Contract-Based Parsers
For Component-Based System Analysis And Transformation*

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

[1] Software Improvement Group, Kruislaan 419, NL-1098 SJ Amsterdam
[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

Recovering base-line grammar specifications for a specific language is one side of the coin. This paper focuses on the other side of the coin: obtaining a parser in an industrial context. The following side conditions characterise our setting. Firstly, we are interested in component-based development of tools for the analysis and transformation of Cobol software systems. Secondly, we commit ourselves to ASF+SDF technology as for the implementation of parsers and components for system analysis and transformation.

In this setting, we describe a process for parser development. It is centred around the observation that components for analysis and transformation contract only a small part of the Cobol language. The simplicity of contract-based parsers makes them easy to define, to debug, and to compile. The tolerance of these parsers lets them cope with the many Cobol dialects without further ado.

1 Introduction

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 'maintainable' in the sense that they have to cope with large portfolios from different clients using different dialects and language cocktails while the clients' requirements might evolve over time. A few typical problem classes in software analysis and transformation are the following:

- 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 as for dialect, API, idioms,
- software re-documentation,
- metrics-based software assessment.

Such projects are challenging due to a rich interaction of technological, management, and sociological aspects as exemplified in [KLV02a]. In the present paper, we deal with technological aspects surrounding the development of source code analysers, say *parsers*. Here, we restrict ourselves to the analysis and transformation of *Cobol* software systems. Further, we assume *component-based* tool development. Finally, we commit ourselves to *ASF+SDF technology* [B$^+$01] as for the implementation of parsers and components.

Contract-based parsing Since components for analysis and transformation interact with the syntax of the underlying language cocktail, *in principle*, one would like to employ a *context-free* as opposed to a lexical approach. However, in the view of the up-front investment for suitable parsers, lexical or mixed approaches are quite common in practice. There is a fuzzy borderline where lexical approaches stop to be tractable [BSV00]. To give an example, a proper Cobol `GOTO` elimination [SSV02] interacts heavily with the Cobol syntax, and hence a lexical approach is impractical. The present paper continues the line of research on *grammar engineering*[1] to make context-free technology more attractive. We contribute the notion of a *contract-based parser*. This is an improvement over previous work in that these largely context-free parsers abstract from all syntactical structure that is irrelevant for a given client of a source code analyser. In this manner, contract-based parsers get radically tolerant and simple.

*Draft as of October 28, 2002

[1] See the grammar engineering page at CS.VU.NL: `http://www.cs.vu.nl/grammars/`

Road-map In Sec. 2, we present a sample suite of simple components for the analysis and transformation of Cobol systems. They are also meant to illustrate the ASF+SDF technology that we use in our setting. In Sec. 3, we identify the problems with parser development in an industrial context. These problems triggered the idea of contract-based parsing. In Sec. 4, we describe an automated process for the development of contract-based parsers. In Sec. 5, related work will be reviewed. In Sec. 6, implementation issues of contract-based parsing are discussed. In Sec. 7, the paper is concluded.

2 Samples of component-based development

In figures 1–4, the ASF+SDF rewriting rules for several components for system analysis and transformation are shown. The specifications are executable within the ASF+SDF Meta-Environment [B$^+$01] based on SDF [HHKR89] as the syntax definition formalism, and ASF [BHK89] as the rewriting formalism. These samples will be useful for the subsequent motivation of our contract-based parser notion. The rewrite rules use concrete syntax to process the relevant Cobol patterns. For brevity, we omit the signatures of the functions for analysis and transformations. We should note that we rely on a recent extension of ASF, that is, traversal functions [BKV01] — a fully transparent approach to generic traversal as a refinement of an earlier generative approach [BSV97, BSV00]. With this extension, we can entirely focus on the problem-specific patterns. Otherwise, we had to define the functions for analysis and transformation for the hundreds of Cobol patterns. We will now discuss the samples in some detail.

`CONTINUE` elimination This is a transformation to eliminate the Cobol no-op `CONTINUE`. Such no-ops naturally arise in the course of program restructuring. The shown component is part of a suite for `GOTO` elimination with about 50 components. Rule [1] covers the case that `CONTINUE` occurs in an ordinary statement sequence. The side condition tests that the remaining statement sequence is not empty because we favour CONTINUE as the representation of the empty statement sequence. The other two rules [2a] and [2b] deal with `CONTINUE`s in the `THEN` and `ELSE` branches of `IF` statements. Here we allow for the removal of `CONTINUE`s even if this leads to an empty branch. This is sensible because subsequent transformations are assumed to simplify `IF` statements with empty branches accordingly.

`NEXT SENTENCE` to `CONTINUE` conversion In its full generality, the `NEXT SENTENCE` statement denotes a jump statement to the 'next sentence', that is, after the period of the current, possibly nested statement. The shown conversion rule assumes ANSI-74 expressiveness where conditional statements can only be nested in ways that `NEXT SENTENCE` can be replaced by a no-op CONTINUE. However, we have to remove dead code following `NEXT SENTENCE` (see rule [1]) because it would be re-animated once `NEXT SENTENCE` is replaced by `CONTINUE`. This component, like the previous one, is part of a suite for `GOTO` elimination with about 50 components.

Type-of-usage analysis A recurring theme in system analysis is to determine data fields with the same type of usage. This kind of analysis is usually based on code patterns that mention two or more data names in a context such that the same type can be assumed for them. Type-of-usage analysis is inherent to Cobol because the language lacks a strong type system. The shown component performs the analysis on the basis of `MOVE` statements (cf. rules [1], [2]) and `SET` statements (cf. rules [3], [4]). All the rewrite rules follow the same pattern. Given a `Seed` set, we are looking for data names that are used in pattern that involve `Seed` names. If we uncover such data names, then they are added to `Accu`. There are two rules per statement forms to deal with two possible memberships of operands in `Seed`. Type-of-usage analysis is completely fundamental. We place such components in a library of reusable components.

