Natural and Flexible Error Recovery for Generated Parsers

Maartje de Jonge[1], Emma Nilsson-Nyman[2], Lennart C.L. Kats[1], and Eelco Visser[1]

[1] Dept. of Software Technology, Delft University of Technology, The Netherlands
m.dejonge@tudelft.nl, l.c.l.kats@tudelft.nl, visser@acm.org
[2] Dept. of Computer Science, Lund University, Sweden
emma@cs.lth.se

Abstract. Parser generators are an indispensable tool for rapid language development. However, they often fall short of the finesse of a hand-crafted parser, built with the language semantics in mind. One area where generated parsers have provided unsatisfactory results is that of error recovery. Good error recovery is both natural, giving recovery suggestions in line with the intention of the programmer; and flexible, allowing it to be adapted according to language insights and language changes. This paper describes a novel approach to error recovery, taking into account not only the context-free grammar, but also indentation usage. We base our approach on an extension of the SGLR parser that supports fine-grained error recovery rules and can be used to parse complex, composed languages. We take a divide-and-conquer approach to error recovery: using indentation, erroneous regions of code are identified. These regions constrain the search space for applying recovery rules, improving performance and ensuring recovery suggestions local to the error. As a last resort, erroneous regions can be discarded. Our approach also integrates bridge parsing to provide more accurate suggestions for indentation-sensitive language constructs such as scopes. We evaluate our approach by comparison with the JDT Java parser used in Eclipse.

1 Introduction

Domain-specific languages offer substantial gains in expressiveness and ease of use for a particular problem domain. To efficiently construct and use domain-specific languages, language development environments should be used, such as IMP [6], the Meta-Environment [27], MontiCore [14], openArchitectureWare [8], or Spoofax/IMP [13]. With these tools, languages are constructed using a grammar as the principal artifact. Using a parser generator, a grammar can be used to automatically generate a parser. When deployed, the parser constructs abstract syntax trees (ASTs) from programs, used to provide the user with syntactical and semantic editor services, such as an outline view and error marking.

Parser generators are an indispensable tool for rapid language development, allowing the language to be quickly changed according to new domain insights and needs. Yet general-purpose programming languages are often still constructed using handcrafted or partially handcrafted parsers. For example, the Java parser used in the popular Eclipse JDT Java editor, is based on a parser generated by JikesPG (now known as LPG) [5].

M. van den Brand, D. Gašević, J. Gray (Eds.): SLE 2009, LNCS 5969, pp. 204–223, 2010.
© Springer-Verlag Berlin Heidelberg 2010

However, the parser employs handwritten recovery rules as well as a number of large, customized Java components.

The reason often stated for not using a purely generated parser is that they fall short of the finesse of a handcrafted one, built with the language semantics in mind. A particular area where generated parsers have provided unsatisfactory results is that of error recovery, which is essential for parsing incomplete and syntactically incorrect programs, and thus indispensable for interactive editors. Problems with error recovery in generated parsers are the quality of the recovered program and the reported errors, and finding a good trade-off between recovery quality and performance.

Some parser generators allow custom recovery rules to improve error recovery quality [2,5,10,12]. Custom recovery rules allow a language engineer to inspect and improve an error recovery strategy. Compared to a handcrafted parser, a rule-based recovery specification is much easier to maintain, especially as languages are changed or reused to build new languages. Another way to improve error recovery is through grammar analysis, such as LPG's scope detection [5].

In previous work we introduced an approach to error recovery that derives properties from grammars to produce explicit, customizable recovery rules [12]. Using scannerless generalized-LR (SGLR) parsing, the approach supports languages with a complex lexical syntax, such as AspectJ [3], and language embeddings and extensions, such as the Stratego program transformation language with embedded Java fragments [30]. Using generalized parsing, SGLR can parse ambiguous grammars. By considering the different ambiguous meanings of a syntactically incorrect program, through inspection of an expanding search space for applying the set of recovery rules, the approach can provide recovery suggestions that local recovery methods cannot.

An open problem we identified with our approach is that some search space-based suggestions are too "creative" and not natural (i.e., as a programmer would suggest them) [12]; in some cases it is simply 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. Another open problem is that for tight clusters of errors, it is not always feasible to provide good suggestions in an acceptable time span.

In order to provide better, more natural suggestions, the present paper proposes an approach to identify the region in which a parse error is found. By restricting the search space for applying the recovery rules to this region, it becomes much less likely that the user is presented with "creative" suggestions that are nowhere near the original error. Using a smaller search space also helps performance. To further help performance, we add a form of "panic mode" [7]: if no solution of applying the recovery rules is found within an acceptable time span, the entire region can be skipped and marked as erroneous. This way, the parser can still continue, report other errors, and construct a partial AST.

We select erroneous regions based on indentation usage. Using indentation, programs typically form logical, nested regions of code. The approach of using layout information for partitioning files has been inspired by the technique of bridge parsing [20]. Bridge parsing is a supplementary technique to grammar-based error recovery. It uses structural information, such as typical use of indentation for bracket placement,

to improve recovery quality. To further improve the quality of recovery suggestions, we adapted the bridge parsing approach to be usable with an SGLR parser.

We have identified and focus our paper on two open issues with error recovery for generated parsers. The first is the quality of corrections, which is often lacking since a generic solution is not aware of the semantics or typical structure of a language. The second is that given high-quality recovery, a good balance with the performance of error recovery must be maintained. To address these issues, this paper provides the following contributions:

- The use of layout to select regions of code that enclose a syntax error. These can be analyzed in detail by a secondary strategy, or discarded if no recovery is found within an acceptable time span.
- The application of bridge parsing based on a context-free (tokenizer) grammar rather than a scanner, showing how bridge parsing can be integrated into a parser rather than used as a preprocessor, improving results.
- The use of grammars for automatic construction of a tokenizer grammar and the heuristic derivation of a bridge parser specification.

We begin this paper with background on error recovery and setting out a number of requirements for good error recovery. In Section 3, we show how regions around a syntax error can be selected and used for coarse-grained error recovery. Section 4 describes how these regions can be used to apply recovery production rules. We refine error recovery for scopes based on bridge parsing in Section 5. Finally, Section 6 evaluates our approach and compares different configurations, using the Eclipse JDT parser as a baseline.

2 Error Recovery

Parsers serve two purposes: determining the grammatical structure of an input program, and syntactically validating it. Given the grammatical structure, the parser constructs an abstract syntax tree (AST), used for semantic analysis in tools such as compilers or editors. While performing syntactic validation, a parser also reports any errors that exist in the input.

