SDF definition of the language. Indeed, Java can be parsed without the use of SGLR, but for extensions and embeddings based on Java, SGLR has proved invaluable. Furthermore, the current problems with a lack of error recovery in SGLR also hold for the stand-alone Java language; currently, if any error is found in a pure Java input, the parser comes to a screeching halt and is unable to continue.

5.1 Chunk-Based Water Recovery Rules

Whereas island grammars have an underlying “chunk” structure, this structure is lacking in complete, well-defined grammars. Rather, these grammars typically have a more hierarchical structure. For example, Java programs consist of a one

[1] Note that parse forests are efficiently represented using the ATerm library [2], which employs hash-consing to achieve maximal sharing of subtrees, ensuring that any identical leaves and branches occupy the same space in memory.

```
module Java-15-Permissive-ChunkBased
imports Java-15
exports
  lexical syntax
    ~[\ \t\12\r\n]+ -> WATER {recover}
  lexical restrictions
    WATER -/- ~[\ \t\12\r\n]
  context-free syntax
    WATER -> Stm {cons("WATER")}
```

Figure 7. Chunk-based recovery rules for Java.

or more classes that each contain methods, which contain statements, etc. Still, it is possible to impose a more chunk-like structure on existing grammars in a coarse-grained fashion: for example, in Java, all statements can be considered as chunks.

Figure 7 extends the standard Java grammar with a coarse-grained chunk structure at the statement level. In this grammar, every `Stm` symbol is considered a “chunk,” which can be parsed as either a statement or as water, effectively skipping over any noise that may exist within method bodies. Note that the standard Java grammar, as shown in Figure 6, already uses an `{avoid}` annotation to explicitly avoid the “dangling else problem,” a notorious ambiguity that occurs with nested if/then/else statements. Therefore, in our recovery rules we use `{recover}` rather than `{avoid}` to distinguish between the two concerns of disambiguation and recovery.

We can extend the grammar of Figure 7 to introduce a chunk-like structure at other levels in the hierarchical structure formed by the grammar, e.g. at the method level or at the class level, in order to cope with syntax errors in different places. However, doing so leads to a large number of possible interpretations of syntactically invalid (but also syntactically valid) programs. For example, any invalid statement that appears in a method could then be parsed as a “water statement.” Alternatively, the entire method could be parsed as a “water method.” A preferred interpretation can be picked by counting all occurrences of the `{recover}` annotation in ambiguous branches, and selecting the variant with the lowest count.

The technique of selectively adding water recovery rules to a grammar allows any existing grammar to be adapted. It avoids having to rewrite grammars from the ground up to be more “permissive” in their inputs. Grammars adapted in this fashion produce parse trees even for inputs that contain syntax errors and cannot be parsed by the original grammar. The `WATER` constructors in the abstract syntax trees indicate the location of errors, which can then be straightforwardly reported back to the user.

While the approach we presented so far is already moderately effective in recovery from syntax errors, there are three disadvantages to the recovery rules as presented here. Firstly, the rules are language-specific and are best implemented by an expert of a particular language and its SDF grammar specification. Secondly, the rules are rather coarse-grained in nature; invalid subexpressions in a statement cause the entire statement to be parsed as water. Lastly, the additional productions alter the abstract syntax of the grammar (introducing new `WATER` terminals), causing the parsed result to be unusable for tools that depend on the original structure.

5.2 General Water Recovery Rules

Adapting a grammar to include water productions at different hierarchical levels is a relatively simple yet effective way to selectively skip over “noise” in an input file. In the remainder of this section, we refine this approach and use it as a basis for our general approach to error recovery. Note throughout this section we use only the standard, unaltered SDF specification language, adding only the `{recover}` annotation and identifying idioms for recovery rules.

Most programming languages feature comments and insignificant whitespace that have no impact on the logical structure of a program. They are generally not considered to be a logical part of the abstract syntax tree. As discussed in Section 4, any form of layout, which may include comments, is implicitly interleaved in the patterns of concrete syntax productions. The parser, in a way, skips over these parts, in a similar fashion to the noise skipping of island grammars. However, layout and comments interleave the context-free syntax of a language at a much finer level than the recovery rules we have discussed so far. Consider for example the Java statement

```
if (temp.greaterThan(MAX)/*API change pending*/)
  fridge.startCooling();
```

in which a comment appears in the middle of the statement. Context-free syntax in SDF is a convenient way to define context-free productions without having to worry about the interleaving of layout. Only in the *kernel syntax* that lies at the heart of SDF, does the production explicitly include the layout:

```
syntax
  "if" <LAYOUT?-CF> "(" <LAYOUT?-CF> <Expr-CF>
       <LAYOUT?-CF> ")" <LAYOUT?-CF> <Stm-CF>
   -> <Stm-CF> cons("If")
```

The parse table generator for SDF automatically converts context-free productions to this form. (The production above was derived from the `If` production in Figure 6). Expressed in kernel syntax, the symbol names in the rule above use angle brackets and explicitly state that they are related to context-free (CF) syntax. The optional layout symbols `<LAYOUT?-CF>` are not considered for the construction of the abstract syntax tree (and may be stored as annotated data instead).

We can use the notion of interleaving context-free productions with optional layout in order to define a new variation of the water-based recovery rules we have shown so far. Consider Figure 8, which combines elements of the comment definition of Figure 6 and the chunk-based recovery rules from Figure 7. It introduces optional water into the

```
module Java-15-Permissive-WaterOnly
imports Java-15
exports
  lexical syntax
    [A-Za-z0-9\_]*          -> WATERWORD {recover}
    ~[A-Za-z0-9\_\ \t\12\r\n]
                            -> WATERSEP  {recover}
    WATERWORD -> WATER
    WATERSEP  -> WATER
    WATER     -> LAYOUT {cons("WATER")}
  lexical restrictions
    WATERWORD -/- [A-Za-z0-9\_]
```

Figure 8. Water-based recovery rules.

grammar, which interleaves the context-free syntax patterns. As such, it skips noise on a much finer grained level than our previous grammar incarnation.

To separate patches of water into small chunks, each associated with its own significant `{recover}` annotation, we distinguish between `WATERWORD` and `WATERSEP` tokens. This ensures that large strings, consisting of multiple words and special characters each, are counted towards a higher recovery cost.

As an example input, consider a programmer who is in the process of introducing a conditional clause to a statement:

```
if (temp.greaterThan(MAX) // missing )
  fridge.startCooling();
```

Still missing the closing brace, the standard SGLR parser would report an error near the missing character, and would stop parsing. Using the adapted grammar, a parse forest is constructed that considers the different interpretations, taking into account the new water recovery rule. Based on the count of the `{recover}` annotations, the following would be the preferred interpretation:

```
if (temp.greaterThan)
  fridge.startCooling();
```

In the resulting fragment both the opening `(` and the identifier `MAX` are discarded, giving a total cost of 2 recoveries. The previous, chunk-based incarnation of our grammar would simply discard the entire `if` clause. While not yet ideal, the new version maintains a larger part of the input. Since it is based on the `LAYOUT` symbol, it also does not introduce new "water" nodes into the abstract syntax tree. For reporting errors, the original parse tree can be inspected instead.

The adapted grammar of Figure 8 no longer depends on hand-picking particular symbols at different granularities to introduce water recovery rules. Therefore, it is effectively language-independent, and can be automatically constructed using only the `LAYOUT` definition of the grammar.

```
module Java-15-Permissive-InsertionsOnly
imports Java-15
exports
  lexical syntax
     -> ")" {recover, cons("INSERT")}
     -> "]" {recover, cons("INSERT")}
     -> "}" {recover, cons("INSERT")}
     -> ">" {recover, cons("INSERT")}
     -> ";" {recover, cons("INSERT")}
  lexical syntax
    INSERTSTARTQ StringPart* INSERTENDQ
              -> StringLiteral {cons("INSERTEND")}
    "\""      -> INSERTSTARTQ {recover}
    "\n"      -> INSERTENDQ
  lexical syntax
    INSERTSTARTC CommentPart* INSERTENDC
                   -> Comment {cons("INSERTEND")}
    "/*"           -> INSERTSTARTC {recover}
    EOF            -> INSERTENDC
```

Figure 9. Literal-insertion recovery rules.

