# Current Parsing Techniques in Software Renovation Considered Harmful

Mark van den Brand

*Department of Software Engineering, CWI*
*Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*

Alex Sellink*, Chris Verhoef

*University of Amsterdam, Programming Research Group*
*Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*

`markvdb@cwi.nl, alex@wins.uva.nl, x@wins.uva.nl`

## Abstract

We evaluate the parsing technology used by people working in the reengineering industry. We discuss parser generators and complete systems like Yacc, TXL, TAMPR, REFINE, CobolTransformer, COSMOS, and ASF+SDF. We explain the merits and drawbacks of the various techniques. We conclude that current technology may cause problems for the reengineering industry and that modular and/or compositional parsing techniques are a possible solution.

*Categories and Subject Description*: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring; D.3.4. [**Processors**]: Parsing.

*Additional Key Words and Phrases*: Reengineering, System renovation, Parsing, Generalized *LR* parsing, compositional grammars, modular grammars.

## 1 Introduction

A hardly controversial statement in the reengineering community is that in order to reengineer software it is convenient to parse it. Maybe due to the overall agreement on this issue, the question which parsing technology is the most suitable for reengineering purposes is not addressed. Indeed, there are well-established techniques for generating parsers from a grammar. They were developed with a focus on the fact that the grammar was known in advance and not likely to change continuously. After all, once a language is defined and a parser is generated that is the end of the story. The condition that the grammar of the language is known in advance is not satisfied in the field of reengineering. Let us first explain that this is the case and then what the consequences are in the realm of proven parser technology.

*Alex Sellink was sponsored by bank ABN AMRO, software house DPFinance, and the Dutch Ministry of Economic Affairs via the Senter Project #ITU95017 "SOS Resolver".

It is not the case that when, say COBOL, needs to be reengineered that we can use an existing grammar and then parse this COBOL source. For, there is a myriad of COBOL dialects, some of them are even home-brewed. Furthermore, there are many extensions to COBOL, for instance, embedded CICS, SQL, and/or DB2. This implies that there is a good chance we have to write a parser before we can start a reengineering project. To phrase it informally, it seems that in reengineering the *language* is the variable and the *problems* are the constants: the Year 2000 problem or the Euro conversion problem occur in virtually every language. Since the language is not known in advance the original condition of well-established parser technology is not satisfied in the field of reengineering. To make the problems a bit more concrete we quote from the German home page of Micro Focus [7]:

> Beim Einsatz von Parsern stoßen Unternehmen in der BRD zumeist auf zwei Probleme: Zum einen sind Parser fast ausschließlich für Programmiersprachen wie COBOL, PL/1 und Assembler erhältlich. Eine Entwicklung von Parsern für "Nischen"-Sprachen rechnet sich für die Tool-Hersteller in der Regel nicht. Zum anderen stammen fast alle gängigen Parsing-Tools aus den USA - sie kommen deshalb mit den Eigenheiten der Entwicklungsumgebungen in deutschen Rechenzentren nicht zurecht. Auf den "Zoo" von Entscheidungstabellen, Programmgeneratoren und Meta-COBOL-Sprachen, der in den vergangenen 20 Jahren hierzulande entstanden ist, sind sie nicht vorbereitet. Der von diesen Werkzeugen erzeugte "Meta"-Code wird von den gängigen Parsern nicht verstanden. Eine Migration dieses Codes zu "echtem", lesbarem COBOL-85-Code wäre oft zu risikoreich: Zum einen würden irreguläre Sprachelemente implementiert, zum anderen verhalten sich die verfügbaren Konverter oft nicht stabil. Drittens kommt hinzu, daß eine Migration zu COBOL-85 das Jahrtausendpro-

blem zunächst nur auf eine “modernere” Sprache verlagert - nicht aber löst.[1]

In the footnote we give the translation in English. Obviously, there is a great need for state-of-the-art parser technology to solve the problems like the ones addressed by Micro Focus.

A well-established technology for the generation of parsers is called *LR*-parsing. The most well-known implementations of this technology are the parser generator Yacc [25] and its GNU variant Bison. All these implementations share the property that the grammar that is specified in the above formalisms should be free of conflicts. We recall that there are two categories of conflicts. There is a so-called shift/reduce conflict, which means that there are two options, namely, an extra token can be shifted on the stack or a reduction can be made by application of a production rule. And there are reduce/reduce conflicts, meaning that two distinct reductions are possible. When defining a grammar using Yacc or Bison, it is necessary to remove the above conflicts. This is not a disaster when the language is fixed: as soon as the conflicts are resolved we are done. The problems occur when the grammar is continuously changing. It is not an exception that 50 conflicts arise as a consequence of adding a single grammar rule to a COBOL grammar. Some of these problems can be resolved by performing a retokenization in the preprocessing phase. This does not change the underlying structure of the grammar, which is an advantage. However, it means that the source code has to be transformed. The remaining problems can only be solved by modifying some context-free grammar rules. The effect of this is that the underlying structure of the grammar becomes more and more artificial, which is undesirable.

Removing a grammar rule is also a problem: apart from removing production rules also the changes to resolve conflicts should be removed: grammar rules and/or preprocessor fixes. This is quite some work and demands sophisticated version management, for instance using CVS [14]. In practice, such things do not happen: the legacy of a retokenization or grammar modification often remains in the grammar specification. In this way a grammar specification easily becomes an unmaintainable business-critical application that needs to be reengineered. Therefore, we consider current parser technology harmful when applied to reengineering purposes.