Borderline-case extension In [KLV02a], we report on a project for data expansion that consists of a dozen components. We need to expand data types and we also have to expand affected literals in the code. The latter activity is exemplified with the component for borderline-case extension. Here we assume that the upper boundary of some data range is normally used in conditions such as comparisons to check for validity. Hence, if the data range was extended, the tests using the maximum value have to be rephrased accordingly. The rules [1] and [2] cover the relevant patterns of relational expressions. The rule [3] for the helper function `expand´` performs the actual replacement of literals based on the passed arguments `MaxOld` and `MaxNew` for the old and the new maximum. The final rule [4] states that literals other than `MaxOld` are preserved.

This inventory of a few typical components is concluded with the observation that underlies the present paper:

> *A component for system analysis and transformation* contracts *only a small part of the Cobol language. Basically, the* patterns *used in the rewrite rules of a component constitute a contract. For the sample components, we show these small pattern contracts in figures 5–8. The majority of language constructs is irrelevant for a given component. This also generalises to non-trivial applications that consist of many components.*

```
[1] empty(Statement*1 Statement*2) = False
    ================================================================================
    eliminate(Statement*1 CONTINUE Statement*2) = eliminate(Statement*1 Statement*2)

[2a] eliminate(IF Condition Statement*1 CONTINUE Statement*2 ELSE Statement*3 END-IF) =
     eliminate(IF Condition Statement*1 Statement*2 ELSE Statement*3 END-IF)

[2b] eliminate(IF Condition Statement*1 ELSE Statement*2 CONTINUE Statement*3 END-IF) =
     eliminate(IF Condition Statement*1 ELSE Statement*2 Statement*3 END-IF)
```

Figure 1. A component for `CONTINUE` elimination

```
[1] replace(Statement*1 NEXT SENTENCE Statement*2) = replace(Statement*1 CONTINUE)
```

Figure 2. A component for `NEXT SENTENCE` to `CONTINUE` conversion

```
[1] Id1 ∈ Seed = true
    ================================================
    propagate(MOVE Id1 TO Id2, Seed, Accu) = Accu ∪ Id2

[2] Id2 ∈ Seed = true
    ================================================
    propagate(MOVE Id1 TO Id2, Seed, Accu) = Accu ∪ Id1

[3] Id1 ∈ Seed = true
    ====================================
    propagate(SET Id1 TO Id2) = Accu ∪ Id2

[4] Id2 ∈ Seed = true
    ====================================
    propagate(SET Id1 TO Id2) = Accu ∪ Id1
```

Figure 3. A component for type-of-usage analysis

```
[1] Id0 ∈ Affected = true,
    expand'(Dec0, MaxOld, MaxNew) = Dec1
    ================================================================
    expand(Dec0 Rel0 Id0, Affected, MaxOld, MaxNew) = Dec1 Rel0 Id0

[2] Id0 ∈ Affected = true,
    expand'(Dec0, MaxOld, MaxNew) = Dec1
    ================================================================
    expand(Id0 Rel0 Dec0, Affected, MaxOld, MaxNew) = Id0 Rel0 Dec1

[3] expand'(MaxOld, MaxOld, MaxNew) = MaxNew

[4] Dec == MaxOld = false
    ================================
    expand'(Dec, MaxOld, MaxNew) = Dec
```

Figure 4. A component for borderline-case extension

```
"CONTINUE"                                                -> Statement
"IF" Condition Statement* "ELSE" Statement* "END-IF" -> Statement
```

Figure 5. A pattern contract for `CONTINUE` elimination

```
"CONTINUE"        -> Statement
"NEXT" "SENTENCE" -> Statement
```

Figure 6. A pattern contract for `NEXT SENTENCE` to `CONTINUE` conversion

```
"MOVE" Identifier "TO" Identifier -> Statement
"SET"  Identifier "TO" Identifier -> Statement
```

Figure 7. A pattern contract for type-of-usage analysis

```
Arithmetic-expression Relational-operator Arithmetic-expression -> Relation-condition
Numeric                                                         -> Arithmetic-expression
Identifier                                                      -> Arithmetic-expression
```

Figure 8. A pattern contract for borderline-case extension

3 Parsing in an industrial context

It is possible to rapidly recover high-quality base-line grammar specifications from language references and other artifacts [LV01b, LV01a]. In an industrial context, the actual implementation of parsers for system analysis and transformation poses extra challenges: implementability, reliability, predictability, flexibility, maintainability or simply *scalability*. A discussion of these concerns will utterly clarify the contribution of the paper.

Technology premises The requirements that shape projects for system analysis and transformation address the languages to be used for tool development and the parsing technology. In our case, we committed ourselves to the Meta-Environment [B$^+$01] technology. This is a good fit for system analysis and transformation because the Meta-Environment integrates parsing, rewriting, and pretty-printing. As for parsing, the Meta-Environment supports a powerful syntax definition formalism and (scannerless) generalised LR parsing (SGLR; [Vis97]). Components for analysis and transformation are developed using a simple and readable executable specification language for rewriting that supports concrete syntax, that is, Cobol syntax is used in the rewrite rules. There are further provisions that make appear the Meta-Environment as a good choice, e.g., its consequent use of an interchange format for component integration, or its support for layout preservation. However, we will identify some current drawbacks of our technology premises below.