5.3 Literal-insertion Recovery Rules

So far, we have focused our efforts on recovery by deletion of erroneous substrings. However, in an interactive environment, most parsing errors may well be caused by *missing substrings* instead. Consider again our previous example:

```
if (temp.greaterThan(MAX) // missing )
  fridge.startCooling();
```

Our use case for this has been that the programmer was still editing the phrase, and did not yet add the missing closing brace. Discarding the opening `(` and the `MAX` identifier allowed us to parse most of the statement and the surrounding file, reporting an error near the missing brace. Still, a better recovery would be to insert the missing `)`.

One way to accommodate for insertion based recovery is by the introduction of a new rule to the syntax to make the closing brace optional:

```
"if" "(" Expr Stm -> Stm {cons("If"), recover}
```

This strategy, however, is rather specific for a single production, and would greatly increase the size of the grammar if we applied it to all productions. A better approach would be to actually "insert" the particular literal into the parse stream. SDF actually allows us to simulate this using separate productions that "insert" literal symbols. We illustrate this in Figure 9. Consider the first lexical syntax section, which lists a number of basic *literal-insertion recovery rules*, each inserting a closing bracket or other literal that ends a production pattern.

Literal-insertion rules have an empty pattern, indicating that they match an empty string. That is, for each of these literals specified in the grammar, an empty string may be matched against instead. Just as in our previous examples, `{recover}` ensures these productions are deferred. The constructor annotation `{cons("INSERT")}` is used as a labeling mechanism for error reporting for the inserted literals. As it is defined in lexical syntax context, it is not used in the resulting abstract syntax tree.

Insertion Rules for Opening Brackets In addition to insertions of closing brackets in the grammar, we can also add rules to insert opening brackets. These literals start a new scope or context. This is particularly important for composed languages, where a single starting bracket can indicate a transition into a different sublanguage, such as the `|[` and `<|` brackets of Figure 1 and Figure 2. Consider for example a syntax error caused by a missing opening bracket in the SQL query of the former figure:

```
SQL stm = // missing <|
    SELECT password FROM Users WHERE name = ${s}
|>;
```

Without an insertion rule for the `<|` opening bracket, the entire SQL fragment could only be recognized as (severely syntactically incorrect) Java code. Thus, it is essential to have insertions for such brackets. However, for the insertion of opening brackets, the regular SGLR implementation no longer suffices, since it could insert infinitely many combinations of opening and closing brackets. We address this in Section 6 by adapting the parsing algorithm to consider the added recovery cases in an on-demand fashion.

On Literals, Identifiers, and Reserved Words Literal-insertion rules can also be used for literals that are not *reserved words*. For example, for the combined Stratego-Java language, a good insertion rule is:

```
lexical syntax
   -> "end"
```

In Java, the string `end` is not a reserved word and is a perfectly legal *identifier*. In Java, identifiers are defined as follows:

```
lexical syntax
  [A-Za-z\_\$][A-Za-z0-9\_\$]* -> ID
```

This lexical rule would match a string `end`. Still, the recovery rule will strictly be used to insert the literal `end`, and never an identifier. The reason why the parser can make this distinction is that the literal `end` itself is defined as an ordinary symbol when normalized to kernel syntax:[2]

```
syntax
  [\e] [\n] [\d] -> "end"
```

The literal-insertion rule simply adds an additional derivation for the `"end"` symbol, providing the parser with an additional way to parse it. As such, the rule does not change how identifiers (`ID`) are parsed. This is an important property when considering composed languages in general. In many cases, some literals in one sublanguage may not be reserved words in another. With a naive recovery strategy that inserts tokens into the stream, this could result in keywords being inserted in place of identifiers (e.g., `end` in Java). But since the insertion rules only apply when a literal is expected, these effects are avoided with out approach.

Insertion Rules for Lexical Symbols Insertion rules can also be used to insert lexical symbols such as identifiers. In our approach, we only focus on a very small set of lexical symbols; missing identifiers generally indicate an error in the enclosing context-free construct and are not addressed separately. Still, using identifier insertions is feasible, but adds extra complexity to the tools that process the abstract syntax tree.

The lower sections of Figure 9 specify insertion rules for terminating the productions of the `StringLiteral` and `Comment` symbols, first seen in Figure 6. Both rules have a `{recover}` annotation on their starting literal. Alternatively, the annotation could be placed on the complete production, but this formulation is beneficial for the runtime behavior of our adapted parser implementation, ensuring that the annotation is considered before construction of the literal.

The recovery rules for string literals and comments match either at the end of a line, or at the end of the file as appropriate, depending on whether newline characters are allowed in the original, non-recovering productions. In contrast, an alternative approach would have been to add a literal insertion production for the quote and comment terminator literals. However, by only allowing the strings and comments to be terminated at the ending of lines and the file, the number of different possible interpretations is severely reduced, thus reducing the overall runtime complexity of the recovery.

5.4 Combining Different Recovery Rules

The water recovery rules of Section 5.2 and the insertion rules of Section 5.3 can be combined to form a unified recovery mechanism that allows both discarding and insertion of substrings:

```
module Java-15-Permissive
imports
  Java-15-Permissive-WaterOnly
  Java-15-Permissive-InsertionsOnly
```

Together, the two strategies maintain a fine balance between discarding and inserting substrings. Since the water-based recovery rules incur additional cost for each water substring, insertion of literals will generally be preferred over discarding multiple input strings. This ensures that most of the original (or intended) user input is preserved.

5.5 Derivation of Permissive Grammars

So far, we only focused on a particular kind of literals for insertion into the grammar, such as brackets, keywords, and string literals. Still, we need not restrict ourselves to only these particular literals. In principle, any literal in the grammar is eligible for use in an insertion recovery rule.

For many literals, automatic insertion can lead to unintuitive results in the feedback presented for the user. For example, we don't want the editor to suggest to insert a "try" or "synchronized" keyword. In those cases, discarding some substrings instead may be a safer alternative. The decision whether to consider particular keywords for insertion may

[2] Actually, in fully normalized kernel syntax form, the character codes `[\101] [\110] [\100]` are used.

depend on their semantic meaning and importance [10]. To take this into account, expert feedback on a grammar is vital. Since we have aimed at maintaining language independence of the approach, our main focus is on more generic, structure-based properties of the grammar.

In this section we have identified and focused on four different distinct, general *classes of literals* that commonly occur in grammars:

- Closing brackets and terminating literals for context-free productions
- Opening brackets and starting literals for context-free productions
- Closing literals that terminate lexical productions where no newlines are allowed (such as most string literals)
- Closing literals that terminate lexical productions where newlines are allowed (such as block comments)

Each has its own particular kind of insertion rule, and each follows its own particular definition pattern. By analysis of a grammar, using heuristic rules to recognize these patterns, we derive water-based recovery rules and recovery rules for insertions of the above categories. Thereby, our system maintains language independence by providing a generic, automated approach towards the introduction of recovery rules.

Automatically deriving recovery rules helps maintain a valid, up-to-date recovery rule set as languages evolve and are extended or embedded into other languages. Particularly, as languages are changed, care must be taken to remove any recovery rules from the grammar that are no longer applicable.

SDF specifications are fully declarative. It is this nature that is essential for automated analysis and transformation of a grammar specification. It is not feasible to do so for other syntax formalisms that use semantic actions to construct abstract syntax trees and may maintain state or call external functions (e.g., to determine operator priorities).

We formulated a set of heuristic rules for the detection of different production patterns based on our experience with different grammars. For instance, *opening bracket and starting literal insertions* are added based on the following criteria. First, we only consider context-free productions. Second, the first and last symbols of the pattern of such a production must be a literal. And last, this literal is not used as the starting literal of any other production.

