Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Deriving tolerant grammars from a base-line grammar

Steven Klusener, Ralf Lämmel

REPORT SEN-E0319 DECEMBER 23, 2003

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Deriving tolerant grammars from a base-line grammar

ABSTRACT
A grammar-based approach to tool development in re- and reverse engineering promises precise structure awareness, but it is problematic in two respects. Firstly, it is a considerable up-front investment to obtain a grammar for a relevant language or cocktail of languages. Existing work on grammar recovery addresses this concern to some extent. Secondly, it is often not feasible to insist on a precise grammar, e.g., when different dialects need to be covered. This calls for tolerant grammars. In this paper, we provide a well-engineered approach to the derivation of tolerant grammars, which is based on previous work on error recovery, fuzzy parsing, and island grammars. The technology of this paper has been used in a complex Cobol restructuring project on several millions of lines of code in different Cobol dialects. Our approach is founded on an approximation relation between a tolerant grammar and a base-line grammar which serves as a point of reference. Thereby, we avoid false positives and false negatives when parsing constructs of interest in a tolerant mode. Our approach accomplishes the effective derivation of a tolerant grammar from the syntactical structure that is relevant for a certain re- or reverse engineering tool. To this end, the productions for the constructs of interest are reused from the base-line grammar together with further productions that are needed for completion.

1998 ACM Computing Classification System: D.3.4
Keywords and Phrases: tolerant parsing; software re-engineering

Deriving tolerant grammars from a base-line grammar

Steven Klusener [1,2] and Ralf Lämmel [2,3]

[1] Software Improvement Group, Muiderstraatweg 58a, NL-1111 PT Diemen
[2] Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
[3] CWI, Kruislaan 413, NL-1098 SJ Amsterdam
Email: (steven|ralf)@cs.vu.nl

Abstract

A grammar-based approach to tool development in re- and reverse engineering promises precise structure awareness, but it is problematic in two respects. Firstly, it is a considerable up-front investment to obtain a grammar for a relevant language or cocktail of languages. Existing work on grammar recovery addresses this concern to some extent. Secondly, it is often not feasible to insist on a precise grammar, e.g., when different dialects need to be covered. This calls for tolerant grammars.

In this paper, we provide a well-engineered approach to the derivation of tolerant grammars, which is based on previous work on error recovery, fuzzy parsing, and island grammars. The technology of this paper has been used in a complex Cobol restructuring project on several millions of lines of code in different Cobol dialects.

Our approach is founded on an approximation relation between a tolerant grammar and a base-line grammar which serves as a point of reference. Thereby, we avoid false positives and false negatives when parsing constructs of interest in a tolerant mode. Our approach accomplishes the effective derivation of a tolerant grammar from the syntactical structure that is relevant for a certain re- or reverse engineering tool. To this end, the productions for the constructs of interest are reused from the base-line grammar together with further productions that are needed for completion.

1. In need of tolerance

Background — System analysis and transformation Tool providers for automated software analysis and transformation develop new components on a daily basis. These components have to be 'scalable', 'tolerant' and 'evolvable'. That is, they have to cope with large portfolios from different clients using various dialects and language cocktails while client requirements might evolve over time.

These are representative analyses and transformations:

- Business rule extraction for software re-engineering.
- Data expansion (think of Y2K as a folklore instance).
- Off-line code restructuring, e.g., goto elimination.
- Interactive code restructuring (think of refactoring).
- Conversion regarding language dialect, API, idioms.
- Software re-documentation.
- Metrics-based software assessment.

In the present paper, we focus on a technological aspect of system analysis and transformation, namely parsing of source code. The technology of this paper has been used in a complex Cobol restructuring project on several millions of lines of code in different Cobol dialects [Vee03].

Problem statement — Tolerant parsing Since components for analysis and transformation interact with the syntax of the underlying language cocktail (e.g., Cobol + Embedded SQL + CICS), one would like to employ a context-free approach to source code analysis as opposed to a lexical approach. There is some borderline where proper parsing becomes mandatory. For example, goto elimination for Cobol [SSV02] involves a non-trivial fragment of the language syntax, and hence a lexical approach is impractical. However, in the view of the up-front investment for parsers, it is still common to employ a lexical or a mixed approach in practice. Another practical limitation of normal parsers is that they cannot easily deal with different dialects, embedded languages, erroneous inputs (such as in an editing session). These problems have triggered work on *tolerant* parsers, also called *robust* parsers elsewhere. Previous approaches to tolerant parsing [BH82, Kop97, DK99, Moo01] are limited in two respects. Firstly, the link between a grammar used for tolerant parsing and the intended language is not completely clear. Secondly, specific technology is assumed to realise tolerant parsing. Both concerns are addressed in the present paper.

Contribution Our approach is based on two concepts:

a) We provide a formal definition of a grammar that is tolerant with respect to another grammar. To this end, we develop an approximation relation on grammars such that 'constructs of interest' are recognised in a reliable manner. Thereby, we address the important problem of false positives and false negatives, which enters the scene when tolerant parsing is used instead of precise parsing.

b) We define a semi-automatic process to derive a tolerant grammar from the constructs that are of interest for a certain re- or reverse engineering tool. The productions for the constructs of interests are reused from a base-line grammar that serves as a point of reference. Further productions are reused to provide a reliable context for constructs of interest. Ultimately, default productions are added to skip irrelevant constructs in a liberal mode.

