

Case Closed: the 500 Language Problem is Cracked

R. Lämmel* and C. Verhoef**

**Centrum voor Wiskunde en Informatica,
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*

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

`ralf@cwi.nl`, `x@wins.uva.nl`

Abstract

We explain what the 500 language problem is, why it is a relevant problem, and why solutions are needed. We discuss its solution, and illustrate it by applying it to two non-trivial but representative languages: an exotic proprietary real-time language from the telecommunications industry, and a well-known dialect of the most popular language in the world: IBM's VS COBOL II. We share our lessons learned based on these examples and on other experiences we had with our work on this problem.

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 Keywords and Phrases: Reengineering, Software renovation, Grammar engineering, Grammar recovery, Grammar reverse engineering, 500 language problem, COBOL.

1 What is the 500 Language Problem?

Capers Jones estimates that there are at least 500 languages and dialects available in commercial form or in the public domain. On top of that, he estimates that some 200 proprietary languages have been developed by corporations for their own use [8, loc. cit. 321]. In his book on estimating the costs of the Year 2000 problem [9] he furthermore indicated that systems written in all those 500 plus 200 languages were affected. The initial estimate of Jones inspired many Y2K whistle blowers to mention his estimates as a major impediment to solve the Year 2000 problem. Let's have a look at what these people had to say. For instance, Ed Yourdon got so many email messages on solutions for the Y2K problem that he had to answer such email with a boilerplate where the 500 language problem is mentioned, and it is worthwhile to quote this part in its entirety:

I recognize that there is always a chance that someone will come up with a brilliant solution that everyone else has overlooked, but at this late date, I think it's highly unlikely. In particular, I think

the chances of a “silver bullet” solution that will solve ALL y2k problems is virtually zero. If you think you have such a solution, I have two words for you: embedded systems. If that’s not enough, I have three words for you: 500 programming languages. The immense variety of programming languages (yes, there really are 500!), hardware platforms, operating systems, and environmental conditions virtually eliminates any chance of a single tool, method, or technique being universally applicable.

The number 500 should be taken poetically, like the 1000 in the preserving process for so-called thousand-year-old eggs, which last only 100 days. For a start, the 200 proprietary languages should be added, moreover other estimates indicate that 700 is rather conservative: Weinberg estimated already in 1971 that in 1972 programming languages will be invented at the rate of one per week—or more, if we consider the ones which never make it to the literature, and enormously more if we consider dialects, too [17, p. 242]. Also Peter de Jager created awareness for the 500 language problem. He writes [7]:

There are close to 500 programming languages used to develop applications. Most of these conversion or inventory tools are directed toward a very small subset of those 500 languages. A majority of the tools are focused on COBOL, the most popular business programming language in the world. Very few tools, if any have been designed to help in the area of APL or JOVIAL for example.

Both Yourdon and de Jager were right about the lack of tool support: Jones also figured out that a small fraction of the languages was covered, while the distribution of languages did not justify that. The distribution of languages is that 30% of the world’s installed software is written in COBOL, 20% in C/C++, 10% in Assembler, and the remaining 40% is written in all those obscure languages. In contrast, for about 50 languages search engines existed, and for about 10 languages there were automated repair engines [9, p. 325]. So only for a very small part of the languages there is automated modification support. Thus the 500 language problem got its name. We entered the new millennium without too much trouble so you may conclude: okay, maybe there was this 500 language problem, but whatever it was, it is not relevant anymore. Deadline passed. Good point. Of course the 500 language problem already existed before it was popularized by the Y2K gurus, and it didn’t go away when we entered the new millennium. Here’s our definition:

The (generalized) *500 language problem* is the endemic lack of tool support to analyze and modify existing software systems for the myriad of available languages.