The heuristic rules for the other categories involve a larger set of conditions. The main characteristic of the second category is that it is based on starting literals in context-free productions. We only consider a literal a starting literal if it only ever appears as the first part of a production pattern in all rules of the grammar. For the third category, we only consider productions with identical starting and end literals. Finally, for the fourth category we derive rules for matching starting and ending literals in `LAYOUT` productions.

```
module Java-15
...
context-free syntax
 "{" BlockStm* "}"        -> Block {cons("Block")}
 "(" Expr ")"             -> Expr  {bracket}
 "while" "(" Expr ")" Stm -> Stm {cons("While")}
context-free syntax
 "void" "." "class" -> ClassLiteral {cons("Void")}
 (Anno | ClassMod)* "class" Id ...
                  -> ClassHead {cons("ClassHead")}
```

Figure 10. A selection of context-free productions that appear in the Java grammar.

Note that we found that some grammars (notably the Java grammar of [6]) use kernel syntax for `LAYOUT` productions to more precisely control how comments are parsed. Thus, we consider both lexical and kernel syntax for the comment-terminating rules.

As an example, consider the context-free productions of Figure 10. Looking at the first production, and using the heuristic rules above, we can recognize that `}` qualifies as a closing literal. Likewise, `)` satisfies the conditions we have set. By programmatically analyzing the grammar in this fashion, we collected the set of closing literal insertion rules of Figure 9.

Note that none of the generated inserted closing literals of Figure 9 ever occur as an opening literal in the grammar. We only derive rules from brackets that appear in a balanced fashion with another (possibly different) literal (or a number of other literals). Insertions of literals that are balanced with another literal can lead to undesired results, since such constructs do not form a clear nesting structure. We make an exception for lexical productions that define strings and comments, for which we only derive more restrictive insertion rules.

From the productions of Figure 10 we can further derive the `{` and `(` opening literals. In particular, the “while” keyword is not considered for deriving an opening literal insertion rule, since it is not used in conjunction with a closing literal in its defining production.

No set of heuristic rules is perfect. For any kind of heuristic, an example can be constructed where it fails. We have encountered a number of anomalies that arose from our heuristic rules. For example, based on our heuristic rules, the Java `class` keyword is recognized as a closing literal. (For narrative reasons, we did not include it in Figure 9.) This follows from the void class literal production of Figure 10. The `class` keyword is never used as a starting literal of any production (as seen in the same figure, not even so for class headings), and therefore satisfies our set of rules. In practice, we have found that these anomalies are relatively rare and harmless, or sometimes even beneficial.

We evaluated our set of heuristic rules using Java, Java-SQL, Stratego and Stratego-Java grammars, as outlined in Section 7. For these grammars, a total number of respectively 33, 56, 36, and 130 insertion rules were generated, along with a constant number of water-based recovery rules as outlined in Figure 8. The complete set of derived rules is available from [1].

5.6 Customization of Permissive Grammars

A good error recovery mechanism is not only *language independent*, but is also *flexible* [10]. That is, it allows grammar engineers to use their experience with a language to improve recovery capabilities. Our system, while remaining within the realm of the standard SDF grammar specification formalism, delivers both of these properties. Language engineers can add their own recovery rules using SDF productions similar to those shown earlier in this section.

Using automatically derived rules may not always lead to the best possible recovery for a particular language. Different language constructs have different semantic meanings and importance. Different languages also may have different points where programmers often make mistakes. For example, a common "rookie" mistake in Stratego-Java is to use [| brackets |] instead of |[brackets]|. This may be recovered from by standard deletion and insertion rules. However, the cost of such a recovery is rather high, since it would involve four deletions and two insertions. Other alternatives, less close to the original intention of the programmer, might be preferred by the recovery mechanism. Based on this observation, a grammar engineer can add *substitution recovery rules* to the grammar:

```
lexical syntax
  "[|" -> "|[" {recover, cons("INSERT")}
  "|]" -> "]|" {recover, cons("INSERT")}
```

These rules substitute any occurrence of badly constructed embedding brackets with the correct alternative, at the cost of only a single recovery. Similarly, grammar engineers may add recovery rules for specific keywords, operators, or even placeholder identifiers as they see fit to further improve the result of the recovery strategy.

Modular Definition of Customizations It is good practice to separate the generated recovery rules from the customized recovery rules. This way, the generated grammar does not have to be adapted and maintained by hand. A separate grammar module can import the generated definitions, while adding new, handwritten definitions.

Besides composition, SDF also provides a mechanism for subtraction of languages. The {reject} disambiguation annotation filters all derivations for a particular set of symbols [3]. Using this filter, it is possible to disable some of the automatically derived recovery rules. Consider for example the insertion rule for the class keyword, which arose as an anomaly from the heuristic rules of the previous subsection. Rather than directly removing it from the generated grammar, we can disable it by extending the grammar with a new rule that rejects this recovery. Figure 11 illustrates this with a grammar that imports the generated permissive grammar, and disables the class insertion rule.

```
module Java-15-Permissive-Customized
imports
  Java-15-Permissive
exports
  lexical syntax
     -> "class" {reject}
    ...
```

Figure 11. A customized permissive grammar.

```
i = f ( x ) + 1  ;
i = f ( x   + 1 );
i = f ( x )      ;
i = f (       1 );
i =   ( x ) + 1  ;
i =   ( x   + 1 );
i =     x   + 1  ;
i = f            ;
i =   ( x )      ;
i =     x        ;
i =           1  ;
    f ( x   + 1 );
    f ( x )      ;
    f (       1 );
                 ;
```

Figure 12. Many different interpretations of i=f(x)+1; using literal-insertion recovery rules (underlined) and water-based recovery rules.

```
i   =     2    ;
i   =    (2)   ;
i   =   ((2))  ;
i   =  (((2))) ;
```

Figure 13. Insertion of opening brackets creates infinitely many possible interpretation of expressions.

6. Parsing Permissive Grammars

When all recovery rules are taken into account, permissive grammars provide many different interpretations of the same code fragment. As an example, Figure 12 shows all possible interpretations of the string i=f(x)+1;. The alternate interpretations are obtained by applying recover productions for inserting parentheses or removing text parts. This small code fragment illustrates the explosion in the number of ambiguous interpretations when using a permissive grammar. As illustrated in Figure 13, the option of inserting opening brackets even results in an infinite number of interpretations.

Generalized parsers explore all possible interpretations of a string in parallel. Any alternative that does not lead to a valid interpretation is simply discarded, and remaining branches may be filtered by disambiguation rules. Disambiguation can be performed during parsing, or by a post processor on the created parse tree. A disambiguation filter for

```
void methodX() {
   if (true) // missing {
      foo();
   }
   int j = 0;
   while (j < 8)
      methodY(j++);
}
```

Figure 14. An if statement with a missing opening brace, causing the method to be closed at the end of the statement The error is detected at the `while` keyword.

permissive grammars, should prefer branches with the least number of recover productions.

Theoretically, the use of productions to specify how to recover from errors provides an excellent mechanism for the parsing of erroneous files. However, from a practical point of view, the extra interpretations created by these productions negatively affect time and space requirements. As we have shown above, the combinatorial possibilities would grow exponentially, leading to unacceptable overhead on performance. Therefore, we must adapt the parser strategy, to process the alternatives introduced by the recovery rules.

It is not practical to consider all recovery interpretations in parallel with the ordinary grammar productions (or even impossible as in the case of Figure 13). As an alternative to parsing different interpretations in parallel, backtracking LR parsers revisit points of the file that allow multiple interpretations (the choice points). For normal grammars, these parsers are less practical since they exhibit exponential behavior in the worst case [17]. As such, we introduce a selective form of backtracking to GLR parsing that is only used for the concern of error recovery. We ignore all recovery productions during normal parsing, and employ backtracking to apply the recovery rules only once an error is detected.