[1]Companies in Germany mainly encounter two problems in using parsers. Firstly, parsers are mostly only available for programming languages like: COBOL, PL/1 and Assembler. Normally, the development of parsers for non-mainstream languages is unprofitable for tool developers. Secondly, most popular parsing tools have been developed in the USA and therefore are incompatible with the peculiarities of the development environments used in german computer centers. They are not prepared for the myriads of decision tables, program generators and Meta-COBOL languages that have been developed in this country during the last 20 years. The meta-code that is generated by aforementioned tools cannot be interpreted by off-the-shelf parsers. The migration of this code into real legible COBOL-85 code often would be to risky. Firstly, it would mean that irregular elements of the language are implemented. Secondly, the available converter tools often are unstable. Thirdly, a migration to COBOL-85 would only transfer the Y2K-problem into a modern language to begin with, not solve it.

**Organization** In Section 2 we explain modularity and compositionality of grammar specifications—two important properties that are not present in current parsing technology but that are useful for reengineering. Then in Section 3 we evaluate the current situation of parser technology that is in use for reengineering purposes. In Section 4 we discuss a possible solution for the problems called Generalized *LR* parsing. This technique is implemented in the system that we use use for reengineering. It supports modularity and compositionality of grammar specifications. We discuss the background of the algorithm, we give an idea of the algorithm, and its efficiency. We discuss our experience using this technology to define COBOL grammars with extensions. Furthermore, we address the issue of reusability. In section 5 we give a real world example of grammar pollution that sooner or later leads to grammar reengineering. We show that our approach does not lead to such situations. Finally, in Section 6 we give our conclusions.

**Acknowledgements** We thank Ira Baxter (Semantic Designs), Joachim Blomi (Micro Focus), Tim Chou (Reasoning), Fred Hirdes (TechForce), Paul Klint (CWI/UvA), Vadim Maslov (Siber Systems), Patrick McParland (NORTEL), and Harry Sneed (SES Software-Engineering Service) for stimulating discussions and useful comments. Thanks to Gert Veltink (DeTeWeKommunikationssysteme) for translating the German text.

## 2 Modularity and Compositionality of Grammars

A well-known result in formal language theory is that if we combine two context-free grammars we obtain again a context-free grammar. Combination means here the set-theoretic union of the grammar rules. We call this *compositionality* of context-free grammars. Suppose we have an algorithm that is able to parse a certain subclass of context-free grammars. If that subclass is compositional we can use the algorithm on any combination of grammars of this subclass. This is an interesting property for reengineering purposes. If we wish to construct a COBOL/CICS grammar, we can simply obtain it by combining a COBOL grammar and a CICS grammar.

If a grammar can be constructed in more than one file using an import mechanism we call this property *modularity.* This is also a useful property in reengineering. In this way, it is possible build a repository of grammar components. So we can reuse grammar definitions. For a reengineering company it is convenient to have grammar components in a repository so that a parser for a specific reengineering project can be constructed on the fly. If we wish to construct it simply by importing a COBOL module and a CICS module.

We stress that the combination of compositionality and modularity is a desirable combination for reengineering purposes: modularity enables reuse and compositionality guarantees that the combined modules give rise to correct parsers. So if we import COBOL and CICS modules to obtain a COBOL/CICS grammar *and* the grammars are compositional, we can effortlessly generate a parser for them.

The problem with current parsing technology is that it is not compositional: most of the parser generators do not parse arbitrary context-free grammars but certain subclasses that are not closed under compositionality. They are mostly indicated with abbreviations like $LL$, $LL(1)$, $LL(k)$, $LR$, $LR(1)$, $LR(k)$, $LALR$, $LALR(1)$, $LALR(k)$, etc. We recall that the first $L$ in $LL$ or $LR$ stands for left-to-right scanning of input, and that the $R$ in $LR$ stands for rightmost derivation in reverse, the second $L$ in $LL$ stands for leftmost derivation, the $k$ stands for the number of input symbols of lookahead that are used. If there is no $k$ we assume $k = 1$. Finally, the $LA$ stands for lookahead and is synonym with the $(k)$ postfix, therefore it is sometimes omitted. For all these classes an efficient parsing algorithm exists. Therefore, we speak of proven technology: it is robust and efficient. The problem with those subclasses is that they are not compositional. This means that, for instance, combining two $LR$ grammars does not imply that the resulting grammar is also an $LR$ grammar[2]. So, if we use an $LR$ parser generator and combine two $LR$ grammars, we can not successfully use the generator again. First we have to resolve the conflicts in the $LR$ table. This is a fundamental problem of the process of modifying grammars for reengineering purposes: the grammars are subject to continuous change. Let us give a small example. Suppose we have the grammar consisting of the rules `E ::= a B C`, `B ::= b`, `C ::= c` and a grammar constisting of `E ::= a B D`, `B ::= b`, `D ::= d`. Both grammars are $LR$. If we combine them, we obtain the grammar `E ::= a B C`, `E ::= a B D`, `B ::= b`, `C ::= c`, `D ::= d`, note that combining is just simply the set-theoretic union. The resulting grammar is no longer $LR$ because we need a look ahead of more than 1 to decide which rule matches `a b c`. For other combinations, similar examples exist. For more details on these issues and well-established parsing technology we refer to [2], [1], or [43].