Why does this problem exist anyway? A prerequisite for tool-supported analysis or automated modification of software, is that the code has to be converted from text format into a tree format so that tools can more easily analyze and manipulate code. To make this conversion you need a so-called parser. Constructing a parser for a language is a major effort, and in many cases the up-front investment is hampering initiatives for commercial tool builders, which

clarifies the lack. You don't have to be a rocket scientist to realize that implementing the McCabe complexity metric for JOVIAL will not pay off. Indeed, Tom McCabe told us that McCabe & Associates developed parsers for 23 languages, which was already a huge investment. He also called the 500 language problem "the number one problem in software renovation".

Okay, before you stop reading to fire up your favorite search engine to figure out what the hack JOVIAL is, we better tell you. JOVIAL stands for *Jules' Own Version of the International Algebraic Language*. So far, things sound fairly innocent: some private implementation of Algol [1]. Speaking of privates, using the JOVIAL language was prescribed in the US Air Force's standard MIL-STD-1589 for embedded systems during the late 1970s and early 1980s when the majority of the Air Force's fielded weapon systems were developed. Needless to say that the Department of Defense (DoD) wants tools for JOVIAL, and they are right. This is probably also one of the reasons why Peter de Jager mentioned JOVIAL. During talks and demonstrations of our prototype software renovation factories, the first question we get from DoD software engineers is always: nice stuff, but does it also work for JOVIAL? Of course, this is anecdotal evidence, but nevertheless it is indicative. There are support tools for JOVIAL, but on the official JOVIAL home page we can read that "Maintaining these software support tools is an extremely complex task".

The 500 language problem is not a DoD problem. JOVIAL is just one of the "500". We mention another example that will come back in the paper. Ericsson needs tool support for a proprietary language designed in the late 1970s and early 1980s that was used when the majority of public telephony switches were developed. It is called PLEX, short for Programming Language for EXchanges. As an indication, only in 1996 the in-house developed McCabe complexity metric tool for PLEX was evaluated and found to be erratic [5]. For mainstream languages such evaluations have long been performed, and language specific improvements are available in abundance, also for some of those languages many tools are available. Seen in that light, it is not a surprise that efforts are underway to move away from developing in PLEX [2].

Hey, that is an idea: the solution for the 500 language problem is to just convert from obscure languages to mainstream ones where tool support is available. Eliminating all these languages will make the 500 language problem go away. Not. Language conversion is much more difficult than many of us think. The average language converter is a procrustean bed for software: it violently forces code into arbitrary conformity to compile under the new language, but it does not fit the idiom of the new language at all, resulting in unnatural amputated code that is usually unmaintainable and whose behavior is not equivalent to the original system. So conversion is not a solution. For more information on the realities of language conversions, have a look at [16]. But even if conversion were a solution, you need a full-blown tool set for a conversion, including a serious parser. And obtaining a parser is part of the 500 language problem. So language conversion will not *eliminate* the 500 language problem, au contraire, you need a *solution* for the 500 language problem to aid in solving conversion problems.

JOVIAL and PLEX are just two examples, typical for the tool support situation for 40% of the existing software written in exotic languages. These examples may give you the impression that the 500 language problem is restricted to embedded software systems written in obscure languages, like Ed

Yourdon indicated, too. This impression is wrong. Let's have a look at another 30%: COBOL. There are many COBOL dialects, each compiler product has a few versions, with many patch levels. On top of that, COBOL often contains embedded languages like DMS, DML, CICS and SQL. Ron Kral of the Source Recovery Company told us that a COBOL executable could have been compiled in at least 300 different ways (which gives you an idea of the immense variety in COBOL dialects). In fact, there is no such thing as *the COBOL language*. COBOL is a polyglot: a confusing mixture of dialects and embedded languages. This leads us to state that parsing COBOL is a 500 language problem in itself. The demand for COBOL grammars is large: it is a frequently asked question in the Usenet group `comp.compilers`. Here's one such question:

```
Anybody know where I might find a COBOL Grammar (preferably in
Bacchus-Naur Form)?
-- douglas368@aol.com
```

