Delft University of Technology
Software Engineering Research Group
Technical Report Series

Natural and Flexible Error Recovery for Generated Modular Language Environments

Maartje de Jonge, Lennart C.L. Kats, Emma Soderberg, and Eelco Visser

Report TUD-SERG-2012-021

TUD-SERG-2012-021

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/

Note: Accepted for publication in TOPLAS

A

Natural and Flexible Error Recovery for Generated Modular Language Environments

MAARTJE DE JONGE, LENNART C. L. KATS and EELCO VISSER, Technical University Delft
EMMA SÖDERBERG, Lund University

Integrated development environments (IDEs) increase programmer productivity, providing rapid, interactive feedback based on the syntax and semantics of a language. 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 languages composed from separate grammar modules. To apply this algorithm in an interactive environment, this paper introduces a novel error recovery mechanism. Our approach is language-independent, and relies on automatic derivation of recovery rules from grammars. By taking layout information into consideration it can efficiently suggest natural recovery suggestions.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*program editors*; D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax*

General Terms: Languages, Algorithms, Design

Additional Key Words and Phrases: Error recovery, generalized parsing

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 programmers 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.

One burden in particular for textual languages is the development of a parser. Modern IDEs use a parser to obtain the syntactic structure of a program with every change that is made to it, ensuring rapid syntactic and semantic feedback as a program is edited. As programs are often in a syntactically invalid state as they are edited, parse error recovery is needed to diagnose and report parse errors, and to construct a valid abstract syntax tree (AST) for syntactically invalid programs. Thus, to successfully apply a parser in an interactive setting, proper parse error recovery is essential.

The development and maintenance costs of complete parsers with recovery support are often prohibitive when general-purpose programming languages are used for their construction. Parser generators address this problem by automatically generating a working parser from a grammar definition. They significantly reduce the development time of the parser and the turnaround time for changing it as a language design evolves.

In this paper we show how generated parsers can both be general – supporting the full class of context-free languages – and automatically provide support for error recovery. Below we elaborate on these aspects, describe the challenges in addressing them together, and give an overview of our approach.

Generalized parsers. A limitation of most parser generators is that they only support certain subclasses of the context-free grammars, such as LL(k) grammars or LR(k) grammars, reporting conflicts for grammars outside that grammar class. Such restrictions on grammar classes make it harder to change grammars – requiring refactoring – and prohibit the composition of grammars as only the full class of context-free grammars is closed under composition [Kats et al. 2010].

Generalized parsers such as generalized LR support the full class of context-free grammars with strict time complexity guarantees[1]. By using scannerless GLR (SGLR) [Visser 1997b], even scanner-level composition problems such as reserved keywords are avoided.

[1]Generalized LR [Tomita 1988] parses deterministic grammars in linear time and gracefully copes with non-determinism and ambiguity with a cubic worst-case complexity.

Error recovery. To provide rapid syntactic and semantic feedback, modern IDEs interactively parse programs as they are edited. A parser runs in the background with each key press or after a small delay passes. As the user edits a program, it is often in a syntactically invalid state. Users still want editor feedback for the incomplete programs they are editing, even if this feedback is incomplete or only partially correct. For services that apply modifications to the source code such as refactorings, errors and warnings can be provided to warn the user about the incomplete state of the program. These days, the expected behavior of IDEs is to provide editor services, even for syntactically invalid programs.

Parse error recovery techniques can diagnose and report parse errors, and can construct a valid AST for programs that contain syntax errors [Degano and Priami 1995]. The recovered AST forms a speculative interpretation of the program being edited. Since all language specific services crucially depend on the constructed AST, the quality of this AST is decisive for the quality of the feedback provided by these services. Thus, to successfully apply a parser in an interactive setting, proper parse error recovery is essential.

Challenges. Three important criteria for the effectiveness and applicability of parser generators for use in IDEs are *1)* the grammar classes they support, *2)* the performance guarantees they provide for those grammar classes, and *3)* the quality of the syntax error recovery support they provide. Parse error recovery for generalized parsers such as SGLR has been a long-standing open issue. In this paper we implement an error recovery technique for generalized parsers, thereby showing that all three criteria can be fulfilled.

The scannerless, generalized nature of SGLR parsers poses challenges for the diagnosis and recovery of errors. We have identified two main challenges. First, generalized parsing implies parsing 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, increasing the difficulty of diagnosis and recovery. Second, scannerless parsing implies that there is no separate scanner for tokenization and that errors cannot be reported in terms of *tokens*, but only in terms of *characters*. This results in error messages about a single erroneous character rather than an unexpected or missing token. Moreover, common error recovery techniques based on token insertion and deletion are ineffective when applied to characters, as many insertions or deletions are required to modify complete keywords, identifiers, or phrases. Together, these two challenges make it harder to apply traditional error recovery approaches, as scannerless and generalized parsing increases the search space for recovery solutions and makes it harder to diagnose syntax errors and identify the offending substring.

Approach overview. In this paper we address the above challenges by introducing additional "recovery" production rules to grammars that make it possible to parse syntax-incorrect inputs with added or missing substrings. These rules are based on the principles of island grammars (Section 3). We show how these rules can be specified and automatically derived (Section 4), and how with small adaptations to the parsing algorithm, the added recovery rules can be activated only when syntax errors are encountered (Section 5). By using the layout of input files, we improve the quality of the recoveries for scoping structures (Section 6), and ensure efficient parsing of erroneous files by constraining the search space for recovery rule applications (Section 7).

Contributions. This paper integrates and updates our work on error recovery for scannerless, generalized parsing [Kats et al. 2009; de Jonge et al. 2009] and draws on our work on bridge parsing [Nilsson-Nyman et al. 2009]. We implemented our approach based on the modular syntax definition formalism SDF [Heering et al. 1989; Visser 1997c] and JSGLR[2], a Java-based implementation of the SGLR parsing algorithm. The present paper introduces general techniques for the implementation of an IDE based on a scannerless, generalized parser, and evaluates the recovery approach using automatic syntax error seeding to generate representative test sets for multiple languages.

[2] `http://strategoxt.org/Stratego/JSGLR/`.

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

Fig. 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*
```

Fig. 2. Program transformation using embedded object language syntax.

2. COMPOSITE LANGUAGES AND GENERALIZED PARSING

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 [Ducasse et al. 2006] or aspects [Kiczales et al. 1997] 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's enumerations and lambda expressions.

Examples of language embeddings include database query expressions integrated into an existing, general-purpose language such as Java [Bravenboer et al. 2010]. Such an embedding both increases the expressiveness 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 meta-programming for quotation of their object language [Visser 2002]. Transformation languages such as Stratego [Bravenboer et al. 2008] and ASF+SDF [van den Brand et al. 2002] 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 meta-variables (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.

2.1. Parsing Composite Languages

The key to effective realization of composite languages is a modular, reusable language description, which allows 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 [Salomon and Cormack 1989; 1995], these issues can be elegantly addressed [Bravenboer et al. 2006].

The *Scannerless Generalized-LR* (SGLR) parsing algorithm [Visser 1997b] realizes scannerless parsing by incorporating the generalized-LR parsing algorithm [Tomita 1988]. 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 by efficiently trying 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.[3]

Language composition often results in grammars that contain ambiguities. Generalized parsing allows declarative disambiguation of ambiguous interpretations, implemented as a filter on the parse tree, or rather the *parse forest*. As an alternative to parsing different interpretations in parallel, *backtracking parsers* revisit points of the file that allow multiple interpretations. Backtrack parsing is not generalized parsing since a backtracking parser only explores one possible interpretation at a time, stopping as soon as a successful parse has been found. In the case of ambiguities, alternative parses are hidden, which precludes declarative disambiguation.

Non-determinism in grammars can negatively affect parser performance. With traditional backtracking parsers, this would lead to exponential execution time. Packrat parsers use a form of backtracking with memoization to parse in linear time [Ford 2002]; but, as with other backtracking parsers, they greedily match the first possible alternative instead of exploring all branches in an ambiguous grammar [Schmitz 2006]. In contrast, GLR parsers explore all branches in parallel and run in cubic time in the worst case. Furthermore, they have the attractive property that they parse the subclass of deterministic LR grammars in linear time. While scannerless parsing tends to introduce additional non-determinism, the implementation of parse filters during parsing rather than as a pure post-parse filter eliminates most of this overhead [Visser 1997a].

2.2. Defining Composite Languages

The syntax definition formalism SDF [Heering et al. 1989; Visser 1997c] integrates lexical syntax and context-free syntax supported by SGLR as the parsing algorithm. Undesired ambiguities in SDF2 definitions can be resolved using declarative *disambiguation filters* specified for associativity, priorities, follow restrictions, reject, avoid and prefer productions [van den Brand et al. 2002]. Implicit disambiguation mechanisms such as 'longest match' are avoided. Other approaches, including PEGs [Ford 2002], language inheritance in MontiCore [Krahn et al. 2008], and the composite grammars of ANTLR [Parr and Fisher 2011], 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, and explicitly addressed by a developer. This characteristic benefits the definition of non-trivial grammars, in particular the definition of grammars that are composed from two or more independently developed grammars.

SDF has been used to define various composite languages, often based on mainstream languages such as C/C++ [Waddington and Yao 2007], PHP [Bravenboer et al. 2007], and Java [Bravenboer and Visser 2004; Kats et al. 2008]. The example grammar shown in Figure 3 extends Java with embedded SQL queries. It imports both the Java and SQL grammars, adding two new productions that integrate the two. In SDF, grammar productions take the form $p_1 \ldots p_n$ `->` s and specify that

[3]Note that Schwerdfeger and Van Wyk [2009] have shown that for some LR grammars it is possible to statically determine whether they compose. They claim that if you accept some restrictions on the grammars, the composition of the "independently developed grammars" will not produce conflicts.

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

Fig. 3. Syntax of Java with embedded SQL queries, adapted from [Bravenboer et al. 2010]. The 'cons' annotation defines the name of the constructed ATerm.

a sequence of strings matching symbols p_1 to p_n matches the symbol s. The productions in this particular grammar specify a quotation syntax for SQL queries in Java expressions, and vice versa an anti-quotation syntax for Java expressions inside SQL query expressions. 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.

3. ISLAND GRAMMARS

Island grammars [van Deursen and Kuipers 1999; Moonen 2001; 2002] 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 [Moonen 2001]. In this paper we use island grammars as inspiration for error recovery using additional production rules.

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 [van Deursen and Kuipers 1999; Moonen 2001]. 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. A parser using an island grammar behaves similar to one that implements a noise-skipping algorithm [Lavie and Tomita 1993]. It can skip over any form of noise in the input file. However, using an island grammar, this logic is entirely encapsulated in the grammar definition itself.

Figure 4 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 sequence 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 define a language that matches *any* given character stream.

The parse tree produced for an island is constrained using disambiguation filters that are part of the original SDF specification [van den Brand et al. 2002]. First, the `{avoid}` annotation on the `WATER` rule specifies a disambiguation filter for these productions, indicating that the production is to be avoided, e.g., at all times, a non-water `Chunk` is to be preferred. Second, the lexical restrictions section specifies a restriction for the `WATER` symbol. This rule ensures that water is always greedily matched, and never followed by any other water character.

The following example illustrates how programs are parsed using an island grammar:

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

Given this COBOL fragment, a generalized parser can construct a parse tree — or rather a parse *forest* — that includes all valid interpretations of this text. Internally, the parse tree includes the

```
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-zA-Z][a-zA-Z0-9]* -> Id
  lexical restrictions
    WATER -/- [A-Za-z0-9]
```

Fig. 4. An island grammar for extracting calls from a legacy application; adapted from [Moonen 2001].

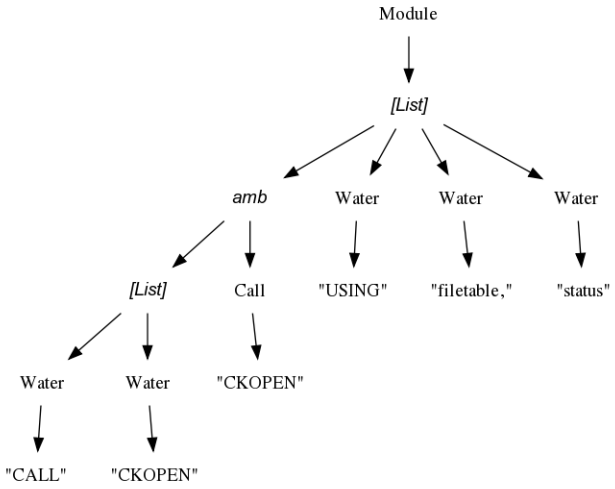


Fig. 5. The unfiltered abstract syntax tree for a COBOL statement, constructed using the ExtractCalls grammar.

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 AST 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. Normally, these filters are applied automatically during or after parsing.

4. 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. While the technique of island grammars is targeted only towards partial grammar definitions, this observation suggests that the basic principle behind island grammars may be adapted for use in recovery for complete, well-defined grammars.

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. We are developing this material in an example-driven way in the sections 4.1 to 4.3. Then, in Section 4.4, we explain how different forms of recovery

```
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 {cons("IfElse"), avoid}
  ...
```

Fig. 6. Part of the standard Java grammar in SDF; adapted from [Bravenboer et al. 2006].

can be combined. Finally, in Section 4.5 we discuss automatic derivation of recovery rules from the grammar, while Section 4.6 explains how the set of generated recovery rules can be customized by the language developer.

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 Java language. SDF allows the definition of concrete and abstract syntax in a single framework. The mapping between concrete syntax trees (parse trees) and abstract syntax trees is given by the `{cons(`*name*`)}` annotations. Thus, in the given example, the `{cons("If")}` and `{cons("IfElse")}` annotations specify the name of the constructed abstract syntax terms. Furthermore, the abstract syntax tree does not contain redundant information such as layout between tokens and literals in a production. The `{avoid}` annotation in the second context-free production is used to explicitly avoid the "dangling else problem", a notorious ambiguity that occurs with nested if/then/else statements. Thus, the `{avoid}` annotation states that the interpretation of an `IfElse` term with a nested `If` subterm, must be avoided in favour of the alternate interpretation, i.e. an `If` term with a nested `IfElse` subterm. Indeed, Java can be parsed without the use of SGLR, but SGLR has been invaluable for extensions and embeddings based on Java such as those described in [Bravenboer and Visser 2004; Bravenboer et al. 2006].

4.1. Chunk-Based Water Recovery Rules

Island grammars rely on constructing a grammar based on coarse-grained chunks that can be parsed normally or parsed as water and skipped. This structure is lacking in normal, complete grammars, which tend to have a more hierarchical structure. For example, Java programs consist of one 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, each `Stm` symbol is considered a "chunk," which can be parsed as either a regular statement or as water, effectively skipping over any noise that may exist within method bodies. To ensure that water is always greedily matched, a follow restriction is specified (`-/-`), expressing that the WATER symbol is never followed by another water character.

From Avoid to Recover Productions. As part of the original SDF specification, the `{avoid}` annotation is used to disambiguate parse trees produced by grammar productions. An example is the "dangling else" disambiguation shown in Figure 6. In Figure 7, we use the `{avoid}` annotation on the water production to indicate that preference should be given to parsing statements with regular productions. The key insight of permissive grammars is that this mechanism is sufficient, in principle, to model error recovery.

However, in practice, there are two problems with the use of `{avoid}` for declaring error recovery. First, `{avoid}` is also used in regular disambiguation of grammars. We want to avoid error recovery productions *more* than 'normal' `{avoid}` productions. Second, `{avoid}` is implemented as a post-parse filter on the parse forest produced by the parser. This is fine when ambiguities are relatively local and few in number. However, noise-skipping water rules such as those in Figure 7

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

Fig. 7. Chunk-based recovery rules for Java using `avoid`.