Current parser technology is also not modular. This is not very surprising since the lack of compositionality disencourages modular specifications of grammars. However, with the aid of a preprocessor it is possible to obtain a form of noncompositional modularity. In the next section we will see an example where Siber Systems uses noncompositional modularity.

[2]We speak of an $LR$ grammar if the grammar can be parsed by an $LR$ parsing algorithm.

## 3 Current Practice in Parsing

Let us take a look at current practice in parsing technology that is used in reengineering. There are many systems around that use off-the-shelf parser generators, or other parsing technology to aid in reengineering tasks. The list we give is not exhaustive. There are many more tools for the construction of compilers, interpreters, and parsers. See for instance the USENET Frequently Asked Questions (FAQ) list for the newsgroup `comp.lang.misc` where freely available language related tools are listed. What the not mentioned tools have in common, is that they all use some form of noncompositional and/or nonmodular parsing technology. We note that our list can be seen complementary to the FAQ since we mainly discuss commercial systems.

- Lex+Yacc [25, 28] uses $LALR(1)$ parser technology. Lex+Yacc are well-known formalisms to generate scanners and parsers. They have been used to define COBOL grammars. But since $LR$ grammars are not compositional it is difficult to extend an $LR$ grammar. There is no tool support for locating conflicts in the grammar. Moreover, Yacc is not modular.

- TXL [15] is a program transformation system. It is used for reengineering purposes. The technology for parsing is recursive descent parsing with an upper bound to the depth of backtracking. We recall that recursive decent parsing is $LL(1)$. In fact, this is a very simple algorithm: it does not use a parse table. A drawback is that the backtracking leads to an exponential behaviour. Therefore, an upper bound is set to the depth of backtracking. This implies that unexpected parse trees can be obtained. This technique does not allow left-recursive grammar rules, which is a severe restriction. It is not modular and it is not compositional.

- ANTLR/DLG ANTLR stands for another tool for language recognition [32]. It is an $LL(k)$ parser generator and is part of PCCTS (Purdue Compiler Construction Tool Set [31]). DLG is an abbreviation for DFA-based lexical analyzer generator—it is also part of PCCTS. The ANTLR/DLG part of PCCTS has been used by several researchers in the reengineering area to define grammars. This technology does not not allow left-recursive grammar rules, is not compositional, and not modular.

- TAMPR is a general purpose program transformation system developed in the seventies [8]. It has been used to restructure COBOL source [19]. One of the authors of [19], McParland, informed us that they used Yacc for parsing. This was unsatisfactory for them since it was a problem to deal with the dialects of COBOL. They are looking for alternative parsing technology.

- CobolTransformer is a product meant for the transformation of twelve popular COBOL dialects [38]. In their white paper we can read that their COBOL grammar cannot be fully described with $LALR(1)$ grammar used by plain vanilla Yacc. Therefore they use BtYacc (Back Tracking Yacc) which is original Berkeley Yacc modified by Chris Dodd (`chrisd@collins.com`) and then enhanced by Vadim Maslov of Siber Systems. Maslov [29] mailed us that Yacc with backtracking is somewhat limited in what it can do for parsing different dialects for COBOL. Manual resolving of conflicts is still necessary. The coming release of CobolTransformer (version 1.4.1) supports modularity in the sense that all grammar rules are contained in separate files to be included into one large grammar file. This means that this system supports noncompositional modularity. Maslov communicated to us that even this form of modularity helps when dealing with different dialects of data manipulation languages (CA-IDMS and DMS DML). With the coming release of CobolTransformer, there will be a new release of BtYacc (version 2.1) that supports C-like modularity constructs (`%define, %ifdef, %endif, %include`). BtYacc can be downloaded for free from the home page of Siber Systems. BtYacc is not compositional.

- REFINE is a reengineering environment [33]. The current implementation makes use of $LALR$ parsing with escapes to handle non-$LALR$ language features. From a recent email discussion with Tim Chou from Reasoning we learned that reasoning is working on a next generation Reasoning Advanced Parser Generator based on $GLR$ technology. So the next implementation will be compositional and (therefore) also modular.

- SEMANTIC DESIGNS, Inc. The director of this Texas located company, Ira Baxter [3], communicated to us they are currently implementing $GLR$ parsing technology in order to improve reengineering and maintenance tasks. After this paper was put on the Web we got an email from Ira Baxter, stating: "We have a full $GLR$ parser engine running with a C++ grammar, complete error recovery and fully automated syntax tree building. We like it `:-}`" Obviously, this parser is both compositional and modular. At the time of writing this paper they are working on the incremental part of the parser generator.