Apparently, some people are so desperately seeking for a COBOL parser, that they confuse the Greek god of wine with the inventor of syntax notation (J.W. Backus [1]). More importantly, what is the answer? Despite the 40 year history of COBOL and its omnipresent use, there has never been a freely available COBOL grammar (until we cracked the 500 language problem). It is a major pain to construct a COBOL grammar: first of all *which* variant? Secondly, the language is quite elaborated, so implementing a quality COBOL parser takes you 2 to 3 years, as Vadim Maslov of Siber Systems estimated (he constructed COBOL parsers for about 16 dialects). We immediately believe that already the thought of having to implement a COBOL parser turns you into a fan of Bacchus, if only to forget the traumatic experience. So once your life has been ruined by such a project, you're a fool to disclose such software to the general public: it is worth a lot of money (which you need for the wine).

For the 20% C/C++ code the problem is less dramatic. Still, there are easily 20 dialects of C, especially in the telecommunications industry this is a nuisance.

Finally, tool support for the remaining 10% which is assembler code is often not parser-based. For, assembler is row and column oriented, which makes it amendable for lexical approaches. But when assembler is used embedded in some host language, you will sometimes need a parser. Since there are many different assembly languages, assembler also contributes to the 500 language problem.

Summarizing, both for obscure languages and mainstream languages the availability of grammars is very sparse. The 500 language problem is a real problem, it was a real problem before the Y2K gurus baptized it, and it did not go away after the millennium passed. Its solution is a significant step in the direction of enabling tool support for analyzing and modifying our 7 billion function points of existing software assets written in numerous languages.

Organization The rest of this paper is organized as follows. In Section 2 we indicate how the 500 language problem is cracked. Then in Sections 3 and 4 we provide empirical evidence of our proposed solution: for two nontrivial languages with totally different characteristics we illustrate how we solved the problem of

obtaining a parser for them in an easy way. Based on our experience with many grammar recovery cases, we summarize some lessons learned in Section 5.

2 Cracking the 500 Language Problem

While the Ed's and Peter's of this world were creating the necessary awareness for the 500 language problem, we worked on solutions. The latter is not meant pejorative: bridging the KAP-Gap, i.e. removing the contradiction between the level of *knowledge and attitudes* compared with *practice* that characterizes *any* preventive measure, is utterly necessary.

Ed Yourdon thought that the large number of programming languages would virtually eliminate any chance of a single tool, method, or technique being universally applicable. It turns out that the 500 language problem does have a single solution. And it is totally trivial once you get the hang of it. No matter how trivial, the solution has been overlooked by many, as we will indicate below. Here's what we mean with the word *solution*:

The 500 language problem is cracked when there is a cheap, rapid and reliable method to produce parsers for the myriad of languages so that existing code can be analyzed and modified using tools that work on the trees produced by the parsers.

What is cheap? What is rapid? What is reliable? Cheap is in the 20.000–40.000 US dollar range, rapid is in the 2–4 weeks range, and reliable is that the parser passes the test of parsing millions of lines of code provided by the customer.

And now for the solution: what means cracked? Parser construction is a typical compiler subject. It is common practice to use parser generators to construct parsers. The input for such generators typically is a BNF [1] (Backus-Naur Form) description of the language. So without loss of generality, the 500 language problem reduces to the 500 *grammars* problem. Okay, so how to obtain all these grammars? The solution is:

don't create them, steal them, and then massage them to your needs.