Implementation barriers When deriving a grammar implementation from a grammar specification or a language reference implementation, one has to overcome barriers that are either related to the high-level of the specification, or to shortcomings of the chosen technology. Both issues are illustrated by the Cobol artifact of abbreviated combined relation conditions (cf. Fig. 9). The fact that the construct is hard to define challenges any technology. In fact, SGLR saves us here to some extent because the absence of grammar class restrictions make an executable specification more promising. On the other hand, the need for type-directed parsing (see the discussion at the top of the figure) clashes with SGLR while corresponding tweaking is not difficult with other technology. Another, more general implementation barrier is triggered by the SGLR parsing regime which is deliberately 'picky' about ambiguities. To give an example, there are 35 productions for 'conditional' statements in Ansi 85 Cobol. They all cause one or more 'dangling-else' problems. With SGLR, the resulting ambiguities need to be resolved by the smart grammar implementor.[2] With less advanced parsing technology, the 'prefer-shift-over-reduce' rule of a non-generalised LR parser or the 'greediness' of a top-down parser resolves this issue without much ado.

Scalability Depending on the grammar format and on the language cocktail (Cobol + SQL + CICS + ...), a Cobol grammar might easily consist of as much as 1000 productions. Non-trivial Cobol portfolios are measured in million lines of code. A transformation framework will normally involve dozens to hundreds of components. These figures pose several scalability challenges for parsing technology — some of them being less obvious:

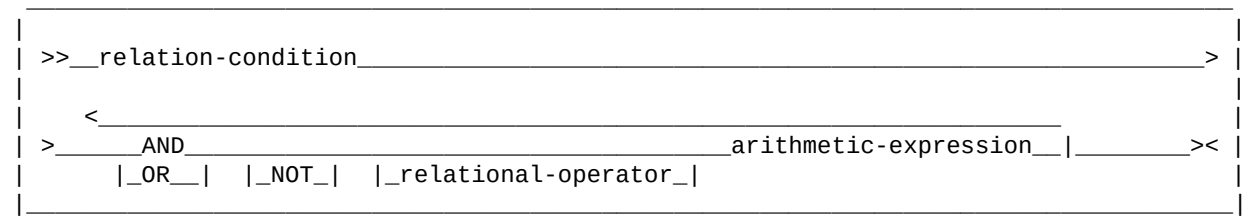
[2] Currently, the Meta-Environment is being extended with the ability to define equations that resolve ambiguities.

Examples

```
IF A = B OR [> C] ... expands to IF A = B OR [A > C] ...
IF A = B OR [C] ...   expands to IF A = B OR [A = C] ...
```

Note that the second expansion can only be approved on the basis of the type of C. If C is a data name, the expansion is correct. If C was a condition name, we are faced with an ordinary conjunction of a relation condition and a 'condition name condition'. This adds to the mere complexity of rules that govern the proper format of the expression form (see one rule below).

The basic format



An informal side condition (1 out of 12; quoted from [IBM93a])

"When a left parenthesis appears immediately after the relational operator, the relational operator is distributed to all objects enclosed in the parentheses. In the case of a "distributed" relational operator, the subject and relational operator remain current after the right parenthesis which ends the distribution. The following three restrictions apply to cases where the relational operator is distributed throughout the expression:

a. *A simple condition cannot appear within the scope of the distribution.*

b. *Another relational operator cannot appear within the scope of the distribution.*

c. *The logical operator NOT cannot appear immediately after the left parenthesis, which defi nes the scope of the distribution."*

Figure 9. A nightmare: abbreviated combined relation conditions in Cobol

1. Development effort
   (a) Grammar implementation
   (b) Coverage of a new portfolio
2. Make effort
   (a) Parser generation
   (b) Component compilation
3. Program efficiency
   (a) Parser
   (b) Component

Cobol grammar implementation (cf. (1.a)) with SGLR suffers from scalability problems regarding the overall parsing regime with its treatment of ambiguities as discussed above. This is an issue because of the size of the Cobol grammar and the complexity of some Cobol constructs. Coverage of a new portfolio (cf. (1.b)) by another client or even using another dialect triggers continuous grammar adaption and hence grammar implementation effort as just discussed. This is a general problem with any parsing approach that assumes a full grammar specification. With SGLR, there is the additional concern that ambiguities might pop up for any new program that is parsed. As for make effort, let us first state why the time for parser generation (cf. (2.a)) is an issue anyway. Whenever the underlying grammar evolves, a debug cycle is needed in order to test and to disambiguate the grammar. Parser generation is too slow in this context (see Sec. 6 for figures). The same holds for component compilation (cf. (2.b)) because component compilation actually involves parser generation as a consequence of concrete syntax in rewriting. Recall that we deal with *many* components in our setting, and that the development and maintenance of a component requires a debug cycle and hence repeated compilation. As for the compiled parsers and components ((3.a) and (3.b)), the speed of parsing and rewriting is sufficient in our setting. SGLR is known to be outperformed by less advanced parsing technology but the time that parsing takes is not a bottleneck in our setting.

To summarise the section:

> *The merits of a given technology compete with other requirements in the broad context of tool development. There is usually a trade-off between generality, efficiency, maturity, support, complexity, and others. Our technology premises provide good provision for parsing and component-based development in general, but our setting asks for improved scalability regarding development and make effort.*

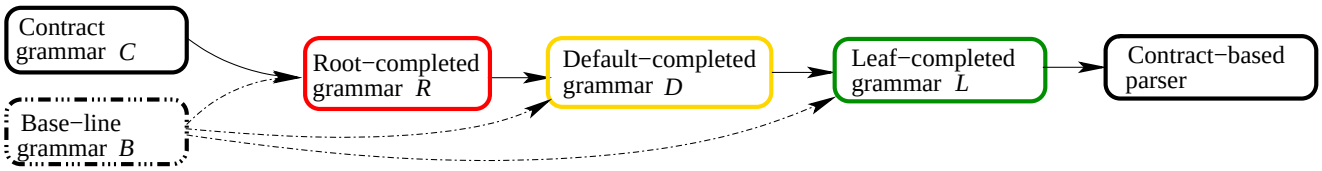


Figure 10. Stepwise derivation of a tolerant parser from a contract and an optional base-line grammar