- SES Software-Engineering Service GmbH. The technical director Harry Sneed [39] told us that he has parsers for COBOL, PL/1, Assembler, NATURAL, CICS, DLI and SQL. The parsers for CICS, DLI and SQL are called from the COBOL, PL/1 and Assembler parsers when the command `EXEC CICS`, `EXEC DLI`, or `EXEC SQL` is recognized. So SES works with parsers and subparsers, which is a form of modularity. We quote from an email from Harry Sneed: "I do not use any standard parsing tools such as Lex or Yacc. They do not serve my purpose. Everything has been handcoded by myself. Nevertheless, I do have my own parsing patterns, code frameworks which I inherit and adjust according to local dialects." Harry Sneed manages the problems with parser technology by handcoding the complete parsers so that he has complete control when modifications are necessary for dealing with dialects. He writes: "The other problem is with the many dialects of COBOL. No one company has the same set of COBOL instructions. If I were not able to adjust my parsers to the user dialects, I would be lost."

- Revolve/2000 is a parser tool of Micro Focus that can handle a number of German specific dialects. From Joachim Blomi [6], we learned that Lex+Yacc is `not` used to implement this parser. We hope to hear from him whether the used technology is modular or compositional.

- COSMOS is the product that provides, according to the Gartner Group, the most flexible and most automated Year-2000 solution [18]. COSMOS can parse a large number of different languages and dialects [40]. From Fred Hirdes [24] we learned that currently, they use Flex and Bison as parsing techniques. We are investigating with TechForce whether it is feasible to migrate to more modular and or compositional parsing techniques.

- ASF+SDF this is a modular algebraic specification formalism for the definition of syntax and semantics of (programming) languages. It is a combination of two formalisms ASF (Algebraic Specification Formalism [4]), and SDF (Syntax Definition Formalism [20]). The ASF+SDF formalism is supported by an interactive programming environment, the ASF+SDF Meta-environment [26]. This system is called *meta-environment* because it supports the design and development of programming environments. The ASF+SDF Meta-Environment uses $GLR$ technology [34]. More precisely, it uses a Generalized $LR$-based parser generator based on modifications of Tomita's algorithm [41], for *arbitrary* context-free languages. The parser generator generates parsers incrementally. That is, it it modifies the parser if modifications to the grammar were made and/or never used production rules are needed to parse a program. This technology is called incremental generation of generalized *LR*-parsers [21, 22]. The Asf+Sdf Meta-Environment has been used to define COBOL dialects for reengineering purposes [13]; it has been used to generate components for software renovation factories that are dialect proof [12]; and it has been applied to construct an assembly line for a software renovation factory [11]. ASF+SDF supports modular and compositional grammar construction. It has no error recovery. The parser generator is incremental. It has support for locating ambiguities that cannot be resolved by the *GLR* parsing algorithm.

From the above observations we can draw the conclusion that it is more and more recognized that current parser technology gives many problems in the field of reengineering languages like COBOL. We expect that this is due to the fact that such languages come in many dialects, with many extensions, and with home brewed constructs. Another observation is that companies begin to solve their parse problems using compositional and/or modular techniques. We note that in [10] it has been argued that generic language technology like modular *GLR* parsing is necessary for reengineering purposes. In the next section we will elaborate on this technology.

## 4 Generalized *LR* Parsing

In the early seventies Bernard Lang provided the theoretical foundations for Generalized *LR* parsing [27]. This result is worked out by Tomita [41]. In accordance with Stigler's law of eponymy[3] Generalized *LR* parsing is sometimes called Tomita parsing. We prefer to use generalized *LR* parsing since the result is due to Lang, at least not to Tomita. As far as we know, Rekers [34] was the first who implemented the *GLR* parsing algorithm as part of a system intended for practical applications (the Asf+Sdf Meta-Environment). Also Rekers extended the algorithm to the full class of general context-free grammars. The algorithm that Tomita presents loops on cyclic grammars, which was solved by Rekers. Furthermore, in [30] a modification to the algorithm that Tomita describes was made—a modification that is present in Rekers' implementation. The GLR algorithm returns a forest of parse trees, however, for reverse engineering purposes we are only interested in one tree. Therefore, a number of syntactic disambiguation rules must be applied to reduce the number of yielded trees. This can be done by, e.g., using the associativity and priority rules for binary operators, for more details see the Sdf reference manual [20]. If eventually more than one parse tree is left over, this is reported to the user and she is obliged to make a selection. The grammar rules which cause this ambiguity are reported as well, so in this way the ambiguities in the grammar are located.

[3]No scientific discovery is named after its discoverer.

Thus far, we are not aware of other systems than the Asf+Sdf Meta-Environment that have well-tested and heavily used implementation of a *GLR* parser generator as part of a system. We note that the latest version of the so-called Ibpag2 system (Icon-Based Parser Generation System 2) can handle GLR grammars for Icon. Icon is a high-level, general purpose programming language [17].

### 4.1 The *GLR* algorithm

The *GLR* algorithm uses an ordinary *LR* parse table. This table may contain shift/reduce and/or reduce/reduce conflicts. The *GLR* algorithm operates in exactly the same way as the *LR* algorithm, until either a shift/reduce or reduce/reduce conflict in the parse table is encountered. The algorithm forks as many parses as there are possibilities to proceed. If a conflict was due to the need for lookahead, those forked parses will die. All these parses proceed in parallel but synchronize on the shift actions. This gives rise to the possibility of merging parses which are in the same *LR* state. We note that the above algorithm is a special case of work that has been presented by Lang [27].