6.1 Selecting Choice Points for Backtracking

To employ backtracking for applying error recovery rules it is important that the right choice point is selected. In particular, simply trying different interpretations at the point of failure is ineffective, especially when considering a scannerless parser. This is because the point of failure rarely reflects the location of the original error, let alone the point where the error can be repaired. Consider for example the code in Figure 14. Due to the missing opening brace of the if-block, the closing brace after the enclosed `foo();` statement is misinterpreted as closing the method. At that point, the parser simply continues, interpreting the remaining statements as class-body declarations. This causes it to fail at the reserved `while` keyword, which can only occur inside a method body. More precisely, our scannerless parser fails at the unexpected space after `w-h-i-l-e`; the character cannot be shifted and all branches (interpretations at that point) are discarded.

The difference between the point of detection and the actual location of the error is a well-known factor that poses a challenge for error recovery techniques [10]. In order to properly recover from non-local errors, they have to consider the text that precedes the point of detection. Backtracking can be used to inspect this text in reverse order, starting at the point of detection, gradually moving backwards to the start of the input file. Using a reverse order helps maintain efficiency, since the actual error is most probably near the failure location.

As SGLR parses different interpretations in parallel, it uses a more complicated stack structure than normal LR parsers. Instead of a single, linear stack, it maintains a graph-structured stack that efficiently stores the different interpretation branches. As characters are shifted, some of these branches may be discarded. This poses a challenge for applying backtracking, since all the discarded branches must be stored in case the old state is revisited. We found that it is prohibitive (in terms of performance) to maintain the complete stack state for every shifted character. Therefore, we only selectively record choice points to minimize the overhead introduced. In the current implementation, we construct a choice point at every new line.

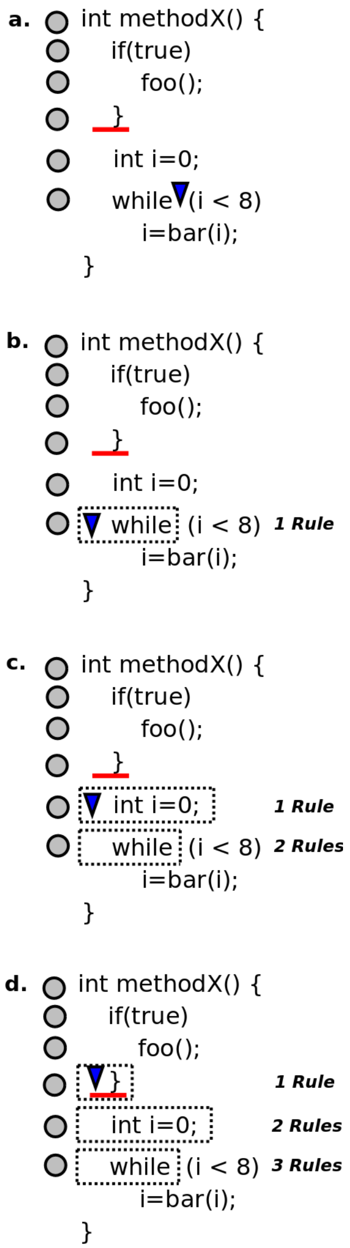
6.2 Applying Recovery Rules

Our backtracking algorithm iteratively explores the input stream in reverse order, starting at the nearest choice point. At each point, different candidate recoveries are attempted and discarded if a valid interpretation is not possible. An interpretation is considered valid after the error detection line is parsed. Once a valid candidate is selected, normal parsing continues. Permissive grammars typically account for many possible interpretations, which means that the order of exploration determines the final result. Generally, corrections that employ fewer recover productions are preferred. Therefore, candidate recoveries that use fewer recovery rules are selected first.

With each iteration of the algorithm, different candidate recoveries are explored in parallel for a restricted area of the file and for a restricted number of recovery rule applications. For every following iteration the size of the area and the maximum number of recovery rule applications are increased. Figure 15 illustrates a number of iterations of the algorithm for the Java method of Figure 14.

Figure 15a shows the parse failure after the `while` keyword. The point of failure is indicated by the triangle. The actual error, at the closing brace after the if-block, is shown underlined. The figure shows the different choice points that have been stored during parsing using circles in the left margin.

The first iteration of the algorithm (Figure 15b) focuses on the line where the parser failed. The parser is reset to the choice point at the start of the line, and enters recovery mode. At this point, only candidate recoveries that use one recovery production are considered; alternative interpretations formed



```
int methodX() {
    if(true)
        foo();
  }
    int i=0;
    while(i < 8)
        i=bar(i);
}
```

Figure 15. Expanding search space.

by a second recover production are cut off. Their exploration is postponed until the next iteration. The figure visualizes the search space using boxed lines with a number indicating the amount of recovery rules that may be applied. In this example scenario, the first iteration does not lead to a valid solution.

For the next iteration, in Figure 15c, the search space is expanded with respect to the size of the inspected area and the number of recovery rules that may be applied. The new search space consist of the line that precedes the point of detection, plus the error detection line where recovery candidates with two rules are now also considered. For the latter, interpretations that were previously cut off can now be resumed.

In Figure 15d, the search space is again expanded to inspect the line where the error was detected and the two preceding lines. This time, a valid recovery is found: discarding the closing brace, by application of a water-based recovery rule, leads to a valid interpretation of the erroneous code fragment.

Rather than a purely backtracking-based approach, we use a parallel search for recovery candidates with each backtracking step. Ordering of recovery candidates is only imposed by the different iterations of the backtracking algorithm. This means that in some cases more than one valid recovery candidate is identified. In this case we pick the candidate with the smallest number of recovery rule applications, or pick an arbitrary candidate if multiple candidates have the same number of applications.

6.3 Implementation

The implementation of the recovery algorithm requires a number of (relatively minor) modifications of the SGLR algorithm used for normal parsing. Firstly, productions marked with the `{recover}` attribute are ignored during normal parsing. Secondly, a choice point is stored for every newline character. And thirdly, if all branches are discarded and no accepting state is reached, the parser enters recovery mode. Once the recovery is successful, normal parsing resumes with a newly constructed stack.

Figure 16 shows pseudo code for the recovery algorithm. The `Recover` function controls the iterative search process described in Section 6.2. The function starts with some initial configuration (lines 3-5): it enables the recovery productions that are ignored in normal parsing mode, selects the most recent choice point, and initializes the `candidates` variable. The choice points from the point of failure are then visited in reverse order (lines 6-10). Candidates for the current choice point are collected and explored using the `RecoverParse` function (line 7), until finally a valid interpretation is found (line 10).

The `RecoverParse` function tries to construct a valid interpretation by reparsing the area starting from the choice point and revisiting recovery candidates that were previously cut off. Each iteration resets the parser to the previous state stored for the choice point (lines 20-21). It consumes characters from the choice point location until the original point of failure is reached (lines 23-32). By parsing the area again, any previously cut off candidate stacks are explored further and new candidates can be collected. For each character, the stack states of candidates at that point are merged with the current stack (line 25-27). As such, the revisited candidate recoveries are processed in parallel, using the standard SGLR parser (line 28), but with the recovery rules enabled. At that point, any new stack branches created using recovery rules are excluded from further exploration and stored to

```
RECOVER($sglr$)
1   ▷ Input: $sglr$ - SGLR parser
2
3   $sglr.ignoreRecoverProductions \leftarrow False$
4   $choicePoint \leftarrow$ Last inserted choicePoint
5   $candidates \leftarrow$ Initialize list
6   do
7       $candidates \leftarrow$ RECOVERPARSE(
8               $sglr, candidates, choicePoint$)
9       $choicePoint \leftarrow$ Previous choicePoint
10    until $sglr.stacks\ not\ empty$
11  $sglr.ignoreRecoverProductions \leftarrow True$

RECOVERPARSE($sglr, candidates, choicePoint$)
12  ▷ Input:
13      $sglr$ - SGLR parser
14      $candidates$ - Unexplored recover branches with
15              associated location, created on previous loop
16      $choicePoint$ - Start location for recover parse
17  ▷ Output: New candidates created
18      by one extra recover production
19
20  $sglr.stacks \leftarrow choicePoint.stacks$
21  $sglr.streamLocation \leftarrow choicePoint.streamLocation$
22  $newCands \leftarrow$ Initialize list
23  while $sglr.streamLocation \leq sglr.failureLocation$
24  do
25      $locCands \leftarrow$ { c | c in $candidates\ and$
26              $c.streamLocation$ = $sglr.streamLocation$ }
27      $sglr.stacks \leftarrow sglr.stacks \cup locCands$
28      $sglr.parseCharacter()$
29      $createdCands \leftarrow$ stacks created by
30                          recover production
31      $sglr.stacks \leftarrow sglr.stacks$ / $createdCands$
32      $newCands \leftarrow newCands \cup createdCands$
33  return $newCands$
```

Figure 16. The recovery algorithm.

be revisited in the next iteration (line 31-32). The algorithm ends once the character at the point of failure can be successfully parsed. The parser then resumes in normal parsing mode (line 11) using the newly constructed stack.