Let us explain. Recall that we need to produce grammars for *existing* software. The goal of the grammar is to enable analysis and modification of the software with tools. This implies that there must be a compiler or an interpreter. For, if there is no such artifact, it does not make sense to think of modifying your source code (it does make sense to rethink your software process though). So we can safely assume that there is an executable containing the grammar. Then there are two possibilities: either the source code of this artifact is available to you or not. If the source code is at your disposal the only thing you have to do is find the part of the compiler/interpreter that turns the text into an intermediate form. That part now contains the grammar. There are three possibilities: either this part is written by hand, using something like a recursive descent algorithm, or some parser generator is used, or both (in a complex compiler for instance). We only need to cover the first two cases. In the first case, it is about a week of effort to extract the actual grammar from the hand-written code. Most of the times, in the comments (erroneous) BNF

rules are provided giving you an indication what the grammar comprises. This closes the case where a hand-written parser was implemented. If the parser is generated, there must be some BNF description of it. With a simple tool that parses the BNF itself you can automatically extract the BNF. So in both cases we have recovered the grammar. This finishes the case when you have access to the source code of a compiler or an interpreter.

The remaining case is that you do not have the compiler/interpreter sources. In such a case, it is reasonable to assume that some company is behind the product, and therefore, a language reference manual, and sometimes even an official language standard is available. Such documents are known to be full of errors. We have experienced that, presumably you too. However, we also discovered that the myriads of errors are of a repairable category. A major problem in language reference manuals is concerned with how the parts of the grammar are connected together. In a manual these connections are usually erroneous, but with simple tools this can be analyzed and repaired. Another problem is that sometimes the formal descriptions are too restricted, and only informally extended in the accompanying text. With testing we find those errors, and reading part of the manual usually helps in finding a solution. We give an example of that in Section 4. Also other restriction problems were revealed, we refer to [12] for elaborate information. Summarizing, it is possible to extract a BNF description from a language reference or a standard, and there is a process to repair the errors. This also leads to a correct BNF grammar of the language. QED.

Discussion That's all there is. But you don't buy this. For instance, there is always a case one can think of such as a quick messy implementation of a language tool, for which grammar extraction is a difficult task. We like to stress that the goal is to build tool support for analysis and renovation of business critical software systems. In the cases where it is maybe not possible to use our approach, it is also possible that we are not dealing with business critical software. But one can never exclude this.

A constructive proof of our claim would involve a list of all grammars of all languages. We opt for a proof by extrapolation. We tell you how we cracked it for two languages: one obscure nontrivial language at the one end of the spectrum and one well-known large language at the other end of the gamut (where no compiler sources are to our avail). Both languages are heavily used for business critical systems. We hope that both cases convince you so that you extrapolate the 2 examples to your particular case, whether it is PL/I or JOVIAL. In the next two sections we deal with the examples, but before we do that there are a few points we want to make.

Probably some readers immediately got the idea that if you have access to the compiler sources you can *reuse* the parser component as is. In fact, this is what Prem Devanbu is doing with his GENOA/GENII system [4]. He developed a language supported by a tool that can turn a certain idiosyncratic output format of a parser into another format that is more suitable for code analysis. There is however, one major drawback to this approach: as Devanbu points out in his paper [4] the GENOA system does not allow for modifying code. There is no sense in modifying an AST without having a grammar, namely, after modification the tree has to be converted to text again, for which you usually

need a grammar. Since we ultimately want tools to *modify* code it is better to reuse the source code of the parser rather than its output. For, then you also have the possibility to implement tree-based modification tools.

After plain *extraction* we need to adapt the grammar to our present needs. This is what we called massaging the grammar. For instance, if we need to parse sources that we want to adapt with tools, the comments should not be wiped out, like is the case in a compiler grammar. Adapting existing grammars is less a problem than getting a first base-line version. To do these things right, you need to know of grammars, you need to know how not to break the grammar while modifying it. Because we wish to focus on awareness that grammar stealing is a solution we will not dive into the technical aspects of grammar transformations. More information on grammar embellishment is available in the cited literature.

If things were that simple, why is our approach not common practice? History learns that many simple but useful inventions were not turned into common practice for a long time. So this may be a reason. Another reason might be that we use a unique combination of powerful techniques that together enable our approach. They are:

- automated grammar extraction;
- powerful parsing techniques;
- automated testing;
- having grammar analysis and transformation tool support.