Road-map In Sec. 2, we review island grammars with which we share our motivation for problem dedication in source code analysis. This review provides us with observations that shape our approach. In Sec. 3, we formally define the notion of a tolerant grammar, which is a generalisation of the notion of a (well-behaved) island grammar. In Sec. 4, we describe the semi-automatic process for the derivation of a tolerant grammar from a base-line grammar and constructs of interest. In Sec. 5, practical experiences with tolerant parsing are reported.

2. Islands in the stormy ocean

An island grammar [DK99, Ver00, Moo01, Moo02] consists of 'island' productions for the precise analysis of constructs of interest, and 'water' productions for skipping the irrelevant rest of the input. Island grammars combine the flexibility of lexical analysis with the power of context-free parsing. The available body of research on island grammars employs SDF [HHKR89] and scannerless generalised LR parsing [Vis97, BSVV02] for the implementation of island grammars. In this section, we will review island grammars in detail using an example. We will motivate the necessity of a so far missing correspondence between an island grammar and a base-line grammar which serves as a point of reference. This missing correspondence should guarantee that there are no false positives and false negatives when trying to select constructs of interest with a tolerant parser.

An example of an island grammar The grammar in Fig. 1 defines the structure of data declarations in a given Cobol program. Hence, a parser that is derived from the grammar can be used to select all data declarations in a given Cobol source file. For example, the three data declarations `"01 REC1"`, `"03 FLD1"`, and `"03 FLD2"` will be recognised for the following Cobol program:

```
module Layout
  lexical syntax
    [\ \t\n]              -> LAYOUT

module Water
  imports Layout
  context-free syntax
    Chunk*                -> Input
    Water                 -> Chunk
  lexical syntax
    ~[\ \t\n]+            -> Water {avoid}

module DataParts
  lexical syntax
    [0][1-9]              -> Level
    [1-4][0-9]            -> Level
    [A-Z][A-Z0-9\-]*      -> DataName

module DataFields
  imports Water DataParts
  context-free syntax
    Level DataName        -> Chunk
```

Figure 1. An island grammar (in SDF notation) for extracting data declarations; adopted from [Moo01].

```
IDENTIFICATION DIVISION.
 PROGRAM-ID.  TST.
DATA DIVISION.
 WORKING-STORAGE
  01    REC1.
    03  FLD1 PIC 99.
    03  FLD2 PIC S9(4) USAGE COMP.
PROCEDURE DIVISION.
 ...
```

We will now explain the grammar in detail. The productions in module `DataParts` define the lexical sorts `Level` and `DataName`. In the shown Cobol snippet, there are two level numbers, i.e., `01` and `03`; there are three data names, i.e., `REC1`, `FLD1`, and `FLD2`. The production

```
Level DataName -> Chunk
```

in the module `DataFields` defines the context-free syntax of constructs of interest, i.e., data declarations. The remaining productions in the modules `Layout` and `Water` are needed to skip layout and irrelevant input (aka water or ocean). In particular, the production `~[\ \t\n]+ -> Water {avoid}` defines that every token without spaces, tabs and newlines can be parsed as water, while the `avoid` attribute assigns a low priority to this production. Thereby, chunks of the form `Level DataName` (i.e., islands) are strictly preferred over water. This grammar is very concise because it only defines the structure of constructs of interest.

```
Level-number
data-name-or-filler?
Data-description-entry-clauses
"."
      ->   Data-description-entry
```

Figure 2. The base-line production for data declarations; adopted from [LV99].

```
module DataFieldsWithContext
  imports Water DataParts
  context-free syntax
    "DATA DIVISION"
    DataChunk*
    "PROCEDURE DIVISION" -> Chunk
    Level DataName       -> DataChunk
    Water                -> DataChunk
```

Figure 3. A revision of Fig. 1 to rule out false positives outside the data division; adopted from [Moo02].

The relation with a base-line grammar We will now review the relation of the island grammar from above with a suitable base-line grammar for Cobol. In general, we want to be sure that if we used the base-line grammar to do the extraction of constructs of interest, then we obtain the same results as with the island grammar. We will here consider the grammar for VS Cobol II as of [LV99] as the base-line grammar for Cobol. In Fig. 2, the base-line production for data declarations is presented. This inspection reveals that the island production for `DataField` chunks actually covers some part of the base-line production for data description entries. In particular, the clauses of such an entry (e.g., `PIC` ...), and the terminating period "." are omitted from the chunk production. Also, `DataField` chunks are simply assumed to occur in a flat sequence of chunks, while `Data-description-entry` occurs in the context of other base-line productions for the data division of a Cobol program. The general question is now if the island grammar admits false positives and false negatives when trying to recognise constructs of interests.

A false positive is a substring which is parsed by the island grammar as an island, but which does not correspond to a construct of interest (judging on the basis of the base-line grammar). The following fragment triggers a false positive:

```
01 FLD1  PIC 99.
01 FLD2  PIC 99 VALUE 42 COMP-3.
```

That is, `"42 COMP-3"` matches with the island production `Level DataName -> Chunk` but not with `Data-description-entry` in the base-line grammar. This false positive can be fixed by rejecting `COMP-3` as a data name. In SDF this can be done as follows:

```
"COMP-3" -> DataName {reject}
```

Eventually, all or most reserved Cobol words should be rejected like this. This is error-prone when it is done manually, and without reference to a base-line grammar. For other sorts of false positives, different provisions are needed. For example, in [Moo02], a revision of the island grammar for data declarations is proposed which makes sure that we only search for `DataField` chunks within the data division, and not accidentally in the procedure division. Indeed, a numeric literal followed by a name can also very well form part of some statements, e.g., in:

```
DISPLAY 01 FLD1.
```

The required revision is shown in Fig. 3. There are now two levels of chunks. At the top level, we search for the content of the data division, and at the inner level, we search for data fields in a list of data chunks.