```
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")}
```

Fig. 8. Chunk-based recovery rules using `recover`.

cause massive numbers of ambiguities; each statement can be interpreted as water or as a regular statement, i.e. the parse forest should represent an exponential number of parse trees. While (S)GLR is equipped to deal with ambiguities, their construction has a performance penalty, which is wasteful when there are no errors to recover from.

Thus, we introduced the `{recover}` annotation in SDF to distinguish between the two different concerns of recovery and disambiguation (Figure 8). The annotation is similar to `{avoid}`, in that we are interested in parse trees with as few uses of `{recover}` productions as possible. Only in case all remaining branches contain recover productions, a preferred interpretation is selected heuristically by counting all occurrences of the `{recover}` annotation in the ambiguous branches, and selecting the variant with the lowest count. Parse trees produced by the original grammar productions are always preferred over parse trees containing recover productions. Furthermore, `{recover}` branches are disambiguated at runtime, and, to avoid overhead for error-free programs, are only explored when parse errors occur using the regular productions. The runtime support for parsing and disambiguation of recover branches is explained in Section 5.

Throughout this section we use only the standard, unaltered SDF specification language, adding only the `{recover}` annotation.

Limitations of Chunk-Based Rules. We can extend the grammar of Figure 8 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 based on the number of occurrences of the `{recover}` annotation in the ambiguous branches.

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 with syntax errors that cannot be parsed by the original grammar. The `WATER` constructors in the ASTs indicate the location of errors, which can then be straightforwardly reported back to the user.

While the approach we presented so far can already provide basic syntax error recovery, 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.

4.2. 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, 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 part of the AST. As discussed in Section 3, any form of layout, which may include comments, is implicitly interleaved in the patterns of concrete syntax productions. The parser 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.

The key idea discussed in this section is to declare water tokens that may occur anywhere that layout may occur. Using this idea, permissive grammars can be defined with noise skipping recovery rules that are language-independent *and* more fine grained than the chunk-based recovery rules above. To understand how this can be realized, we need to understand the way that SDF realizes 'character-level grammars'.

Intermezzo: Layout in SDF. In SDF, productions are defined in *lexical syntax* or in *context-free syntax*. Lexical productions are normal context-free grammar productions, i.e. not restricted to regular grammars. The *only* distinction between lexical syntax and context-free syntax is the role of layout. The characters of an identifier (lexical syntax) should not be separated by layout, while layout *may* occur between the sub-phrases of an if-then-else statement, defined in context-free syntax.

The implementation of SDF with scannerless parsing entails that individual characters are the lexical tokens considered by the parser. Therefore, lexical productions and context-free productions are merged into a single context-free grammar with characters as terminals. The result is a character-level grammar that explicitly defines all the places where layout may occur. For example, the `If` production is defined in Kernel-SDF [Visser 1997c], the underlying core language of SDF, as follows[4]:

```
syntax
  "if" LAYOUT? "(" LAYOUT? Expr LAYOUT? ")" LAYOUT? Stm -> Stm {cons("If")}
```

Thus, optional layout is interleaved with the regular elements of the construct. It is not included in the construction of abstract syntax trees from parse trees. Since writing productions in this explicit form is tedious, SDF produces them through a grammar transformation, so that, instead of the explicit rule above, one can write the `If` production as in Figure 6:

```
context-free syntax
  "if" "(" Expr ")" Stm -> Stm {cons("If")}
```

Water as Layout. We can use the notion of interleaving context-free productions with optional layout in order to define a new variation of the water recovery rules we have shown so far. Consider Figure 9, which combines elements of the comment definition of Figure 6 and the chunk-based recovery rules from Figure 8. It introduces optional water into the 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. The production for the `WATERWORD` token allows to skip over identifier strings, while the production for the `WATERSEP` token allows to skip over special characters that are neither part of identifiers nor whitespace characters. The latter production is defined as an inverse pattern, using the negation operator (`~`). This distinction ensures that large strings, consisting of multiple words and special characters, 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:

[4]We have slightly simplified the notation that is used for non-terminals in Kernel-SDF.

```
module Java-15-Permissive-Water
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\_]
```

Fig. 9. Water recovery rules.

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

Still missing the closing bracket, 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 number of `{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 AST. For reporting errors, the original parse tree, which *does* contain "water" nodes, can be inspected instead.

The adapted grammar of Figure 9 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.

4.3. 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 bracket. 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 bracket. 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 bracket optional:

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

This strategy, however, is rather specific for a single production, and would significantly increase the size of the grammar if we applied it to all productions. A better approach would be to *insert* the particular literal into the parse stream.

Literal Insertion. SDF allows us to simulate literal insertion using separate productions that virtually insert literal symbols. For example, the lexical syntax section in Figure 10 defines a number of basic *literal-insertion recovery rules*, each inserting a closing bracket or other literal that ends a production pattern. This approach builds on the fact that literals such as `")"` are in fact non-terminals that are defined with a production in Kernel-SDF:

```
  syntax
    [\41] -> ")"
```

```
module Java-15-Permissive-LiteralInsertions
imports Java-15
exports
lexical syntax
   -> ")" {cons("INSERT"), recover}
   -> "]" {cons("INSERT"), recover}
   -> "}" {cons("INSERT"), recover}
   -> ">" {cons("INSERT"), recover}
   -> ";" {cons("INSERT"), recover}
```

Fig. 10. Insertion recovery rules for literal symbols.

Thus, the character 41, which corresponds to a closing brace in ASCII, reduces to the nonterminal ")". A literal-insertion rule extends the definition of a literal non-terminal, effectively making it optional by indicating that they may match the empty string. 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 the `INSERT` constructor is defined in lexical syntax, it is not used in the resulting AST.

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 = ${user}
|>;
```

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:

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

On Literals, Identifiers, and Reserved Words. Literal-insertion rules can also be used for literals that are not *reserved words*. This is an important property when considering composed languages since, in many cases, some literals in one sublanguage may not be reserved words in another. As an example, we discuss the insertion rule for the `end` literal in the combined Stratego-Java language.

In Stratego, the literal `end` is used as the closing token of the `if ... then ... else ... end` construct. To recover from incomplete `if-then-else` constructs, a good insertion rule is:

```
lexical syntax
   -> "end" {cons("INSERT"), recover}
```

In Java, the string `end` is not a reserved word and is a perfectly legal *identifier*. In Java, identifiers are defined as follows:[5]

```
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 with the name "end". 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:

```
syntax
  [\101] [\110] [\100] -> "end"
```

[5] In fact this production is a simplified version of the actual production. Java allows many other (Unicode) letters and numbers to appear in identifiers.

```
module Java-15-Permissive-LexicalInsertions
imports Java-15
exports
lexical syntax
  INSERTSTARTQ StringPart* "\n"    -> StringLiteral {cons("INSERTEND")}
  "\""                             -> INSERTSTARTQ  {recover}
  INSERTSTARTC CommentPart* EOF    -> Comment       {cons("INSERTEND")}
  "/*"                             -> INSERTSTARTC {recover}
```

Fig. 11. Insertion recovery rules for lexical symbols.

The reason that SDF allows this production to be defined in this fashion is that in the SGLR algorithm, the parser only operates on characters, and the `end` literal has no special meaning other than a grouping of character matches.

The literal-insertion recovery rule simply adds an additional derivation for the `"end"` symbol, providing the parser with an additional way to parse it, namely by matching the empty string. As such, the rule does not change how identifiers (`ID`) are parsed, namely by matching the pattern at the left hand side of the production rule for the `ID` symbol. With a naive recovery strategy that inserts tokens into the stream, identifiers (e.g., `end` in Java) could be inserted in place of keywords. With our approach, these effects are avoided since the insertion recovery rules only apply when a literal is expected.

Insertion Rules for String and Comment Closings. Figure 11 specifies recover 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:

```
lexical syntax
   "\"" StringPart* "\n" -> StringLiteral {cons("INSERTEND"), recover}
```

However, the given formulation is beneficial for the runtime behavior of our adapted parser implementation, ensuring that the annotation is considered before construction of the starting 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. 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 end of file, the number of different possible interpretations is severely reduced, thus reducing the overall runtime complexity of the recovery.

Insertion Rules for Lexical Symbols. Insertion rules can also be used to insert lexical symbols such as identifiers. However, lexical symbols do have a representation in the AST, therefore, their insertion requires the introduction of placeholder nodes that represent a missing code construct, for example a `NULL()` node. Since placeholder nodes alter the abstract syntax of the grammar, their introduction adds to the complexity of tools that process the AST. However, for certain use cases such as content completion in an IDE, lexical insertion can be useful. We revisit the topic in Section 8.

4.4. Combining Different Recovery Rules

The water recovery rules of Section 4.2 and the insertion rules of Section 4.3 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-Water
  Java-15-Permissive-LiteralInsertions
  Java-15-Permissive-LexicalInsertions
```

Together, the two strategies maintain a fine balance between discarding and inserting substrings. Since the water recovery rules incur additional cost for each water substring, insertion of literals will

generally be preferred over discarding multiple substrings. This ensures that most of the original (or intended) user input is preserved.

4.5. Automatic Derivation of Permissive Grammars

Automatically deriving recovery rules helps to 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, all recovery rules that are no longer applicable are automatically removed from the grammar and new recovery rules are added. Thus, automatic derivation helps to maintain language independence by providing a generic, automated approach towards the introduction of recovery rules.

SDF specifications are fully declarative, which allows automated analysis and transformation of a grammar specification. We formulate a set of heuristic rules for the generation of recovery rules based on different production patterns. These rules are applied in a top-down traversal to transform the original grammar into a permissive grammar. The heuristics in this section focus on insertion recovery rules, since these are language specific. The water recovery rules are general applicable and added to the transformed grammar without further analysis. The heuristics discussed in this section are based on our experience with different 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. However, for many literals, automatic insertion can lead to unintuitive results in the feedback presented to the user. For example, in the Java language "synchronized" is an optional modifier at the beginning of a class declaration. We don't want the editor to suggest to insert a "synchronized" keyword. In those cases, discarding some substrings instead may be a safer alternative. The decision whether to consider particular keywords for insertion may depend on their semantic meaning and importance [Degano and Priami 1995]. To take this into account, expert feedback on a grammar is needed.

Since we have aimed at maintaining language independence of the approach, our main focus is on more generic, structure-based properties of the grammar. We have identified four different 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. We base our generic, language independent recovery technique on these four categories.

By grammar analysis, we derive recovery rules for insertions of the categories mentioned above. With respect to the first and second category, we only derive rules for opening and closing terminals that appear in a balanced fashion with another literal (or a number of other literals). Insertions of literals that are not balanced with another literal can lead to undesired results, since such constructs do not form a clear nesting structure. Furthermore, we exclude lexical productions that define strings and comments, for which we only derive more restrictive insertion rules given by the third and fourth category.

Insertion rules for the first category, *closing bracket* and *terminating 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, e.g., the closing literal appears in a balanced fashion. Finally, the last literal is not used as the starting literal of any other production. 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

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

Fig. 12. A selection of context-free productions that appear in the Java grammar.

production pattern in all rules of the grammar. For the third category, we only consider productions with identical starting and end literals where no newlines are allowed in between. Finally, for the fourth category we derive rules for matching starting and ending literals in `LAYOUT` productions. Note that we found that some grammars (notably the Java grammar of [Bravenboer et al. 2006]) 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 12. Looking at the first production, and using the heuristic rules above, we can recognize that `}` qualifies as a closing literal. Likewise, `)` satisfies the conditions for closing literals we have set. By programmatically analyzing the grammar in this fashion, we collected the closing literal insertion rules of Figure 10 which are a subset of the complete set of closing literal insertion rules for Java. From the productions of Figure 12 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[6], which follows from the "void" class literal production of Figure 12, and from the fact that the `class` keyword is never used as a starting literal of any production. In practice, we have found that these anomalies are relatively rare and in most cases harmless.

We evaluated our set of heuristic rules using the Java, Java-SQL, Stratego, Stratego-Java and WebDSL grammars, as outlined in Section 10. For these grammars, a total number of respectively 19 (Java), 43 (Java-SQL), 37 (Stratego), 47 (Stratego-Java) and 32 (WebDSL) insertion rules were generated, along with a constant number of water recovery rules as outlined in Figure 9. The complete set of derived rules is available from [Kats et al. 2011].

4.6. Customization of Permissive Grammars

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. Therefore a good error recovery mechanism is not only *language independent*, but is also *flexible* [Degano and Priami 1995]. 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. 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 two 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:

[6]Note that for narrative reasons, we did not include an insertion rule for this keyword in Figure 10.

```
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 );
                 ;
```

Fig. 13. Interpretations of `i=f(x)+1;` with insertion recovery rules (underlined) and water recovery rules.

```
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.

Besides composition, SDF also provides a mechanism for subtraction of languages. The `{reject}` disambiguation annotation filters all derivations for a particular set of symbols [van den Brand et al. 2002]. 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 disables the `class` insertion rule.