4 Contract-based parser design

Contract-based parsing radically improves scalability within the scope of our technology premises. Hence, can refrain from toggling these premises that are valuable for numerous other reasons. Our approach is to build parsers that focus on the contracts of their client components while they make few assumptions regarding the remaining language syntax. Basically, such a contract-based parser can be specified in terms of the contracted productions plus 'skeleton productions' that ensure the global consistency of parsing. This style exhibits several merits. Contract-based parsers are more resistant to changing grammars. One is not even obliged to make a reasonably complete base-line grammar available but it can be accumulated over time. Furthermore, contract-based parsers are tolerant to cope with different dialects. Also, a contract-based parser is very simple just in terms of the number of productions and the depth of parse trees. Hence, compiling such a parser is done much faster. Also, the simpler structure immediately shortcuts all kinds of implementation barriers. In particular, disambiguation is much less of an issue if at all. In the sequel, we describe how to design contract-based parsers.

A process for parser development The process (see Fig. 10) starts from a contract grammar C — normally a few productions (recall the examples from figures 5–8). The process also refers to a base-line grammar B which might be actually available or which might just come in the form of an informal language reference or other knowledge about the intended language. Depending on this, the entire process of parser development can be more or less automated based on grammar transformations [Läm01, LW01, KLV02b]. The first step of extending the contract grammar C is 'root completion'. That is, C is put into the right upper context. The resulting root-completed grammar R is further extended by 'default completion'. That is, we liberalise the definition of nonterminals as to allow for parsing alternative constructs. The resulting default-completed grammar D is further extended by 'leaf completion'. That is, all the 'holes', say occurrences of undefined nonterminals are resolved. The resulting leaf-completed grammar L is the input for a rather direct parser implementation. We will now discuss the steps to obtain the grammars R, D, and L from a given contract grammar C in more detail.

Root completion Given a contract grammar C, productions are added that are eventually needed to reach the contract productions from the root, say, the start symbol. These added productions are called 'skeleton productions' because they are meant to enforce the right shape of the parse tree needed for contractual matches. To give an example, in Fig. 11, we show the few skeleton productions that are triggered by contracts at the Cobol statement level as, for example, by the `MOVE` and `SET` patterns in the case of the type-of-usage analysis from Fig. 3. There are productions to place statements in the context of Cobol sentences, in turn inside paragraphs and sections, and finally, inside the procedure division of a complete Cobol program. The derivation of the root-completed grammar R is made precise in Fig. 12.[3] For an automated root completion, we have indeed to assume that a base-line grammar B is available. In a manual approach, we would recover the required piece of syntax on demand. One might even think of using different base-line grammars, for example, a very simple one that does not pay attention to the complex nested structure of Cobol statements, and another one which is used whenever the precise structure is an issue. The shown algorithm is completely general (i.e., no side conditions, no heuristics), and is trivially implemented. It effectively minimises the number of base-line productions that we are going to deal with in the final parser. The contextual obligations enforced by root completion are meant to rule out false matches of the contract patterns.

[3] **Context-free grammar notation:** A context-free grammar is a tuple $\langle N, T, P, S\rangle$ with nonterminals N, terminals T, productions P, and a set of start symbols $S \subseteq N$. We need a set of start symbols because for contract grammars we need to express what the various 'entry points' are as opposed to nonterminals that only play a role in nested positions of patterns. A context-free production p is of the form $n \to u$ with $n \in N$ and $u \in (N \cup T)^*$. We use the following notation to inquiry a grammar G following [Läm01]:

- $\mathcal{T}(G)$ —terminals in G
- $\mathcal{N}(G)$ —nonterminals in G
- $\mathcal{D}(G) \subseteq \mathcal{N}(G)$ —nonterminals with defining productions.
- $\mathcal{B}(G) \subseteq \mathcal{N}(G)$ —used nonterminals without defining productions.

```
Id-division? Env-division? Data-division? Proc-division? End-program? -> Program
"PROCEDURE" "DIVISION" Using-phrase Declaratives? Sections           -> Proc-division
Paragraphs Section*                                                  -> Sections
Section-header "." Paragraphs                                        -> Section
Sentence* Paragraph*                                                 -> Paragraphs
Paragraph-name "." Sentence*                                         -> Paragraph
Statement+ "."+                                                      -> Sentence
```

Figure 11. Skeleton productions for statement contracts

- Input:
  - Base-line grammar $B = \langle N_B, T_B, P_B, \{s\}\rangle$
  - Contract grammar $C = \langle N_C, T_C, P_C, S_C\rangle$ with $P_C \subseteq P_B$
- Output:
  - Root-completed grammar $R = \langle N_R, T_R, P_R, S_R\rangle$
- Algorithm:
  1. Initialisation $R := C$
  2. Repeat steps (a)–(d) until a fi x-point is reached.
     (a) Pick a production $p \in P_B \setminus P_R$ where p is of the form $n \to u\, n'\, v$, and $n' \in S_R$ (i.e., an 'entry point' is reached) and there is a derivation of the following form
     $$s \Rightarrow_B \cdots \Rightarrow_B x\, n\, y \overset{p}{\Rightarrow}_B x\, u\, n'\, v\, y.$$
     (b) Add p to P_R.
     (c) Add n to S_R.
     (d) Extend N_R and T_R accordingly.
  3. $S_R := \{s\}$ (i.e., 'root' reached)

Figure 12. Automated root completion

Default completion We now look at all the nonterminals *defined* in R as opposed to nonterminals with lacking definitions, and we liberalise the definitions as to allow parsing of patterns other than those admitted by the skeleton and contract productions. We prefer to assume user-provided skip rules per defined nonterminal. There are the following forms ω of skip rules:

- Skip until $T \subseteq T_B$ including T, i.e., skip $(T_B \setminus T)^*\, T$.
- Skip until $T \subseteq T_B$ excluding T, i.e., skip $(T_B \setminus T)^*$.
- Skip everything, i.e., ${T_B}^*$.