**Backtracking versus *GLR*** We explain the differences between *GLR* and backtracking in some more detail in order to make clear that they are not the same, as was assumed by some discussions we had with people about this paper. With backtracking techniques, the lexicals are read more than once for ambiguous parses: for each new possibility to parse the code, the lexicals are read again. Using *GLR* technology, the lexicals are read only once. Sharing is not an issue using backtracking, since this is a sequential process. So with backtracking, it is not possible to share parse steps or subtrees. The *GLR* algorithm makes use of the fact that common parse steps in distinct parses can be shared. Moreover, it is possible to share common subtrees of distinct ambiguous parses, this is called a shared forest. For a more elaborate discussion on shared forests of ambiguous parsing and their efficiency we refer to [5]. These differences all imply that the efficiency of the *GLR* approach is, in general, better than that of backtracking. Moreover, the *GLR* approach is parallel, and the backtracking approach is sequential. In the next paragraph we elaborate on efficiency some more.

**Efficiency** A natural question that arises is that of efficiency of the *GLR* parsing algorithm. There are at least two ways of measuring this efficiency. First, how does a *GLR* parsing algorithm compare to mainstream technology like Yacc? Secondly, since with *GLR* parsing we can handle ambiguities, how do various algorithms compare as a function of the number of ambiguities? Both questions are answered in [34]. We will give a short indication of the measurements so that the reader has an idea of the efficiency. Roughly, a 1 KLOC Pascal program was parsed by Yacc in 0.25 seconds and by the *GLR* algorithm in 0.75 seconds. However, the Yacc grammar had to be disambiguated: 357 shift/reduce conflicts had to be solved. The *GLR* algorithm used the extra 0.5 seconds to solve those 357 conflicts on the fly. On disambiguated grammars *GLR* and *LR* have the same performance. The merit of *GLR* parsing is that grammar development time is far less than that of a Yacc grammar. In the field of reengineering this latter property is extremely valuable, since grammars are continuously changing in this area.

Next we compare the performance of the *GLR* algorithm to Earley's algorithm [16]. Earley's algorithm can handle ambiguities as well. Below $10^7$ ambiguities the *GLR* parsing algorithm wins from the Earley algorithm and above this number the Earley algorithm wins. See [34] for details.

From all these measurements, the conclusion can be drawn that when there is not a single conflict in a grammar the cheapest solution is to use an existing *LALR* parser generator like Yacc. When the number of ambiguities is extremely high, it is best to use Earley's algorithm—it misbehaves on conflict-free grammars. In most practical reengineering cases the number of ambiguities is far less than $10^7$ but more than more than zero (viz. the 357 conflicts in the Pascal grammar). So in those cases it is a good idea to use *GLR* parsing technology, as we will see shortly in the next section.

## 4.2 Applications

In [13], we propose a methodology to obtain a COBOL grammar for reengineering purposes. As an example we implemented an OS/VS COBOL grammar using the Asf+Sdf Meta-Environment that contains Rekers' [34] implementation of incremental Generalized *LR* parsing. The Asf+Sdf Meta-Environment also contains incremental generation of lexical scanners [23]. Those features made this system for us a perfect candidate to prototype our ideas on parsing for reengineering. We think that successfully defining a COBOL grammar plus some dialects and extensions using *GLR* parsing technology gives clear evidence that this technology scales up to real-world applications. We have no no maintenance problems and we can easily extend or restrict grammar definitions.

We took care of handling dialects of COBOL in [11]. Due to the modularity and the compositionality we were able to simply modify the grammar defined in [13] to a grammar that parsed both dialects, which is convenient for dialect conversions. To give the reader an idea of the modularity we depicted the import-graph of the grammar in Figure 1. An arrow from module *A* to *B* means that all grammar rules of module *A* are imported in module *B*. The module `PROGRAM`, depicted at the right of Figure 1 is the top module of the definition. This module gathers all the grammar rules that are describing the syntax of the four divisions of COBOL. Similarly, the top modules `IDENT-DIV`, `ENV-DIV`, `DATA-DIV`, and `PROC-DIV` gather the grammar rules that are specific for the four divisions that are available in COBOL. In the modules `IDENT-BASIS`, `ENV-BASIS`, `DATA-BASIS`, and `PROC-BASIS`, the structure of those divisions is defined. In twelve modules ranging from `INFO` to `RU-STAT` we define the more local syntax, like `MOVE` statements, or `IF` statements. The four `BASIS` modules import the module `PHRASES`. This module gathers the grammar part that is not specific to a particular division. Natural numbers, decimals, strings, literals, identifiers, data-names, arithmetic and logical expressions (`A-EXP`, `L-EXP`), relation symbols, predicates, pseudo-text, copy statements, and on the lowest level comments. This module contains the grammar rules for layout and comment.