```
lexical syntax
   -> "class" {reject}
```

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. SDF allows modular composition of grammar definitions.

5. PARSING PERMISSIVE GRAMMARS WITH BACKTRACKING

When all recovery rules are taken into account, permissive grammars provide many different interpretations of the same code fragment. As an example, Figure 13 shows many possible interpretations of the string `i=f(x)+1;`. The alternative interpretations are obtained by applying recovery 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. The option of inserting opening brackets results in even more possible interpretations, since bracket pairs can be added around each expression that occurs in the program text.

Conceptually, the use of grammar productions to specify how to recover from errors provides an attractive mechanism to parse erroneous fragments. All possible interpretations of the fragment are explored in parallel, using a generalized parser. Any alternative that does not lead to a valid interpretation is simply discarded, while the remaining branches are filtered by disambiguation rules applied by a post processor on the created parse forest. However, from a practical point of view, the extra interpretations created by recovery productions negatively affect time and space requirements. With a generalized parser, all interpretations are explored in parallel, which significantly increases the workload for the parser, even if there are no errors to recover from.

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

Fig. 14. The superfluous closing bracket is detected at the `while` keyword.

In this section we address the performance problems introduced by the multiple recover interpretations. We extend the SGLR algorithm with a selective form of backtracking that is only applied when actually encountering a parsing error. The performance problems during normal parsing are simply avoided by ignoring the recover productions.

5.1. Backtracking

As it is not practical to consider all recovery interpretations in parallel with the normal grammar productions, we need a different strategy to efficiently parse with permissive grammars. As an alternative to parsing different interpretations in parallel, *backtracking parsers* revisit points of the file that allow multiple interpretations (the choice points). Backtrack parsing is not a correct implementation of generalized parsing, since a backtracking parser only produces a single possible parse. However, when applied to error recovery, this is not problematic. For typical cases, parsing only a single interpretation at a time suffices; ultimately, only one recovery solution is needed.

To minimize the overhead of recovery rules, we introduce a selective form of backtracking to (S)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. Backtracking parsers exhibit exponential behavior in the worst case [Johnstone et al. 2004]. For pathological cases with repetitive backtracking, the parser is aborted, and a secondary, non-correcting, recovery technique is applied.

5.2. Selecting Choice Points for Backtracking

A parser that supports error recovery typically operates by consuming tokens (or characters) until an erroneous token is detected. At the point of detection of an error, the recovery mechanism is activated. A major problem for error recovery techniques is the difference between the actual location of the error and the point of detection [Degano and Priami 1995]. Consider for example the erroneous code fragment in Figure 14. The superfluous closing bracket (underlined) after the `foo();` statement is obviously intended as a closing bracket for the `if` construct. However, since the `if` construct misses an opening bracket, the closing bracket is misinterpreted as closing the method instead of the `if` construct. At that point, the parser simply continues, interpreting the remaining statements as class-body declarations. Consequently, the parser fails at the reserved `while` keyword, which can only occur inside a method body. More precisely, with a scannerless parser, it fails at the unexpected space after the characters `w-h-i-l-e`; the character cannot be shifted and all branches (interpretations at that point) are discarded.

In order to properly recover from a parse failure, the text that precedes the point of failure must be reinterpreted using a correcting recovery technique. Using backtracking, this text is inspected 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 likely near the failure location.

As generalized LR parsers process different interpretations in parallel, they use a more complicated stack structure than regular LR parsers. Instead of a single, linear stack, they use a graph-structured stack (GSS) that efficiently stores the different interpretation branches, which are discarded as input tokens or characters are shifted [Tomita 1988]. All discarded branches must be restored in case the old state is revisited, which poses a challenge for applying backtracking.

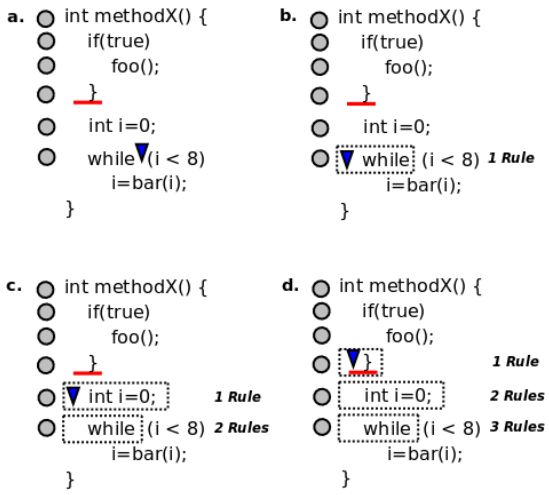


Fig. 15. Applying error recovery rules with backtracking. The initial point of failure and the start of the recovery search space is indicated by a triangle. The entire search space is indicated using dashed lines, where the numbers to the side indicate the number of recovery rules that can be applied at that line.

To make it possible to resume parsing from a previous location, the complete stack structure for that location is stored in a choice point. We found that it is prohibitive (in terms of performance) to maintain the complete stack state for each shifted character. To minimize the overhead introduced, we only selectively record the stack structure. Lines have meaning in the structure of programs as units of editing. Typically, parse errors are clustered in the line being edited. We base our heuristic for storing choice points on this intuition. In the current implementation, we create one backtracking choice point for each line of the input file.

5.3. Applying Recovery Rules

A parse failure indicates that one or more syntax errors reside in the prefix of the program before the failure location. Since it is unlikely that the parser can consume many more tokens after a syntax error, these errors are typically located near the failure location. To recover from multiple errors, multiple corrections are sometimes required. To recover from syntax errors efficiently, we implement a heuristic that expands the search space with respect to the area that is covered and with respect to the number of corrections (recover rule applications) that are made.

Figure 15 illustrates how the search heuristic is applied to recover the Java fragment of Figure 14. The algorithm iteratively explores the input stream in reverse order, starting at the nearest choice point. 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 each following iteration the size of the area and the number of recovery rule applications are increased.

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 bracket after the `if` statement, is 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 by a second recovery production are cut off. Their exploration is postponed until the next iteration. 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 applied recovery rules. The new search space consists of the line that precedes the point of detection, plus the error detection line where the recovery candidates with two changes are considered, resuming the interpretations that were previously cut off.

In Figure 15d, the search space is again expanded with the preceding line. This time, a valid recovery is found: the application of a water recovery rule that discards the closing bracket leads to a valid interpretation of the erroneous code fragment. Once the original line where the error was detected can be successfully parsed, normal parsing continues.

5.4. Algorithm

The implementation of the recovery algorithm requires a number of (relatively minor) modifications of the SGLR algorithm used for normal parsing. First, productions marked with the `{recover}` attribute are ignored during normal parsing. Second, a choice point is stored at each newline character. And finally, if all branches are discarded and no accepting state is reached, the `Recover` function is called. Once the recovery is successful, normal parsing resumes with the newly constructed stack structure.

Figure 16 shows the recovery algorithm in pseudo code. The `Recover` function controls the iterative search process described in Section 5.3. The function starts with some initial configuration (line 2–3), initializing the `candidates` variable, and selecting the last inserted choice point. The choice points are then visited in reverse order (line 4–7), until a valid interpretation (non-empty stack structure) is found (line 7).

For each choice point that is visited, the `ParseCandidates` function is called. The `ParseCandidates` function has a twofold purpose (line 16, 17): first, it tries to construct a valid interpretation (line 16) by exploring candidate recover branches; second, it collects new candidate recover branches (line 17) the exploration of which is postponed until the next iteration. Candidate recover branches are cut off recover interpretations of a prefix of the program. The `ParseCandidates` function reparses the fragment that starts at the choice point location and ends at the accept location (line 19–26). We heuristically set the `ACCEPT_INTERVAL` on two more lines and at least twenty more characters being parsed after the failure location. For each character of this fragment, previously cut off candidates are merged into the stack structure (line 23) so that they are included in the parsing (line 24); while new candidates are collected by applying recover productions on the stack structure (line 24–25, line 31).

The main idea, implemented in line 23-25 and the `ParseCharacter` function (line 28–32), is to postpone the exploration of branches that require multiple recover productions, thereby implementing the expanding search space heuristic described in Section 5.3.

After the algorithm completes and finds a non-empty set of stacks for the parser, it enters an optional disambiguation stage. In case more than one valid recovery is found, stacks with the lowest recovery costs are preferred. These costs are calculated as the sum of the cost of all recovery rules applied to construct the stack. We employ a heuristic that weighs the application of a water recovery rule as twice the cost of the application of an insertion recovery rule, which accounts for the intuition that it is more common that a program fragment is incomplete during editing than that a text fragment was not intended and therefore should be deleted. Ambiguities obtained by application of a recovery rule annotated with `{reject}` form a special case. The reject ambiguity filter removes the stack created by the corresponding rule from the GSS, thereby effectively disabling the rule.

6. LAYOUT-SENSITIVE RECOVERY OF SCOPING STRUCTURES

In this section, we describe a recovery technique specific for errors in scoping structures. Scoping structures are usually recursive structures specified in a nested fashion [Charles 1991]. Omitting brackets of scopes, or other character sequences marking scopes, is a common error made by programmers. These errors can be addressed by common parse error recovery techniques that insert missing brackets.

```
RECOVER(choicePoints, failureOffset)
1  ▷ Constructs a recovery stack structure (GSS) for the parse input after the failure location
2  candidates ← {}
3  choicePoint ← Last inserted choicepoint
4  do
5     (stacks, candidates) ← PARSECANDIDATES(candidates, choicePoint, failureOffset)
6     choicePoint ← Preceding choicepoint (or choicePoint if none)
7  until | stacks | > 0
8  return stacks

PARSECANDIDATES(candidates, choicePoint, failureOffset)
9  ▷ Parses in parallel previously collected candidate recover branches,
10    while cutting off and collecting new recover candidates
11 ▷ Input:
12    candidates - Unexplored recover branches that were created in previous loop
13    choicePoint - The start configuration for the parser
14    failureOffset - Location were the parser originally failed
15 ▷ Output:
16    stacks - recovered stacks at the accept location
17    newCandidates - new unexplored recover branches for the parsed fragment
18
19 stacks ← choicePoint.stacks
20 offset ← choicePoint.offset
21 newCandidates ← {}
22 do
23    stacks ← stacks ∪ { c | c ∈ candidates ∧ c.offset = offset}
24    (stacks, recoverStacks) ← PARSECHARACTER(stacks, offset, true)
25    newCandidates ← newCandidatess ∪ recoverStacks
26    offset = offset +1
27 until offset = (failureOffset + ACCEPT_INTERVAL)
28 return (stacks, newCandidates)

PARSECHARACTER(stacks, offset, inRecoverMode)
29 ▷ Parses the input character at the given offset.
30 ▷ Output:
31    parseStacks - stacks created by applying the normal grammar productions
32    recoverStacks - stacks created by applying recover productions (in recover mode)
33 return (parseStacks, recoverStacks)
```

Fig. 16. A backtracking algorithm to apply recovery rules.

However, as scopes can be nested, there are often many possible positions where a missing bracket can be inserted. The challenge is to select the most appropriate position. As an example, consider the Java fragment in Figure 17. This fragment could be recovered by inserting a closing bracket at the end of the line with the second opening bracket, or at any line after this line. However, the use of indentation suggests the best choice may be just before the `int x;` declaration.