A false negative is a substring which is a proper construct of interest according to the base-line grammar, but which is not recognised by the island grammar. Let us suppose that the constructs of interests are all data declarations — not just named ones. Then the following fragment contains two false negatives:

```
01    REC1.
  03          PIC X.
  03  FILLER  PIC X.
```

The substring `"03 PIC X."` matches with the base-line production of `Data-description-entry` because it is an entry without any data name. By contrast, the island grammar will not recognise this special form. Also, the substring `"03 FILLER PIC X."` is not recognised by the island grammar. Here we assume that reserved words such as `FILLER` were rejected as data names as needed above for handling false positives.

The cure — Sharing structure We contend that it is not straightforward to determine if an island grammar admits false positives or negatives. Once false positives or negatives were identified, one still has to revise the island grammar by imposing more structure on chunks, by adding restrictions, priorities, or others. Our key insight, which we concluded from these observations, is that the status of tolerant parsing becomes radically more clear if the link between a tolerant grammar and its base-line grammar can be clearly identified in terms of shared structure. For example, we could refactor the island grammar for data declarations so that it becomes compliant with the base-line grammar. In this case, we will obtain chunk productions as follows:

```
Data-description-island -> Chunk
Level (DataName|"FILLER")? Water* "."
   -> Data-description-island
```

The nonterminal `Data-description-island` can be directly mapped onto the base-line nonterminal `Data-description-entry`. (If we wanted to express that only named data fields are relevant, then we would remove the relevant branches of the production by a grammar transformation.) It remains to share more structure in order to enforce the right context for data description entries.

3. A formal definition of a tolerant grammar

We will now define the notion of a tolerant grammar. To this end, we define an approximation relation on grammars. Here we employ a base-line grammar as a point of reference. As we will discuss at the end of the section, the notion of a tolerant grammar is more general and abstract when compared to the notion of an island grammar. The specific intention of (well-behaved) island grammars suggests one possible style of encoding tolerant grammars.

A nonterminal mapping The main idea underlying our formalisation is that the correspondence between two grammars is specified by a mapping m between some of the nonterminals of both grammars. We write $G' \sqsubseteq_m G$ to denote that G' approximates G, where some of the nonterminals of G are mapped to a counterpart in G' via m. Our notion of a tolerant grammar is visualised in Fig. 4. We first go from *left to right* ("$\Rightarrow$"). Given is a parse tree T_G according to the base-line grammar G. The shown subtree inside T_G comprises a construct t of interest rooted by a nonterminal n. If G' is a tolerant grammar, then there must be a parse tree $T_{G'}$ with a subtree rooted by $m(n)$ for the same construct of interest t in the same context x and y. If this left-to-right correspondence holds, there will be *no false negatives*. We now go from *right to left* ("$\Leftarrow$"). Basically, the inverse condition must hold, and then there will be *no false positives*. Here, it is important that we restrict both "$\Rightarrow$" and "$\Leftarrow$" to the language generated by G (i.e., $L(G)$) when we require that the tolerant grammar G' and its base-line grammar G agree on the substrings t for constructs of interest. The fact that G' is tolerant is modelled by the fact that G' can still recognise more substrings as constructs of interest for inputs that are in $L(G')$ but not in $L(G)$. The details of the definition follow.

Context-free grammars A grammar G is a tuple $\langle N, T, P, s\rangle$, where N is the set of *nonterminals*, T is the set of *terminals*, P is a set of *productions* and $s \in N$ is the start symbol as usual. In the sequel, we let t, x, y range over the set of *strings* T^* whereas u, v, w range over the set of *sentential forms* $\{N \cup T\}^*$, and n ranges over the set of nonterminals N. We assume the standard derivation relation on sentential forms. That is, $u\,n\,w \Rightarrow_G u\,v\,w$ holds if there is a production $n \to v$. The relations $\overset{*}{\Rightarrow}_G$ and $\overset{+}{\Rightarrow}_G$ denote the common closures of $\Rightarrow_G$. For example, the language defined by G is the set of strings t such that $s \overset{+}{\Rightarrow}_G t$.

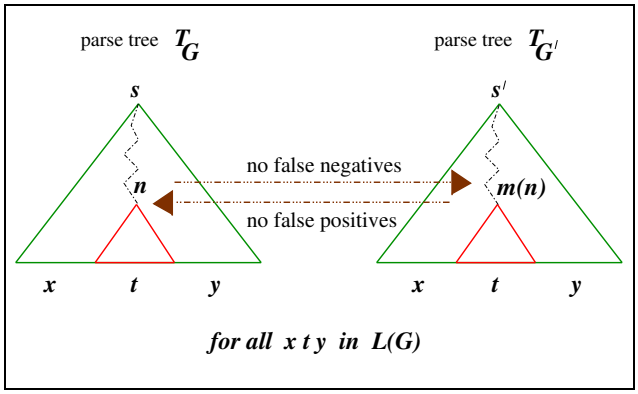

Figure 4. Correspondence between a tolerant grammar and a base-line grammar.