If one of the above ingredients is omitted things will become less easy. Extraction by hand is error prone. Simple parsing technology limits you to work with grammars in severely limited formats. With powerful technology you can work with arbitrary context-free grammars. Without automated testing you never find so many errors in a short time. Without tool support to reengineer grammar specifications, analyses are inaccurate, corrections are not done consistently, and without transformations you cannot repeat what you have done, or change initial decisions easily (for we record transformations in scripts). This also gives you an idea of what kind of people are capable of stealing grammars. They should know about grammars, powerful parsing techniques, how to set up testing, and know about program transformations. Of course, it would be great if there were some integrated tool set to carry out such tasks. We used a variety of tools and techniques to accomplish our results. We learned that implementation of the grammar reengineering tools is very simple, mainly because the syntax and semantics of BNF is small and simple. This is one of the reasons why is not hard to implement analysis and transformation tools for BNF.

3 Grammar Recovery from Compiler Sources

We applied our approach to the exceptionally complex proprietary language PLEX used by Ericsson to program public telephone switches. PLEX consists of about 20 (sub)languages, called sectors. In fact, we are dealing with a mixed language containing high level programming sectors, assembly sectors, finite state machine sectors, marshaling sectors, and what have you. Here's what we did:

- we reverse engineered the PLEX compiler on-site (63 Meg source code);
- we found the majority of the grammars in some BNF form;
- we found a hand-written proprietary assembly parser with erroneous BNF in the comments;
- we wrote 6 BNF parsers (there were 6 different BNF dialects used);
- we extracted the BNF, and converted it to another syntax definition formalism (SDF);
- we found the lexer files;
- we converted the lexical files to SDF;
- we combined all the converted grammars into one overall grammar;
- we generated a parser with a sophisticated parser generator;
- we successfully parsed 8 million lines of PLEX code as a test; (this test was passed immediately).

This effort took us about 2 weeks in total (two persons), including constructing tools, testing time, etc. It was done for 25.000 US Dollar. We heard from Ericsson that Cordell Green (the founder of the cutting edge reengineering company Reasoning) made a ball-park estimate of a few million US Dollar. He told us later that 25.000 US Dollar was *nothing* for such a grammar.

To give you an idea of the limited complexity of our solution, here's some of the compiler source code:

```
<plex-program>
  = <program-header> <statement-row> 'END' 'PROGRAM' ',';
  %% xnsmtopg(1) ; %%
--   <= sect
--     Compound(
--       Reverse(STATEMENT-ROW.stat_list) =>
--         PROGRAM-HEADER.sect.as_prog_stat : ix_stat_list_p
--       PROGRAM-HEADER.sect : ix_sect_node_p)
--   ;
```

We first converted this to some common BNF while removing the idiosyncratic semantic actions:

```
plex-program ::= program-header statement-row 'END' 'PROGRAM' ',';
```

Then we converted to SDF:

```
Program-header Statement-row "END" "PROGRAM" ";" -> Plex-program
```

In this way we recovered the majority of the 3000+ production rules in an afternoon. The rest of the time was necessary to build one of the 5 assembly sector grammars, the overall grammar (combining all the sector grammars), and to do the testing. On top of that we generated a complete web-enabled version of the BNF description that could serve as basis for a complete and correct manual.