```
class C {
  void m() {
    int y;
  int x;
}
```

Fig. 17. Missing `}`.

One approach to handle this problem is to take secondary notation like indentation into account during error recovery. Bridge parsing, introduced by Nilsson-Nyman

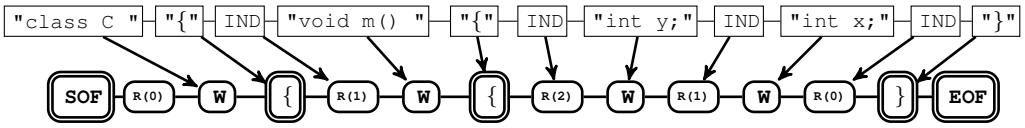


Fig. 18. A tokenization of the example program in Figure 17 where text is mapped to islands (double edges), water (**W**), and reefs (**R(**n**)**). The number n in a reef R(n) represents the indentation level of the reef.

et al. [2009][7], uses this particular approach. This scope recovery approach can be combined with the permissive grammar approach presented in the previous section.

6.1. Bridge Parsing

Bridge parsing provides a technique specifically targeted at improved recovery of scope errors using secondary notation such as indentation. The technique as such is independent of any specific parsing formalism. It may be used as a standalone processor of erroneous files where recovery otherwise fails: given an erroneous file, or section of a file, the bridge parser analyses the content and provides suggestions on where to insert missing brackets. Based on a set of rules that describe the typical relation between scopes and layout for Java, a bridge parser can correctly recover cases such as the example above.

Internally, a bridge parser contains three parts: a *tokenizer*, a *model builder*, and a *repairer*. The tokenizer provides a list of interesting tokens from an input text. Tokens starting and ending scopes are referred to as *islands*; tokens interesting for construction of scopes, or recovery of scopes, are referred to as *reefs*; and remaining tokens are considered to be *water*. The terms island and water are used in the same fashion as in island grammars [van Deursen and Kuipers 1999; Moonen 2001; 2002]. Reefs, added for bridge parsing, are tokenwise like islands, but have a different role in the model constructed from the token list. Figure 18 shows an example of a token list for the program fragment in Figure 17. Each part of the fragment is mapped to either an island, reef, or water. For the benefit of the model builder algorithm, the token list is padded with some additional tokens at the start and end.

After tokenization, the model builder constructs scopes based on information in the token list. For instance, each reef in the token list in Figure 18 has a number indicating indentation level.[8] This indentation information is key to construction of scopes, represented as bridges, connecting two islands, in the model. The model builder decides which two islands to connect using an algorithm that considers patterns of tokens surrounding islands, and rules for when patterns match. For instance, in Figure 19 bridges have been added to the token list in Figure 18. The added bridges connect the start and end of the fragment, and two of the islands, while one island remains unmatched. In this example, islands are matched based on the indentation of the first reef to their left. For the two matched islands their corresponding reef shares the same indentation, while their is no such match for the island without a bridge. Islands like this one, without a bridge, are considered broken and representatives of broken scopes.

After construction of bridges, the repairer takes over. The purpose of the repairer is to recover broken scopes based on as set of patterns and rules. The purpose of the patterns is to identify appropriate so called *construction sites* for a recovery. Once such a construction site has been found, the rules are used to decide how to insert a matching so called *artificial island* and create a bridge. For instance, in Figure 19 a construction site is found based on a pattern identifying indentation shifts, and an artificial islands is inserted to match the broken island and recover the scope error. Insertion of islands, like in this example, correspond to the recovery suggestions a bridge parser provides after it is done.

[7]Emma Nilsson-Nyman, the first author of the cited bridge parser paper, has changed her name to Emma Söderberg and is one of the authors of this paper.

[8]Note that in our implementation, we determine the indentation level by counting the number of spaces, treating tabs as a fixed number of spaces. The relation between tabs and spaces could also be determined from the editor settings.

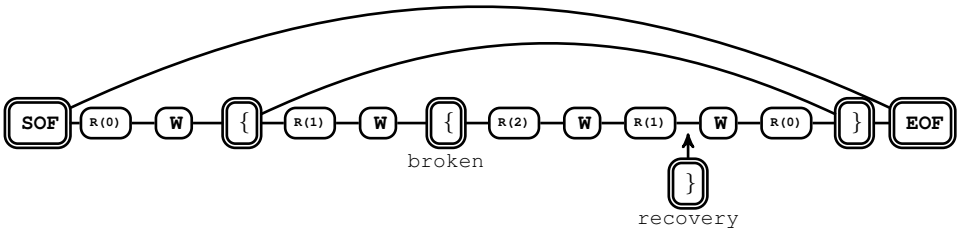


Fig. 19. A bridge parser model with bridges (arches) between matching islands (double edged nodes). Islands missing a bridge correspond to broken scopes (`broken`). The bridge repairer will try to recover such scopes by insertion of new matching islands (`recovery`).

A more complete description of the algorithm, incrementally constructing multiple bridges, is given by Nilsson-Nyman et al. [2009].

6.2. Combining Permissive Grammars and Bridge Parsing

As a recovery technique, bridge parsing forms a supplementary approach that can be used together with permissive grammars introduced in Section 4. Permissive grammars and bridge parsing share their inspiration from island grammars [van Deursen and Kuipers 1999; Moonen 2001; 2002], with the difference that a bridge parser employs a scanner.

The use of a scanner in bridge parsing may appear contrary to the scannerless nature of SGLR. One could imagine that a scannerless version of a bridge parser would be better suited for an integration to SGLR. That is, based on an accurate (scannerless) lexical analysis, additional reefs could be identified using the keywords of a language. However, previous results showed that doing so only marginally improves recovery quality [de Jonge et al. 2009]. Also, practical experience has shown that a bridge parser is most time and memory-efficient when independent from a specific grammar, focusing just on the scoping structures of the language. For this reason and for simplicity, the bridge parsers used in this paper only include scope tokens and layout reefs.

This combined approach has limitations with regard to embedded languages, where a token may have different syntactic meanings: `{` might be a scope delimiter in one language and an operator in another. Still, the layout-sensitive bridge model gives an approximation of the scoping structure in those cases, which can improve recovery results when used in combination with recovery rules. As a layout-sensitive technique, bridge parsing served as an inspiration to the layout-sensitive regions discussed in the next section.

7. LAYOUT-SENSITIVE REGION SELECTION

In this section we describe a layout-sensitive region recovery algorithm that improves recovery efficiency and helps cope with pathological cases not easily addressed with only permissive grammars, backtracking, and bridge parsing. Relying on the increasing search space of permissive grammars and backtracking, it is not always feasible to provide good recovery suggestions in an acceptable time span. Problems can arise when the distance between the error location and the detection location is exceptionally large, or when the recovery requires many combined recovery rule applications. The latter can occur when multiple errors are tightly clustered, or when no suitable recovery rule is at hand for a particular error. In general, a valid parse can be found after expanding the search space, but at a risk of a high performance cost, and potentially resulting in a complex network of recovery suggestions that do not lead to useful feedback for programmers. Section 4.3 discusses an example in which an entire SQL fragment would be parsed as (severely incorrect) Java code.

To address these concerns, this section introduces an approach to identify the *region* in which the actual error is situated. By constraining the recovery suggestions to a particular part of the file, *region selection* improves the efficiency as well as the quality of the recovery, avoiding suggestions that are spread out all over the file.

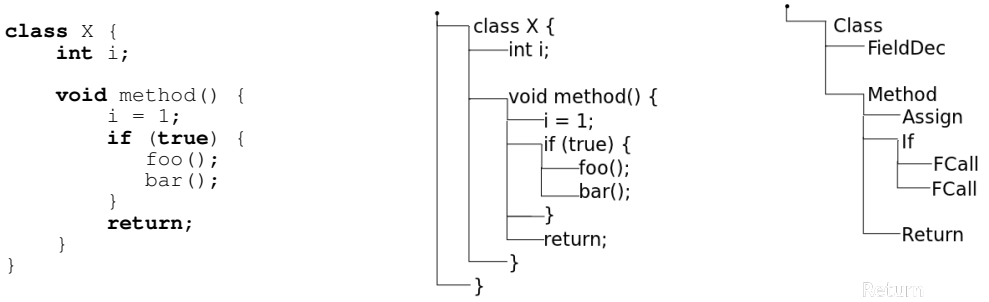


Fig. 20. Indentation closely resembles the hierarchical structure of a program.

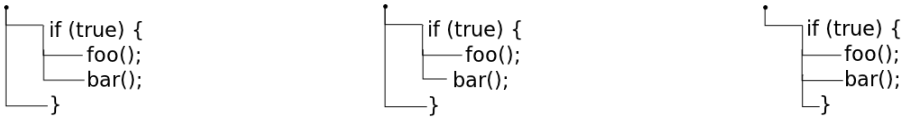


Fig. 21. Parent child relations between lines with consistent layout (left) and inconsistent layout (mid, right). `if(true){` is the parent line of the siblings `foo();` and `bar();` (left, mid, right), and the inconsistently indented `}` (right).

In some cases it is better to ignore a small part of the input file, rather than to try and fix it using a combination of insertions and discarded substrings. As a second application of the regional approach, *region skipping* is used as a fallback recovery strategy that discards the erroneous region entirely in case a detailed analysis of the region does not lead to a satisfactory recovery.

7.1. Nested Structures as Regions

Language constructs such as statements and methods are elements of list structures. List elements form free standing blocks, in the sense that they can be omitted without influencing the syntactic interpretation of other blocks. It follows that erroneous free standing blocks can simply be skipped, providing a coarse recovery that allows the parser to continue. We conclude that list elements are suitable regions for regional error recovery.

The bridge parsing technique discussed in Section 6 exploits layout characteristics to detect the intended nesting structure of a program. In this section, we present a region selection technique that, inspired by bridge parsing, uses indentation to detect erroneous structures. Indentation typically follows the logical nesting structure of a program, as illustrated in Figure 20. The relation between constructs can be deduced from the layout. An indentation shift to the right indicates a *parent-child* relation, whereas the same indentation indicates a *sibling* relation. The region selection technique inspects the parent and sibling structures near the parse failure location to detect the erroneous region.

Indentation usage is not enforced by the language definition. Proper use of layout is a convention, being part of good coding practice. We generally assume that most programmers apply layout conventions, which is reinforced by the application of automatic formatters. Furthermore we assume that indentation follows the logical nesting structure. However, we should keep in mind the possibility of inconsistent indentation usage which decreases the quality of the results. The second assumption we make is that programs contain free standing blocks, i.e. that skipping a region still yields a valid program. Most programming languages seem to meet this assumption. If both assumptions are met, layout-sensitive region selection can improve the quality and performance of a correcting technique, and offer a fallback recovery technique in case the correcting technique fails.

7.2. Regions based on Indentation

We view the source text as a tree-structured collection of lines, whereby the parent-child relation between lines are determined by indentation shifts. Thus, given a line l, line p is the *parent* of l if and

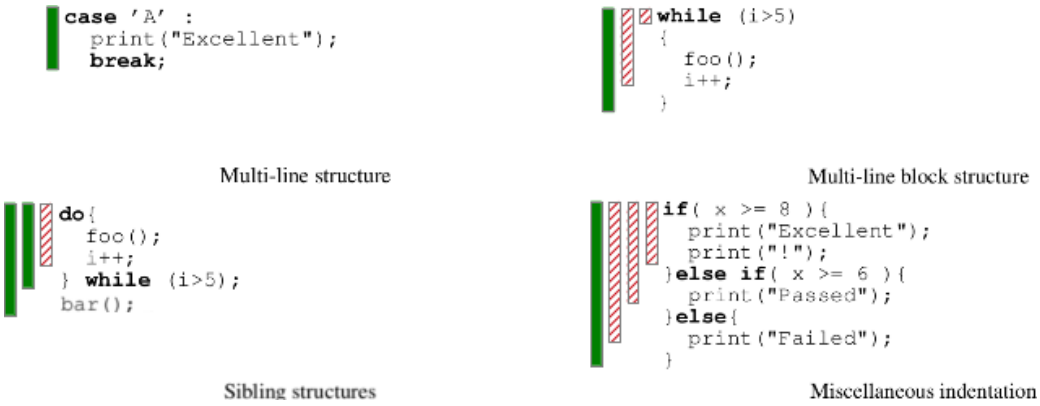


Fig. 22. Multi-line Java constructs with various indentation patterns. The solid bars indicate layout regions that correspond to code regions, the hatched bars indicate layout regions that are in fact unwished artifacts.

only if l is strictly more indented than p, and line l succeeds line p, and no lines exist between l and p that have less indentation than l. Lines with the same parent are *siblings* of each other. Figure 21 illustrates the parent-child relation for some small code fragments. The line `if(true){` in the left fragment is the parent of the sibling lines `foo();` and `bar();`. The mid and right fragment illustrate how the parent-child relation applies in case of inconsistent indentation; by definition, child nodes are more indented than their parent, however, the siblings in these fragments do not all have the same indent value.

A parent-child relation between two lines is a strong indication that the code constructs associated to these lines are also in parent-child relation. Similarly, a sibling relation between two lines indicates that either their associated code constructs are siblings as well, or that both lines belong to the same multi-line construct. Figure 22 provides some examples of multi-line constructs with various indentation patterns. For all constructs in the figure it holds that a parent-child relation between two lines reflects a parent-child relation between the code constructs associated to these lines. The shown constructs are different with respect to the number of siblings (of the first line) that are part of the construct. Another type of multi-line constructs are constructs that wrap over to the subsequent, more indented line. In that case, a parent child relation exists between two lines that actually belong to the same construct. This is an example of a small inconsistency that is not harmful to the overall approach.

We decompose a code fragment into candidate regions, based on the assumption that parent-child relations between lines reflect parent-child relations between the associated constructs, e.g., if a line is contained in a region then its child lines are also contained in that region. Unfortunately, indentation alone does not provide sufficient information to demarcate regions exactly. The main limitation is the ambiguous interpretation of sibling lines, which, by assumption, either belong to the same code construct or to separate constructs that are siblings. Given a single line, we construct multiple indentation-based regions: the smallest region consist of the line plus its child lines, the alternate regions are obtained by subsequently including sibling lines, including their children. The bars in Figure 22 show the different regions that are constructed for the first line of the given fragments. Only the regions corresponding to the solid bars represent actual code constructs or (sub)lists of code constructs. The other bars are unwanted artifacts that, based on indentation alone, can not be distinguished from real regions. Notice that most of these ambiguities could be solved by using language specific information, for example about the use of curly braces in Java; lines that start with a curly brace are most likely to be part of the region being constructed. However, we implemented the algorithm in a language independent way.

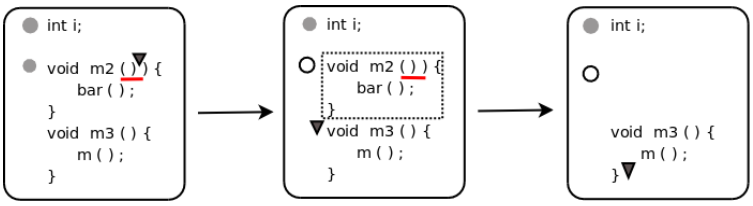
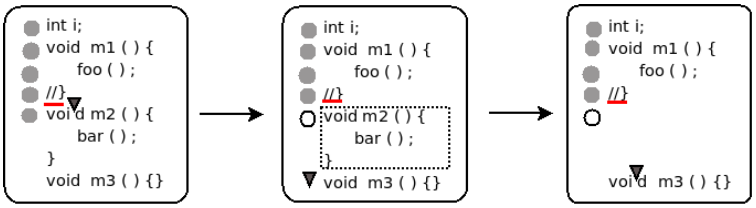
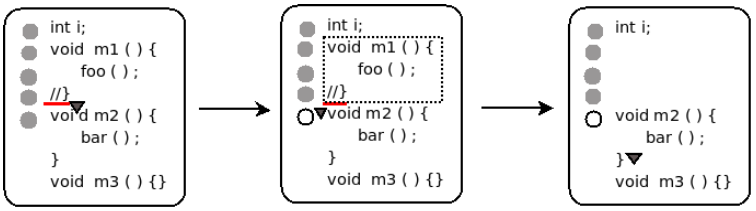


Fig. 23. A candidate region is validated and successfully discarded.



(a) A candidate region is rejected.



(b) An alternative candidate region is validated and successfully discarded.

Fig. 24. Iterative search for a valid region.

7.3. Region Selection

We follow an iterative process to select an appropriate region that encloses a syntax error. In each iteration, a different *candidate region* is considered. This candidate is then *validated* and either accepted as erroneous or rejected; in case of a rejected candidate, another candidate is considered.

The selection of candidate regions faces two challenges: First, the start line of the erroneous code construct is not known, second, multiple unsuitable regions are constructed because of the ambiguous interpretation of sibling lines. We adopt a pragmatic approach, subsequently selecting candidate regions for a different start line location with a different number of sibling lines. We start with validating small regions near the failure location, then we continue with validating regions of increased size as well as regions that are located further away from the failure location. More details are provided in Section 7.4 that describes the region selection algorithm.

A region is validated as erroneous in case discarding of that region solves the syntax error, e.g., parsing continues after the original failure location. We show example scenarios in Figure 23 and Figure 24. Figure 23 shows a syntax error and the point of detection, indicated by a triangle (left). A candidate region is selected based on the alignment of the `void` keyword and the closing bracket (middle figure), and validated by discarding the region. Since the parsing of the remainder of the fragment is successful (right), the region is accepted as erroneous. Figure 24(a) shows an example where a candidate region is rejected. Based on the point of detection, an obvious candidate region is the `m2` method (middle), which is discarded (right). However, the attempt to parse the succeeding construct leads to a premature parse failure (right), therefore the region is rejected. In Figure 24(b) an alternative candidate region is selected. This region is validated as erroneous.

The region validation criterion should balance the risk of evaluating a syntactically correct candidate region as erroneous, and the risk of evaluating an erroneous candidate region as syntactically correct. Both cases lead to large regions and/or spurious syntax errors, which should be avoided.

```
SELECTERRONEOUSREGION(failureLine)
 1  ▷ Input: Line where the parse failure occurs (or a parent of this line)
 2  ▷ Output: Region that contains the error
 3
 4  ▷ MAX_SIBLINES_COUNT: Max number of sibling lines in the candidate regions
 5  ▷ MAX_BW_INDEX: Max number of sibling lines that are backtracked
 6  for sibCount in 0 to MAX_SIBLINES_COUNT
 7     for bwSibIndex in 0 to MAX_BW_INDEX
 8           startLine ← GETPRECEDINGSIBLINE(failureLine, bwSibIndex)
 9           sibLine ← GETFOLLOWINGSIBLINE(startLine, sibCount)
10           endLine ← GETLASTDESCENDANTLINE(sibLine)
11           if startLine, endLine exist and TRYSKIPREGION(startLine, endLine) then
12             return (startLine, endLine) ▷ erroneous region
13           end
14     end
15  end
16  return SELECTERRONEOUSREGION(GETPARENTLINE(failureLine))

TRYSKIPREGION(startline, endline)
17  ▷ Output: true iff discarding the region startline ... endline
      lets parsing continue after the failure location

GETPRECEDINGSIBLINE(line, bwCount)
18  ▷ Output: Sibling line that preceeds line by bwCount siblings

GETFOLLOWINGSIBLINE(line, fwCount)
19  ▷ Output: Sibling line that succeeds line by fwCount siblings

GETLASTDESCENDANTLINE(line)
20  ▷ Output: Last descendant line of line, or line if no descendants exist

GETPARENTLINE(line)
21  ▷ Output: Parent line of line
```

Fig. 25. Algorithm to select a discardable region that contains the syntax error.

The underlying problem are multiple errors; it is not possible to distinguish a secondary parse failure from a genuine syntax error that happens to be close-by. We address the issue of multiple syntax errors by implementing a heuristic accept criterion. The criterion considers a candidate region as erroneous if discarding results in two more lines of code parsed correctly. The criterion is established after some experimentation and has shown good practical results.

7.4. Algorithm

Figure 25 shows the region selection algorithm in pseudo-code. The function `SelectErroneousRegion` takes as input the failure line and returns as output the erroneous region described by its start line and end line. The nested `for` loops (line 6,7) implement the iterative search process described in Section 7.3. The iteration starts with the smallest region (line 6, `sibCount=0`) that can be constructed for the failure line (line 7, `bwSibIndex=0`). In the first iteration (line 7), regions are selected at increasing distance from the failure location. The second iteration (line 6) increases the size of the selected regions. The iteration stops in case a selected region is validated as erroneous (lines 11-13). If no erroneous region is found, the search

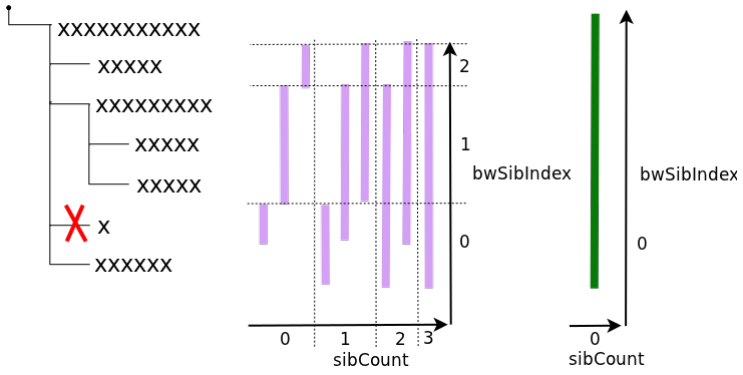


Fig. 26. Candidate regions subsequently tested for the indented code fragment at the left. Candidate regions are selected by backtracking ($bwSibIndex$) and by extending the number of sibling lines that are contained in the region ($sibCount$). Finally, the parent line is visited in the recursion step.

process continues by recursively visiting the parent of the failure line (line 16). For performance reasons, we restrict the maximum size of the visited regions (line 4) and the maximum number of backtracked lines (line 5). Good practical results were obtained with a maximum size of 5 sibling lines and 5 backtracking steps.

Figure 26 illustrates the region selection procedure applied to a small code fragment with a parse failure at the marked line. The vertical bars represent the regions that are subsequently visited by increasing the backtracking distance (`bwSibIndex`) and the region size (`sibCount`). The right most bar represents the parent region visited in the recursion step.

7.5. Practical Considerations

Separators and Operators. Region selection works for structures that form free standing blocks in the grammar, e.g., list elements and optional elements such as the `else` block in an `if-else` statement. A practical consideration are separators and operators that may reside between language constructs. For example, the constructs `FAILED` and `score <= 8` in this Java fragment can only be discarded if the separator (`,`), respectively the operator (`&&`) that connects these constructs with their preceding constructs are discarded as well. To address this issue, we have extended the region selection schema with a candidate region consisting of the original region plus the lexical token at the end of the preceding sibling line.

```
public enum Grade {
  EXCELLENT ,
  PASSED ,
  FAILED
}

Grade getGrade(){
  ...
  if(
    6 <= score &&
    score <= 8
  ) return Grade.PASSED;
  ...
}
```

Multi-line Comments and Strings. The selection procedure can generally select erroneous regions that are not located at the failure location. However, if the distance between the error and the failure location is too large, the region selection schema fails to locate the error. A particularly problematic case commonly seen in practice are unclosed flat structures such as block comments or multi-line strings. After the opening of the block comment (`/*`), the parser accepts all characters until the block comment is ended (`*/`) or the end of the file is reached. As a consequence, a missing block comment ending is typically detected at a large distance from the error location. The stack structure of the parser in these scenarios is characterized by a reduction that involves many characters starting from the characters that open the flat construct (`/*`). If this stack structure is recognized, a candidate region is selected from the start of the reduction, making it possible to cope with flat multi-line structures such as block comments for which errors may cause a parse failure far from the actual error location.