Definition 1 (Approximation relation on grammars) *G and G' are two grammars with the start symbols s and s', respectively. m is a partial mapping from $N(G)$ to $N(G')$. We say that G' approximates G, denoted by $G' \sqsubseteq_m G$, if the following holds for all $x\,t\,y \in L(G)$, $n \in N(G)$ with $m(n) \neq \bot$:*

$$\begin{array}{rlll} & s \overset{*}{\Rightarrow}_G x\ n\ y & \wedge & n \overset{+}{\Rightarrow}_G t \\ \iff & s' \overset{*}{\Rightarrow}_{G'} x\ m(n)\ y & \wedge & m(n) \overset{+}{\Rightarrow}_{G'} t \end{array}$$

□

Decidability The approximation relation defines equality of generated languages except for the provision of the nonterminal mapping m and the asymmetry to consider strings from $L(G)$ only. This property is obviously undecidable. However, in Sec. 4, we present a process for the derivation of tolerant grammars from a base-line grammar such that the approximation property follows from the construction.

Filtering derivations In reality, parsing is not just plain simulation of the derivation relation for context-free grammars. Actual parsing technology relies on a number of idioms such as 'prefer-shift-over-reduce' for LR parsers, semantic predicates for top-down parsers, or disambiguation constructs for generalised LR parsing. We reflect the potential of such idioms in our setting by a single provision. That is, we add a *filter* F, which is a predicate over derivations. Given a grammar $G = \langle N, T, P, s, F\rangle$, a derivation $s \Rightarrow_G \ldots \Rightarrow_G \ldots \Rightarrow_G t$ is constrained to also pass the filter F. Def. 1 does not need any modification. This provision effectively abstracts from idiosyncratic issues of specific parsing technology.

Tolerant grammars vs. island grammars Moonen gives a definition of island grammars in [Moo01]. His definition requires that the island grammar accepts more input strings than a reference grammar. More specifically, it requires that the island grammar accepts constructs of interest in strings that are not accepted by the reference grammar. However, the nonterminal mapping that we have introduced is not part of Moonen's definition. And because this mapping is missing, reasoning about false positives and negatives is not facilitated. In particular, degenerated grammars are island grammars. For example, for every $G = \langle N, T, P, s \rangle$ we can construct the following degenerated island grammar which is obviously by far too liberal because it accepts T^*:

$$G_I = \langle \{s, n\}, T, \{s \to \epsilon, s \to n\, s\} \cup \{n \to t | t \in T\}, s \rangle$$

Furthermore, while Moonen's definition refers to a reference grammar, there is no systematic process which would utilise this reference grammar (say, base-line grammar) for the derivation of island grammars. The subsequent section present such a process for tolerant grammars.

Styles of tolerant grammars Moonen's definition also states that the reference grammar has a higher complexity than the island grammar. We omit such intentions from our abstract, formal definition of tolerant grammars. Nevertheless, different styles of tolerant grammars can be identified with (well-behaved) island grammars as one example. Here, we propose to reserve the notion of an island grammar for grammars with little structure: most of the input is parsed as *water* and only certain substrings are parsed as islands. (This is well in line with the island grammars in the literature.) Then, tolerant grammars of a different style are *skeleton grammars*. These are grammars that share their context-free structure with a base-line grammar down-to a certain depth in the parse tree. In the next section, we will present a constructive approach for deriving tolerant grammars that are in fact skeleton grammars.

4. The derivation of a tolerant grammar

We will now describe a process for the derivation of tolerant grammars (in fact, skeleton grammars) from a base-line grammar. To this end, we will employ a Cobol re-engineering exercise as the running example.

Contract productions We will first motivate the transformation underlying our running example so that the relevant grammar productions can be determined. The transformation is meant to eliminate Cobol's jump statement `NEXT SENTENCE`. (This is a typical operation of a restructuring suite [Vee03].) Here, is a Cobol snippet:

```
context-free syntax
  "CONTINUE"          -> Statement
  "NEXT" "SENTENCE"   -> Statement
  Statement*          -> Statement-list

variables
  "Stat*" [0-9]* -> Statement
```

```
    replace(Stat*1 NEXT SENTENCE Stat*2)
  = replace(Stat*1 CONTINUE)
```

Figure 5. A transformation rule and the associated contract productions.

```
IF A>B THEN
  MOVE 1 TO B
ELSE
  NEXT SENTENCE
  DISPLAY "WILL NEVER BE REACHED".
DISPLAY "START OF NEXT SENTENCE".
```

Because of the way the jump statement `NEXT SENTENCE` statement is used in the sample code, it can be translated into a no-op `CONTINUE` statement subject to removal of the dead code following `NEXT SENTENCE`. This leads to the following result:

```
IF A>B THEN
  MOVE 1 TO B
ELSE
  CONTINUE.
DISPLAY "START OF NEXT SENTENCE".
```

The definition of the transformation and the relevant productions are given in Fig. 5. The actual transformation is implemented using term rewriting in the ASF+SDF Meta-Environment [BDH+01]. Since the corresponding productions express to what extent the transformation depends on the base-line grammar, we will refer to them as the *contract productions*.