7. Evaluation

We add error recovery to a parser to improve the user experience. This is particularly important in an interactive environment where many editor services depend on proper parse error recovery, as opposed to batch use. With this in mind, we have focused the evaluation of our approach on the following criteria:

- **Quality of Recovery**: recovery should be as close as possible to the intention of the user. A bad recovery may introduce spurious errors and lead to misleading error feedback.
- **Performance** of the parser, with regards to parse time and space, is important and should not disturb the work flow of the user. A significant factor in interactive scenarios is that parser input will, more often than not, contain errors.
- **Quality of Feedback**: good error feedback should point out an error as accurately as possible and also give good suggestions on possible corrections. Quality of feedback depends on quality of recovery.
- **Language Independence and Flexibility**: the recovery solution should be independent of a particular language, yet should be customizable to the needs and insights of language designers.
- **Transparency**: it should be clear why a particular recovery is presented. The language designer should have insight into how the recovery works for a given grammar.

As a basis for testing we have used a set of automatically derived, permissive grammar variants for Java, Java-SQL, Stratego and Stratego-Java. For each language we have tested the following permissive variants: Water (W), Insertion of close (C), W + C, C + Insertion of open (O) and W + C + O, along with standard grammars for comparison. These grammars have been tested on a set of sample files taken from the following projects:

- **The JUnit framework**: A library for defining unit tests [15], providing Java 1.5 code used to test the Java grammars.
- **The Dryad compiler**: An open compiler for the Java platform [19], providing Stratego and Stratego-Java code which we use to test both the Stratego and Stratego-Java grammars.
- **The StringBorg project**: A tool and grammar suite that defines different embedded languages [4], providing Java-SQL code used to test the Java-SQL grammars.

Code from these projects provide correct samples which we use to test the parser performance for syntactically correct files. We introduced errors to the samples to simulate editing scenarios that result in syntactically erroneous code. These error samples are used to test performance and quality of recovery in the presence of syntactical errors.

Both the development of a representative error sample suite and its evaluation have been a challenging task. There are many factors involved: the type of error, type of grammar, locations of errors, etc. For example, the distribution of errors has an impact on time spent in recovery and the number of recoveries triggered. Clustered errors are expected to trigger fewer recoveries compared to scattered errors. Still, the actual runtime behavior of the parser remains hard to predict.

To evaluate the recovery quality we have opted to focus on single errors. For error performance we also focus on small sets of clustered errors, as these have a large impact on the runtime behavior. All measures related to time have been collected as averages after several runs, using a "pre-heated" JVM. All figures shown in this paper should be considered as preliminary with the purpose of showing the effectiveness of our approach, not as a part of an in-depth study.

7.1 Quality of Recovery

For the evaluation of recovery quality we first measure the tree alignment distance [16], i.e., comparing the reference AST to ASTs obtained through recovery. This is one possible way to get a number on the difference of two trees as opposed to, e.g, running diffs on pretty printed ASTs and measure number of lines. The reference AST is obtained from a correct sample file, while the recovered ASTs are created from the error samples, following [28].

Given the tree distances we study recoveries in more detail in cases where the distance is larger than zero. We label the quality of a recovery as either *poor*, *good* or *excellent*, as suggested by Pennello and DeRemer [30], also used in the comparative study of Degano and Priami [10]. A tree alignment distance equal to zero indicates an excellent recovery while a larger distance indicates a poor or good recovery. Poor recoveries are recoveries that introduce spurious errors (as observed through manual inspection of the parse result).

We study the permissive grammars for Stratego-Java in more detail for sample files containing one error. The set of error samples capture six different types of errors:

- **Missing Close** and **Added Close**: `)`, `}`, `>`, `]`, `]\!|`, or `*/`
- **Missing Open**: `(`, `{`, or `|[`
- **Missing Delim** and **Added Delim**: `;` or `,`
- **Changed** order of symbols: `[|` instead of `|[`, or `|]` instead of `]|`

Figure 17 shows distance results per error and grammar type, while Figure 18 shows the quality of recoveries as a percentage of the whole error set.

Figure 18 shows that the standard grammar fails to parse all inputs (as expected), and that most of the permissive grammar succeed in parsing the complete set. An exception is the C variant which only manages to recover form a subset of all errors. The combined variants WC, CO and WCO even get excellent recoveries for a majority of cases.

The diagram in Figure 17 shows the effect of using a grammar variant on different error types. For the W variant we see that it is robust but gives large distances, with the exception of the added delimiter cases. The C variant, on the other hand, is less robust but gives excellent recoveries for a majority of the missing close cases. The O variant is not tested separately, but we see that it provides excellent recoveries for a majority of the missing open cases when it is included in a combination.

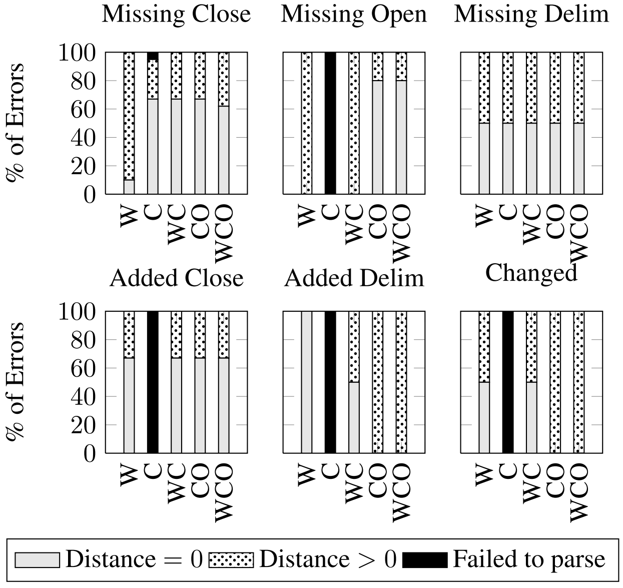


Figure 17. Distance: each diagram shows results for one error type and the following grammar variants; W, C, WC, CO and WCO. Each bar shows for what percentage the distance was equal to zero, larger than zero or parsing failed. **W** - Water, **C** - Insertion of close, **O** - Insertion of open.

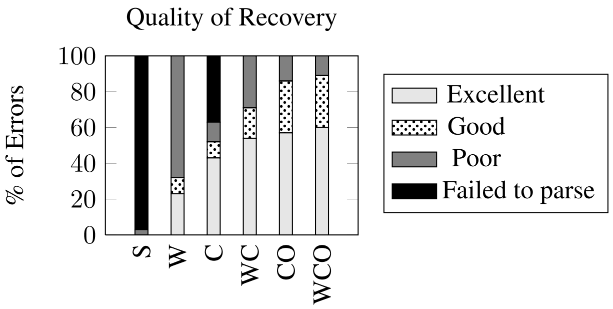


Figure 18. Quality of recovery using various grammars for Stratego-Java. Each bar shows for what percentage recovery was excellent, good, poor or parsing failed. **S** - Standard grammar, **W** - Water, **C** - Insertion of close, **O** - Insertion of open.

