Cracking the 500-Language Problem

Ralf Lämmel and Chris Verhoef, *Free University of Amsterdam*

Parser implementation effort dominates the construction of software renovation tools for any of the 500+ languages in use today. The authors propose a way to rapidly develop suitable parsers: by stealing the grammars. They apply this approach to two nontrivial, representative languages, PLEX and VS Cobol II.

At least 500 programming languages and dialects are available in commercial form or in the public domain, according to Capers Jones.[1] He also estimates that corporations have developed some 200 proprietary languages for their own use. In his 1998 book on estimating Year 2000 costs, he indicated that systems written in all 700 languages would be affected.[2] His findings inspired many Y2K whistle-blowers to characterize this situation as a major impediment to solving the Y2K problem; this impediment became known as the 500-Language Problem.

In 1998, we realized that we had discovered a breakthrough in solving the 500LP—so we had something to offer regarding the Y2K problem. We immediately informed all the relevant Y2K solution providers and people concerned with the Y2K awareness campaign. In answer to our emails, we received a boilerplate email from Ed Yourdon explaining that the 500LP was a major impediment to solving the Y2K problem (which we knew, of course). Ed was apparently so good at creating awareness that this had backfired on him: he got 200 to 300 messages a day with Y2K questions and was no longer able to read, interpret, and answer his email other than in "write-only" mode. Although he presumably missed our input, his response regarding the 500LP is worth quoting:

> *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 1,000 in the preserving process for so-called 1,000-year-old eggs, which last only 100 days. For a start, we

0740-7459/01/$10.00 © 2001 IEEE

should add the 200 proprietary languages. Moreover, other estimates indicate that 700 is rather conservative: in 1971, Gerald Weinberg estimated that by the following year, programming languages would be invented at the rate of one per week—or more, if we consider the ones that never make it to the literature, and enormously more if we consider dialects.[3]

Peter de Jager also helped raise awareness of the 500LP. He writes this about the availability of Y2K tools:[4]

> *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.*

If everyone were using Cobol and only a few systems were written in uncommon languages, the 500-Language Problem would not be important. So, knowing the actual language distribution of installed software is useful. First, there are about 300 Cobol dialects, and each compiler product has a few versions—with many patch levels. Also, Cobol often contains embedded languages such as DMS, DML, CICS, and SQL. So there is no such thing as "the Cobol language." It is a polyglot, a confusing mixture of dialects and embedded languages—a 500-Language Problem of its own. Second, according to Jones, the world's installed software is distributed by language as follows:

- Cobol: 30 percent (225 billion LOC)
- C/C++: 20 percent (180 billion LOC)
- Assembler: 10 percent (140 to 220 billion LOC)
- less common languages: 40 percent (280 billion LOC)

In contrast, there were Y2K search engines for only about 50 languages and automated Y2K repair engines for about 10 languages.[2] Thus, most languages had no automated modification support, clarifying the concerns of Jones, Yourdon, McCabe, de Jager, and others. These alarming figures underscored the 500LP's importance.

What is the 500-Language Problem?

We entered the new millennium without much trouble, so you might conclude that whatever the 500LP was, it is not relevant now. Of course, the problem existed before the Y2K gurus popularized it, and it has not gone away.

Why is the problem still relevant? If you want tools to accurately probe and manipulate source code, you must first convert the code from text format to tree format. To do this, you need a so-called syntactic analyzer, or parser. But constructing a parser is a major effort, and the large up-front investment hampers initiatives for many commercial tool builders. Indeed, a few years ago Tom McCabe told us that his company, McCabe & Associates, had made a huge investment in developing parsers for 23 languages. Noting that 500 would be insurmountable, he dubbed this problem "the number one problem in software renovation." Thus, the 500LP is the most prominent impediment to constructing tools to analyze and modify existing software assets. Because there are about a trillion lines of installed software written in myriad languages, its solution is a step forward in managing those assets.