From the contract productions to a skeleton grammar Clearly, the contract productions have to be complemented to obtain a complete grammar that can be used to parse actual source code. The overall process for the derivation of a skeleton grammar is shown in Fig. 6. There are basically three steps. Firstly, the contract productions are complemented by all base-line productions that are needed to reach constructs of interest from the start symbol. This step is called *root completion*. Secondly, all undefined or partially defined nonterminals are provided with liberal defaults. This step is called *default completion*. Thirdly, tolerance can be customised to favour parse errors for certain symptoms of inappropriate contract productions. This step is called *default restriction*. We will discuss these steps in turn. To illustrate what is coming, the resulting skeleton grammar for our running example is shown in Fig. 7.

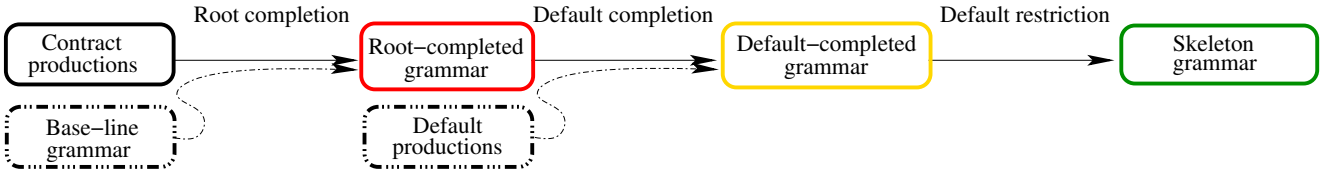

Figure 6. Construct a tolerant grammar (in fact, a skeleton grammar) from contract productions, a base-line grammar, and default productions; this is a semi-automatic process.

contract.1	`"CONTINUE"`	`-> Statement`
contract.2	`"NEXT" "SENTENCE"`	`-> Statement`
contract.3	`Statement*`	`-> Statement-list`
root.1	`Statement-list "."`	`-> Sentence`
root.2	`Label-name "." Sentence*`	`-> Paragraph`
root.3	`Paragraph-without-header? Paragraph*`	`-> Paragraphs`
root.4	`Section-header "." Paragraphs`	`-> Section`
root.5	`Section-without-header Section*`	`-> Sections`
root.6	`Proc-division-header Sections`	`-> Proc-division`
root.7	`Id-division? Env-division? Data-division? Proc-division?`	`-> Program`
default.1	`Token-start-verb Token-stat* "."`	`-> Statement`
default.2	`Integer | User-defined-word`	`-> Label-name`
default.3	`Sentence+`	`-> Paragraph-without-header`
default.4	`Label-name "SECTION" Integer?`	`-> Section-header`
default.5	`Paragraphs`	`-> Section-without-header`
default.6a	`"PROCEDURE" "DIVISION" Using-phrase "." Declaratives?`	`-> Procedure-division-header`
default.6b	`"USING" User-defined-word*`	`-> Using-phrase`
default.6c	`"DECLARATIVES" "." Token-excl-end* "END" "DECLARATIVES" "."`	`-> Declaratives`
default.7	`"IDENTIFICATION" "DIVISION" "." Token-excl-env-data-procedure*`	`-> Id-division`
default.8	`"ENVIRONMENT" "DIVISION" "." Token-excl-data-proc*`	`-> Env-division`
default.9	`"DATA" "DIVISION" "." Token-excl-proc*`	`-> Data-division`

Figure 7. The productions of the constructed grammar; auxiliary token productions are omitted.

The productions are grouped for the contract, root completion, and default completion. The productions for root completion were reused from the base-line grammar for VS Cobol II [LV99]. The default productions were derived semi-automatically from the base-line grammar.

Automated root completion In order to share context-free structure with the base-line grammar, we add all productions that are needed to connect the nonterminals for constructs of interest with the start symbol. This is an important provision for making sure that constructs of interest will only be recognised in the right context. In our example, we need to connect `Statement` with `Program`. In Fig. 8, root completion is defined as an algorithm that constructs the *root-completed* grammar. This algorithm is completely general (i.e., no side conditions, no heuristics), and it is trivially implemented. The algorithm transitively selects all the base-line productions that are needed for reaching the start symbol. Note that the automation of root completion implies that its added value comes for free. In Fig. 7 all productions root.1 – root.7 are shown that will be collected for the contract productions contract.1 – contract.3 during root completion.

Extent of default completion Some of the nonterminals in the root-completed grammar can be completely undefined. Both contract productions and the productions included by root completion can involve such unresolved references. In our running example, `Data-division` in the production root.7 is such an undefined nonterminal. Default completion shall provide a definition for these undefined nonterminals. Defined nonterminals also necessitate a default completion for two reasons. Firstly, some nonterminals might be partially defined, e.g., `Statement` in our example, because the contract productions only cover some forms. Secondly, to improve tolerance, it is beneficial to provide defaults for most nonterminals anyway.

Notation: $\mathcal{D}(P)$ vs. $\mathcal{U}(P)$ denote the nonterminals that are defined vs. used by the productions P. $N(P)$ denotes the union of $\mathcal{D}(P)$ and $\mathcal{U}(P)$.