A good parser does not only report the first character or token that is not valid according to the grammar, but also provides the user with a more sophisticated diagnosis. It can for example report missing constructs (e.g., "} expected here"). An even better parser also supports error recovery: based on the analysis of an error, it can recover from an error and continue parsing the rest of a file. Recovery techniques can be divided into correcting error recovery, which tries to transform the input string into a syntactically correct one, and non-correcting error recovery, which tries to continue the analysis by skipping parts of the input [7].

Error recovery plays an important role in modern, interactive development environments (IDEs). IDEs parse a file as it is typed in, making incomplete programs and syntax errors the common case rather than the exceptional one. Using error recovery, a parser can still construct a partial abstract syntax tree, allowing the IDE to perform semantic analysis and provide the user with interactive feedback (e.g., error marking, content completion).

In their comparative study, Degano and Priami [7] set out a number of quality criteria for good error recovery strategies, on which we will elaborate here. We distinguish between aspects that impact users and developers of a language. Firstly, there are three main criteria with respect to the end user's experience:

– Constructing a good AST: The recovered program should be as close to the program as intended by the programmer as possible. Since the AST is used for syntactic and semantic editor services in the IDE (e.g., the outline and error markers), the quality of the reconstructed AST is of great importance for the user experience.
– Providing good feedback: The parser should provide the user with good suggestions of how to fix the program. Spurious error messages should be avoided; instead, a small number of natural suggestions should be reported.
– Delivering adequate performance: For interactive use, the error recovery mechanism must not incur an unacceptable overhead. As their last criterion, Degano and Priami have suggested to only take performance degradation into account only if greater than a fixed maximum value.

Important criteria for developers of a language or an IDE (plugin) are:

– Flexibility: The approach must be easily adaptable to language insights and language changes.
– Language independence: an error recovery algorithm should be independent of a particular language. It should be usable with any given grammar, without introducing a prohibitive amount of work.
– Transparency: it should be clear why a particular recovery is presented. The grammar engineer should have insight into how the recovery works for a given grammar.

3 Coarse-Grained Error Recovery

A parser that supports error recovery typically operates by consuming tokens (or characters) until an erroneous token is found. At the point of detection of an error, the recovery mechanism is activated. Simple, local approaches to error recovery will then attempt to make a modification to the input so that at least one more original symbol can be parsed [7]. For most cases, this works quite well. There are cases, however, particularly for complex languages, where these algorithms choose a poor repair that leads to further problems as the parser continues ("spurious errors").

Spurious errors are the result of one of the major problems in error recovery: the difference between the point of detection and the actual location of an error in the source program [7]. In contrast to local methods, global recovery methods examine the entire program and make a minimum of changes to repair all syntax errors [2,17]. While these give the "best" repair, they are not efficient.

An alternative approach to local or global recovery is to consider only the direct context of the error, by identifying the region of code in which the errors reside [16,18,21]. Using regions for error recovery has three main advantages. Firstly, they reduce the search space for a recover algorithm. Secondly, they constrain the recovery suggestions to a particular part of the file, avoiding suggestions that are spread out all over the file. And thirdly, they can be used as a secondary recovery strategy [7], i.e. erroneous regions can be discarded entirely if a detailed analysis of the region does not provide a better recovery solution.

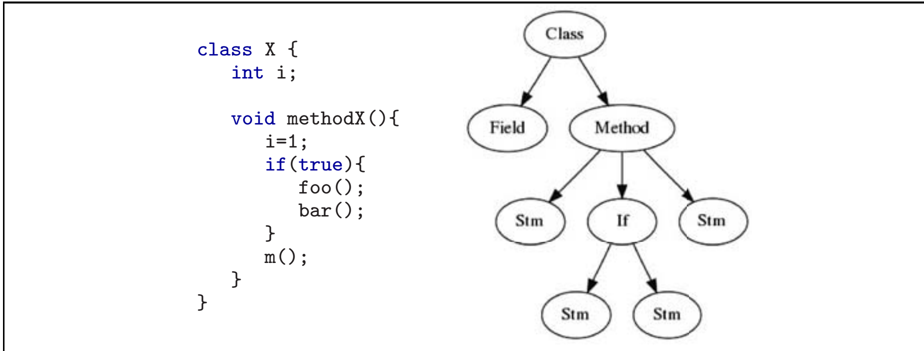


Fig. 1. Indentation closely follows the hierarchical structure of a program

3.1 Nested Structures as Regions

Code constructs such as “while” statements and method bodies form good regions for regional error recovery. They form free standing blocks, in the sense that they can be omitted without influencing the interpretation of other blocks. Erroneous free standing blocks can simply be skipped, providing a coarse recovery that allows the parser to continue. A typical technique to select such regions is to look for certain marker tokens in the context of an error, such as the fiducial tokens of Pai and Kieburtz [21]. These tokens depend on the language used. For example, for Java, keywords such as `class` and `while` could be used. We will take a more language-independent approach in this paper.

The method presented in this section is based on the use of indentation to detect code constructs. Indentation typically follows the logical nesting structure of a program, as illustrated in Figure 1. The relation between constructs can be deduced from the layout. An indentation shift to the right indicates a parent-child relation; the same indentation indicates a sibling relation.

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, but should keep in mind the possibility of inconsistent indentation usage.

Proper recognition of nesting structures prevents bad recoveries, obtained by merging structures that do not belong together. Figure 2 illustrates this idea with the example of a method that is missing a closing brace. The parser tries to parse the method header of the second method as a statement, which leads to a failure at the open brace in the method header. Indentation suggest that both methods should be considered as separate constructs. An indentation-based region selector will detect the erroneous if-block; which leads to the recovery presented in the middle part of the figure. An inferior recovery would be obtained by removing tokens surrounding the error detection point. The example at the right shows the result, merging the erroneous method with the correct method.

3.2 Indentation-Based Region Selection

We follow an iterative process to select an appropriate region that encloses a syntax error. Each iteration, a different *candidate region* is considered. This candidate is then

```
class X {                      class X {                      class X {

   int methodX(){                 int methodX(){                 int methodX(){
      if(true){                                                     if(true){
         foo(); //}                                                    foo(); //}
      return 5;                      return 5;                      return 5;
   }                              }                              }

   void methodY(){                void methodY(){
      int i=5;                       int i=5;                       int i=5;
      bar(i);                        bar(i);                        bar(i);
   }                              }                              }
}                              }                              }
```

Fig. 2. Erroneous code (left), discarded erroneous region (middle), and merged constructs (right)

either validated or rejected; in case of a rejected candidate, another candidate is considered. We show example scenarios in Figure 3.