```
/* Comments ...
int foo(){
    ...
}
...
EOF
```

8. APPLYING ERROR RECOVERY IN AN INTERACTIVE ENVIRONMENT

A key goal of error recovery is its application in the construction of IDEs. Modern IDEs rely heavily on parsers to produce abstract syntax trees that form the basis for editor services such as the outline view, content completion, and refactoring. Users expect these services even when the program has syntactic errors, which is very common when source code is edited interactively. Experience with modern IDEs shows that for most services it is not a problem to operate on inaccurate or incomplete information as a consequence of syntax errors; for some services such as refactorings, errors and warnings can be presented to the user. In this section, we describe the role of error recovery in different editor services and show language-parametric techniques for using error recovery with these services.

8.1. Efficient Construction of Languages and Editor Services

While IDEs for languages have been constructed and used for several decades, only recently did they become significantly more sophisticated and indispensable for productivity of software developers. In early 2001, IntelliJ IDEA [Saunders et al. 2006] revolutionized the IDE landscape [Fowler 2005b], setting a new standard for highly interactive and language-specific IDE support for textual languages. Since then, providing good IDE support for new languages has become mandatory, posing a significant challenge for language engineers.

As IDEs become both more commonplace and more sophisticated, it becomes increasingly important to lower the threshold of creating new languages and developing IDEs for these languages. In order to make this possible, *language workbenches* have been developed that combine the construction of languages and editor services. Language workbenches improve the productivity of language engineers by providing specialized languages, frameworks, and tools [Fowler 2005a]. Examples of language workbenches for textual languages include EMFText [Heidenreich et al. 2009], MontiCore [Krahn et al. 2008; Grönniger et al. 2008], Spoofax [Kats and Visser 2010], TCS [Jouault et al. 2006], and Xtext [Efftinge and Voelter 2006].

The central artifact that language engineers define in a language workbench is the grammar of a language, which is used to generate a parser. The generated parser runs in the background with each key press or after a small delay passes, and provides a basis for all interactive editor services. Traditionally, IDEs used handwritten parsers or only did a lexical analysis of source code for syntax highlighting in real-time. By using a generated parser that runs every time the source code changes, they have access to more accurate, more up-to-date information, but they also crucially depend on the parser's performance and its support for error recovery.

8.2. Guarantees on Recovery Correctness

Using permissive grammars, bridge parsing and regional recovery, the parser can construct ASTs for syntactically incorrect inputs. These trees can be constructed using generated or handwritten recovery rules, and may have gaps for regions that could not be parsed. Ultimately, error recovery provides a speculative interpretation of the intended program, which may not always be the desired interpretation. As such, it is both unavoidable and not uncommon that editor services operate on inaccurate or incomplete information. Experience with modern IDEs shows that this is not a problem in itself, as programmers are shown both syntactic and semantic errors directly in the editor.

While error recovery is ultimately a speculative interpretation of an incorrect input, our approach does guarantee well-formedness of ASTs. That is, it will only produce ASTs with tree nodes that conform to the abstract structure imposed by production rules of the original (non-permissive) grammar. This property is maintained for all our recovery techniques. With respect to permissive grammars (Section 4 and 5), water recovery rules (Section 4.2) and literal insertion recovery rules (Section 4.3 and 4.5) do not contribute AST nodes, while insertion recovery rules for lexical productions (Section 4.3, 4.5) only contribute lexical tree nodes that correspond to the recovered lexicals. Bridge parsing (Section 6) and region recovery (Section 7) do not compromise the well-formedness property of the parse result since both techniques only modify the input string respectively by adding a literal and by skipping over a text fragment.

The property of well-formedness of trees significantly simplifies the implementation and specification of editor services, as they do not require any special logic to handle badly parsed constructs with missing nodes or special constructors. This approach also ensures separation of concerns: error recovery is purely performed by the parser, while editor services do not have to treat syntactically incorrect programs differently. This separation of concerns means that all editor services could be implemented without any logic specific for error recovery. Still, there are a number of editor services that inherently require some interaction with the recovery strategy, which we discuss next.

8.3. Syntactic Error Reporting

Syntax errors are reported to users by means of an error location and an error message. In traditional compilers, the error location was reported as a line/column offset, while modern IDEs use the location for the placement of error markers in the editor. We use generic error messages that depend on the class of recovery (Section 4.5). For water recovery rules and for region recoveries, we use "[string] not expected," for insertion rules we use "expected: [string]," and for insertion rules that terminate a construct we use "construct not terminated." The location at which the errors are reported is determined by the location at which a recovery rule is applied, rather than by the location of the parse failure. For region recoveries, where no recovery rule is applied, the start and end location of the region, plus the original failure location is reported instead.

Figure 27 shows a screenshot of an editor for Stratego with embedded Java. The shown code fragment contains two syntax errors. Due to error recovery, the editor can still provide syntax highlighting and other editor services, while it marks all the syntax errors inline with red squiggles.

8.4. Syntax Highlighting

Syntax highlighting has traditionally been based on a purely lexical analysis of programs. The most basic approach is to use regular expressions to recognize reserved words and other constructs and assign them a particular color. Unfortunately, for language engineers the maintenance of regular expressions for highlighting can be tedious and error prone; a more flexible approach is to use the grammar of a language. Using the grammar, a scanner can recognize tokens in a stream, which can be used to assign colors instead.

More recent implementations of syntax highlighting do a full context-free syntax analysis, or even use the semantics of a language for syntax highlighting. For example, they may assign Java field accesses a different color than local variable accesses.

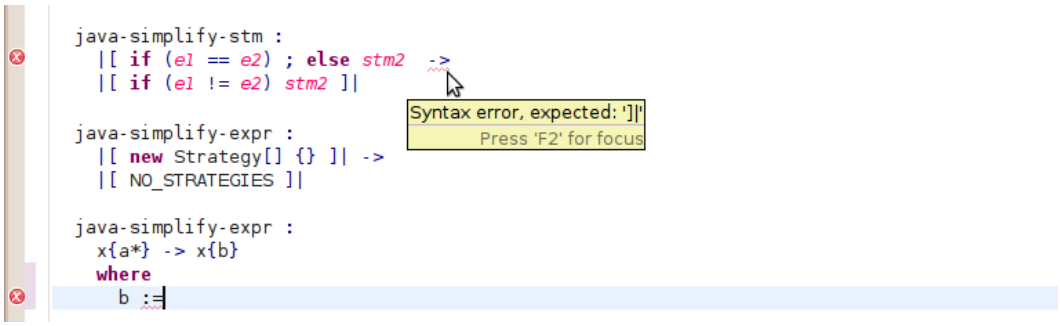


Fig. 27. An editor for Stratego with embedded quotations of Java code.

Scannerless syntax highlighting. When using a scannerless parser such as SGLR, a scanner-based approach to syntax highlighting is not an option; files must be fully parsed instead. This makes it important that a proper parse tree is available at all times, even in case of syntactic errors. To illustrate this, consider the following incomplete Java statement:

```
Tree t = new
```

Using a scanner, the word `new` can be recognized as one of the reserved keywords and can be highlighted as such. In the context of scannerless parsing, a well-formed parse tree must be constructed for the keyword to be highlighted. In situations like this one, that may not be possible, resulting in no highlighting for the `new` keyword.

Fallback syntax highlighting. Syntax highlighting is equally or possibly more important for syntactically incorrect programs than for syntactically correct programs, as it indicates how the editor interprets the program as a programmer is editing it. A *fallback syntax highlighting* mechanism is needed to address this issue.

A natural way of implementing fallback syntax highlighting is by using a lexical analysis for those cases where the full context-free parser is unable to distinguish the different words to be highlighted. This analysis can be performed by a rudimentary tokenizer that can recognize separate words such that they can be distinguished for colorization. Simple coloring rules can then be applied to any tokens that do not belong to recovered tree nodes, e.g. highlighting all the reserved keywords and string literals. Consequently, programmers get highly responsive syntax highlighting as they are typing, even if the program is not (yet) syntactically correct. A limitation of the approach is that with a tokenizer it cannot distinguish between keywords in different sublanguages, making the approach only viable as a fall-back option. We use the fallback syntax highlighting for discarded regions and in case the combined recovery technique fails, e.g. no AST is constructed for the erroneous program.

8.5. Content Completion

Content completion, sometimes called content assist, is an editor service that provides completion proposals based on the syntactic and semantic context of the expression that is being edited. Where other editor services should behave robustly in case of incomplete or syntactically incorrect programs, the content completion service is almost exclusively targeted towards incomplete programs. Content completion suggestions must be provided regardless of the syntactic state of a program: an incomplete expression '`blog.`' does not conform to the syntax, but for content completion it must still have an abstract representation.

Completion recovery rules. In case context completion is applied to an incomplete expression, the syntactic context of that expression must be recovered. This is especially challenging for language constructs with many elements, such as the "for" statement in the Java language. Even if only part of such a statement is entered by a user, it is important for the content completion service that there is an abstract representation for it. Based on the recovery rules of Section 4 this is not always the case. Water recovery rules interpret the incomplete expression as layout. As a consequence, the syntactic

```
context-free syntax

  "for" "(" FormalParam ":" Expr ")" Stm  ->
     Stm {cons("ForEach")}

  "for" "(" FormalParam ":" Expr ")"?  ->
     Stm {ast("ForEach(<1>, <2>, NULL())"), completion}

  "for" "(" FormalParam ":"? ")"?  ->
     Stm {ast("ForEach(<1>, NULL(), NULL())"), completion}
```

Fig. 28. Java `ForEach` production and its derived completion rules.