Solutions that don't work

A solution sometimes suggested for the 500LP is just to convert from uncommon languages to mainstream ones for which tool support is available. However, you need a full-blown tool suite—including a serious parser—to do this. And obtaining a parser is part of the 500LP. So language conversion will not eliminate the problem—on the contrary, you need a solution for the 500LP to solve conversion problems.

A Usenet discussion on comp.compilers offered a second suggestion to solve the 500LP: generating grammars from the source code only, in the same way linguists try to generate a grammar from a piece of natural language. In search of solutions, we studied this idea and consulted the relevant literature. We did not find any successful effort where the linguistic approach helped to create a grammar for a parser in a cost-effective way. We concluded that the linguistic approach does not lead to useful grammar inferences from which you can build parsers.[5]

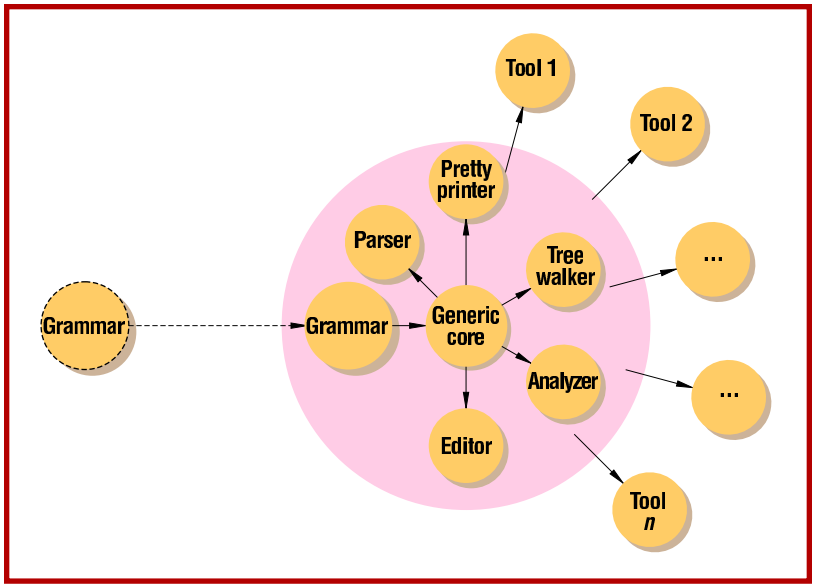

Figure 1. Effort shift for renovation tool development. The longer the arrow, the more effort is needed. The dashed line represents the greater effort needed if the traditional approach is used.

Another, more reasonable suggestion is to reuse the parser from compilers: just tap a compiler's parser output and feed it to a modification tool. Prem Devanbu's programmable GENOA/GENII tool can turn a parser's idiosyncratic output format into a more suitable format for code analysis.[6] There is, however, one major drawback to this approach: as Devanbu points out, the GENOA system does not allow code modification. This is not a surprise: a compiler's parser removes comments, expands macros, includes files, minimizes syntax, and thus irreversibly deforms the original source code. The intermediate format is good enough for analysis in some cases, but the code can never be turned into acceptable text format again. Hence, the approach does not help regarding mass modifications, for which the Gartner Group recommends tool support to handle a larger code volume.[7,8] Obviously, this concerns Y2K and Euro conversions, code restructuring, language migrations, and so on. Another real limitation of Devanbu's approach is that, even if you only want to do code analysis, you often cannot get access to a compiler company's proprietary source.

How we are cracking the 500LP

Recall that Yourdon claimed that the large number of programming languages would virtually eliminate any chance of a single tool, method, or technique being universally applicable. Nevertheless, there is a single, feasible solution for the 500LP. It is cracked when there is a cheap, rapid, and reliable method for producing grammars for the myriad languages in use so that existing code can be analyzed and modified. *Cheap* is in the US$25,000 ± $5,000 range, *rapid* is in the two-week range (for one person), and *reliable* means the parser based on the produced grammar can parse millions of LOC.