7.2 Parsing Performance

For evaluation of parsing performance we focus on parse time. Space is also of interest but not addressed in this paper. As mentioned earlier, a parser used in an interactive scenario should not slow down the work flow of the user. Adding recovery to a parser will potentially give overhead with regard to time and space.

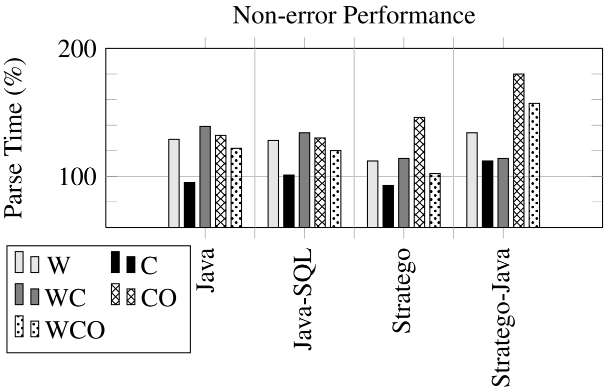


Figure 19. Non-error performance for permissive Java, Java-SQL, Stratego and Stratego-Java grammars. Values show time percentage compared to a standard grammar for each corresponding language. **W** - Water, **C** - Insertion of close and **O** - Insertion of open.

Note that the current implementation of JSGLR is still in active development and has not been optimized. Particularly, algorithmic optimizations such as SRNGLR [12] have not been integrated yet, and an overuse of object allocations and wrapper objects currently causes a lack of memory locality and increase in heap usage.

We are interested in the performance overhead of using permissive production rules on both correct and erroneous input. Figure 19 shows non-error parse time, i.e. the average parse time for permissive grammars in relation to the standard grammar for each language. We observe that the overhead varies between variants and languages. The W variant gives more or less the same overhead independent of language, while the C variant gives the best performance for all languages, even better performance in two cases.

An important property of any parser used in an interactive environment is its ability to handle erroneous input. Using the WC grammar for Stratego-Java, which gives us the best balance between quality and performance, we measure how much additional time that is needed to parse files with one, two, or three errors. The results are shown in Figure 20. The figure shows that the amount of time spent in recovery increases with the number of clustered errors. This is the expected result since clustered errors are bound to trigger more recoveries.

The diagram in Figure 21 shows us that the number of errors in a sample will result in an equal number of recoveries for a majority of the cases. For occasional errors, clustered and single, recovery time can exceed 1 second. The percentage increases with the number of clustered errors. There are some pathological cases for which an additional strategy is required to avoid overly long parse times. This is further addressed in Section 7.3.

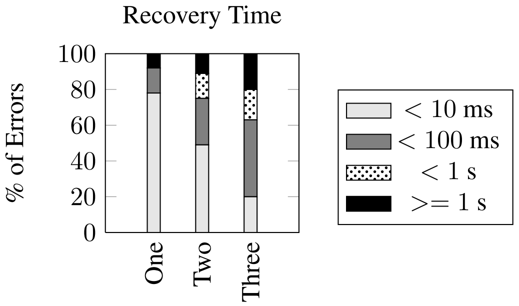


Figure 20. Recovery time for samples with one, two or three errors, using the Stratego-Java W+C grammar variant (water and closing-literal insertion). Each bar shows for what percentage recovery time is less than 10 ms, 100 ms, 1 s or greater than 1 s.

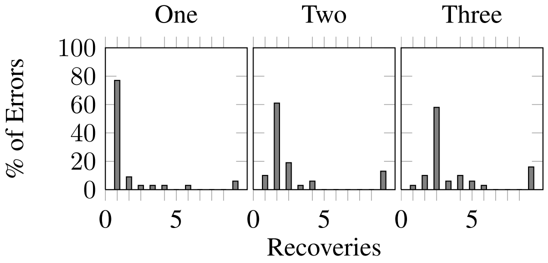


Figure 21. Recovery count for one, two and three errors using the Stratego-Java W+C grammar variant, i.e., water and insertion of close.

7.3 Recovery from Pathological Cases

A good error recovery strategy maintains a fine balance between response time and the quality of a recovery. Based on our test set, we have recognized that there are certain pathological cases where the complete recovery rule set takes too long to find a proper recovery.

Through experimentation, we found that these pathological cases arise seldomly and are hard to predict or recognize. They only occur for particular combinations of syntax errors and surrounding context. To avoid slow response due to long-running recovery attempts, we abort attempts that take more than a set amount of time.

We speculate that these cases arise from a combination of syntax errors that can only be resolved by a multitude of recovery operations and the presence of particularly liberal productions in the base grammar. For instance, string literals in Stratego may contain any character, including newlines, except an unescaped closing quote. Any unterminated string literal, or a string literal that has been partially discarded by a water-based recovery rule, can greatly increase the possible number of interpretations of an input file, i.e., almost any character in the remainder of the file could be part of the string (or not).

Successful error recovery typically combines multiple strategies, using a secondary strategy if the first does not suffice [10]. In order to still provide feedback for the pathological cases, such a secondary strategy can be used. This strategy should focus purely on performance, not quality of recovery. We have identified three viable secondary strategies for error recovery with SGLR.

The first approach is simply to abort the current recovery attempt and report all offending characters encountered so far. With this approach, it is not possible to recover a partial AST, but the user is still presented with feedback of errors up to the point of where the parser is aborted.

A second fall-back strategy is to construct a partial parse tree based on the reductions performed by the parser so far. This approach is described in [36], where a standard, unaltered SDF grammar is used. The available syntax tree at the point of abortion is combined with remaining parts of the file to create a partial parse result.

Finally, a third fall-back strategy is to confine errors to selected regions of code. Each region can constrain the search space used to recover from errors in that region. If recovery is not possible within a set time limit, an entire region can be discarded. Regions that enclose syntax errors may be detected using coarse-grained, water-based recovery grammars similar to those discussed in Section 5.1. These grammars parse any of the pathological cases almost as quickly as a non-erroneous input, identifying and skipping over erroneous regions of code. In ongoing experimentation we have also seen good results in using indentation and the SGLR graph-structured stack to identify regions while parsing. We plan to refine these strategies in future work.

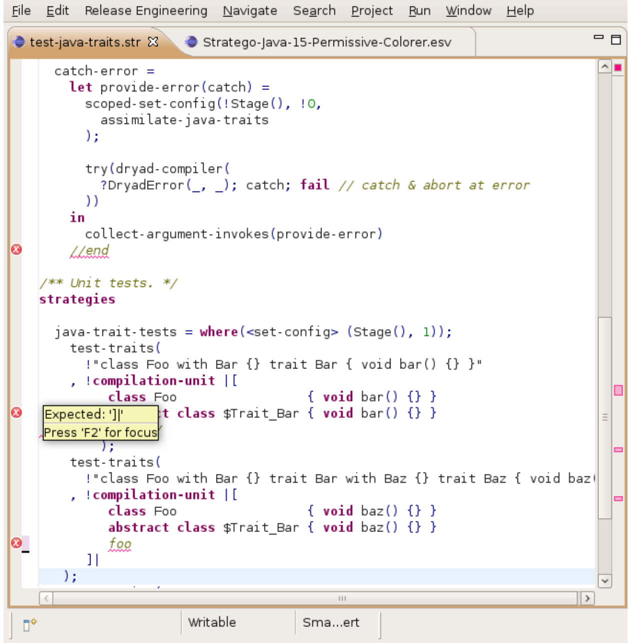
7.4 Interactive Editing and Error Reporting

A parser used in an interactive scenario should provide a user with good error feedback. Perhaps, just as important is the placement of error markers (“squiggles”) in the editor. Our current implementation does this based on the layout of the source code, as illustrated in Figure 22.