These forms suffice in practice but one could think of a more general definition of ω. Skip rules of the first two kinds are best hand-crafted but this is not difficult. Default completion is made precise in Fig. 13. Note that we use right-recursive helper productions to encode the regular expressions "$(T_B \setminus T)^*\, T$", "$(T_B \setminus T)^*$", and "${T_B}^*$" from above. The shown algorithm is very simple again. As an aside on implementation, to avoid a trivial kind of ambiguous parsing, we have to enforce a lower priority for these

- Input:
  - Base-line grammar $B = \langle N_B, T_B, P_B, S_B\rangle$
  - Root-completed grammar $R = \langle N_R, T_R, P_R, S_R\rangle$
  - A set Γ of skip rules each of the form $n \to \omega$. (There is at most one skip rule per $n \in \mathcal{D}(R)$.)
- Output:
  - Default-completed grammar $D = \langle N_D, T_D, P_D, S_D\rangle$
- Algorithm:
  1. Initialisation $D := R$, $T_D := T_B$
  2. For all $n \in \mathcal{D}(R)$ add productions as follows: (In all cases, n' is a fresh nonterminal to be included subsequently into N_D.)
     (a) Case $n \to$ 'Skip until incl. T' $\in \Gamma$:
     $- n \to n'$
     $- n' \to t$ — one for each $t \in T$
     $- n' \to t\, n'$ — one for each $t \in {T_B} \setminus T$
     (b) Case $n \to$ 'Skip until excl. T' $\in \Gamma$:
     $- n \to n'$
     $- n' \to \epsilon$
     $- n' \to t\, n'$ — one for each $t \in {T_B} \setminus T$
     (c) Case $n \to$ 'Wild card' $\in \Gamma$:
     $- n \to n'$
     $- n' \to \epsilon$
     $- n' \to t\, n'$ — one for each $t \in {T_B}$
     (d) Case $\nexists \omega.\ n \to \omega \in \Gamma$: Add nothing.

Figure 13. Algorithm for default completion

'default productions'. This is taken care of when mapping the derived grammar to an actual parser specification. To illustrate default completion, we show prime examples of reusable skip rules for Cobol in Fig. 14. The first three skip rules deal with the most obvious units of code in a Cobol program, namely statement sentences in the `PROCEDURE DIVISION` and data description entries in the `DATA DIVISION`. The other skip declarations cover the remaining divisions. As one can see, the amount of skip declarations is very small despite the large base-line grammar for Cobol. In practice, we do not specify any further skip rules than those from the table. To make default completion more automatic, we should assume a refined case (d) in Fig. 13 to handle nonterminals without an explicitly assigned skip rule differently. That is, if the nonterminal is defined by several

productions in B, then default completion according to the wild-card skip rule should be assumed.

Nonterminal	**Stop token**
Sentence	incl. period
Statement	excl. period and statement verb
Data description entry	incl. period
File and sort description entry	incl. period
File control entry	incl. period
Identifi cation division	excl. ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION
Confi guration section	excl. INPUT-OUTPUT SECTION, DATA DIVISION, and PROCEDURE DIVISION

Figure 14. Skip rules for Cobol

- Input:
  - Base-line grammar $B = \langle N_B, T_B, P_B, S_B \rangle$
  - Default-completed grammar $D = \langle N_D, T_D, P_D, S_D \rangle$
  - A set Γ of skip rules each of the form $n \to \omega$. (There is at most one skip rule per $n \in \mathcal{B}(D)$.)
- Output:
  - Leaf-completed grammar $L = \langle N_L, T_L, P_L, S_L \rangle$
- Algorithm:
  1. Initialisation $L := D$
  2. For all $n \in \mathcal{B}(L)$ add productions as follows:
     (a) Case $n \to$ 'Skip until incl. T' $\in \Gamma$:
     $-\ n \to t$ — one for each $t \in T$
     $-\ n \to t\,n$ — one for each $t \in T_B \setminus T$
     (b) Case $n \to$ 'Skip until excl. T' $\in \Gamma$:
     $-\ n \to \epsilon$
     $-\ n \to t\,n$ — one for each $t \in T_B \setminus T$
     (c) Case $n \to$ 'Wild card' $\in \Gamma$:
     $-\ n \to \epsilon$
     $-\ n \to t\,n$ — one for each $t \in T_B$
     (d) Case $\nexists \omega.\ n \to \omega \in \Gamma$: Assume 'Wild card'.

Figure 15. Algorithm for leaf completion

Leaf completion It remains to define the bottom nonterminals of D denoted by $\mathcal{B}(R)$ (i.e., all nonterminals that are used but not defined [Läm01]). This delivers the leaf-completed grammar L. For a suggestive terminology, the occurrences of bottom nonterminals will be called 'holes', and the parts of a parse tree that correspond to these holes will be called 'cut-off trees'. The definitions of bottom nonterminals can simply be defined via the forms of skip rules as introduced above for default completion. In fact, default completion and leaf completion follow the same scheme but we separate them to differentiate between liberalisation of existing definitions vs. the trivial definition of bottom nonterminals. In the former phase, liberalisation is option whereas in the latter phase, the definition by a wild card is the common case. This is clarified in Fig. 15. Unless requested differently via the skip rules, leaf completion establishes that the set of strings derivable from a bottom nonterminal coincides with ${T_B}^*$. This approach is obviously 'complete' in the sense that whatever the definition of n is in B, at least all the strings derivable via B are derivable via L. It is easy to see that this step only vacuously relies on the actual availability of the base-line grammar because the required T_B could also be delivered in different ways than by extraction from an actual grammar. Most importantly, this step is the key to isolation from irrelevant structural details. The input for holes is simply skipped.