Figure 3(a) shows a syntax error and the point of detection, indicated by a triangle (left figure). A candidate region can be selected based on the alignment of the `void` keyword and the closing bracket (middle figure). The candidate is then successfully validated by discarding the region, and attempting to parse the remainder of the file (right figure). After validation, the parser can be reset to its previous state (indicated by the circle, which represents a choice point for the parser). A detailed analysis of the region may be used to attempt to repair the erroneous region, as we will see in the following sections.

Figure 3(b) illustrates a rejected candidate region. Based on the point of detection, an obvious candidate region may be the `m2` method (middle figure). However, an attempt to parse the construct that follows it leads to a premature parse failure; the region is rejected. Figure 3(c) revisits the example. Another candidate region is selected, this time one preceding the point of detection. This region is successfully validated.

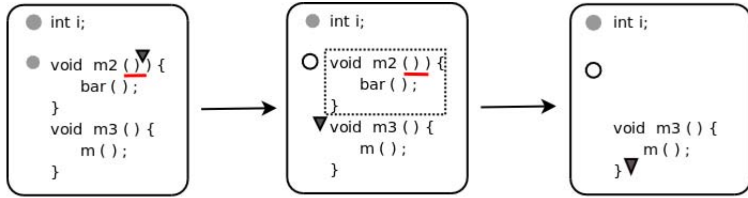
The region validation criteria should balance the risk of selecting the wrong candidate, which may lead to spurious errors, and the risk of rejecting a correct candidate region. The latter typically occurs in the context of multiple errors, in which a new, unrelated error causes the parser to fail again. Both cases lead to large regions, which should be avoided. We currently consider a region valid if the two lines of code succeeding it parse correctly, which has shown good practical results.

Selection Schema. The candidate regions are explored in an ordered fashion, with the aim to find the smallest fragment enclosing the error first.

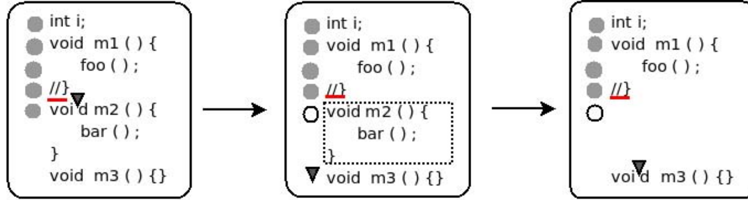
Current structure. The first candidate region is the construct starting from the error detection location. The region is recognized by a forward skip until the end of the construct is found. The construct ends with the last child (more indentation), including the closing bracket after the last child (same indentation). In Figure 4, the parser fails after reading the mistakenly inserted second brace. Discarding the entire `while` statement resolves the error.

```
while(true)){
    foo();
}
```

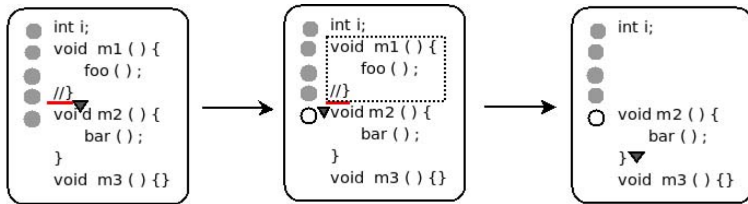
Fig. 4. Extra)



(a) A candidate region is validated and successfully discarded.



(b) A candidate region is rejected.



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

Fig. 3. Recovery by discarding of regions

Previous structure. The second candidate is the structure preceding the error detection location. The region is detected by a backwards skip, using the indentation information stored in the choice points. Typical problems that are solved by discarding the previous structure are uncompleted lines and scope errors caused by a missing closing brace. The error in Figure 5 is detected after the `bar();` statement, while the preceding line caused the error.

```
void methodX() {
    foo(
    bar();
}
```

Fig. 5. Missing);

Siblings. Regions that are mutually dependent should be discarded as a whole. A typical example is provided in Figure 6. The unclosed “then” clause cannot be discarded, because the “else” clause cannot exist in isolation. The “sibling-procedure” deals with this situation. The procedure starts with the current structure as discarded region. Then it successively includes the prior sibling and the next sibling, until a valid erroneous region is found or all siblings have been considered.

```
if(true){
    foo();
else
    bar();
```

Fig. 6. Missing }

Parent. The next region to consider is the parent structure, identified through a forward and backward search for a decrease in indentation. Identifying the parent structure can be useful when a child that is missing or erroneous. Parent child dependencies are rarely seen in common programming languages, but they can

```
Person
    Name: John
    email: ????
```

Fig. 7. Illegal email property

occur in DSLs. The example in Figure 7 shows a simple person data language with an error in the required field `email`. Apart from solving errors in parent-child dependencies, the parent selection scheme adds some robustness with respect to inconsistent indentation.

While the selection schemata have been designed to be generally applicable, the success of our approach depends on assumptions of indentation conventions and language characteristics. Conventions for widely used programming languages seem to meet the assumption that the indentation follows the logical nesting structure of the program. A more problematic issue is the (mis)use of indentation by programmers. Inconsistent indentation usage decreases the quality of the results, although some robustness for small deviations can be expected. The second assumption we make is that programs have free standing blocks, i.e. that discarding a region still yields a valid program. Again, conventional programming languages seem to meet this requirement. However, some (declarative) languages use constructs that cannot be discarded because they are syntactically obligatory. Such languages can lead to large regions.

3.3 Implementation Considerations

We implemented the region selection method in SGLR, in order to use it in collaboration with recovery rules [12]. The selection method does not depend on specific features of generalized parsing and can be implemented in other LR parsers as well.

Layout conventions for braces. Varying conventions for closing and opening braces are used. They can be omitted in some situations, besides the position of the opening brace can vary. The figure illustrates the problem with a concrete example. Two code fragments with the same indentation characteristics, have a different decomposition in regions.

```
bar();
while(true)
    foo();
doX();

while(true)
{
    foo();
}
```

Fig. 8. Different indentation styles

The need to cover all different notations in a language independent way, has greatly increased the complexity of the implementation. A simple solution would be provided by an explicit recognition of those tokens. This would make the algorithm more precise and the implementation straight forward. However, this will introduce a language dependency. We have chosen to stick with our language independent approach, and deal with the support for various brace-conventions in the code. In case of doubt, we assume the notation including the braces on separate lines. The main disadvantage is that sometimes one or two correct lines are included in the selected region.