Discussion There is a case where our solution doesn't help you. If you lose all compilers and all language reference manuals for your business critical software you have a problem. For, if all that is gone and you must adapt the software and cannot resort to zapping the binaries, you have no choice but to convert. So you need a parser, and since all valuable knowledge sources that could possibly contain a grammar are nonexistent, there is no solution other than writing a parser by hand. We have never encountered this situation, but Capers Jones mailed us that: "For a significant number of applications with Y2K problems, the compilers may no longer be available either because the companies that wrote them have gone out of business or for other reasons." Fortunately this was not the case at Ericsson. This was our second attempt to obtain a PLEX grammar [14]. In a first attack [13, 15] we failed to recover the PLEX grammar from on-line PLEX manuals. Those manuals were not good enough to reconstruct the language from. But from the compiler sources it was easy meat. Once you realize that this is a solution it seems to be ultra trivial. As one reviewer of [14] pointed out, "the real contribution here is perhaps the entire mechanization of the process for a real system, without resorting to any manual steps; and the reason for their success is basically the general absence of strong [grammar] re-engineering tools rather than the deepness of their insight." We were very happy with these qualifications: this means that *anyone* can do it, including us. Still the obvious solution is often overlooked. For instance, in Usenet discussions on how to obtain grammars for existing source code, Spencer Rugaber (a researcher active in the reverse engineering field) suggested to look for linguistic solutions, namely to generate grammars from the source code only, in the same way linguists try to generate a grammar from a piece of natural language. This is a laborious and error prone approach leading to a totally useless grammar (if at all). Recall Prem Devanbu who reused parser *output* for analysis but could not do automated modifications. He worked at that time for AT&T, so presumably, he had access to the sources of the C++ compiler. Still he did not get the idea to extract the BNF and generate semantic actions specialized to analyze and modify C++ legacy systems. He was pleasantly surprised by our approach, which he dubbed *grammar stealing*. The point we want to make here is that the seemingly trivial solution is apparently overlooked by specialists. We think the solution is as trivial as a matchbox. Someone had to invent it so that others could take it for granted. Let's face it, when was the last time you looked at a matchbox in total devotion acknowledging the deep insight of this invention?

4 Grammar Recovery from a Manual

Some of our colleagues felt a little fooled by the results discussed in Section 3: "they're not really constructing a parser, they only convert an existing one. Hey we can do that, too! Now try it without the compiler." Indeed, at first sight, *not* having this valuable knowledge source available is a different issue (after all, we failed doing this with for PLEX). However, we learned that this failure was not due to the tools that we developed, but due to the nature of proprietary manuals: its audience is so limited that major omissions can go unnoticed for a long time. When there is a large audience for a manual things look different. In another two-week effort we recovered the VS COBOL II [6] grammar from a

3.30 SEARCH Statement

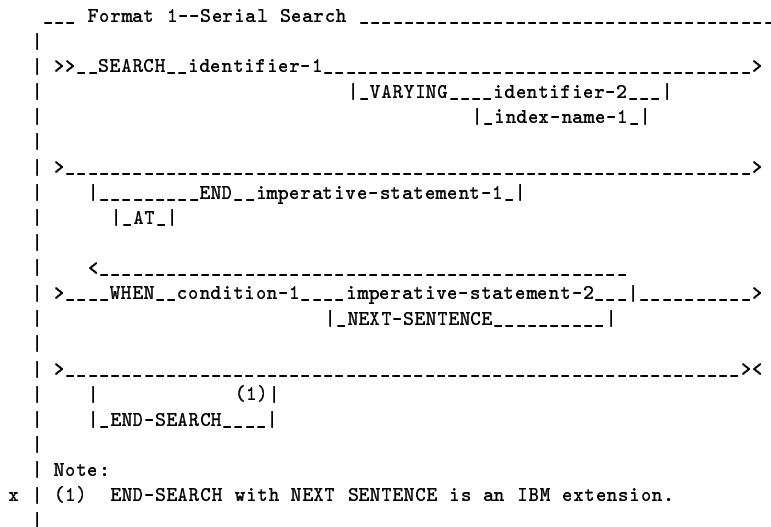


Figure 1: The original syntax diagram for the SEARCH statement.

manual. Here's how:

1. we retrieved the on-line VS COBOL II manual from www.ibm.com;
2. we extracted the syntax diagrams;
3. we wrote a parser for the syntax diagrams;
4. we extracted the BNF from the diagrams;
5. we added 17 lexical rules by hand;
6. we corrected the BNF using grammar transformations;
7. we generated an error-detection parser;
8. we parsed 2 million lines of COBOL II code and reiterated step 6–8 until all errors vanished;
9. we converted the BNF to SDF;
10. we generated a parser;
11. we parsed VS COBOL II code to detect ambiguities;
12. we solved ambiguities using grammar transformations;
13. we reiterated steps 10–12 until no more ambiguities were found.