```
context-free syntax

  "for" "(" FormalParam ":" Expr ")"?  ->
     Stm {ast("ForEach(<1>, <2>, Block([]))"), completion}
```

Fig. 29. Java `ForEach` completion rule with placeholder pattern that matches the signature of the original production.

context is lost. Insertion recovery rules can recover some incomplete expressions, but only insert missing terminal symbols.

We introduce specific recovery rules for content completion that specify what abstract representation to use for incomplete syntactic constructs. These rules use the `{ast(`p`)}` annotation of SDF to specify a pattern p as the abstract syntax to construct. Figure 28 shows examples of these rules. The first rule is a normal production rule for the Java "for each" construct. The second rule indicates how to recover this statement if the `Stm` non-terminal is omitted, using a placeholder pattern `NULL()` in place of the abstract representation of the omission. The third rule handles the case where both non-terminals are omitted.

The completion recovery rules are automatically derived by analyzing the original productions in the grammar, creating variations of existing rules with omitted non-terminals and terminals marked as optional patterns. For best results, we generate rules that use placeholder patterns that reflect the signature of the original production. Since these rules preserve the wellformedness property, they are also applicable for normal error recovery. For example, in the second rule of Figure 28, the pattern `Block([])` can be used instead of the `NULL()` placeholder (Figure 29). Sensible placeholder patterns are constructed by recursively analyzing the production rules for the omitted non-terminals. In the given example, the production rule `"{" Stm* "}" -> Stm {cons("Block")}` provides the pattern `Block([])` as a placeholder for the `Stm` non-terminal, using the the empty list `[]` as the basic default for list productions.

Runtime support. Completion recovery rules are designed to support the special scenario of recovering the expression where content completion is requested. The cursor location provides a hint about the location of the (possible) error. Instead of backtracking after an error is found, we apply completion recovery rules if they apply to a character sequence that overlaps with the cursor location. This approach adequately completes constructs at the cursor location and minimizes the overhead of completion rules in normal parsing and other recovery scenarios. It also ensures that the completion recovery rules have precedence over the normal water and insertion recovery rules for the content completion scenario.

9. IMPLEMENTATION

We implemented our approach in Spoofax [Kats and Visser 2010], which is a language development environment that combines the construction of languages and editor services. Using SDF and JSGLR[9], Spoofax has the distinguishing feature that it supports language compositions and embeddings. In this section we give an overview of the general system and we discuss the adaptations we made for error recovery.

[9] `http://strategoxt.org/Stratego/JSGLR/`

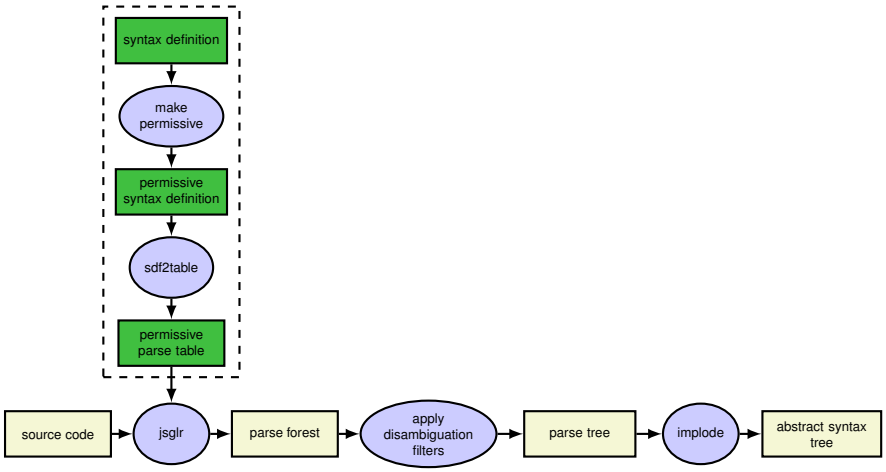


Fig. 30. Overview tool chain. Make-permissive generates a permissive version of the original grammar, for which a parse table is constructed by sdf2tbl. The (permissive) parse table is used by JSGLR to construct a parse tree for a (possible erroneous) input file, which is then imploded into an AST.

Figure 30 gives a general overview of the tool chain that handles parsing in Spoofax with integrated support for error recovery. Given a grammar definition in SDF, the *make-permissive* tool generates a permissive version of this grammar, for which a parse table is constructed by *sdf2table*. This parse table is used by the *JSGLR* parser, which constructs a parse tree for a (possible erroneous) input file. The parse tree is first disambiguated by applying post-parse filters, and then imploded into an AST.

The *make-permissive* tool was added to the tool chain specifically for the concern of error recovery. The tool implements a grammar-to-grammar transformation that applies the heuristic rules described in Section 4.5 and Section 8.5 that guide the generation of recovery rules. The tool is implemented in Aster [Kats et al. 2009], a language for decorated attribute grammars that extends the Stratego transformation language.

We adapted the JSGLR parser implementation so that it can efficiently parse correct and incorrect syntax fragments using the productions defined by the permissive grammar. For this reason, we implemented a selective form of backtracking specificly for recover productions. Furthermore, we implemented two additional recovery techniques, namely, bridge parsing and region selection. All mentioned techniques are implemented in Java and integrated in the JSGLR implementation. To summarize, we made the following adaptations to the Java based JSGLR parser:

— ignore productions labeled with the recover annotation during normal parsing
— ignore productions labeled with the completion annotation, unless the production applies to a character sequence that overlaps with the cursor location, and the completion service is triggered by the user.
— runtime disambiguation filter that selects the branch with the lowest number of recover/completion productions, preferring insertions over water productions.
— implementations for the different recovery techniques described in Sections 5, 6, and 7.
— some code to integrate the different recovery techniques, as described below.

Integrating recovery techniques. We combine the different techniques described in this paper in a multi-stage recovery approach (Figure 31). Region selection (RS) is applied first to detect the erroneous region. In case region selection fails to select the erroneous region, the whole file is selected instead. In the second stage, the erroneous region is inspected by one of the correcting techniques, bridge parsing (BP) or permissive parsing (PG). Since bridge parsing provides the most natural

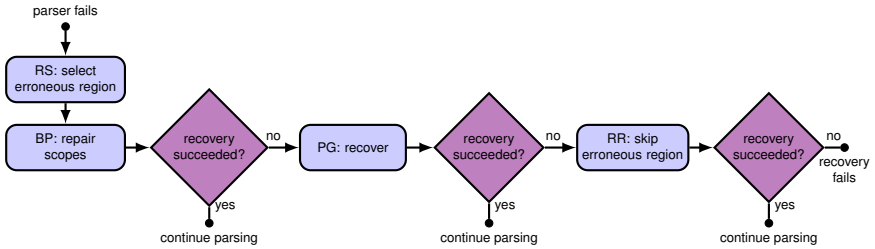


Fig. 31. Overview integrated recovery approach implemented in JSGLR.

recoveries from a user perspective, it is applied first. The bridge parser returns a set of recovery suggestions based on bracket insertions, which are applied during a re-parse of the erroneous region. In case the bridge parser suggestions do not lead to a successful recovery, the permissive grammars approach described in Sections 4 and 5 is used, where backtracking is restricted to the erroneous region. In case both correcting techniques fail, the erroneous region is skipped (region recovery, RR) as a fallback recovery strategy.

10. EVALUATION

We evaluate our approach with respect to the following properties:

— **Quality of recovery**: How well does the environment recover from input errors?
— **Performance and scalability**: What is the performance of the recovery technique? Is there a large difference in parsing time between erroneous and correct inputs? Does the approach scale up to large files?
— **Editor feedback**: How well do editor services perform based on the recovered ASTs?

In the remainder of this section we describe our experimental setup, experimentally select an effective combination of techniques and recovery rules, and show the quality and performance results of the selection.

10.1. Setup

In this section we describe our experimental setup; we explain how we construct a realistic test set, and how we measure recovery quality and performance.

10.1.1. Syntax Error Seeding. The development of representative syntax error benchmarks is a challenging task, and should be automated in order to minimize the selection bias. There are many factors involved for selecting the test inputs, such as the type of grammar, the type of error, distribution of errors over the file, and the layout characteristics of the test files. With these factors in mind, we have taken the approach of generating a reasonably large set of syntactically incorrect files from a smaller set of correct base files. We seed syntax errors at random locations in the base files, using a set of rules that cover different types of common editing errors. These rules were established after a statistical analysis of collected edit data for different languages [de Jonge and Visser 2012]. We distinguish the following categories for seeded errors:

— *Incomplete constructs*, language constructs that miss one or more symbols at the suffix, e.g. an incomplete `for` loop `for (x = 1; x`.
— *Random errors*, constructs that contain one or more token errors, e.g. *missing*, *incorrect* or *superfluous* symbols.
— *Scope errors*, constructs with missing or superfluous scope opening or closing symbols.
— *String or comment errors*, block comments or string literals that are not properly closed, e.g., `/*...*`
— *Large erroneous regions*, severely incorrect code fragments that cover multiple lines.

— *Language specific errors*, errors that are specific for a particular language.
— *Combined errors*, two or more errors from the above mentioned categories, randomly distributed over the source file.

10.1.2. Test Oracle. To measure the quality and performance of a recovery, we compare the results obtained for the recovered file against the results for the base file or expected file. In some cases, the base file does not realistically reflect the expected result, as information is lost in the generated erroneous file. For these cases we construct an expected result, a priori. For example, for a "for" loop with an *Incomplete construct* error – such as `for (x = 1; x` – the original body of the construct is lost. For this "for" loop, we complete the construct with the minimal amount of symbols possible, which results in the expected construct `for (x = 1; x; ) {}`.

10.1.3. Measuring Quality. We use two methods to measure the quality of the recovery results. First, we do a manual inspection of the pretty-printed results, following the quality criteria of Pennello and DeRemer [1978]. Following these criteria, an *excellent* recovery is one that is exactly the same as the intended program, a *good* recovery is one that results in a reasonable program without spurious or missed errors, and a *poor* recovery is a recovery that introduces spurious errors or involves excessive token deletion. The Pennello and DeRemer criteria represent the state of the art evaluation method for syntactic error recovery applied in, amongst others, [Pennello and DeRemer 1978; Pai and Kieburtz 1980; Degano and Priami 1995; Corchuelo et al. 2002].

Since human criteria form an evaluation method that is arguably subjective, as a second method, we also do an automated comparison of the abstract syntax. For this, we print the AST of the recovered file to text using the ATerm format [van den Brand et al. 2000], formatted so that nested structures appear on separate lines. We then count the number of lines that differ in the recovered AST compared to the AST of the expected file (the "diff"). The advantage of this approach is that it is objective, and assigns a larger penalty to recoveries for which a larger area of the text does not correspond to the expected file, where structures are nested improperly, or when multiple deviations appear on what would be a single line of pretty-printed code. Furthermore, using this approach the comparison can be automated, which makes it feasible to apply to larger test sets.

The scales for the figures we show are calibrated such that "no diff" corresponds to the *excellent* qualification, a "small diff" (1–10 lines of abstract syntax) roughly corresponds to the *good* qualification, and a "large diff" ($>$ 10 lines) approximately corresponds to the *poor* qualification. After a selection of recovery techniques and recovery rule sets, we show both metrics together in a comprehensive benchmark in Section 10.2.3.

10.1.4. Measuring Performance. To compare the performance of the presented recovery technique under different configurations, we measure the additional time spent for error recovery. That is, we compute the extra time it takes to recover from one or more errors (the recovery time) by subtracting the parse time of the correct base file or expected file from the parse time of the incorrect variation of this file.

To evaluate the scalability of the technique, we compare the parse times for erroneous and correct files of different sizes in the interval $1,000$–$15,000$ LOC.

For all performance measures included in this paper, an average, collected after three runs, is used. All measuring is done on a "pre-heated" JVM running on a laptop with an Intel(R) Core(TM) 2 Duo CPU P8600, 2.40GHz processor, 4 GB Memory.

10.1.5. Test sets. To evaluate quality and performance of the suggested recovery techniques we use a test set of programs written in WebDSL, Stratego-Java, Java-SQL and Java, based on the following projects:

— *YellowGrass*: A web-based issue tracker written in the WebDSL language.[10]

[10] `http://www.yellowgrass.org/`.

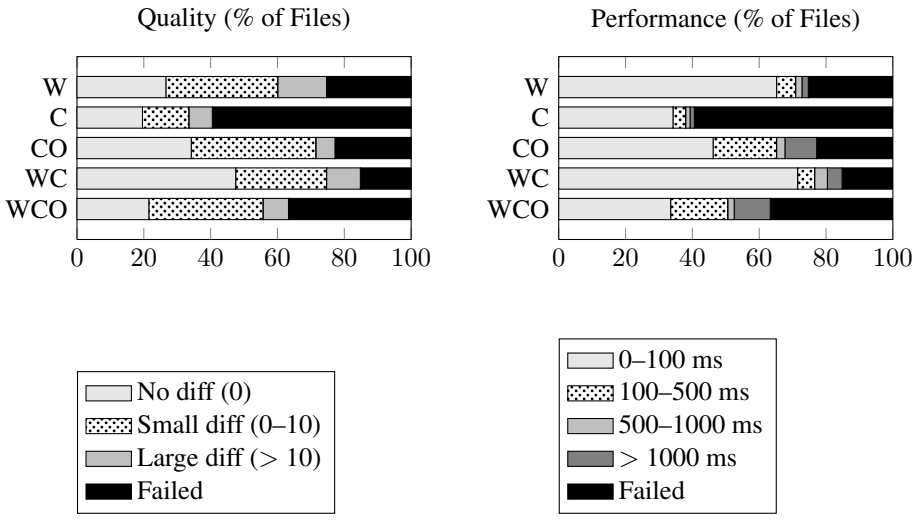


Fig. 32. Quality and performance (recovery times) using a permissive grammar with different recovery rule sets for Stratego-Java. **W** - Water, **C** - Insertion of closing brackets, **O** - Insertion of opening brackets.

— *The Dryad compiler*: An open compiler for the Java platform [Kats et al. 2008] written using Stratego-Java.
— *The StringBorg project*: A tool and grammar suite that defines different embedded languages [Bravenboer et al. 2010], providing Java-SQL code.
— *JSGLR*: A Java implementation of the SGLR parser algorithm.[11]

We selected five representative base files from each project, and generated test files using the error seeding technique. We applied a sanity check to ensure that generated test cases are indeed syntactically incorrect and that there are no duplicates. In total, we generated 334 Stratego-Java test cases, 190 WebDSL test cases, 195 Java-SQL test cases, and 329 Java test cases. In addition, we generated a second test set consisting of 314 Stratego-Java test cases in the *Incomplete construct* and *Erroneous context* categories specifically to evaluate the content completion editor service. Finally, for testing of scalability, we manually constructed a test set consisting of 28 erroneous Stratego-Java files of increasing size in the interval of $1,000$–$15,000$ LOC.

10.2. Experiments

There are a large number of configurations to consider in evaluating the presented approach: combinations of languages, recovery rule sets, and recovery techniques. In order to limit the size of the presented results, we first concentrate on one language and experiment with different configuration of recovery rule sets and recovery techniques. For these initial experiments we use the Stratego-Java language – a fairly complex language embedding. After selecting an effective configuration, we perform additional experiments with other languages.

10.2.1. Selecting a Recovery Rule Set. In this experiment we focus on selecting the most effective recovery rule set for a permissive grammar with respect to quality and performance. The permissive grammar technique is used in combination with region selection, described in Section 7. That is, the recovery rules are applied on a selected erroneous region, but the fallback region recovery technique is disabled since it obscures failed recoveries obtained for the evaluated rule sets. In this experiment, we set a time limit of 5 seconds to cut off recoveries that take an (almost) infinite time to complete.

[11] `http://strategoxt.org/Stratego/JSGLR/`.

For the permissive grammars approach of Section 4, there are three recovery rule sets that we evaluate in isolation and in combination – *Water* (W), *insertion of Closing brackets* (C), and *insertion of Open brackets* (O). Results from the experiment are shown in Figure 32. The figure includes results for W, C, CO, WC and WCO for a Stratego-Java grammar. The remaining combinations, O and WO, were excluded since it is arguably more important to insert closing brackets than to insert open brackets in an interactive editing scenario.

The results show that the insertion of closing brackets (C) and the application of water rules (W) both contribute to the quality of a recovery. Combined together (WC) they further improve recovery results. The insertion of opening brackets (O) does improve the recovery quality for insertion-only grammars, which follows from comparing C to CO. However, when all rules are combined (WCO), the recovery quality decreases in comparison with the WC grammar. This slightly unexpected result is partly explained by the fact that the insertion rules for opening brackets prove to be too costly with respect to performance, which leads to failures because of exceeding of the time limit set. A second explanation is that the combined rule set (WCO) allows many creative recoveries that often do not correspond to the human intended recoveries. We conclude that WC seems to be the best trade off between Quality and Performance.