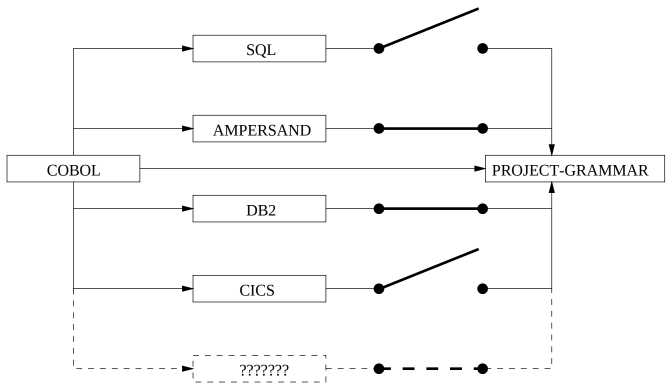

Figure 2: Hyper-graph of a typical reengineering project

We also constructed a CICS grammar using the methodology reported on in [13]. It took two hours to construct the CICS grammar, to merge it with a COBOL grammar and to test it on a COBOL program that contained all the CICS code that was available in a 70 KLOC COBOL/CICS mortgage system. For another project we needed to extend a certain COBOL dialect with SQL. It took eight hours to construct a grammar for embedded SQL, to integrate it, and to test it on the system. We stress that—due to the *GLR* parser generator—not a single ambiguity had to be solved during the extension of CICS or SQL. Note that with this approach we are able to provide multilingual support for reengineering tools. For an example of tools for reengineering a COBOL/CICS system we refer to [11].

In Figure 2 we depict a hyper-graph of a typical project grammar. It is a hyper-graph since the nodes of this graph can represent themselves again a (hyper-)graph. For instance, the COBOL node in our hyper-graph represents the entire COBOL import-graph as depicted in Figure 1. The other nodes represent grammars for possible extensions to COBOL, like embedded SQL, embedded CICS, embedded

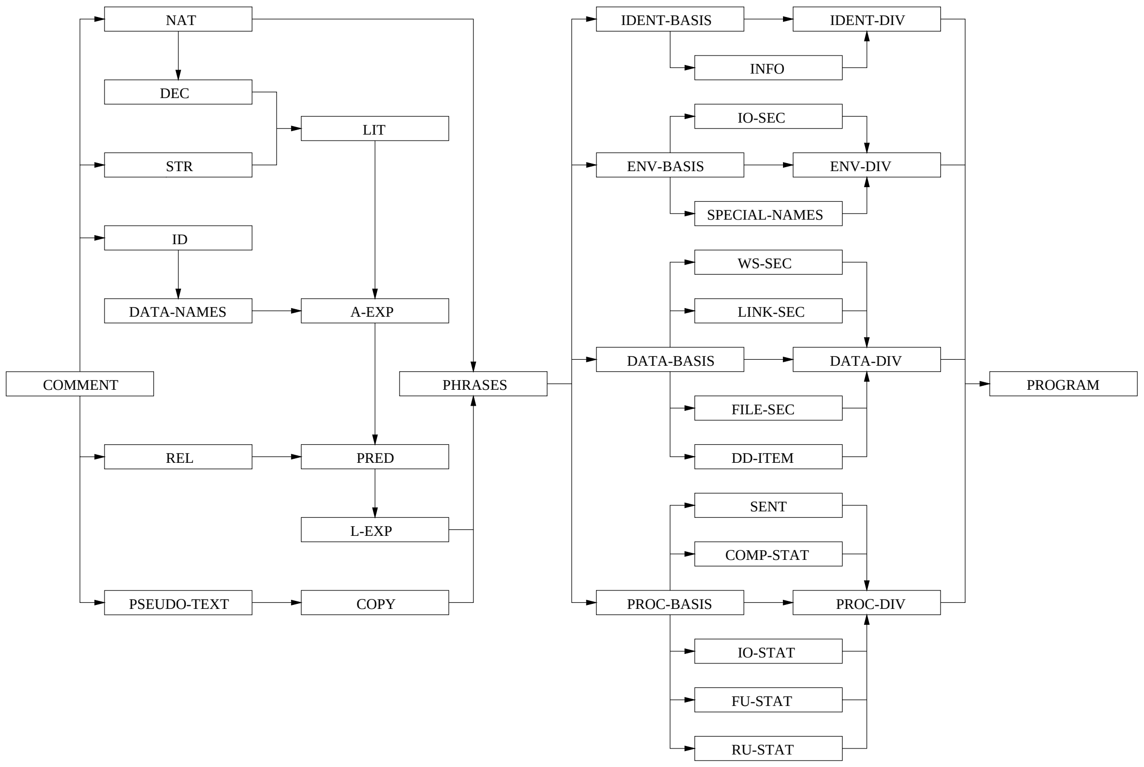

Figure 1: Import-graph of a modular COBOL grammar

DB2, a yet unknown extension, or some temporal nonstandard extension like the Ampersand module that we will discuss in Section 5. The switches in Figure 2 denote the possibility to add or remove extensions to COBOL. The implementation of this switchboard is just a simple module that contains the right imports for a particular project. The module `PROJECT-GRAMMAR` for the above situation consists of the following line:

```
imports COBOL DB2 Ampersand
```

Addition of CICS and removal of DB2 and Ampersand for another project is simply a matter of changing the import structure:

```
imports COBOL CICS
```

Intuitively, we see this file as a switchboard to combine COBOL with arbitrary extensions at wish.

We use this switchboard to reuse large parts of grammars to construct new ones. We believe that our approach gives some evidence that modular *GLR* parsing provides the necessary flexibility for reengineering purposes. At least, we are building a repository of grammar modules that can be effortlessly combined at wish to readily parse new projects.