As you see, apart from some error correction cycles and ambiguity removal sessions, the process is the same. It also took us about two weeks of effort (two persons) in total, including the construction of tools, testing, and so on. It was done for zero US dollars. In that way we could freely publish the grammar on the Internet [10], as a gift for the 40's birthday party of COBOL. But, it is a cheap present, maybe worth between 20.000 and 40.000 US Dollar.

To give you an idea of the low complexity of this effort, we depicted an original syntax diagram (taken from [6]) in Figure 1. After conversion to BNF, and correction it looks like this:

```
search-statement =
  "SEARCH" identifier ["VARYING" (identifier | index-name)]
  [{"AT"} "END" statement-list]
  {"WHEN" condition (statement-list | "NEXT" "SENTENCE")+
  ["END-SEARCH"]
```

A dash is removed between NEXT and SENTENCE. Furthermore, both occurrences of imperative-statement are replaced by list sorts. This is an example of a diagram that was too restrictive: only one occurrence was mentioned but in the informal text we learned that: “A series of imperative statements can be specified whenever an imperative statement is allowed.” We modified this in the BNF. Both errors were found using our error-detection parser: we parsed code where NEXT SENTENCE was used but without a dash. Upon inspection of the manual and grammar we wrote a grammar transformation repairing the error. Also in the second case, we parsed code where more statements were allowed by the compiler but not by the manual. We repaired the error with a grammar transformation. The resulting production rule does not lead to ambiguities.

We also give an example of an ambiguous rule. In the COBOL CALL statement the following fragment of a syntax diagram is present:

```
-----identifier-----
|_ADDRESS_OF_identifier_|
|_file-name_|
```

The above stack of 3 alternatives can lead to an ambiguity. How? Both identifier and file-name eventually reduce to the same lexical category. So when we parse a COBOL CALL statement without an occurrence of ADDRESS OF both other alternatives are valid, leading to an ambiguity. The description in the manual is mixing the syntax description with providing type information: an identifier vs. a file-name. With a grammar transformation we remove in a sense some typing information by eliminating the file-name alternative. We first show you the ambiguous extracted BNF fragment.

```
(identifier | "ADDRESS" "OF" identifier | file-name)
```

With a grammar transformation we eliminated the file-name alternative.

```
(identifier | "ADDRESS" "OF" identifier)
```

Note that with the adapted grammar the same language is recognized as before, only an ambiguity is gone. In this way we recovered the entire VS COBOL II grammar and tested it with all the code we obtained from earlier software renovation projects and code from colleagues who were curious to the outcome of this project. All in all we parsed about 2 million lines of VS COBOL II code. As in the PLEX case, we generated a fully web-enabled version of both the corrected BNF and the syntax diagrams that could serve as the core for a complete and correct language reference manual. It is freely available on the Internet [10]. Directly after its publication the URL was added to the `comp.compilers` FAQ, many sites linked it, and at the time of writing this paper it is retrieved about a thousand times a month.