To summarise the section:

> *Root completion is the key to identifying proper input fragments for contract pattern matching. Default completion is the key to skipping over input that does not admit any contract pattern matching. Leaf completion is the key to skipping over input with non-contracted structure. The full process provides us with safe tolerance based on simply-structured grammars that are easily implemented. As for safety, the implicit assumption is here that the skeleton productions sufficiently split up the input to navigate to the contracted parts of the input. Cobol, but also many 4GLs, make this assumption robust because of their 'keyword-intensive' productions.*

5 Related work

Grammars as contracts In [JV00], it is argued elaborately that grammars should be viewed as contracts for component-based development of language tools where 'component' stands for parsers, pretty printers and tree processors. It is strikingly obvious that grammars serve as a kind of contract in this setting, in the same sense as interface descriptions constrain well-typed modular programs. The authors further discuss the combined use of concrete and abstract syntax, and they argue that the 'grammars as contracts' view disciplines meta-tooling: parsers, pretty printers, APIs, and concrete / abstract mediators can be generated from the contracts — this also has been called 'generic language technology' for long.

Our notion of contract does not start from a grammar but from a component. In our setting, 'component' stands for rewriting components for system analysis and transformation. Our contracts are meant to capture the patterns that are used by a component as opposed to a complete gram-

mar in the above setting. A crucial insight is here that rewriting components clearly exhibit the problem-specific patterns due to the virtue of generic traversal with traversal functions [BKV01]. Our contracts leave the context of the relevant patterns and the details of the unmatched subtrees unspecified. A contract-based parser refines the underlying contract in the sense that contract grammars are completed into grammars for parsing complete input programs. The whole purpose of this process is to get simple and tolerant parsers. This is not an issue in 'grammars as contracts'.

Fuzzy parsing In [Kop97], the notion of fuzzy parsing is systematically developed. These parsers perform syntactical analysis on selected portions of the input, mostly for the purpose of the extraction of a partial source code model. The key idea is here to identify 'anchor terminals' that trigger the application of context-free productions. The input is skipped until an anchor a is found, and then context-free analysis is attempted using a production starting with a. Extra idioms can be used to perform error-handling, e.g., if inconsistent source files must be analysed, e.g., in an editing session. Fuzzy parsing does not employ a base-line grammar as a point of reference. In particular, there is no correspondence to our root completion which can be viewed as an alternative to the very basic anchor approach. More generally, with fuzzy parsing, input is skipped by a lexer until an anchor is found while with contract-based parsing, the control is never withdrawn from the parser.

Island parsing A potent refinement of the fuzzy parsing notion is island parsing [DK99, Moo01, Moo02].[4] Island grammars are structured as follows. At the top, the input is viewed as a list of 'chunks' with optional layout (say, white spaces; see module Layout in Fig. 16) in between. A chunk is either 'water', that is, everything we are not interested in, or a problem-specific chunk very much in the sense of our contract patterns. Water is defined in a way that it can potentially consume any string except layout but its definition is prioritised such that other chunks will be preferred over water (see module Water in Fig. 16; see the attribute avoid). Problem-specific patterns are simply injected into the sort Chunk (see the context-free structure of data description entries in module DataFields of Fig. 16: a level followed by a data name). Chunks can be nested to specify boundaries of certain regions in the input (like "the puddle on a little island in a lake on a bigger island in an ocean"; see the module DataFieldsWithContext that refines module DataFields by working out a differentiation of top-level Chunk and DdChunk). Island grammars

[4] The authors only use the term island *grammar* but not island *parser*. We prefer to normalise the terminology even if this might cause a clash with a similar approach in natural language processing.

```
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
  [A-Z][A-Z0-9\-]* → DataName
  [0][1-9] → Level
  [1-4][0-9] → Level

module DataFields
imports Water DataParts
context-free syntax
  Level DataName → Chunk {cons(Data)}

module DataFieldsWithContext
imports Water DataParts
context-free syntax
  "DATA DIVISION"
  DdChunk*
  "PROCEDURE DIVISION" → Chunk
  Level DataName → DdChunk {cons(Data)}
  Water → DdChunk
```

Figure 16. An island grammar for selecting data fields adopted from [Moo02]

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 style does immediately lead to very tolerant parsers. The prime differences between island and contract-based parsing are the following. Firstly, island parsing does not involve a base-line grammar as point of reference. Secondly, contract-based parsing is predominantly context-free whereas island parsing amalgamates lexical and context-free analysis rather heavily (see the lexical definition of Water in Fig. 16 which tends to compete with the several tokens of problem-specific forms of chunk).

Degrees of context-free parsing In Fig. 17, we place various approaches on a chart regarding their relative position in between lexical analysis and conservative context-free analysis. Fuzzy parsers involve a rather basic lexical criterion to switch to the context-free mode. Island parsers can skip over input and mix lexical vs. context-free style in