## 4.3 Reusability of Grammars and/or Parsers for Reengineering

Reusability of parsers is not easy. Apart from the lack of modularity, a reason for this is that the input and output formats of various parsers do not necessarily fit to the front-ends of reengineering tools. In [35] this problem is already mentioned: they propose that parsing should be a separate component for a reverse engineering tool in order to deal with dialects. They write that:

> Expanding a tool's capabilities to include additional source languages and additional analyses, can often be quite difficult. The statement that "all you have to do is add a new parser" is deceptively appealing.

So not only parsing of obscure dialects is a problem, but also the connection of these parsers to existing tools.

Another problem is that most parsers have been used in order to be the front-end of a compiler so sometimes they are completely integrated within this implementation. If this is the case it is not possible to reuse the parser of that particular language for reengineering purposes.

The parser that we use for reengineering purposes has as input format SDF [20]. We recall that this stands for *Syntax Definition Formalism.* The output format is ATF, this stands for *Annotated Term Format* [9]. ATF is specifically designed for the data exchange between (possibly) heterogeneous or distributed components in a software system. It is a powerful format in which it is possible to represent, for instance, parse trees that can be annotated with all kinds of information like textual coordinates, access paths, or the results of data flow, control flow or other program analysis. It is not hard to transform a term in ATF to another output format. In this way our parser generator can, in principle, be used as a front-end for other tools. The ASF+SDF Meta-Environment is primarily an interactive system, but it can be run batch oriented using a small shell script. So in principle it can be used as a *GLR* parser generator. We use the batch oriented mode to automatically restructure complete systems. We are interested in the in- and output formats (if any) and the implementations that Reasoning and Semantic Designs are currently implementing; so we can compare them to the implementation in the ASF+SDF Meta-Environment.

## 4.4 scannerless *GLR* parsing

[42] discusses *scannerless GLR* parsing. The idea of scannerless parsing is due to Salomon [36, 37]. Scanning and parsing are separate processes. The parser asks the scanner to give the next lexical token. However, at the lexical level there can be ambiguities as well (for instance the longest-match ambiguity [36]). The scanner has to make a decision that can be in conflict with what the parser expects. This can lead to syntax errors in correct programs. This problem is solved by using scannerless parsing because the parser takes over the scanning and makes the disambiguation decisions. The basic idea is to transform the lexical syntax (defined by regular expressions) into context-free grammar rules. So scannerless *GLR* parsing is even more desirable for reengineering. This is not implemented in the ASF+SDF Meta-Environment. It is the intention that a scannerless *GLR* parser generator will become a component in the new version of the ASF+SDF Meta-Environment. A prototype is both algebraically specified in ASF+SDF and implemented in C [42]. It works on small examples but some problems need to be solved.

According to [37], scannerless parsing is efficient since it can be implemented using a simple two-stack push-down automaton. Indeed this is efficient: push and pop operations on a stack are inexpensive.

# 5 Grammar Reengineering

Due to the lack of modularity of mainstream parser technology, sooner or later a grammar, say a COBOL grammar with extensions will become unmaintainable. Let us give a typical example that we experienced in practice. When parsing an OS/VS COBOL system our parser did not recognize some COBOL programs. After inspection we observed that about 40% of a subsystem contained ampersand signs (`&`) on locations where an identifier was expected. We recall that an ampersand sign is not part of any COBOL syntax we know of. Still it occurred in real world COBOL code. The ampersands were used by a home grown preprocessor that used ampersands as a parameter mechanism for generating programs. This implies that we needed a grammar that recognizes this particular dialect of COBOL. Note that this case resembles the situation that we quoted in the introduction from Micro Focus [7]. To be able to parse this code we made an SDF module named Ampersand that recognizes identifiers containing series of ampersands or that consisted solely of one or more ampersands. This is the complete module:

```
imports Program
exports
  sorts Id-ampersand

  lexical syntax
    Left* [&]+ Right* -> Id-ampersand

  context-free syntax
    Id-ampersand -> Id
```

Before we continue, let us explain this module. We import the entire COBOL grammar by importing the top module in COBOL called `Program`. We export the new sort `Id-ampersand` so that tools like a parser can use it. The sorts `Left` and `Right` are defined in the low-level module `ID` that defines identifiers for COBOL. They are known to the ampersand module due to the import structure of the COBOL grammar, see Figure 1. They represent that an identifier in COBOL may not start with a minus sign and may not end with a minus sign. The same holds for the identifiers containing the ampersands. In the context-free syntax definition we then lift this special sort to the known sort `Id`, so that not only normal identifiers are recognized but also if they contain an ampersand. Using this additional module we can recognize identifiers like `ABC-&&&&&&&-DEF`.

The merit of this modular approach is that for this particular project we added it to the switchboard. The core grammar is not polluted with this idiosyncratic syntax. In fact, the switchboard approach enables us to easily undo a change to the grammar. Without modularity it would have been necessary to modify the identifiers inside the COBOL definition, which is dangerous: after quickly modifying the grammar and resolving the conflicts, the resulting grammar is often not changed back to its original form due to timing constraints. Tomorrow someone else uses an ampersand-sign for another purpose and then it becomes more and more difficult to maintain such a grammar. A number of such changes leads to a grammar with a maintenance problem: it will become harder and harder to resolve conflicts. We expect that in the near future serious reengineering companies will face the fact that their own business-critical applications need to be reengineered. Hence, the title of this paper: we believe that mainstream technology indeed can be harmful for reengineering companies. To phrase it informally: if such grammars exist, we believe them to have a year 1999 problem: before that year they need to be reengineered so that they can parse and analyze the programs with a possible year 2000 problem. Therefore, we do not believe it to be a coincidence that, for instance, Reasoning or Semantic Designs are implementing *GLR* parsing technology.