Whitespace parse. A simple discarding of erroneous regions will offer a recovery and allows the parser to continue the analysis, however the information about line and column numbers may be lost. This will cause problems in the scenario of interactive editor support. A simple solution is offered by whitespace parsing. All symbols, except newlines and tabs are parsed as whitespace. Information about skipped regions can be used to generate error messages.

```
class X{
    ....
    void methX(){...}
    void methY(){...}
    void methZ(
```

Fig. 9. An uncompleted class

Parse tree completion. We maintain only a limited number of choice points to backtrack to, to ensure that there is only a negligible overhead when parsing (parts of) files without errors. This limitation means that in some cases the layout-based region selection cannot provide a candidate region. For example, the class construct in Figure 9 is unfinished, and can only be discarded as a whole. Complementary to the region selection schemata, we implemented a technique that completes the parse tree for an unfinished code fragment. In this way, at least the already recognized part of the code can be reported to the programmer. A program prefix missing only a few closing tokens at the end, can be completed to a valid program by inspection of the parse table. Although the missing next token is not known, a list with possible tokens can be retrieved from the parse table. The completion method creates separate stack branches for each possible "next state", deduced from the list of possible tokens. After a number of parse steps using this branching mechanism, an accepting state will be found. Generalized parsers like SGLR and GLR provide native support for branching. The method works efficient if only a few general branching steps are required, which corresponds to a small number of missing tokens. Therefore, we apply the method on the location of the last big reduction, the closing brace of `methY`, in the example.

4 Fine-Grained Error Recovery

We can improve upon the coarse-grained recovery approach by using it in conjunction with a more fine-grained, correcting error recovery method. In this section we outline how the error recovery productions of [12] can be used to perform fine-grained error recovery inside erroneous regions.

Error recovery productions allow for a high-level, grammar-oriented way of customizing a recovery strategy [2,10]. Because the language engineer must design them a priori, they have sometimes been criticized for being language-dependent [7]. In [12] we introduced a way to derive recovery rules from a grammar, and added general rules that can simply skip over erroneous code fragments.

Following [12], recovery productions are written just as any other production, annotated with the `{recover}` annotation. We use the flexible SDF syntax definition formalism [29] for the specification of grammars and their recovery rules. As an example of an SDF production, consider the following Java production:

```
"{" BlockStm* "}" -> Block {cons("Block")}
```

This rule specifies that a `{` literal, followed by a list of `BlockStm` symbols and a closing `}` literal, can be parsed as a `Block`. The `{cons}` annotation specifies the name used for the node in the abstract syntax tree. Based on this rule (and taking global properties of the grammar into consideration, as outlined in [12]), the following recovery production rule can be derived:

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

This production specifies that a possible recovery is to parse the empty string (hence the empty left-hand side) instead of the closing `}` literal. Annotated `{recover}`, this *insertion recovery rule* is only used when recovery is required.

In addition to insertion recovery rules, [12] also specifies lexical "catch-all" production rules to discard unparsable substrings. Together, these rules could parse any string, distinguishing only "words" and "separators":

```
[A-Za-z0-9\_]*          -> WATERWORD {recover}
~[A-Za-z0-9\_\ \t\r\n] -> WATERSEP  {recover}
```

Each application of recovery rule incurs a cost of 1. A minimal-cost solution may be the best possible match with the programmer's intention. However, considering all candidate recoveries for a complete file results in a search space that is too large to inspect within reasonable time. In [12] we applied an unbounded, expanding search space to discover a recovery solution with a minimum cost. For most cases, this approach is effective, but for a number of pathological cases, the unbounded search leads to unacceptable recovery times, or to far-fetched, non-local recovery suggestions.

By restricting the search space to a selected region of code, recovery performance and locality can be improved. Smaller regions (fewer than four lines) are reparsed, applying a bounded number (three in the current implementation) of recover productions. For larger regions, we assume that the error can be corrected in the three lines nearest to the parse failure, which seems to be the case in most practical examples. If the application of recovery rules does not lead to a successful repair, the entire region can be discarded using the whitespace parse approach discussed in the previous section.

5 Bridge Parsing

```
1 class C {
2   void m() {
3     int y;
4   int x;
5 }
```

Fig. 10. Missing }

One of the most common errors made by programmers is omitting closing brackets of scopes, since scopes are recursive structures need to be properly balanced [5]. A parser can recover in these cases by inserting the missing braces. Unfortunately, there are often many possible locations where a missing brace can be inserted. Consider for example the Java fragment of Figure 10. This fragment might be recovered by inserting a closing brace at the start of line 2, 3, or 4. However, the use of indentation suggests the best choice may be just before the `int x;` declaration. Bridge parsing [20] provides an algorithm to improve error recovery based on indentation. Provided with knowledge of typical usage in Java programs, it can correctly recover cases such as the example above. It can be configured to work for any given language, and works independently of a particular parser technology.

Inspired by island grammars [28,19], a bridge parser employs a scanner that only recognizes tokens that make up scoping structures ("islands") and important tokens for determining how those islands should be connected ("reefs"). All other tokens ("water") are skipped. Given a list with these kind of tokens, the bridge parser constructs a bridge model, which captures the scopes in the input. A scope in this context corresponds to two islands connected with a bridge. Two islands will only match if a predefined set of conditions is fulfilled. Missing bridges in the bridge model reveal broken scopes. They can be repaired by locating an appropriate "construction site" for inserting a new, artificial island, matching the island in need of recovery. A new bridge can then be constructed. An algorithm for incrementally constructing multiple bridges is given in [20].

```
module Java-SQL-Tokenizer

context-free start-symbols
  Class Stm Expr ...

context-free syntax %% token list definitions for all start symbols
  ClassToken Class -> Class {cons("Cons")}
                   -> Class {cons("Nil")}

context-free syntax %% tokens and the {cons} name of their production
  EnumDecHeadToken -> ClassToken {cons("EnumDecHead")}
  SQLId            -> ClassToken {cons("Id")}

lexical syntax %% lexical token definitions
  "enum"     -> EnumDecHeadToken
  [A-Za-z]+ -> SQLId
```

Fig. 11. A (partial) generalized tokenizer definition for the Java-SQL language

5.1 Scannerless Bridge Parsing

Composed languages and languages with a complex lexical structure (such as AspectJ) cannot or can only with great difficulty be parsed using a separate scanner [3]. For example, the scanner for the Java language recognizes `enum` as a keyword. This means that it can never be parsed as an identifier. When Java is extended or composed with another language, this restriction also applies for the combined language. Using the same scanner, a composition of Java and SQL cannot parse programs where `enum` is an SQL identifier. Using scannerless parsing [24], these issues can be elegantly addressed [3].