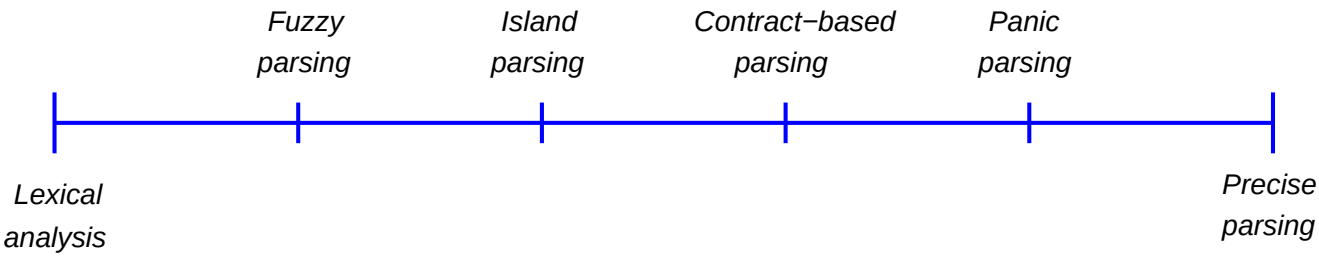


Figure 17. A spectrum of approaches for source code analysis

more sophisticated ways. Still the islands are found in plain lists of chunks. The way how the boundary of islands is normally defined (cf. the two `DIVISION` keywords at the bottom of Fig. 16) remind us of the anchors in fuzzy parsing approach. Contract-based parsers use ordinary context-free productions to split up the input and to parse fragments of interest. Lexical skips only occur to cut-off precise parsing in the interest of tolerance. A standard technique in parser implementation is panicking using stop symbols [ASU86]. Such panic parsers are even a bit more context-free than our parsers. Lexical skips only occur for recovery from parser errors in the interest of tolerance.[5]

We save some concluding remarks regarding this inventory for the paper's final section.

6 Parser implementation

We have applied our process for parser development to Cobol where we use a base-line grammar for VS COBOL II [LV99]. This grammar was recovered earlier from IBM's industry standard [IBM93a, IBM93b] using other grammar engineering techniques as reported in [LV01b]. In fact, we automated the process using grammar transformations [Läm01, LW01] as supported by the Grammar Deployment Kit (GDK) [KLV02b].[6] Using GDK, we can generate parsers based on different parsing technology but our focus was on making the process work for (scannerless) generalised LR parsing (SGLR) as available in ASF+SDF. Extra experiments were necessary to approve the only modest technology dependency of the approach. We will first discuss the potential issues in going from the completed grammar to a parser implementation. We will then discuss benchmarks.

Conflicts and ambiguities We do not expect our parsers to adhere to any strict grammar classes such as LL(1) and LALR(1). This will not be sensible because we do not even expect the base-line grammar to be that restricted. However, we expect that our scheme of grammar completion should not introduce new ambiguities. In fact, this is systematically avoided by assigning a lower priority to productions that were newly introduced during default and lead completion. In bottom-up parsing, the relative priorities of productions can usually be specified directly. In top-down parsing, we can rely on the order of productions. Ambiguities could also arise because of the liberal definitions of bottom nonterminals which might lead to an overlap of alternative productions. We did not experience the problem in our Cobol project. Finally, ambiguities might arise because sibling holes compete for input to be skipped. This problem can be eliminated by a simple grammar transformation that fuses sibling holes.

False positives and negatives The former could happen if our context-free skeletons 'lack bones', say synchronisation tokens such as keywords. This is again not at all a problem with the Cobol grammar but we have to admit that the strength of the skeleton somewhat depends on the actual format of the productions. Keywords only make into the skeleton if they occur directly in the skeleton productions in the sense of root completion. We plan to design an analysis which spots potential problems in this sense, and a normalisation to minimise format sensitivity. False negatives are not an issue because the completed grammar is at least as tolerant as the underlying base-line grammar which is the only trustworthy reference we have available.

Performance degradation The only non-standard feature of our completed grammars is that they involve skip rules with the 'wild card' option being the most challenging one. In a naive reading, this style might create 'black holes' which attempt to eat up all the input. In fact, with SGLR we do not experience any performance degradation. Using simpler parsing technology, this concern deserves special treatment. If we think of a backtracking parser, then the black holes will eventually be triggered to emit most of the input so that it can be re-inserted into the input buffer. It is

[5] Panicking is normally meant for recovery from plain syntax errors. Although panicking is used in commercial source code analysers, we are not aware of any published exposition of the panic technique for designing tolerant source code analysers.

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

Grammar	Productions	LOC	Keywords	Parser generation	Parse table
Simple statement skeleton	51	209	82	1.15 s	205 KB
Nested statement skeleton	268	438	129	21.87 s	836 KB
Base-line grammar	888	1228	325	220.75 s	2.53 MB

Figure 18. Some benchmark results for contract-based parsing

however possible to enforce a non-greedy expansion regime on the basis of a suitable encoding scheme. Then, the completed grammar is executable with top-down and bottom-up parsing with backtracking without further ado.

Benchmarking In Fig. 18, we compare some figures of two different contract-based parsers with a full Cobol grammar. We do not include actual parsing benchmarks because they all run at very similar speed. We show code size figures: number of productions, lines of code, and number of keywords for the generated SDF specification. We also show make figures: the time spent for parser generation, and the size of the generated parse table. The grammar 'simple statement skeleton' basically corresponds to the few rules in Fig. 11. The higher number of productions is implied by the inclusion of the productions for lexical sorts, preprocessing statements and some forms that were ignored in Fig. 11 for simplicity. Components that operate at the level of simple statements as opposed to nested statements can contract this grammar. The grammar 'nested statement skeleton' is much more precise than 'simple statement skeleton' because the proper nesting structure of compound statements is enforced. At that level we have enough structure to perform all kinds of control-flow analysis. Both skeleton grammars provide a good approximation of the typical contracts in system analysis and transformation. The base-line grammar is our recovered VS Cobol II grammar according to [LV99, LV01b] after some tweaking for SGLR.