From the screen shot we can gather that the editor shows several errors, due to successful recovery. The first and the second errors are recovered by a literal-insertion rule, while the third error has been recovered using the water-based recovery rule. We also observe that errors are detected correctly in both Stratego and embedded Java.

Due to the different characteristics of the error recovery rules, described in Section 5.5, we use different error messages based on the constructor of the recovery rule:

- **WATER rules**: “[string] not expected.”
- **INSERT rules**: “[string] expected.” The placement of the error marker is particularly important in this case. For constructs with an opening and closing literal, this should be at the same level of indentation as the other literal.
- **INSERTEND rules**: “construct not terminated.” Highlights entire construct from start to end.



```
  catch-error =
    let provide-error(catch) =
      scoped-set-config(!Stage(), !0,
        assimilate-java-traits
      );

      try(dryad-compiler(
        ?DryadError(_, _); catch; fail // catch & abort at error
      ))
    in
      collect-argument-invokes(provide-error)
    //end

/** Unit tests. */
strategies

  java-trait-tests = where(<set-config> (Stage(), 1));
    test-traits(
      !"class Foo with Bar {} trait Bar { void bar() {} }"
    , !compilation-unit |[
         class Foo                { void bar() {} }
          t class $Trait_Bar { void bar() {} }
       );
    test-traits(
      !"class Foo with Bar {} trait Bar with Baz {} trait Baz { void baz
    , !compilation-unit |[
         class Foo                { void baz() {} }
         abstract class $Trait_Bar { void baz() {} }
         foo
      ]|
   );
```

Expected: ']|'
Press 'F2' for focus

Figure 22. An editor for Stratego-Java using a permissive W+C grammar variant (water and closing-literal insertion).

Syntax highlighting in Spoofax/IMP is implemented in a scannerless fashion [20]. This means that the parse tree is used for colorization, rather than the token stream. This makes it important that a proper parse tree is available at all times. Our experience shows us that even when interactively editing a program, the coloring remains consistent. However, since highlighting is based on successful parsing of a string, any erroneous segments of code that have been discarded cannot use keyword highlighting.

Compared to hand-written parsers, which are commonly used in interactive editing, our error reports tend to be of a more generic nature. For example, for an unterminated string, our editor gives a generic message “construct not terminated” instead of a language-specific message “string literal not terminated.”

7.5 Language Independence and Flexibility

The permissive grammars we evaluated in this section were automatically constructed using the heuristic rules described in Section 5.5, showing that the approach works independently of a particular language. Yet by using derived recovery rules, specified as normal SDF productions, transparency is maintained.

In Section 5.6 we discussed the customizability of derived grammars; additional recovery rules can be added and undesired rules can be removed to improve the recovery quality. Based on the results of a test set as we constructed for our evaluation, permissive grammars can be manually tuned in order to improve the results.

8. Related Work

The SGLR parser, implementing the scannerless generalized-LR algorithm, previously lacked any form of error recovery, limited to reporting only the character and location at the point of a parse failure. However, there has been some exploratory work on the subject by Valkering [36]. Valkering enhances the standard SGLR error reports with information based on the current stack at the point where an error occurs, reporting the set of possible strings that could be inserted at that point. Providing good feedback this way is non-trivial since scannerless parsing does not employ tokens; often it is only possible to report a set of expected *characters* instead. Furthermore, these error reports are still biased with respect to the location of errors; because of the scannerless, generalized nature of the parser, the point of failure rarely is a good indication of the actual location of a syntactic error. In contrast, our error reporting strategy is based on the notions of backtracking and considering least-cost error recovery productions to determine the most likely cause of the error in the context.

Using artificial reduce actions, Valkering constructs a partial parse tree that precedes the point of failure. Furthermore, he constructs partial parse trees for the fragments that follow an error using substring parsing, introduced by Rekers and Koorn [31]. Based on these partial parsing techniques, the result of the parser is a set of (often highly ambiguous) partial parse trees. In our approach, a single, well-formed parse tree is constructed instead.

Lavie and Tomita developed GLR*, a noise skipping algorithm for context-free grammars [25]. Based on traditional GLR with a scanner, their parser determines the maximal subset of all possible interpretations of a file by systematically skipping selected tokens. The parse result with the fewest skipped words is then used as the preferred interpretation. In principle, the GLR* algorithm could be adapted to be scannerless, skipping characters rather than tokens. However, doing so would lead to an explosion in the number of interpretations. In our approach, we restrict these by using backtracking to only selectively consider the alternative interpretations, and using water-based recovery rules that skip over chunks of characters. Furthermore, our approach supports insertions in addition to discarding noise and provides more extensive support for reporting errors.

Island Grammars The basic principles of our permissive grammars are based on the water productions from island grammars. Island grammars [39, 26] have traditionally been used for different reverse and re-engineering tasks. For cases where a baseline grammar is available (i.e., a complete grammar for some dialect of a legacy language), Klusener et al [22] present an approach of deriving *tolerant grammars*. Based on island grammars, these are partial grammars that contain only a subset of the baseline grammar's productions, and are more permissive in nature. Unlike our permissive grammars, tolerant grammars are not aimed at application in an interactive environment. They do not support the notion of reporting errors, and, like parsing with GLR*, are limited to skipping content. Our approach supports recovery rules that insert missing literals and provides an extended set of error reporting capabilities.

More recently, island grammars have also been applied to parse composite languages. Synytskyy et al [34] composed island grammars for multiple languages to parse only the interesting bits of an HTML file (e.g., JavaScript fragments and forms), while skipping over the remaining parts. In contrast, we focus on composite languages constructed from complete constituent grammars. From these grammars we construct permissive grammars that support tolerant parsing for complete, composed languages.

Error Handling and Recovery in Other Parsers There are several different forms of error recovery techniques for LR parsing [10]. These techniques can be divided in *correcting* and *non-correcting* techniques. The most common non-correcting technique is *panic mode*. On detection of an error, the input is discarded until a synchronization token is reached. Then, states are popped from the stack until the state at the top enables the resumption of the parsing process. Panic mode does not provide a proper diagnosis of the error and may skip large fragments of an input. Correcting methods try to improve on this by attempting to correct the flawed part of the input string. Correcting methods for LR parsers typically attempt to insert or delete tokens nearby the location of an error, until parsing can resume. Successful recovery mechanisms often combine more then one technique. For example, panic mode is often used as a fall back method if the correction attempts fail.

While our approach to error recovery follows along the same lines as is common to general correcting methods, there are also two significant differences due to the nature of SGLR. Other error reporting methods use tokens for reporting parse errors. Lacking tokens, our method is based on the identification of errors – such as missing literals – through parsing with the recovery production rules. GLR parsing also introduces the notion of multiple branches that are processed in parallel. Parse errors can only be identified by a failure of the last remaining branch, which may not be local to the actual root cause of an error. While other approaches can simply identify the offending token, we apply backtracking to track back to the offending point in the code.

An alternative approach to scannerless parsing is used for parsing expression grammars (PEGs) [14]. The class of languages PEGs can express has no relation to the context-free grammars supported by SDF: instead of the commutative choice (`|`) operator, PEGs use an ordered choice (`/`). PEGs lack the explicit disambiguation facilities [3] that SDF provides for SGLR, and instead use ordered choice to enforce an ordering of production alternatives, combined with greedy matching. To our knowledge, no automated form of error recovery has been defined for PEGs. However, based on the ordering property, a "catch all" clause is sometimes added to productions, which is used if no other alternative succeeds. Such a clause can be used to skip erroneous content up to a specific point (such as a newline) but does not offer the flexibility of our approach.

IDE support for composite languages We integrated our recovery approach into the Spoofax/IMP [20] language development environment. A related project, also based on SDF and SGLR, is the Meta-Environment [37, 38]. It currently does not employ interactive parsing, and only parses files after a "save" action from the user. Using the traditional SGLR implementation, it also provides no error recovery.