## 6 Conclusions

In this paper we evaluated current parsing technology in the case it will be used for reengineering purposes. We showed that the mainstream technology has restrictions that cause problems for the reverse engineering or reengineering software systems. Those drawbacks are that it is not easy to construct conflict-free grammars and that current technology leads to maintenance problems of grammars. We discussed *GLR* parsing technology that is modular and compositional. We showed with several case studies reported on in other papers that these techniques can be applied to easily construct grammars, to merge them, to reuse parts of them, and to maintain them. We showed that the techniques scale up to real-world grammars. We observed that companies such as Siber Systems, Reasoning, and Semantic Designs and TechForce are in the middle of departing current parsing technology: they move or consider moving to modular and/or compositional parsing techniques.

## References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

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

[3] I.D. Baxter. Director of Semantics Design, Inc., Personal communication, October 1997.

[4] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[5] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics, Vancouver*, pages 143–151, 1989.

[6] J. Blomi. Business Manager Y2K, Micro Focus Germany, Personal Communication, November 1997.

[7] J. Blomi. *Metamorphosen des Datumsfeldes.* Micro Focus GmbH, München, Germany, 1997. Available in German at http://www.microfocus.de/y2000/articles/y2k-info.htm.

[8] J.M. Boyle. A transformational component for programming language grammar. Technical Report ANL-7690, Argonne National Laboratory, Argonne, Illinois, 1970.

[9] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[10] M.G.J. van den Brand, P. Klint, and C. Verhoef. Reengineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997.

[11] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy systems. Technical Report P9714, University of Amsterdam, Programming Research Group, 1997. Available at http://adam.wins.uva.nl/˜x/cfn/cfn.html.

[12] 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, *proceedings of the fourth working conference on reverse engineering*, pages 144–153, 1997. Available at http://adam.wins.uva.nl/˜x/trans/trans.html.

[13] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In M.P.A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, electronic Workshops in Computing. Springer verlag, 1997. To appear, Available at http://adam.wins.uva.nl/˜x/coboldef/coboldef.html.

[14] Per Cederqvist. *Version Management with CVS.* Signum Support AB, Box 2044, S-580 02 Linkoping, Sweden, cvs 1.9 edition, 1993. Available at: http://www.acm.uiuc.edu/sigsoft/workshops/cvs/cvs.ps.

[15] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[16] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[17] R.E. Griswold and M.T. Griswold. *The Icon Programming Language.* Prentice-Hall, Englewood Cliffs (NJ), 1990.

[18] B. Hall. Year 2000 tools and services. In *Symposium/ITxpo 96, The IT revolution continues: managing diversity in the 21st century*, page 14. Gartner Group, 1996.

[19] T. Harmer, P. McParland, and J. Boyle. Using knowledge-based transformations to reverse engineer COBOL programs. In *11th Knowledge-Based Software Engineering Conference*. IEEE-CS-Press, 1996.

[20] 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.

[21] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *SIGPLAN Notices*, 24(7):179–191, 1989.

[22] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. An earlier version appeared as [21].

[23] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems*, 14(4):490–520, 1992.

[24] F.P. Hirdes. Director of techforce, the Netherlands, personal communication, December 1997.

[25] S.C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[26] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[27] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[28] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator*. Bell Laboratories, UNIX Programmer's Supplementary Documents, volume 1 (PS1) edition, 1986.

[29] V. Maslov. Siber Systems, Personal Communication, December 1997.

[30] R. Nozohoor-Farshi. Handling of ill-designed grammars in Tomita's parsing algorithm. In *Proceedings of the International Parsing Workshop 89*, pages 182–192, 1989.

[31] T.J. Parr, H.G. Dietz, and W.E. Cohen. *PCCTS Reference Manual*, 1.00 edition, 1991.

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

[33] Reasoning Systems, Palo Alto, California. *Refine User's Guide*, 1992.

[34] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.

[35] H. Reubenstein, R. Piazza, and S. Roberts. Separating parsing and analysis in reverse engineering tools. In *Proceedings of the 1st Working Conference on Reverse Engineering*, pages 117–125, 1993.

[36] D.J. Salomon. *Metalanguage Enhancements and Parser-Generation Techniques for Scannerless Parsing of Programming Languages*. PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1989.

[37] D.J. Salomon and G.V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, 24(7):170–178, 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[38] Siber Systems Inc. *CobolTransformer—Peek Under the Hood: Technical White Paper*, 1997. Available at http://www.siber.com/sct/tech-paper.html.

[39] H.M. Sneed. Director of SES Software-Engineering Service, GmbH, Personal Communication, October–December 1997.

[40] TechForce B.V., P.O. Box 3108, 2130 KC Hoofddorp, The Netherlands. *COSMOS 2000 White paper*, 1997. Available at http://www.cosmos2000.com/whitepap.pdf.

[41] M. Tomita. *Efficient Parsing for Natural Languages—A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1986.

[42] Eelco Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997. Available at http://www.wins.uva.nl/pub/programming-research/reports/1997/P9707.ps.

[43] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1996.