Since bridge parsing as presented in [20] is based on the notion of a scanner, it cannot support languages that depend on scannerless parsing or parsing with a context-sensitive scanner. Still, bridge parsing only depends on a small set of tokens, such as brackets and keywords, not on a full scanner definition. So why can we not just construct a scanner for those literals in the grammar? The problem is that each sequence of characters, there can potentially be many different lexical and literal interpretations. Again consider `enum`, which is keyword in Java, but may also be an identifier in the composed Java-SQL or Stratego-Java languages.

To overcome the difficulties of a scanner-based approach, we introduce the notion of a *generalized tokenizer*. This tokenizer constructs all possible token interpretations, forming an ambiguous token stream. We implement this tokenizer based on the grammar of a language. For example, given the Java-SQL definition, we mechanically strip all context-free productions and retain only definitions for literals and lexical symbols. For each sort in the grammar, we then generate a start symbol that parses the different lexicals and literals reachable from that state. A (partial) tokenizer grammar for Java-SQL is illustrated in Figure 11. Using the `Class` start symbol, this grammar constructs the following token stream (a list of `Cons` and `Nil` nodes) for the string `enum Color{}`:

```
[ amb([EnumDecHead("enum"), Id("enum"])
, LAYOUT(" "), Id("Color")
, amb([EnumBody("{"), ClassBody("{"), Block("{"), ...])
, amb([EnumBody("}"), ClassBody("}"), Block("}"), ...])]
```

where ambiguities in the token stream are indicated with an `amb` term. For composed languages, these token streams quickly grow more complex as the number of different token interpretations increases.

```
grammar Layout;

abstract island LayoutStart;
abstract island LayoutEnd;
abstract reef Layout;

bridge from LayoutStart to LayoutEnd;

attr Layout LayoutStart.indent =
    [first left Layout];
attr Layout LayoutEnd.indent =
    [first left Layout];
java-attr int Layout.pos =
    ...embedded java code...

recover LayoutStart find [a:Layout]
where (a.pos <= this.indent.pos)
insert bridge-end before a;
...
```

Fig. 12. A generic bridge grammar

```
grammar SimpleJava;

import Layout;

island LBrace : LayoutStartIsland = "{"
  for-sglr("EnumBody"|"ClassBody"|...);
island RBrace : LayoutEndIsland = "}"
  for-sglr("EnumBody"|"ClassBody"|...);
reef Indent : LayoutReef =
  NEWLINE|(WS|TAB)+

bridge from LBrace to RBrace;
...
```

Fig. 13. A bridge grammar for Java

We simplify the token stream by considering only those tokens that are of interest to the bridge parser, and by flattening the ambiguities to create multiple, possible interpretations that have no deep ambiguities. After that, the bridge parser can assign different island classes, reef classes, and water to the tokens:

```
[KeywordReef("enum"), LayoutReef(" "), Water("Color"), LBrace, RBrace]
```

In this list, the `LBrace` and `RBrace` classes encompass all interpretations for the `{` literal. In the remainder of this section we will discuss how the binding between these tokens and the bridge parsing island classes is specified.

5.2 The Bridge Parsing Specification

A bridge parser is generated from a bridge parser specification (or bridge grammar) [20]. It defines all islands, reefs, and rules for matching and recovering islands. Attributes can be added to islands and reefs to help with matching in rule expressions. These specifications are composable and can be extended in several steps. Generic behavior such as "closest match recovery" or "layout-based recovery" is defined in a generic specification that can be reused and redefined by other grammars. Figure 12 lists parts of a generic bridge grammar that specifies layout-based recovery. The grammar specifies abstract `LayoutStart` and `LayoutEnd` islands that must be connected by a bridge. It also adds `indent` and `pos` attributes that can be used to do layout-based matching, as described in [20]. The `recover` rule uses these attributes to construct an artificial `LayoutEnd` island to repair bridges from `LayoutStart` islands.

We extended the bridge parser specification language for the purpose of integrating it with SGLR, adding a new `for-sglr` clause to capture the different possible interpretations for one token or character. For example, braces in Java have several possible interpretations according to the SGLR tokenizer grammar. Figure 13 shows how these can be captured using the `for-sglr` clause. The arguments of the clause correspond to the node types in the generalized token stream, seen in the previous subsection. The grammar in Figure 13 imports the generic layout grammar, and provides concrete implementations for the abstract islands and reef defined in that grammar.

Using the bridge grammar, the bridge parser derives a bridge model from the token stream (illustrated in the previous subsection). Because of ambiguous interpretations, there may be multiple possible token streams. By assigning island and reef classes that may encompass multiple node types, some of these can be eliminated. In case more than one alternative remains, we currently pick the interpretation with the fewest number of broken bridges.

5.3 Deriving Bridge Parser Specifications

SDF grammars are fully declarative, and do not allow semantic actions or callbacks to native code. This property makes SDF grammars well-suited for analysis. In previous work we applied automated analysis of SDF grammars to derive recovery productions [12]. To help language engineers efficiently employ bridge parsing with an SDF grammar, we do the same for a bridge parser specification.

Island definitions are central to the bridge parser specification. Typical candidates for island definitions are scoping constructs, such as `{ }` in curly brace programming languages. Scoping constructs are generally nestable structures, which means that their grammar productions are recursive. For example, scopes in Java are defined as follows:

```
"{" BlockStm* "}" -> Block {cons("Block")}
Block             -> Stm
Stm               -> BlockStm
```

We consider a production $\overline{p}\ \alpha\ \overline{q}\ \beta\ \overline{r}$ `->` S to define a scoping construct for opening literal α and closing literal β to form a scoping construct if the following conditions are satisfied:

- the production is recursive;
- literals α and β are not identical;
- literals α and β appear only in productions of the form $\overline{p}\ \alpha\ \overline{q}\ \beta\ \overline{r}$ `->` S where α and β are not part of patterns $\overline{p}$, $\overline{q}$, or $\overline{r}$;
- the literals do not appear in a production with the `{bracket}` annotation.

The second condition excludes literals such as ` in shell scripts, since they are typically not nestable. The third condition ensures that we only select literals that appear in a balanced fashion throughout the grammar, ensuring that the bridge parser does not try to introduce opening or closing literals for unbalanced literals. The final condition ensures that we do not select constructs that define parentheses. Unlike scopes, parentheses (marked with the `{bracket}` annotation in SDF) have no direct semantic meaning other than modifying the priority of other operators. Because of this property, parentheses are typically not indented the same way as scopes.