Discussion This work has been reported on in two extensive research papers [12, 11]. In our attempt to recover the PLEX grammar from a manual we failed since the language reference manual was erroneous and incomplete (but the tools were okay). Of course, there are many more errors in manuals than in compiler source code. To give you an idea, we also recovered a raw PL/1 and a COBOL 2000 grammar from IBM manuals, but we did not yet correct the errors (there are still hundreds of errors to deal with, which gives you a nice comparison how quickly such errors are repaired using our process). The extraction effort took five hours. We had to adapt the tools since IBM used a different syntax diagram notation in the PL/1 manual. Many problems are still present in the raw extracted grammars, but the individual production rules are almost all correct: they can be used to implement so-called island parsers that are suitable for rapid system analysis [3]. Also the tools for the COBOL case study are simple and straightforward, and similar to the ones defined in [15]. As one reviewer of [15] put it “the so-called ‘tools’ described in the paper seem to be only small functions used on the grammar. Maybe the word ‘tool’ is a bit overstated.” This remark is the best compliment: the approach is so simple that it is not even worth calling the ultra simple functions tools, like it is not even worth calling a matchbox an invention.

5 Conclusion

We explained what the 500 language problem is, how it can be solved in general, and how this general solution works for two very complex and representative cases. We sketched a cost-effective and accurate method to quickly produce parsers for the myriad of languages so that existing code can be analyzed and modified using tools that work on the trees produced by the parsers. We worked on more grammar extraction projects and from those efforts some interesting lessons can be learned.

- Never try to crack the 500 language problem by initiating a language conversion.
- The more obscure a language is, the more chance that you have direct access to the compiler sources which is an excellent source for grammar recovery.

- The more mainstream a language is, the more chance that you have direct access to a reasonably good language reference manual that is debugged by its many users which is an excellent source for grammar recovery.

Case closed.

References

- [1] J.W. Backus. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In S. de Picciotto, editor, *Proceedings of the International Conference on Information Processing*, pages 125–131. Unesco, Paris, 1960.
- [2] M. Berg. Reverse engineering PLEX-C code to SDL 10 code. Master's thesis, Lund Institute of Technology, Department of Communication Systems, 1999.
- [3] A. van Deursen and T. Kuipers. Building documentation generators. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society Press, 1999.
- [4] P.T. Devanbu. GENOA - A Customizable, front-end Retargetable Source Code Analysis Framework. *ACM Transactions on Software Engineering and Methodology*, 8(2):177–212, 1999.
- [5] J. Evans and R. Verbruggen. The Application of Code Based Metrics to a Proprietary Language. Technical Report CA1796, Dublin City University, School of Computer Applications, 1996. Retrieved via: <ftp://ftp.compapp.dcu.ie/pub/w-papers/1996/CA1796.ps.Z>.
- [6] IBM Corporation. *VS COBOL II Reference Summary*, 1.2. Publication number SX26-3721-05 edition, 1993.
- [7] P. de Jager. You've got to be kidding!, 1997. Retrieved via: <http://www.year2000.com/archive/NFkidding.html>.
- [8] C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.
- [9] Capers Jones. *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences*. Addison-Wesley, 1998.
- [10] R. Lämmel and C. Verhoef. *VS COBOL II grammar Version 1.0.3*, 1999. Available at: <http://adam.wins.uva.nl/~x/grammars/vs-cobol-ii/>.
- [11] R. Lämmel and C. Verhoef. Rejuvenation of an Ancient Visual Language. Draft; available at <http://www.cwi.nl/~ralf/>; submitted for publication, April 2000.
- [12] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery, 2000. Available via: <http://adam.wins.uva.nl/~x/ge/ge.html>.
- [13] M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions – extended abstract. In B. Nuseibeh, D. Redmiles, and A. Quilici, editors, *Proceedings of the 13th International Automated Software Engineering Conference*, pages 314–317, 1998. For a full version see [15]. Available at: <http://adam.wins.uva.nl/~x/ase/ase.html>.
- [14] M.P.A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 245–255. IEEE Computer Society Press, 1999. Available via <http://adam.wins.uva.nl/~x/com/com.html>.
- [15] M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE Computer Society, March 2000. Full version of [13]. Available at: <http://adam.wins.uva.nl/~x/cale/cale.html>.
- [16] A.A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, 2000. To Appear. Available at <http://adam.wins.uva.nl/~x/cnv/cnv.html>.
- [17] G.M. Weinberg. *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971.