Another language development environment is MontiCore [23, 24]. Based on ANTLR [29], it uses traditional LL(k) parsing. As such, MontiCore offers only limited support for language composition and modular definition of languages. Combining grammars can cause conflicts at the context-free or lexical grammar level. For example, any keyword introduced in one part of the language is automatically recognized by the scanner as a keyword in another part. MontiCore supports a restricted form of embedded languages through run-time switching to a different scanner and parser for certain tokens. Using the standard error recovery mechanism of ANTLR, it can provide error recovery for the constituent languages. However, recovery from errors at the edges of the embedded fragments (such as missing quotation brackets), is more difficult using this approach. This issue is not addressed in the papers on MontiCore [23, 24]. In contrast to MontiCore, our approach is based on scannerless generalized-LR parsing, which supports the full set of context-free grammars, and allows composition of grammars without any restrictions.

9. Conclusion

The SDF formalism allows for the specification and composition of modular, declarative language definitions. Parse error recovery for parsing SDF grammars with SGLR has previously been identified as an open issue. In this paper, we presented a flexible, language-independent approach to error recovery to resolve this issue. The three pillars of our work have been to use standard SDF productions to specify error recovery rules; to derive such error recovery rules from SDF grammars; and to adapt the SGLR parser to efficiently cope with the added complexity of grammars with recovery rules. Using these techniques, we can support rapid syntactic and semantic feedback for compositional languages as programs are edited.

We evaluated our approach using a set of existing, nontrivial grammars, showing a low performance overhead in case there are no syntax errors, and acceptable (although at times unpredictable) overhead in case of inputs with errors. The results also show that our approach achieves excellent recovery quality in a majority of the cases.

Our backtracking algorithm employs a growing search space and a heuristic approach to systematically explore different possible recoveries in case of an error. We expect that in the future we can further tune the weights of the different factors that play a role in this process to provide more intuitive recoveries. For example, while any single insertion may be preferred to discarding a substring (as suggested in [10]), larger clusters of insertions are often less desirable – and harder to identify in a growing search space – than discarding longer substrings. Overall, we expect to improve both the performance and quality through ongoing experimentation and evaluation.

Acknowledgements This research was supported by NWO/JACQUARD projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*. This work would not have been possible without the efforts of Karl Trygve Kalleberg, whose Java implementation of SGLR has been invaluable for the present work and the integration of SGLR into the Eclipse environment. We thank Mark van den Brand, Martin Bravenboer, Giorgios Rob Economopoulos, Jurgen Vinju, and the rest of the SDF/SGLR team for their work on SDF.

References

[1] The permissive grammars project. `http://strategoxt.org/Stratego/Permissive-Grammars`, 2009.

[2] 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.

[3] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC 2002)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 2002.

[4] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. A host and guest language independent approach. In J. Lawall, editor, *Generative Programming and Component Engineering (GPCE 2007)*, pages 3–12. ACM, 2007.

[5] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[6] M. Bravenboer, E. Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. A case for scannerless generalized-LR parsing. In W. R. Cook, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2006)*, pages 209–228. ACM, 2006.

[7] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2004)*, pages 365–383. ACM, 2004.

[8] P. Charles, R. M. Fuhrer, and S. M. Sutton, Jr. IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *Automated Software Engineering (ASE 2007)*, pages 485–488. ACM, 2007.

[9] P. Charles, R. M. Fuhrer, S. M. Sutton, Jr., E. Duesterwald, and J. Vinju. Accelerating the creation of customized, language-specific IDEs in Eclipse. In G. T. Leavens, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2009)*. ACM, 2009.

[10] P. Degano and C. Priami. Comparison of syntactic error handling in LR parsers. *Software – Practice and Experience*, 25(6):657–679, 1995.

[11] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

[12] G. Economopoulos, P. Klint, and J. Vinju. Faster scannerless GLR parsing. In O. de Moor and M. I. Schwartzbach, editors, *Compiler Construction (CC'09)*, pages 126–141. Springer-Verlag, 2009.

[13] S. Efftinge et al. openArchitectureWare User Guide. Version 4.3. Available from `http://openarchitectureware.org/pub/documentation/`, 2008.

[14] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *International Conference on Functional Programming (ICFP'02)*, volume 37 of *SIGPLAN Notices*, pages 36–47. ACM, October 2002.

[15] E. Gamma and K. Beck. JUnit: A cook's tour. *Java Report*, 4(5):27–38, 1999.

[16] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. In *CPM '94: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, volume 807 of *LNCS*, pages 75–86, London, UK, 1994. Springer-Verlag.

[17] A. Johnstone, E. Scott, and G. Economopoulos. Generalised parsing: Some costs. *Lecture Notes in Computer Science*, 2985:89–103, 2004.

[18] K. T. Kalleberg. JSGLR. `http://www.spoofax.org/`.

[19] L. C. L. Kats, M. Bravenboer, and E. Visser. Mixing source and bytecode. A case for compilation by normalization. In G. Kiczales, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2008)*, pages 91–108. ACM, 2008.

[20] L. C. L. Kats, K. T. Kalleberg, and E. Visser. Domain-specific languages for composable editor plugins. In T. Ekman and J. Vinju, editors, *Language Descriptions, Tools, and Applications (LDTA 2009)*, ENTCS. Elsevier Science Publishers, 2009.

[21] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'07)*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[22] S. Klusener and R. Lämmel. Deriving tolerant grammars from a base-line grammar. In *International Conference on Software Maintenance (ICSM '03)*, pages 179–189. IEEE Computer Society, 2003.

[23] H. Krahn, B. Rumpe, and S. Völkel. Efficient editor generation for compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, technical report TR-38, pages 218–228. University of Jyväskylä, 2007.

[24] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In R. Paige and B. Meyer, editors, *TOOLS EUROPE 2008*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer-Verlag, June 2008.

[25] A. Lavie and M. Tomita. GLR*-an efficient noise skipping parsing algorithm for context free grammars. In *Third International Workshop on Parsing Technologies*, pages 123–134, 1993.

[26] L. Moonen. Generating robust parsers using island grammars. In *Working Conference on Reverse Engineering (WCRE'01)*, pages 13–22. IEEE Computer Society Press, Oct 2001.

[27] L. Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th IEEE International Workshop of Program Comprehension*, pages 219–228. IEEE Computer Society, 2002.

[28] E. Nilsson-Nyman, T. Ekman, and G. Hedin. Practical scope recovery using bridge parsing. In D. Gasevic, R. Lämmel, and E. V. Wyk, editors, *Software Language Engineering (SLE 2008)*, volume 5452 of *LNCS*, pages 95–113. Springer, 2008.

[29] T. Parr and R. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[30] T. J. Pennello and F. DeRemer. A forward move algorithm for LR error recovery. In *Principles of programming languages (POPL '78)*, pages 241–254. ACM, 1978.

[31] J. Rekers and W. Koorn. Substring parsing for arbitrary context-free grammars. *SIGPLAN Not.*, 26(5):59–66, 1991.

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

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

[34] N. Synytskyy, J. Cordy, and T. Dean. Robust multilingual parsing using island grammars. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 266–278. IBM Press, 2003.

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

[36] R. Valkering. Syntax error handling in scannerless generalized LR parsers. Master's thesis, University of Amsterdam, 2007.

[37] M. G. J. van den Brand, M. Bruntink, G. R. Economopoulos, H. A. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J. J. Vinju. Using the Meta-Environment for maintenance and renovation. In *The European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 331–332. IEEE Computer Society, 2007.

[38] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[39] A. van Deursen and T. Kuipers. Building documentation generators. In *IEEE International Conference on Software Maintenance (ICSM '99)*, page 40. IEEE Computer Society, 1999.

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

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

[42] D. Waddington and B. Yao. High-fidelity C/C++ code transformation. *Sci. Comput. Program.*, 68(2):64–78, 2007.

TUD-SERG-2009-020a
ISSN 1872-5392