- Input:
  – Base-line grammar $G_B = \langle N_B, T, P_B, s \rangle$.
  – Contract productions $P_C \subseteq P_B$.
- Output:
  – Root-completed grammar $G_R = \langle N_R, T, P_R, s \rangle$.
- Algorithm:
  1. $P_R := P_C$ (initialisation).
  2. Repeat steps (a) and (b) as long as possible.
     (a) Pick a production $p \in P_B \setminus P_R$ where p is of the form $n \to u\, n'\, v$, and $n' \in \mathcal{D}(P_R), n \notin \mathcal{D}(P_R)$.
     (b) $P_R := P_R \cup \{p\}$.
  3. $N_R := N(P_R)$

Figure 8. Automated root completion.

Parsing with defaults In the default productions from Fig. 7, we use nonterminals the names of which start with `Token-...`. Parts of the parse tree in which we are not interested are parsed as sequences of such tokens. So these nonterminals root waterish areas in the parse tree. Their definitions are derived automatically by enumerating all the kinds of tokens that can possibly occur in the relevant context. With the default productions from Fig. 7 we can parse a fragment like the following:

```
ACCEPT CURRTIME FROM DATE
CONTINUE
ADD A TO B
```

This fragment contains three statements: the first and the last statement are parsed via the liberal default production for statements, and the `CONTINUE` statement in the middle is covered by one of the contract productions. We generally assume that default productions are associated with priorities such that non-default productions are preferred over default productions.

Naive defaults The ultimate approach is to use the structure of the base-line grammar in deriving default productions that respect the approximation property. On the other hand, the default productions are meant to be very liberal in the interest of tolerance. A naive approach is to define defaults such that they generate `Token*`. This would suggest default productions as follows for the running example:

```
default.1   Token*   -> Statement
...
default.9   Token*   -> Data-division
```

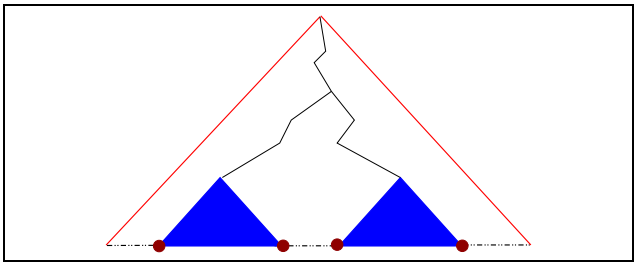

Figure 9. Stable recognition of waterish subtrees in a parse tree. The filled triangles represent subtrees that originate from default productions. The bullets represent synchronisation tokens.

Such default productions will immediately guarantee the "⇒" direction of the approximation relation because the completed grammar will be strictly more general than the base-line grammar. The "⇐" requires a refinement of this naive approach as discussed in the sequel.

Synchronisation — By example Essentially, we should refine the simple default productions in a way that some minimal synchronisation with the input is enforced. This idea is illustrated in Fig. 9. All the default productions from Fig. 7 contain synchronisation tokens at the beginning or the end or both. Here is, for example, the default production for the data division (default.9 in Fig. 7):

```
"DATA" "DIVISION" "."
Token-excl-proc*
      -> Data-division
```

Here, we assume that `Token-excl-proc` is defined to comprise all possible tokens excluding `"PROCEDURE"`. This production expresses that the content of the data division starts necessarily with `"DATA" "DIVISION" "."` and it ends when the subsequent division starts. This is a very liberal definition but still it is safe since it is in accordance with the base-line grammar. For example, one can check that the token sequence `"DATA" "DIVISION" "."` does indeed not occur anywhere else but at the beginning of the data division. Another example is the default production for Cobol statements:

```
Token-start-verb Token-stat* ->  Statement
```

Here, `Token-start-verb` comprises all start verbs of Cobol statements, i.e., `"ACCEPT"`, `"ADD"`, etc., whereas `Token-stat` is defined to comprise all tokens that can possibly occur in Cobol statements. This will exclude start verbs and the "." to finish a Cobol sentence.

Automated synchronisation The kind of synchronisation information that we used in the examples given above can be derived from the base-line grammar by a systematic analysis. To this end, we need to compute some sets of tokens for each nonterminal very much in the sense of the standard first and follow sets known from parsing theory [AU73, ASU86]. The following token sets are of use:

- $first_G(n)$ — first in the strings derived from n.
- $last_G(n)$ — last in the strings derived from n.
- $follow_G(n)$ — following n in sentential forms.
- $precede_G(n)$ — preceding n in sentential forms.
- $prefix_G(n)$ — anywhere in prefixes of n.
- $postfix_G(n)$ — anywhere in postfixes of n.
- $body_G(n)$ — anywhere in the strings derived from n while ignoring the first and the last token.

Using these sets, we can establish safe defaults. For example, for a given nonterminal n, we can define its default as "$f\ t{*}\ l$" subject to the following side conditions:

- $n \not\Rightarrow^+_G \epsilon$ (nonempty derivation)
- $first_G(n) = \{f\}$ (definite begin)
- $last_G(n) = \{l\}$ (definite end)
- $f \notin prefix_G(n)$ (distinguished begin)
- $l \notin postfix_G(n)$ (distinguished end)
- $l \notin body_G(n)$ (distinguished end cont'd)
- $t = body_G(n)$ (all possible tokens)

The nonterminal `Declaratives` meets these specific criteria, and hence the production default.6c is approved. Using similar criteria, we can provide default productions in a systematic manner. (We need to consider more look ahead than just one token in some cases. Also, we need to consider more relaxed scenarios than the one above.) A detailed study of the completeness and correctness of this approach is beyond the scope of the present paper. Cobol is certainly suited for this approach to the definition of synchronisation tokens because of its pervasive use of reserved keywords. In general, the use of a token for synchronisation is maybe better subject to approval by the grammar engineer because the automated insistence on certain synchronisation tokens could make tolerant parsing insufficiently tolerant.