For each opening and closing literal, we generate island definitions and bridge rules similar to those in Figure 13. To complete the bridge parsing specification, we also generate reef rules for all reserved words in the language. Reserved words in SDF are defined using a `{reject}` annotation that indicates they cannot be used as an identifier. For composed languages where these words may not be globally reserved; the bridge parser then considers both interpretations.

Automatically deriving recovery rules helps maintain a valid, up-to-date recovery rule set as languages evolve and are extended or embedded into other languages. Grammar engineers may also customize the derived specification to handle further cases and to introduce different indentation styles.

5.4 Combining Fine-Grained Error Recovery and Bridge Parsing

Bridge parsing excels at correcting scope errors, while fine-grained recovery is the designated approach to recover more localized errors like a missing semicolon. In case the erroneous region contains both types of errors, a combination of both techniques is required to find an optimal recovery. To do this, we extend the fine-grained recovery process to handle suggestions provided by the bridge parser. Each suggestion gives rise to an extra stack branch that is explored in parallel with the other recovery branches. In this way, the bridge parser suggestions are taken into account, but only applied if they lead to a least-cost recovery.

6 Evaluation

We implemented our approach based on JSGLR, a Java implementation of SGLR [11], extending it with support for coarse-grained recovery and refining the support for recovery rules of [12]. The bridge parser implementation, also written in Java, is based on the implementation of [20], and adapted to support ambiguous token streams and recovery of regions rather than complete files. We evaluate our error recovery according to the criteria set out in Section 2. We study the quantitative criteria through evaluation of the parser using a set of test files written in Java. Java was selected because of its ubiquity in software development and in modern IDEs such as the Eclipse JDT, offering a challenging comparison. We will also argue that our approach satisfies the qualitative criteria of providing good feedback, flexibility, language independence, and transparency.

Construction of the Test Set. We evaluate using an extended version of the test set used in [20]. The base test set was originally constructed for testing structural recovery of Java code, and focused on syntax errors such as missing braces pr parenthesis. The extended test set includes tests for both structural and non-structural errors, and is available from [1]. We intentionally included some cases with inconsistent use of indentation, since those are difficult to handle for the bridge parser, i.e. basic layout recovery depends on good indentation information. The test set contains three major categories of tests; *missing* – structural tokens for grouping, closure or division are missing (65 tests), *extra* – there are too many structural tokens (8 tests), and *other* – remaining errors like erroneous statement, or missing comment end (3 tests). Together, these total to a set of 76 test cases.

Setting up the Experiment. All tests are run in an automated fashion, comparing the pretty-printed ASTs for the erroneous files to the pretty-printed ASTs for the original, correct files they were derived from. We use two methods for comparison: First, we do a manual inspection, following the quality criteria of Pennello and DeRemer [22]. Following these criteria, an *excellent* recovery is one that is exactly the same as the intended program, a *good* recovery is close to this result, and a *poor* recovery introduces spurious errors. Since this is arguably a subjective comparison, we also count the number of lines of code that changed in the recovered result (the "diff"). The advantage of this approach is that it is objective, and assigns a larger penalty to recoveries where a larger area of the text does not correspond or is placed in an incorrect scope. The resulting figures are also arguably easier to interpret than comparing tree distances.

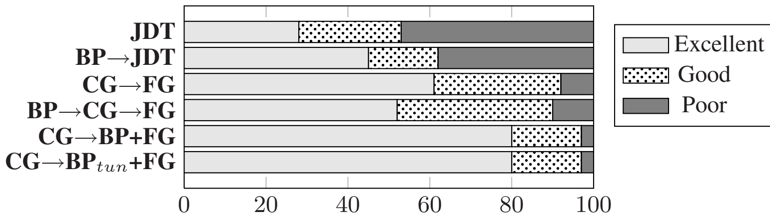


Fig. 14. Quality of Recovery The x axis shows percentage of tests. Each bar shows the percentage of recoveries which where excellent, non-excellent or failed for each approach. **CG** - Coarse Grained, **FG** - Fine Grained, **BP** - Bridge Parsing, **JDT** - Java Developer Toolkit

Various Approaches. We compare the integrated recovery approach presented in this paper to different configurations of the individual techniques and to the parser used by Eclipse's JDT. We apply the test set with the following parser configurations; the JDT parser; the JDT parser with a bridge parser (BP) preprocessor, as suggested in [20] (BP→JDT); our approach without using bridge parsing (Course Grained (CG) → Fine Grained (FG)); our approach with the bridge parser as a preprocessor (BP→CG→FG); the fully integrated approach (CG→BP+FG); and finally the same approach with a tuned bridge parser specification (CG→BP$_{tun}$+FG).

Except for the final configuration, the three SGLR-based parsers use fully automatically derived recovery specifications. In contrast, specialized, handwritten recovery rules and classes related to recovery are used for the JDT parser. For the tests we used the JDT parser with statement-level recovery enabled, following [15]. In some of the test cases, particularly those with multiple errors, the parser was unable to recover the entire body of a method. For content completion, Eclipse uses a secondary parser that can analyze these method bodies. Because of its specialized nature, we have not included it in our experiments. Both the bridge parser used as a preprocessor and the one integrated into SGLR use the same recovery rules and node types.

Results. The diff values acquired for the various approaches are shown in Figure 15 and the same values with a quality distinction are shown in Figure 14. Considering both diagrams, we see that the SGLR parser, parsing using different steps and granularity, consistently outperforms the JDT parser. When fully integrated with the bridge parser, the best results are obtained.

Using the bridge parser in a preprocessor setting was shown to be effective for a number of different parsers in [20]. For the JDT parser, we can see that the results are improved using the bridge parser as a preprocessor. When combined with SGLR, however, we see that the preprocessing approach does not work well. We speculate that these results arise because the bridge parser can only insert braces to recover scopes, never remove them, since it does not have enough knowledge of the complete language. However, when it is actually integrated into SGLR, the bridge parser's suggestions lead to the best results. Manual inspection of the non-excellent results for each approach reveals more in-depth knowledge:

– **JDT** (49 *missing*, 4 *extra*, 2 *other*): A majority of the cases are in the missing category. The most common recovery is for JDT to skip the whole content of a