Recovered scalability reads like this:

> *The time for parser generation differs by a factor of 10–100. Recall that this factor is relevant for any component compilation. Also, recall that speed of parser generation comes in combination with improved tolerance of parsing and decreased grammar development effort.*

7 Conclusion

Contribution Contract-based parsers are tolerant and simple. 'Tolerant' means that they accept inputs that use unanticipated phrases in the sense of dialects, syntax errors, and embedded languages. 'Simple' means that the number of productions is typically by factor 3–15 smaller when compared to the ultimate base-line grammar, and that the parsers overcome implementation barriers (cf. 'scalability'). We have applied our approach to Cobol — this is not a narrow focus [LV01a]. Compared to previous work on panicking, fuzzy, and island parsing the following shift of focus and added value can be pointed out:

- context-free skeletons for regions of interest,
- reusable base-line grammar productions, and
- grammar completion by transformation.

Why context-free skeletons matter The use of context-free rules for the navigation to the regions of interest improves clarity and safety of matching. With island parsing, the islands and their boundaries are defined without direct reference to the language structure. The running example in [Moo02] (see Fig. 16) illustrates this concern: the pattern of data field declarations is described in terms of its lexical appearance 'Level DataName'. The first attempt to use this pattern according to the module DataFields turns out to be by far too greedy. So a revised module had to be defined as to rule out false matches on the basis of a `DATA DIVISION ... PROCEDURE DIVISION` boundary. Even this ad-hoc revision admits false matches, e.g., '`42 TIMES`' in an `OCCURS` clause. This specific problem could be fixed by rejecting all reserved Cobol words from DataName.

Why reusable base-line grammar productions matter We assume the accumulation of stable base-line productions over time. Their reuse in component development is the key to *composability* of components. Our contracts are immediately composable because contracts are proper subsets of the base-line grammar. With island parsing, the focus is on rapid parser development for specific purposes. When ad-hoc productions for islands are designed, the corresponding parse trees are not interchangeable between different components. The running example in [Moo02] (see Fig. 16) again illustrates this concern: the pattern for data field declarations only deals with levels and names but there is no provision for the 'data description entry clauses' (think of `PIC`, `OCCURS`, `VALUE`, `REDEFINES`, etc.) that will be needed for many components.

Why grammar transformations matter We use grammar transformations to 'compute' the ultimate parser specification from the contract and the base-line grammar. This

supports conciseness and declarativeness as opposed to a situation where a complete and idiosyncratic parser specification is provided by the component developer. In fact, the transformation engine can optimise the process for the targeted parsing technology. With island parsing, one tends to write less declarative specifications. The nesting of chunks and thereby patterns is expressed in terms of a global naming regime for kinds of chunks (see Chunk vs. DdChunk in Fig. 16). Also, island grammars already anticipate a specific operational model, namely scannerless LR parsing with certain disambiguation idioms (cf. avoid in Fig. 16).

Perspective In our current implementation, we use simple grammar transformation scripts, a simple tool to export grammars to parser generators, and make files to automate the synthesis of contract-based parsers. We envisage a domain-specific language for tolerant parsers with designated constructs for contracts, base-line productions, skip rules, and others. We also plan to define dedicated properties of our grammars such as the 'strength' of a skeleton.

Acknowledgement We are grateful to Mark van den Brand, Paul Klint, Jan Kort, Niels Veerman, Chris Verhoef, and Ernst-Jan Verhoeven for discussions on the overall subject of the paper.

References

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

[B+01] M. van den Brand et al. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction 2001 (CC 2001)*, volume 2027 of *LNCS*. Springer-Verlag, 2001.

[BHK89] J. A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In *Algebraic Specification*, chapter 1, pages 1–66. The ACM Press in cooperation with Addison-Wesley, 1989.

[BKV01] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, CWI, July 2001.

[BSV97] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proc. of Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997.

[BSV00] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000.

[DK99] A. van Deursen and T. Kuipers. Building Documentation Generators. In *Proc. of Int. Conf. on Software Maintenance*, 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.

[IBM93a] IBM Corporation. *VS COBOL II Application Programming Language Reference*, 4th edition, 1993. Publication number GC26-4047-07.

[IBM93b] IBM Corporation. *VS COBOL II Reference Summary*, 1993.

[JV00] M. de Jonge and J. Visser. Grammars as Contracts. In *Proc. of Generative and Component-based Software Engineeringi (GCSE)*, volume 2177 of *LNCS*, pages 85–99, Erfurt, Germany, October 2000. Springer-Verlag.

[KLV02a] S. Klusener, R. L¨ammel, and C. Verhoef. Architectural Modifications to Deployed Software. Submitted for journal publication, June 2002.

[KLV02b] J. Kort, R. L¨ammel, and C. Verhoef. The Grammar Deployment Kit. In M. van den Brand and R. L¨ammel, editors, *Proc. of LDTA'02*, volume 65 of *ENTCS*. Elsevier Science, April 2002.

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

[L¨am01] R. L¨ammel. Grammar Adaptation. In J.N. Oliveira and P. Zave, editors, *Proc. of Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[LV99] R. L¨ammel and C. Verhoef. *VS COBOL II grammar Version 1.0.3*, 1999. Available at: http://www.cs.vu.nl/˜x/grammars/vs-cobol-ii/.

[LV01a] R. L¨ammel and C. Verhoef. Cracking the 500-Language Problem. *IEEE Software*, pages 78–88, November/December 2001.

[LV01b] R. L¨ammel and C. Verhoef. Semi-Automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[LW01] R. L¨ammel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In M. van den Brand and D. Parigot, editors, *Proc. of LDTA'01*, volume 44 of *ENTCS*. Elsevier Science, April 2001.

[Moo01] L. Moonen. Generating Robust Parsers using Island Grammars. In *Proc. of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, October 2001.

[Moo02] L. Moonen. Lightweight Impact Analysis using Island Grammars. In *Proc. of the 10th International Workshop on Program Comprehension (IWPC 2002)*. IEEE Computer Society Press, June 2002.

[SSV02] A. Sellink, H. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. *Science of Computer Programming*, April 2002. To appear.

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