Default restriction At this point, we have derived a proper tolerant grammar. There is one remaining issue, namely the possibility of quality problems with the base-line grammar. That is, we might intend to cover a certain construct by a contract production but its definition in the base-line grammar is incomplete or incorrect. For example, we might intend to cover all add statements while the base-line grammar lacks some form. If this missing form is exercised by source code, it will be skipped via the default. This is a false negative in the special sense that the (unaccessible) intended syntax for constructs of interest deviates from the available base-line grammar. To tackle this problem, one can restrict defaults so that they do not apply too easily. Then, parse errors manifest the inadequacy of the contract productions. For example, to disable the default production for add statements it is sufficient to exclude `"ADD"` from the definition of `Token-start-verb`.

5. Practical experiences

We have applied our process for parser development to Cobol while we used the relatively correct and complete grammar for VS Cobol II [LV99] as base-line grammar. (This grammar was recovered earlier from IBM's industrial standard [IBM93] using other grammar engineering techniques [LV01].) We have automated the process for deriving skeleton grammars using grammar transformations [Läm01, LW01] as supported by the Grammar Deployment Kit (GDK) [KLV02].[1] Using GDK, we can generate parser specifications for different parsing technologies. Our focus was on making the process work for the SDF syntax definition formalism [HHKR89] and scannerless generalised LR parsing [Vis97, BSVV02] as available in the ASF+SDF Meta-Environment [BDH+01]. This focus is triggered by our use of the ASF+SDF Meta-Environment for the implementation of transformation technology. We do not rely on specific SDF features other than `avoid` attributes for assigning lower priorities to default productions. This can be achieved with other parsing technology as well.

We have applied tolerant parsing in a project with a hierarchy of components, each defining several nontrivial transformation rules for restructuring Cobol programs; see [Vee03]. Each component relies on a designated skeleton grammar derived from the component-specific contract productions. Tolerant parsing is compositional. That is, without any changes to the components we can take the union of all the contract productions for the various components and derive a combined skeleton grammar that works for all components. In the project, the components were applied on a source base ($>$ 2,000,000 LoC of some forms of IBM Cobol and $>$ 2,000,000 LoC of some forms of Microfocus Cobol). While precise parsing according to the base-line grammar would have been unsuccessful, we succeeded with tolerant parsing. The various relevant constructs were properly defined by the base-line grammar since all encountered dialects agreed on them. So no dialect-specific customisation was needed.

The skeleton grammars which are derived from the contract productions are typically by factor 3–15 smaller when compared to the base-line grammar. Here are some milestone grammar sizes:

[1] GDK URL: `http://gdk.sourceforge.net`

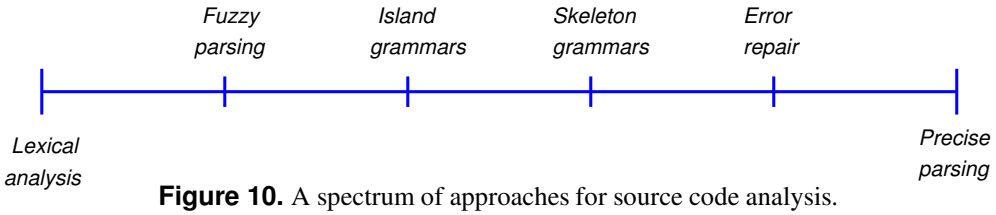

Figure 10. A spectrum of approaches for source code analysis.

Grammar	Productions	LOC	Keywords
Simple statement skeleton	51	209	82
Nested statement skeleton	268	438	129
Base-line grammar	888	1228	325

We have experienced that the runtime for the different parsers lies within the same range; parsing with the base-line grammar is as fast as parsing with a small skeleton grammar. Note, that this was not the case when we started our project because a less systematic default completion caused many (local) ambiguities and in turn penalties. The described scheme for default productions using simple means for synchronisation is very robust.

6. Related work

Fuzzy parsing In [Kop97], the notion of fuzzy parsing is defined and engineered. Fuzzy parsers perform syntactical analysis on selected portions of the input for the purpose of the extraction of a partial source code model. The key idea is to identify 'anchor terminals' that trigger the application of context-free productions. That is, the input is skipped until an anchor a is found, and then context-free analysis is attempted using a production starting with a. This is a rather lexical approach because no context-free structure is employed to determine the context for constructs of interest.

Island grammars A potent refinement of fuzzy parsing is the notion of an island grammar [DK99, Ver00, Moo01, Moo02]. A unified syntax definition formalism is used to specify islands and water. Island grammars from the literature are geared towards very specific parsing technology. Island grammars amalgamate lexical and context-free analysis rather heavily; see the lexical definition of `Water` in Fig. 1 which tends to compete with problem-specific forms of chunk. As discussed in Sec. 2, island grammars can be radically concise for simple analysis and transformation problems when compared to an up-front development of a conservative parser. Furthermore, the island grammar approach does immediately lead to very tolerant parsers.