In this experiment we only set a limit on the number of lines (75) that were inspected during backtracking, and a time limit of 5 seconds to cut off recoveries that take an (almost) infinite time to complete. The performance diagram shows that this leads to objectionable parse times in certain cases, $4.4\% > 1.0$ seconds and $15.2\% > 5.0$ seconds (failures) for WC. For these cases, a practical implementation would opt for an inferior recovery result obtained by applying a fallback strategy (region skipping in our approach). We apply this strategy in the remainder of this section, setting a time limit of 1000 milliseconds on the time spent applying recovery rules.

10.2.2. Selecting Recovery Techniques. In this experiment, we focus on selecting the best parser configuration combining the recovery techniques presented in this paper: the permissive grammars and backtracking approach of Section 4 and 5 (PG), bridge parsing of Section 6 (BP), and the region selection technique of Section 7 (RS), which can be applied as a fall back recovery technique (RR) by skipping the selected region. We use the WC recovery rule set of Section 10.2.1. and the Stratego-Java test set. We first applied the techniques in isolation: first regional recovery by skipping regions (RR), and then parsing with permissive grammars (PG). Bridge parsing is not evaluated separately, since it has a limited application scope and only works as a supplementary method. We then evaluate the approaches together: first parsing with permissive grammars applied to a selected region (RS-PG), then adding region recovery (RR) as a fallback recovery technique (RS-PG-RR), and finally the combination of all three techniques together (RS-BP-PG-RR). Throughout this experiment, we set a time limit of 1 second for applying recovery rules (PG). The results from the experiment are shown in Figure 33.

Figure 33 (Performance) shows the performance results for the different combinations of techniques. The results show that region recovery (RR) gives good performance in all cases, and that region selection (RS) positively affects the performance of the permissive grammar technique (RS-PG versus PG). Furthermore, applying the bridge parsing technique (BP) does not negatively affect performance according to Figure 33 (RS-PG-BP-RR versus RS-PG-RR). Since all techniques give reasonable performance, we focus on quality to find the best combination of techniques.

Considering the Quality part of Figure 33 and the results of PG, we see that it has the largest number of failed recoveries (17%), but regardless of this fact it still leads to reasonable recoveries (< 10 diff lines) in the majority of cases (75%). Restricting PG to a selected erroneous region (RS-PG) leads to more excellent recoveries (48% versus 44%). For regional recovery (RR), the situation is exactly the opposite. As expected, skipping a whole region in most cases does not lead to the optimal recovery. However, the skipping technique does provide a robust mechanism, leading to a successful parse in most cases (94%). Combining both techniques (RS-PG-RR), improves the robustness (96%), as well as the precision (80% small or no diff) compared to both individual techniques.

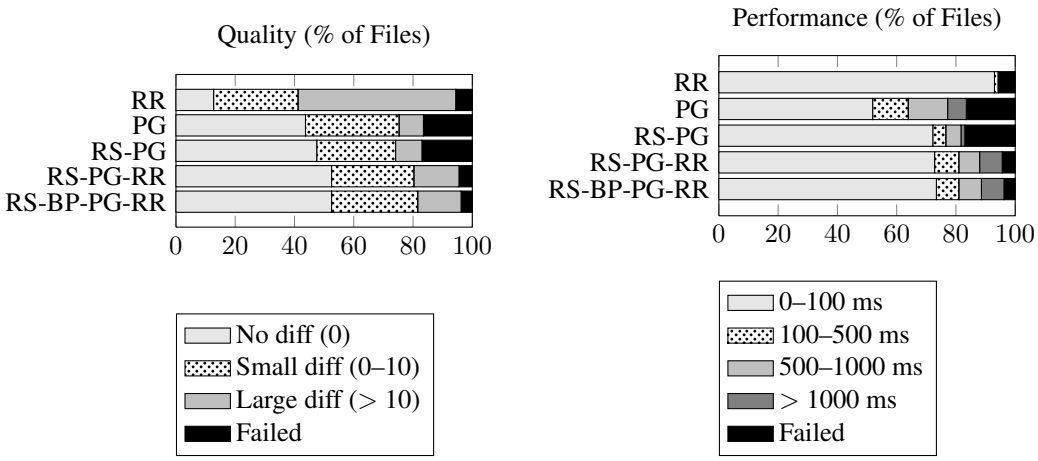


Fig. 33. Quality and performance (recovery times) using combinations of techniques for Stratego-Java. **RR** - Region selection and recovery, **PG** - Permissive grammars, **RS** - Region selection, **BP** - Bridge parsing.

Interestingly, Figure 33 shows little beneficial effects of the bridge parsing method (BP). There is a strong use case for bridge parsing, as it can pick the most likely recovery in case of a syntax error that affects scoping structures. However, the technique is most effective for programs that use deep nesting of blocks, which are relatively rare in Stratego-Java programs. Still, the approach shows no harmful effects. For other languages its positive effects tend to be more pronounced, as we have shown in [de Jonge et al. 2009]. In this previous study, a test set with focus on scope errors is used; showing that bridge parsing improves the results of the permissive grammar technique in 21% of the cases where one or more scope errors occur. The cases where the bridge parser contributes to a better recovery are cases where the region selection technique does not detect the erroneous scope as precisely on its own, which is typical for fragments with multiple clustered scope errors.

10.2.3. Overall benchmark. As an overall benchmark, we compare the quality of our techniques to the parser used by Eclipse's Java Development Tools (JDT). It should be noted that, while our approach uses fully automatically derived recovery specifications, the JDT parser in contrast, uses specialized, handwritten recovery rules and methods. We use the JDT parser with statement-level recovery enabled, following the guidelines given by Kuhn and Thomann [2006].

Both Eclipse and our approach apply an additional recovery technique in the scenario of content completion. Both techniques use specific completion recovery rules that require the completion request (cursor) location as additional information, also, these rules construct special completion nodes that may not represent valid Java syntax. We did not include these techniques in this general benchmark section since they specifically target the use case of content completion and do not work in other scenarios.

Figure 34 shows the quality results acquired for the Java test set, using diff counts and applying the criteria of Pennello and DeRemer [1978]. To ensure that all the results are obtained in a reasonable time span, we set a parse time limit of 1 second. The results show that the SGLR recovery, using different steps and granularity, is in particular successful in avoiding large diffs, thereby providing more precise recoveries compared to the JDT parser. The JDT parser on the other hand managed to construct an excellent recovery in 67% of the cases, which is a bit better than the 62% of the SGLR parser. The SGLR parser failed to construct an AST in less than 1% of the cases, while the JDT parser constructed an AST in all cases. However, manual inspection revealed that in most large diff cases only a very small part of the original file was reconstructed, e.g. only the import lines or the import lines plus the class declaration whereby all declarations in the body were skipped. We conclude that our automatically derived recovery technique is at least on par with practical standards.

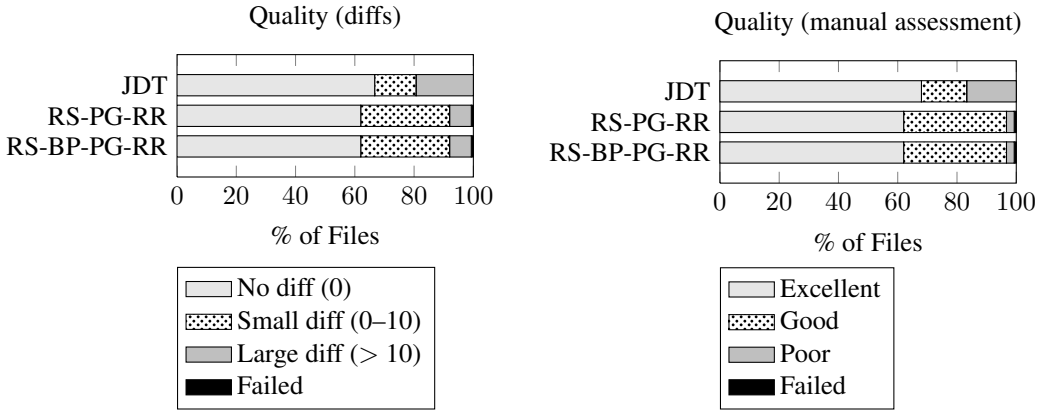


Fig. 34. Quality of our approach compared to JDT. **RS** - Region selection, **RR** - Region recovery, **PG** - Permissive grammars, **BP** - Bridge parsing, **JDT** - Java Developer Toolkit.

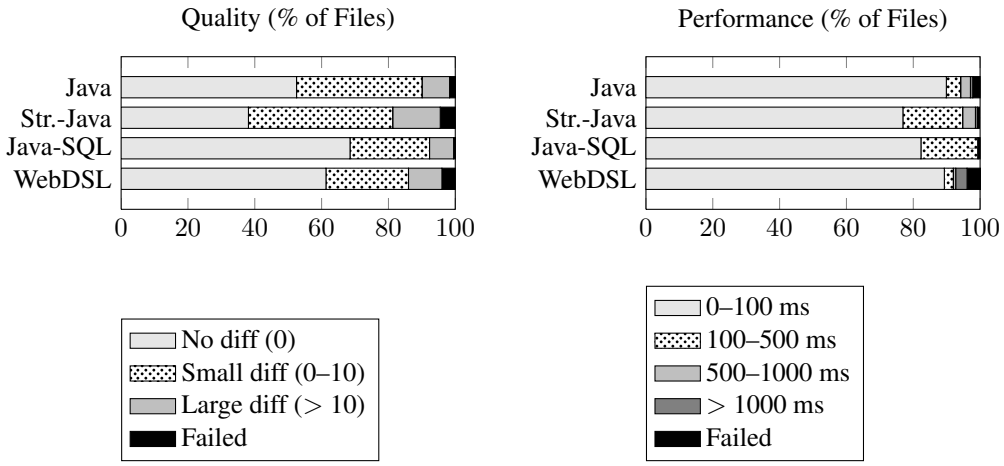


Fig. 35. Quality and performance (recovery times) for different languages.

10.2.4. Cross-language quality and performance. In this experiment we test the applicability of our approach to different languages, using the RS-BP-PG-RR configuration and the WC rule set. For simplicity and to ensure a clear cross-language comparison, we focus only on syntax errors that do not require manual reconstruction of the expected result, i.e., *Random errors*, *Scope errors* and *String or comment errors*. This allows for a fully automated comparison of erroneous and intended parser outputs. The results of the experiment are shown in Figure 35. The figure shows good results and performance across the different languages. From the diagram it follows that the quality of the recoveries varies for the different test sets. More specifically, the recoveries for Java-SQL, in general, are better than the ones for Stratego-Java. Differences like these are both hard to explain and predict, and depend on the characteristics of a particular language, or language combination, as well as the test programs used.

10.2.5. Performance and Scalability. In this experiment we focus on the performance of our approach. We want to study scalability and the potential performance drawbacks of adding recovery rules to a grammar, i.e., the effect of increasing the size of the grammar. We use the Stratego-Java language throughout this experiment with the RS-BP-PG-RR recovery configuration.

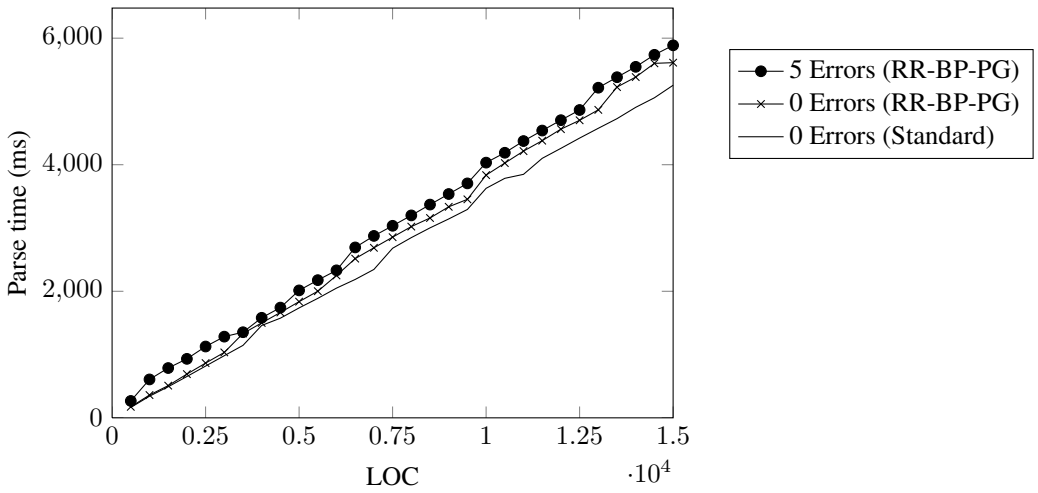


Fig. 36. Parse times for files of different length with and without errors. The files are written in the Stratego-Java language and parsed with the RR-BP-PG recovery configuration.

To test scalability, we construct a test set consisting of files of different size in the interval $1,000$–$15,000$ LOC, obtained by duplicating 500-line fragments from a base file in the Stratego-Java test set. For each test file, the same number of syntax errors are added manually, scattered in such a way that clustering of errors does not occur. We measure parse times as a function of input size, both for syntactically correct files and for files that contain syntax errors. The results, shown as a plot in Figure 36, show that parse times increase *linearly* with the size of the input, both for correct and for incorrect files. Furthermore, the extra time required to recover from an error (recovery time) is independent of the file size, which follows from the fact that both lines in the figure have the same coefficient.

As an additional experiment we study the performance drawbacks in the increased size of a permissive grammar. The extra recovery productions added to a grammar to make it more permissive also increase the size of that grammar, which may negatively affect parse times of syntactically correct inputs. We measure this effect by comparing parse times of the syntactically correct files in the test set, using the standard grammar and the WC permissive grammar. The results show that the permissive grammar has a small negative effect on parse times of syntactically correct files. The effect of modifying the parser implementation to support backtracking was also measured, but no performance decrease was found. We consider the small negative performance effect on parsing syntactically correct files acceptable since it does not significantly affect the user experience for files of reasonable size.

10.2.6. Content Completion. Error recovery helps to provide editor services on erroneous input. Especially challenging is the content completion service, which almost exclusively targets incomplete programs. In Section 8.5 we discussed the strengths and limitations of our current approach with respect to content completion. To overcome the limitations, we introduced a technique to automatically derive special completion rules that are applied near the cursor location. In this section we evaluate how well the current approach (water and insertion rules) serve the purpose of content completion, and how the completion rules improve on this.

We evaluated completion recovery on a set of 314 test cases that simulate the scenario of a programmer triggering the content completion service. Accurate completion suggestions require that the syntactic context, the tree node where completion is requested, is available in the recovered tree. To evaluate the applicability with respect to content completion, we distinguish between recoveries that preserve the syntactic context required for content completion and those that do not.

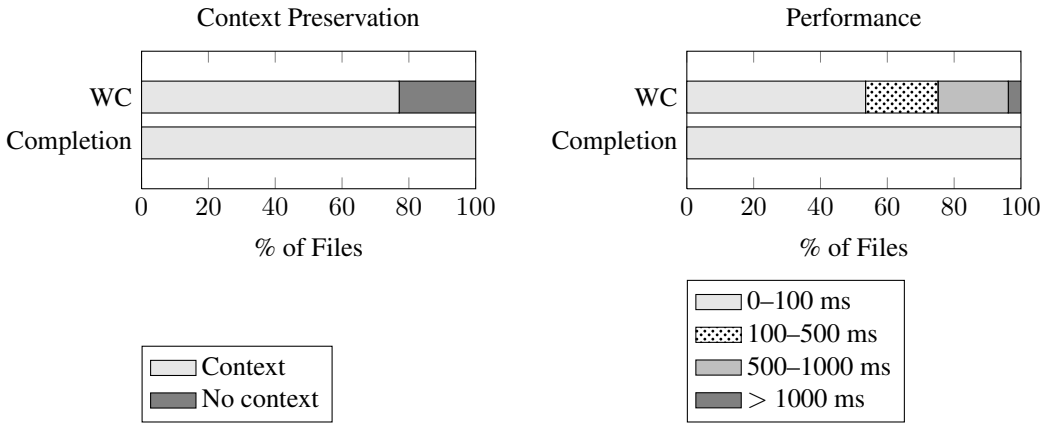


Fig. 37. Context preservation and performance (recovery times) of the Stratego-Java grammar extended with completion rules (**Completion**) and extended with recovery rules (**WC**).

Figure 37 shows the results for our recovery technique with and without the use of completion recovery. Using the original approach (with the WC rule set), the syntactic context was preserved in 77 percent of the cases, which shows that the recovery approach is useful for content completion, but is prone to unsatisfactory recoveries in certain cases. Furthermore, recovering large incomplete constructs can be inefficient since it requires many water and insertion rule applications.

Both problems are addressed by the completion recovery technique, which is specifically designed to handle syntax errors that involve incomplete language constructs. Figure 37 shows the results for the completion recovery strategy of Section 8.5, using a permissive grammar with the WC rule set plus completion rules. Using this strategy, the syntactic context is preserved in all cases, without noticeable time overhead. The low recovery times are a consequence of the (adapted) runtime support that exploits the fact that the cursor location is part of the erroneous construct.

A disadvantage of the completion rules is that they significantly increase the size of the grammar, which can negatively affect the parsing performance for syntactically correct inputs. We compared parse times of syntactically correct inputs for the WC/Completion grammar with parse times for the WC grammar, and measured an overhead factor of 1.2. Given that completion rules are highly effective and essential for the content completion functionality, this overhead seems acceptable. For normal editing scenarios, the completion rules can also be applied as an additional recovery mechanism that is effective only at the cursor location, although we have not focused on this capability in the experiments in this section.

10.3. Summary

In this section we evaluated the quality and performance of different rule sets for permissive grammars, and different configurations for parsing with permissive grammars, region recovery, and bridge parsing. Through experimental evaluation we found that the WC rule set provides the best balance in quality and performance. The three techniques each have their merits in isolation, and work best in combination. Through additional experiments we showed that the recovery quality and performance hold up to the standard set by the JDT, that our approach is scalable, and that it works across multiple languages. In addition, we showed its effectiveness for content completion.

11. RELATED WORK

The problem of handling syntax errors during parsing has been widely studied [Lévy 1971; Mauney and Fischer 1988; Pai and Kieburtz 1980; Barnard and Holt 1982; Tai 1978; Fischer et al. 1980; Degano and Priami 1995; McKenzie et al. 1995; Corchuelo et al. 2002]. We focus on LR parsing

for which there are several different error recovery techniques [Degano and Priami 1995]. These techniques can be divided into *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. When a synchronizing token is reached, states are popped from the stack until the state at the top enables the resumption of the parsing process. Our layout-sensitive regional recovery algorithm can be used in a similar fashion, but selects discardable regions based on layout.

Correcting recovery methods for LR parsers typically attempt to insert or delete tokens nearby the location of an error, until parsing can resume [Tai 1978; McKenzie et al. 1995; Cerecke 2002]. There may be several possible corrections of an error which means a choice has to be made. One approach applied by Tai [1978] is to assign a cost (a minimum correction distance) to each possible correction and then choose the correction with the least cost. This approach of selecting recoveries based on a minimum cost is related to recovery selection in our permissive grammars, where the number of recovery rules used in a correction decides the order in which recoveries are considered (Section 4).

Successful recovery mechanisms often combine more than one technique [Degano and Priami 1995]. For example, panic mode is often used as a fall back method if correction attempts fail. Burke and Fisher [1987] present a correcting method based on three phases of recovery. The first phase looks for simple correction by the insertion or deletion of a single token. If this does not lead to a recovery, one or more open scopes are closed. The last phase consists of discarding tokens that surround the parse failure location. In our work we take indentation into account, for the regional recovery technique and for scope recovery using bridge parsing. In addition, by starting with region selection, the performance as well as the quality of the permissive grammars approach recovery is improved.