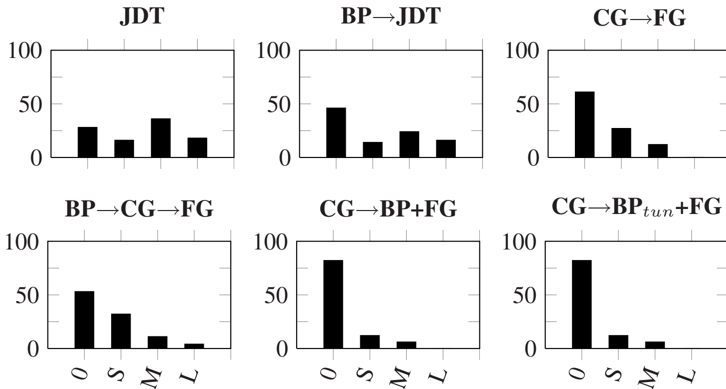


Fig. 15. Diffs for Various Approaches The y axis shows percentage of tests and the y axis shows categories of number of diff lines – No diff (0), **S**mall diff ($1-10$), **M**edium diff ($11-20$) and **L**arge diff (>20). **CG** - Coarse Grained, **FG** - Fine Grained, **BP** - Bridge Parsing, **JDT** - Java Developer Toolkit.

block if there is an error. This explains why the diff values for JDT tend to be higher.

- **BP→JDT** (34 *missing*, 5 *extra*, 2 *other*): The bridge parser helps to reduce the number of cases in the *missing* category. However, it fails to improve cases which are of out of scope for the bridge parser, for example, missing semicolons or extra structural tokens.
- **CG→FG** (24 *missing*, 4 *extra*, 3 *other*): Also has a majority of cases in the *missing* category, particularly missing braces (both start and end).
- **BP→CG→FG** (27 *missing*, 5 *extra*, 3 *other*): This combination does not work out very well. The bridge parser manages to slightly improve cases in the *missing* category, but makes things worse in some of the other cases.
- **CG→BP+FG** (8 *missing*, 3 *extra*, 3 *other*): This is the best option. The robustness of SGLR evens out the rough edges of the bridge parser, using it more like a consultant and discarding bad advice. In practice, this means that tests in the *missing* category see a huge improvement. There is a slight improvement in the *extra* category, while the others categories stay the same.
- **CG→BP$_{tun}$+FG** (8 *missing*, 3 *extra*, 3 *other*): There are no visible changes using this tuned bridge parser. The partial recoveries performed by the bridge parser show a small improvement, i.e., if there is more than one error one of the two gets a better recovery but the end result is the same.

Our experiments have not indicated that using a "tuned" bridge parser specification helps results. Tuning in this context can be quite tricky due to the various uses of, for example, a keyword. Turning all keywords into reefs will potentially ruin recoveries. For example, considering a `for` loop missing a left parenthesis, with too many keywords defined as reefs, the bridge parser might insert a left parenthesis too early. If both `int` and `for` are keywords and there is a rule stating that a recovery shall not pass a reef, then the left parenthesis will be inserted before the `int` and not before the `for`. This indicates that keywords must be chosen with care and the set should probably be quite small.

Concerning the selection of error fragments by the coarse grained recovery approach, manual inspection revealed that the right segment is identified for most of the test cases – both in position and size. Generally, however, there can be cases where selecting the right error fragment is difficult, which can result in a poor recovery.

While we have not performed an in-depth performance study, we set a maximum of 1 second for completing each test run, to allow for good responsiveness when used in an interactive environment (where the parser runs in a background thread). All tests complete within this time limit. The pathological cases previously identified for the Stratego-Java language [30] used to take much longer than this time limit [12], but with the addition of the coarse-grained recovery mechanism now also complete within this time limit. By constraining the expensive fine-grained recovery rules to a small region, setting an upper bound for the number of cases to consider per region, and introducing the possibility to fall back to discarding an entire region, the performance issues seem to have been resolved.

The Impact of Indentation Usage. Since our approach depends on layout, one issue to address is robustness in case of inconsistent indentation. The tab size used greatly affects the indentation levels in a file. The tab size might change, and tabs and white spaces are often mixed. IDEs such as Eclipse can automatically insert spaces for tabs and maintain indentation settings per project, avoiding some of these problems. Possible strategies for more robustness are: 1) Using averages to determine the indentation shift, and in that way handle different indentation shifts within a file or project. 2) Rounding off exact indentation offsets to their approximate indentation level. Some times the exact indentation position has an off-by-one position, e.g., there might be three spaces when the indentation shift is four. This situation can cause indentation matching problems. The two strategies can be combined, normalizing the indentation levels to match the indentation shift in the rest of the file or fragment.

Qualitative Evaluation. When working with interactive parsing the most important thing is to provide a good service to the user. We integrated our approach into Eclipse based on the Spoofax/IMP editor environment [13]. Based on the recovery productions, the editor gives accurate feedback. Following [12], every class of recovery rule is associated with a particular message (e.g., "} expected").

For the language engineer, flexibility, language independence, and transparency of the approach are important qualitative criteria. Our approach is highly flexible as it allows for customization of the high-level bridge parsing and recovery rules specifications. Yet, it maintains language independence by deriving defaults for these specifications, ensuring it is in line with the expectations of parser generators. By deriving explicit, customizable specifications, the approach is also highly transparent.

7 Related Work

In previous work, we introduced error recovery for SGLR, based on parse error productions that can be automatically derived from a grammar [12], and described its integration in Spoofax/IMP [13]. The present paper refines this work, constraining the application of recovery rules to coarse-grained regions and adding support for bridge parsing.

Bridge parsing was previously applied purely as a preprocessor for other parsers [20], ensuring that it repaired scope-related errors before other errors are recovered. We found that this approach was ineffective in combination with the production-based recovery approach of SGLR (see Section 6). Furthermore, using a scanner, the bridge parser was unable to cope with the lexical complexity of composed languages. The present work introduces a scannerless tokenizer and fully integrates the bridge parser into SGLR to address these issues.

Using SGLR parsing, our approach can be used to parse languages with a complex lexical syntax and composed languages. In related work, only a study by Valkering [26], based on substring parsing [23], offered a partial approach to error recovery with SGLR parsing. Composed languages are also supported by parsing expression grammars (PEGs) [9]. PEGs lack the disambiguation facilities [29] 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 [25] 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.

There are several different forms of error recovery techniques for LR parsing [7]. These techniques can be divided in *correcting* and *non-correcting* techniques. The most common non-correcting technique is *panic mode*. On detection of an error, the input is discarded until a synchronization token is reached. Then, states are popped from the stack until the state at the top enables the resumption of the parsing process. Our coarse-grained recovery algorithm can be used in a similar fashion, but selects discardable regions discarded 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. Successful recovery mechanisms often combine more than one technique [7]. For example, panic mode is often used as a fall back method if the correction attempts fail.