Degrees of tolerant parsing In Fig. 10, we place various approaches on a chart regarding their relative position in between lexical analysis and precise parsing. Fuzzy parsers involve a lexical criterion to switch to the context-free mode. Island grammars can mix lexical vs. context-free style in more sophisticated ways. Still the islands are found in lists of chunks with little or no similarities to the parse-tree structure suggested by a base-line grammar. Skeleton grammars employ ordinary context-free productions where lexical skips only occur at subtrees the structure of which is not relevant. Error-repairing parsers can be seen as a way to achieve tolerance. The simple approach is 'panicking' using stop symbols [AU73, ASU86] on top of an otherwise precise grammar. So lexical skips only occur for recovery from parser errors. A sophisticated approach is described in [BH82] where recovery from all errors is guaranteed, and recovery is driven by the grammar structure rather than using a criterion for plain lexical panicking.

7. Conclusion

We have first presented a formal definition of tolerant grammars. The parsers that are derived from our tolerant grammars accept inputs that use unanticipated phrases in the sense of dialects. Our definition specifically addresses the issue of false positives and false negatives, which are to be avoided when performing tolerant parsing. We have then described a semi-automatic process to derive a tolerant grammar for the productions that are needed for a specific grammar-based software tool. We have demonstrated our approach in the context of Cobol re-engineering. The resulting parsers scale as required for use in industrial projects.

Compared to previous work on error repair, fuzzy parsing, and island grammars, the following shift of focus and added value can be pointed out:

- We reuse productions from an existing base-line grammar to define the structure of constructs of interest. That is, we do not advocate the design of problem-specific productions, as in the case of island grammars. Because all our components for system transformation and analysis are based on one base-line grammar, component composition is possible.
- We advocate a form of tolerant grammars which we call skeleton grammars because they share their context-free structure with a base-line grammar down-to a certain depth in the parse tree. Thereby, we establish the right context for constructs of interest, which in turn contributes to reliable tolerant parsing, without false positives and false negatives.

- We use grammar transformations to select, adapt, compose, and generate productions. This contributes to the automation of tolerant grammar development. Without grammar transformations, some elements of our process, e.g., the derivation of auxiliary token nonterminals, would be tedious and error-prone.

In our ongoing work, we attempt to identify a simple domain-specific language for tolerant parser development. It provides the following concepts:

- Selection and specialisation of contract productions.
- Directives for synchronisation tokens.
- Directives for restricting defaults.
- Derivation of a designated abstract syntax.
- Contract-based test case generation.
- Seamless support of different parsing technologies.

Furthermore, the completion approach favoured for the development of skeleton grammars should be dualised. We think of relaxing the structure of the base-line grammar systematically until a certain normal form is reached.

Acknowledgment We are grateful to Mark van den Brand, Jim Cordy, Paul Klint, Jan Kort, Leon Moonen, Niels Veerman, Chris Verhoef, and Ernst-Jan Verhoeven for discussions on the overall subject of the paper. The three anonymous ICSM 2003 reviewers proved to be very helpful.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools*. Addison-Wesley, 1986.

[AU73] A.V. Aho and J.D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Englewood Cliffs (NJ), 1972–73.

[BDH+01] Brand, M.G.J. van den, Deursen, A. van, J. Heering, Jong, H.A. de, Jonge, M. de, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proc. Compiler Construction (CC'01)*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[BH82] D.T. Barnard and R.C. Holt. Hierarchic Syntax Error Repair for LR Grammars. *International Journal of Computer and Information Sciences*, 11(4):231–258, 1982.

[BSVV02] M.G.J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Proc. Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, 2002.

[DK99] A. van Deursen and T. Kuipers. Building Documentation Generators. In *Proc. International Conference on Software Maintenance (ICSM'99)*, pages 40–49, 1999.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[IBM93] IBM Corporation. *VS COBOL II Application Programming Language Reference*, 4th edition, 1993. Publication number GC26-4047-07.

[KLV02] J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. In *Proc. Language Descriptions, Tools, and Applications (LDTA'02)*, volume 65 of *ENTCS*. Elsevier Science, April 2002. 7 pages.

[Kop97] R. Koppler. A systematic approach to fuzzy parsing. *Software Practice and Experience*, 27(6):637–649, 1997.

[Läm01] R. Lämmel. Grammar Adaptation. In *Proc. Formal Methods Europe (FME'01)*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[LV99] R. Lämmel and C. Verhoef. *VS COBOL II grammar Version 1.0.3*, 1999. Available at: http://www.cs.vu.nl/grammarware/vs-cobol-ii/.

[LV01] R. Lämmel and C. Verhoef. Semi-Automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[LW01] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In *Proc. Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *ENTCS*. Elsevier Science, April 2001.

[Moo01] L. Moonen. Generating Robust Parsers using Island Grammars. In *Proc. Working Conference on Reverse Engineering (WCRE'01)*, pages 13–22. IEEE Press, October 2001.

[Moo02] L. Moonen. Lightweight Impact Analysis using Island Grammars. In *Proc. International Workshop on Program Comprehension (IWPC'02)*. IEEE Press, June 2002.

[SSV02] M.P.A. Sellink, H.M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS Legacy Systems. *Science of Computer Programming*, 45(2–3):193–243, 2002.

[Vee03] N.P. Veerman. Revitalizing modifiability of legacy assets. In *Proc. Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 19–29. IEEE Press, 2003.

[Ver00] E.J. Verhoeven. Cobol island grammars in SDF, 2000. Master's thesis, University of Amsterdam.

[Vis97] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.