Regional error recovery methods [Lévy 1971; Mauney and Fischer 1988; Pai and Kieburtz 1980; Barnard and Holt 1982] select a region that encloses the point of detection of an error. Typically, these regions are selected based on nearby marker tokens (also called fiducial tokens [Pai and Kieburtz 1980], or synchronizing symbols [Barnard and Holt 1982]), which are language-dependent. In our approach, we assign regions based on layout instead. Layout-sensitive regional recovery requires no language-specific configuration, and we showed it to be effective for a variety of languages. Similar to the fiducial tokens approach, it depends on the assumption that languages have recognizable (token or layout) structures that serve for the identification of regions.

Barnard and Holt [1982] presents an hierarchic error repair approach using *phases* corresponding to lists of lines. For instance, a phase may be a set of declarations that must appear together. These phases are similar to our regions, with the difference that regions are constructed based on layout. Both approaches have some kind of local repair within phases or regions, and may skip parts of the input.

The LALR Parser Generator (LPG) [Charles 1991] is incorporated into IMP [Charles et al. 2007] and is used as a basis for the Eclipse JDT parser. LPG can derive recovery behavior from a grammar, and supports recovery rules in the grammar and through semantic actions. Similar to our approach, LPG detects scopes in grammars. However, unlike our approach, it does not take indentation into account for scope recovery.

11.1. Recovery for Composite Languages

Using SGLR parsing, our approach can be used to parse composed languages and languages with a complex lexical syntax. In related work, only a study by Valkering [2007], based on substring parsing [Rekers and Koorn 1991], offered a partial approach to error recovery with SGLR parsing. To report syntactic errors, Valkering inspects the stack of the parser to determine the possible strings that can occur 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. Using substring parsing and artificial reduce actions, Valkering's approach could construct a set of partial, often ambiguous, parse trees, whereas our approach constructs a single, well-formed parse tree.

Lavie and Tomita [1993] developed GLR*, a noise skipping algorithm for context-free grammars. 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 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.

Composed languages are also supported by parsing expression grammars (PEGs) [Ford 2002; Grimm 2006]. PEGs lack the declarative disambiguation facilities [Visser 1997c] that SDF provides for SGLR. Instead, they use greedy matching and enforce an explicit ordering of productions. To our knowledge, no automated form of error recovery has been defined for PEGs. However, existing work on error recovery using parser combinators [Swierstra and Duponcheel 1996] may be a promising direction for recovery in PEGs. Furthermore, based on the ordering property of PEGS, a "catch all" clause is sometimes added to a grammar, which is used if no other production succeeds. Such a clause can skip erroneous content up to a specific point (such as a newline) but does not offer the flexibility of our approach.

11.2. IDE support for Composite Languages

We integrated our recovery approach into the Spoofax [Kats et al. 2010] language workbench. A related project, also based on SDF and SGLR, is the Meta-Environment [van den Brand et al. 2002; van den Brand et al. 2007]. 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 does not provide error recovery.

Another language development environment is MontiCore [Krahn et al. 2007; Krahn et al. 2008]. Based on ANTLR [Parr and Quong 1995], 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 [Krahn et al. 2007; Krahn et al. 2008]. 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.

11.3. Island Grammars

The basic principles of our permissive grammars and bridge parsing are based on the water productions from island grammars. Island grammars [van Deursen and Kuipers 1999; Moonen 2001] 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 and Lämmel [2003] 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 extensive set of error reporting capabilities.

More recently, island grammars have also been applied to parse composite languages. Synytskyy et al. [2003] 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.

12. CONCLUSION

Scannerless, generalized parsers support the full set of context-free grammars, which is closed under composition. With a grammar formalism such as SDF, they can be used for declarative specification and composition of syntax definitions. Error recovery for scannerless, generalized parsers 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.

We presented three techniques for error recovery. First, permissive grammars, to relax grammars with recovery rules so that strings can be parsed that are syntactically incorrect according to the original grammar. Second, backtracking, to efficiently parse files without syntax errors and to gracefully cope with errors locally. Third, region recovery, to identify regions of syntactically incorrect code, thereby constraining the search space of backtracking and providing a fallback recovery strategy. Using bridge parsing, this technique takes indentation usage into account to improve recoveries of scoping constructs. We evaluated our approach using a set of existing, non-trivial grammars, showing that the techniques work best when used together, and that they have a low performance overhead and good or excellent recovery quality in a majority of the cases.

Acknowledgments. This research was supported by NWO/JACQUARD projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*. We thank Karl Trygve Kalleberg, whose Java-based SGLR implementation has been invaluable for this work, and 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

BARNARD, D. T. AND HOLT, R. C. 1982. Hierarchic syntax error repair for LR grammars. *International Journal of Computer and Information Sciences 11,* 4, 231–258.

VAN DEN BRAND, M. G. J., DE JONG, H., KLINT, P., AND OLIVIER, P. 2000. Efficient annotated terms. *Software, Practice & Experience 30,* 3, 259–291.

BRAVENBOER, M., DOLSTRA, E., AND VISSER, E. 2007. Preventing injection attacks with syntax embeddings. In *Generative Programming and Component Engineering, 6th International Conference, GPCE 2007*, C. Consel and J. L. Lawall, Eds. ACM, Salzburg, Austria, 3–12.

BRAVENBOER, M., DOLSTRA, E., AND VISSER, E. 2010. Preventing injection attacks with syntax embeddings. *Science of Computer Programming 75,* 7, 473–495.

BRAVENBOER, M., KALLEBERG, K. T., VERMAAS, R., AND VISSER, E. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming 72,* 1-2, 52–70.

BRAVENBOER, M. AND VISSER, E. 2004. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*, J. M. Vlissides and D. C. Schmidt, Eds. ACM, Vancouver, BC, Canada, 365–383.

BRAVENBOER, M., ÉRIC TANTER, AND VISSER, E. 2006. Declarative, formal, and extensible syntax definition for AspectJ. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, P. L. Tarr and W. R. Cook, Eds. ACM, 209–228.

BURKE, M. G. AND FISHER, G. A. 1987. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Trans. Program. Lang. Syst. 9,* 2, 164–197.

CERECKE, C. 2002. Repairing syntax errors in lr-based parsers. In *ACSC*, M. J. Oudshoorn, Ed. CRPIT Series, vol. 4. Australian Computer Society, 17–22.

CHARLES, P. 1991. A practical method for constructing efficient lalr(k) parsers with automatic error recovery. Ph.D. thesis, New York University.

CHARLES, P., FUHRER, R. M., AND SUTTON, JR., S. M. 2007. IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In *Automated Software Engineering (ASE 2007)*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM, 485–488.

CORCHUELO, R., PÉREZ, J. A., CORTÉS, A. R., AND TORO, M. 2002. Repairing syntax errors in LR parsers. *ACM Trans. Program. Lang. Syst. 24,* 6, 698–710.

DE JONGE, M., NILSSON-NYMAN, E., KATS, L. C. L., AND VISSER, E. 2009. Natural and flexible error recovery for generated parsers. In *Software Language Engineering, Second International Conference, SLE 2009*, M. van den Brand, D. Gasevic, and J. Gray, Eds. Lecture Notes in Computer Science Series, vol. 5969. Springer, 204–223.

DE JONGE, M. AND VISSER, E. 2012. Automated evaluation of syntax error recovery. Tech. Rep. TUD-SERG-2012-035, Delft University of Technology, Software Engineering Research Group, Delft, The Netherlands.

DEGANO, P. AND PRIAMI, C. 1995. Comparison of syntactic error handling in LR parsers. *Software – Practice and Experience 25,* 6, 657–679.

DUCASSE, S., NIERSTRASZ, O., SCHÄRLI, N., WUYTS, R., AND BLACK, A. 2006. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems (TOPLAS) 28,* 2, 331–388.

EFFTINGE, S. AND VOELTER, M. 2006. oAW xText: a framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*.

FISCHER, C. N., MILTON, D. R., AND QUIRING, S. B. 1980. Efficient LL(1) error correction and recovery using only insertions. *Acta Inf. 13*, 141–154.

FORD, B. 2002. Packrat parsing: Simple, powerful, lazy, linear time. In *International Conference on Functional Programming (ICFP'02)*. SIGPLAN Notices Series, vol. 37. ACM, 36–47.

FOWLER, M. 2005a. Language workbenches: The killer-app for domain specific languages?

FOWLER, M. 2005b. PostIntelliJ. `http://martinfowler.com/bliki/PostIntelliJ.html`.

GRIMM, R. 2006. Better extensibility through modular syntax. In *PLDI*. 38–51.

GRÖNNIGER, H., KRAHN, H., RUMPE, B., SCHINDLER, M., AND VÖLKEL, S. 2008. Monticore: a framework for the development of textual domain specific languages. In *ICSE*. 925–926.

HEERING, J., HENDRIKS, P. R. H., KLINT, P., AND REKERS, J. 1989. The syntax definition formalism sdf. *SIGPLAN Not. 24,* 11, 43–75.

HEIDENREICH, F., JOHANNES, J., KAROL, S., SEIFERT, M., AND WENDE, C. 2009. Derivation and refinement of textual syntax for models. In *ECMDA-FA*. 114–129.

JOHNSTONE, A., SCOTT, E., AND ECONOMOPOULOS, G. 2004. Generalised parsing: Some costs. *Lecture Notes in Computer Science 2985*, 89–103.

JOUAULT, F., BÉZIVIN, J., AND KURTEV, I. 2006. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Generative and Component Engineering (GPCE'06)*. ACM, 249–254.

KATS, L. C. L., BRAVENBOER, M., AND VISSER, E. 2008. Mixing source and bytecode: a case for compilation by normalization. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, G. E. Harris, Ed. ACM, 91–108.

KATS, L. C. L., DE JONGE, M., NILSSON-NYMAN, E., AND VISSER, E. 2009. Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, G. T. Leavens, Ed. ACM SIGPLAN Notices Series, vol. 44. ACM Press, New York, NY, USA, 445–464.

KATS, L. C. L., DE JONGE, M., NILSSON-NYMAN, E., AND VISSER, E. 2011. The permissive grammars project. `http://strategoxt.org/Stratego/PermissiveGrammars`.

KATS, L. C. L., KALLEBERG, K. T., AND VISSER, E. 2010. Domain-specific languages for composable editor plugins. In *Proceedings of The Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*. ENTCS Series, vol. 253. Elsevier.

KATS, L. C. L., SLOANE, A. M., AND VISSER, E. 2009. Decorated attribute grammars: Attribute evaluation meets strategic programming. In *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, O. de Moor and M. I. Schwartzbach, Eds. Lecture Notes in Computer Science Series, vol. 5501. Springer, 142–157.

KATS, L. C. L. AND VISSER, E. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, W. R. Cook, S. Clarke, and M. C. Rinard, Eds. ACM, Reno/Tahoe, Nevada, 444–463.

KATS, L. C. L., VISSER, E., AND WACHSMUTH, G. 2010. Pure and declarative syntax definition: paradise lost and regained. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, W. R. Cook, S. Clarke, and M. C. Rinard, Eds. ACM, Reno/Tahoe, Nevada, 918–932.

KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'07)*, M. Akşit and S. Matsuoka, Eds. LNCS Series, vol. 1241. Springer, 220–242.

KLUSENER, S. AND LÄMMEL, R. 2003. Deriving tolerant grammars from a base-line grammar. In *International Conference on Software Maintenance (ICSM '03)*. IEEE Computer Society, 179–189.

KRAHN, H., RUMPE, B., AND VÖLKEL, S. 2007. Efficient editor generation for compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*. technical report TR-38. University of Jyväskylä, 218–228.

KRAHN, H., RUMPE, B., AND VÖLKEL, S. 2008. MontiCore: Modular development of textual domain specific languages. In *TOOLS EUROPE 2008*, R. Paige and B. Meyer, Eds. Lecture Notes in Business Information Processing Series, vol. 11. Springer-Verlag, 297–315.

KUHN, T. AND THOMANN, O. 2006. Eclipse corner: Abstract syntax tree. `http://eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html`.

LAVIE, A. AND TOMITA, M. 1993. GLR*-an efficient noise skipping parsing algorithm for context free grammars. In *Third International Workshop on Parsing Technologies*. 123–134.

LÉVY, J.-P. 1971. Automatic correction of syntax errors in programming languages. Ph.D. thesis, Ithaca, NY, USA.

MAUNEY, J. AND FISCHER, C. 1988. Determining the extent of lookahead in syntactic error repair. *ACM Trans. Program. Lang. Syst. (TOPLAS) 10,* 3, 456–469.

MCKENZIE, B. J., YEATMAN, C., AND VERE, L. D. 1995. Error repair in shift-reduce parsers. *ACM Trans. Program. Lang. Syst. 17,* 4, 672–689.

MOONEN, L. 2001. Generating robust parsers using island grammars. In *Working Conference on Reverse Engineering (WCRE'01)*. IEEE, 13–22.

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

NILSSON-NYMAN, E., EKMAN, T., AND HEDIN, G. 2009. Practical scope recovery using bridge parsing. In *Software Language Engineering (SLE 2008)*, D. Gasevic, R. Lämmel, and E. V. Wyk, Eds. LNCS Series, vol. 5452. Springer, 95–113.

PAI, A. AND KIEBURTZ, R. 1980. Global Context Recovery: A New Strategy for Syntactic Error Recovery by Table-Drive Parsers. *ACM Trans. Program. Lang. Syst. (TOPLAS) 2,* 1, 18–41.

PARR, T. AND FISHER, K. 2011. Ll(*): the foundation of the antlr parser generator. In *PLDI*. 425–436.

PARR, T. AND QUONG, R. 1995. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience 25,* 7, 789–810.

PENNELLO, T. J. AND DEREMER, F. 1978. A forward move algorithm for LR error recovery. In *Principles of programming languages (POPL '78)*. ACM, 241–254.

REKERS, J. AND KOORN, W. 1991. Substring parsing for arbitrary context-free grammars. *SIGPLAN Not. 26,* 5, 59–66.

SALOMON, D. AND CORMACK, G. 1995. The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Tech. rep., TR 95/06, Dept. of Comp. Sci., University of Manitoba, Winnipeg, Canada.

SALOMON, D. J. AND CORMACK, G. V. 1989. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Not. 24,* 7, 170–178.

SAUNDERS, S., FIELDS, D. K., AND BELAYEV, E. 2006. *IntelliJ IDEA in Action*. Manning.

SCHMITZ, S. 2006. Modular syntax demands verification. Tech. Rep. I3S/RR-2006-32-FR, Laboratoire I3S, Université de Nice-Sophia Antipolis, France. oct.

SCHWERDFEGER, A. C. AND VAN WYK, E. R. 2009. Verifiable composition of deterministic grammars. *SIGPLAN Not. 44,* 6, 199–210.

SWIERSTRA, S. D. AND DUPONCHEEL, L. 1996. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming, Second International School*, J. Launchbury et al., Eds. LNCS Series, vol. 1129. Springer, 184–207.

SYNYTSKYY, N., CORDY, J., AND DEAN, T. 2003. Robust multilingual parsing using island grammars. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 266–278.

TAI, K.-C. 1978. Syntactic error correction in programming languages. *IEEE Trans. Software Eng. 4,* 5, 414–425.

TOMITA, M. 1988. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Vol. 14. Kluwer Academic Publishers.

VALKERING, R. 2007. Syntax error handling in scannerless generalized LR parsers. M.S. thesis, University of Amsterdam.

VAN DEN BRAND, M., SCHEERDER, J., VINJU, J. J., AND VISSER, E. 2002. Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, R. N. Horspool, Ed. Lecture Notes in Computer Science Series, vol. 2304. Springer, 143–158.

VAN DEN BRAND, M. G. J., BRUNTINK, M., ECONOMOPOULOS, G. R., DE JONG, H. A., KLINT, P., KOOIKER, T., VAN DER STORM, T., AND VINJU, J. J. 2007. Using the Meta-Environment for maintenance and renovation. In *The European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE Computer Society, 331–332.

VAN DEN BRAND, M. G. J., HEERING, J., KLINT, P., AND OLIVIER, P. A. 2002. Compiling language definitions: the ASF+SDF compiler. *ACM Trans. Program. Lang. Syst. 24,* 4, 334–368.

VAN DEURSEN, A. AND KUIPERS, T. 1999. Building documentation generators. In *IEEE International Conference on Software Maintenance (ICSM '99)*. IEEE Computer Society, 40.

VISSER, E. 1997a. A case study in optimizing parsing schemata by disambiguation filters. In *International Workshop on Parsing Technology (IWPT 1997)*. Massachusetts Institute of Technology, Boston, USA, 210–224.

VISSER, E. 1997b. Scannerless generalized-LR parsing. Tech. Rep. P9707, Programming Research Group, University of Amsterdam. July.

VISSER, E. 1997c. Syntax definition for language prototyping. Ph.D. thesis, University of Amsterdam.

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

WADDINGTON, D. AND YAO, B. 2007. High-fidelity C/C++ code transformation. *Sci. Comput. Program. 68,* 2, 64–78.

TUD-SERG-2012-021
ISSN 1872-5392