Burke and Fisher [4] present a 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. We improve on their work by taking indentation into account, for the scope recovery using an adapted version of bridge parsing [20], as well as for the coarse recover technique. In addition, by starting with region selection, the performance as well as the quality of the fine-grained technique [12], is improved.

Regional error recovery methods [16,18,21] 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 [21]), which are language-dependent. In our approach, we assign regions based on layout instead.

The LALR Parser Generator (LPG) [5] is incorporated into IMP [6] 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. Like our

approach, LPG detects scopes in grammars. However, unlike our approach, it does not take indentation into account for scope recovery.

8 Conclusion

Source code has a hierarchical structure that generally is reflected in the usage of layout and indentation. We have shown that this property can be exploited to confine syntax errors to small regions of code, and to provide better, more natural error recovery suggestions. Our approach to error recovery provides language independence by automatically deriving language-specific recovery behavior from grammars. Yet by allowing customization of the recovery behavior, using fine-grained recovery rules and a high-level bridge parsing specification, the approach maintains flexibility.

Acknowledgements. This research was supported by NWO/JACQUARD projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*. 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

1. The permissive grammars project, `http://strategoxt.org/Stratego/PermissiveGrammars`
2. Aho, A., Peterson, T.G.: A minimum distance error-correcting parser for context-free languages. SIAM Journal on Computing 1, 305 (1972)
3. Bravenboer, M., Tanter, E., Visser, E.: Declarative, formal, and extensible syntax definition for AspectJ. A case for scannerless generalized-LR parsing. In: Cook, W.R. (ed.) OOPSLA 2006, pp. 209–228. ACM Press, New York (2006)
4. Burke, M.G., Fisher, G.A.: A practical method for LR and LL syntactic error diagnosis and recovery. ACM Trans. Program. Lang. Syst. 9(2), 164–197 (1987)
5. Charles, P.: A practical method for constructing efficient LALR(K) parsers with automatic error recovery. PhD thesis, New York University (1991)
6. Charles, P., Fuhrer, R.M., Sutton Jr., S.M.: IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) Automated Software Engineering (ASE 2007), pp. 485–488. ACM, New York (2007)
7. Degano, P., Priami, C.: Comparison of syntactic error handling in LR parsers. Software – Practice and Experience 25(6), 657–679 (1995)
8. Efftinge, S., et al.: openArchitectureWare User Guide. Version 4.3 (2008), `http://openarchitectureware.org/pub/documentation/`
9. Ford, B.: Packrat parsing: Simple, powerful, lazy, linear time. In: International Conference on Functional Programming (ICFP 2002). SIGPLAN Notices, vol. 37, pp. 36–47. ACM, New York (2002)
10. Graham, S.L., Haley, C.B., Joy, W.N.: Practical LR error recovery. In: SIGPLAN 1979: Symposium on Compiler Construction, pp. 168–175. ACM, New York (1979)
11. Kalleberg, K.T., et al.: JSGLR, `http://www.spoofax.org/`

12. Kats, L.C.L., de Jonge, M., Nilsson-Nyman, E., Visser, E.: Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In: Leavens, G.T. (ed.) Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009). ACM SIGPLAN Notices, vol. 44, pp. 445–464. ACM Press, New York (2009)
13. Kats, L.C.L., Kalleberg, K.T., Visser, E.: Domain-specific languages for composable editor plugins. In: Ekman, T., Vinju, J. (eds.) Language Descriptions, Tools, and Applications (LDTA 2009). ENTCS. Elsevier Science Publishers, Amsterdam (2009)
14. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: Modular development of textual domain specific languages. In: Paige, R., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 297–315. Springer, Heidelberg (2008)
15. Kuhn, T., Thomann, O.: Eclipse corner: Abstract syntax tree (2006), `http://eclipse.org/articles/article.php?fileĀrticle-JavaCodeManipulationAST/index.html`
16. Lévy, J.-P.: Automatic Correction of Syntax Errors in Programming Languages. PhD thesis, Ithaca, NY, USA (1971)
17. Lyon, G.: Syntax-directed least-errors analysis for context-free languages: a practical approach. Commun. ACM 17(1), 3–14 (1974)
18. Mauney, J., Fischer, C.: Determining the extent of lookahead in syntactic error repair. ACM Trans. Program. Lang. Syst. (TOPLAS) 10(3), 456–469 (1988)
19. Moonen, L.: Generating robust parsers using island grammars. In: Working Conference on Reverse Engineering (WCRE 2001), pp. 13–22. IEEE, Los Alamitos (2001)
20. Nilsson-Nyman, E., Ekman, T., Hedin, G.: Practical scope recovery using bridge parsing. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 95–113. Springer, Heidelberg (2009)
21. Pai, A., Kieburtz, R.: Global Context Recovery: A New Strategy for Syntactic Error Recovery by Table-Drive Parsers. ACM Trans. Program. Lang. Syst. (TOPLAS) 2(1), 18–41 (1980)
22. Pennello, T.J., DeRemer, F.: A forward move algorithm for LR error recovery. In: Principles of programming languages (POPL 1978), pp. 241–254. ACM, New York (1978)
23. Rekers, J., Koorn, W.: Substring parsing for arbitrary context-free grammars. SIGPLAN Not. 26(5), 59–66 (1991)
24. Salomon, D., Cormack, G.: The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical report, TR 95/06, Dept. of Comp. Sci., University of Manitoba, Winnipeg, Canada (1995)
25. Swierstra, S.D., Duponcheel, L.: Deterministic, error-correcting combinator parsers. In: Launchbury, J., Sheard, T., Meijer, E. (eds.) AFP 1996. LNCS, vol. 1129, pp. 184–207. Springer, Heidelberg (1996)
26. Valkering, R.: Syntax error handling in scannerless generalized LR parsers. Master's thesis, University of Amsterdam (August 2007)
27. van den Brand, M.G.J., Bruntink, M., Economopoulos, G.R., de Jong, H.A., Klint, P., Kooiker, T., van der Storm, T., Vinju, J.J.: Using the Meta-Environment for maintenance and renovation. In: European Conference on Software Maintenance and Reengineering (CSMR 2007), pp. 331–332. IEEE, Los Alamitos (2007)
28. van Deursen, A., Kuipers, T.: Building documentation generators. In: IEEE International Conference on Software Maintenance (ICSM 1999), p. 40. IEEE Computer Society, Los Alamitos (1999)
29. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (September 1997)
30. Visser, E.: Meta-programming with concrete object syntax. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 299–315. Springer, Heidelberg (2002)