Why is this a solution? A grammar is hardly a Euro conversion tool or a Y2K analyzer. It is because the most dominant factor in building renovation tools is constructing the underlying parser.

From grammar to renovation tool

Renovation tools routinely comprise the following main components: preprocessors, parsers, analyzers, transformers, visualizers, pretty printers, and postprocessors. In many cases, language-parameterized (or generic) tools are available to construct these components. Think of parser generators, pretty-printer generators, graph visualization packages, rewrite engines, generic dataflow analyzers, and the like. Workbenches providing this functionality include Elegant, Refine, and ASF+SDF, for instance, but there are many more.

Figure 1 depicts a grammar-centric approach to enabling rapid development of renovation tools. Arrow length indicates the degree of effort involved (longer arrows imply more effort). As you can see, if you have a generic core and a grammar, it does not take much effort to construct parsers, tree walkers, pretty printers, and so on. Although these components depend on a particular language, their implementation uses generic language technology: a parser is produced using a parser generator, a pretty printer is created using a formatter generator,[9] and tree walkers for analysis or modification are generated similarly.[10] All these generators rely heavily on the grammar. Once you have the grammar and the relevant generators, you can rapidly set up this core for developing software renovation tools. Leading Y2K companies indeed constructed generic Y2K analyzers, so that dealing with a new language would ideally reduce to constructing a parser. The bottleneck is in obtaining complete and correct grammar specifications. The longest arrow in Figure 1 expresses the current situation: it takes a lot of effort to create those grammars.

Implementing a high-quality Cobol parser can take two to three years, as Vadim Maslov of Siber Systems posted on the Usenet newsgroup comp.compilers (he has constructed Cobol parsers for about 16 dialects). Adapting an existing Cobol parser to cope with new dialects easily takes three to five months. Moreover, patching existing grammars using mainstream parser technology leads to unmaintainable grammars,[11,12] significantly increasing the time it takes to adapt parsers. In contrast, Table 1 lists the effort expended on various phases of a typical Cobol renovation project that used our grammar-centric solution. Notice that the grammar part of this project took only two weeks of effort, so the team could start developing actual renovation tools much more quickly.

This Cobol renovation project concerned one of the world's largest financial enterprises, which needed an automatic converter from Cobol 85 back to Cobol 74 (the 8574 Project).[13] The Cobol 85 code was machine-generated from a fourth-generation-language tool, so the problem to convert back was fortunately restricted due to the code generator's limited vocabulary. It took some time to solve intricate problems, such as how to simulate Cobol 85 features like explicit scope terminators (`END-IF`, `END-ADD`) and how to express the `INITIALIZE` statement in the less-rich Cobol 74 dialect. The developers discussed solutions with the customer and tested them for equivalence. Once they solved these problems, implementing the components was not difficult because they had the generic core assets generated from a recovered Cobol 85 grammar. They cut the problem into six separate tools and then implemented all of them in only five days. The programming by hand was limited (fewer than 500 LOC), but compiled into about 100,000 lines of C code and 5,000 lines of `makefile` code (linking all the generated generic renovation functionality). After compilation to six executables (2.6 Mbytes each), it took 25 lines of code to coordinate them into a distributed, component-based software renovation factory, which then converted Cobol 85 code to Cobol 74 at a rate of 500,000 LOC per hour using 11 Sun workstations.

Table 1

Effort for the 8574 Project

Phase	Effort
Extract the grammar	Two weeks
Generate the parser	One day
Build six tools	Five days
Assemble all the components	One hour
Total	Three weeks

Measuring this and other projects, it became clear to us that the total effort of writing a grammar by hand is orders of magnitude larger than constructing the renovation tools themselves. So the dominant factor in producing a renovation tool is constructing the parser. Building parsers using our approach reduces the effort to the same order of magnitude as constructing the tools. Building parsers in turn is not hard: use a parser generator. But the input for the generator is a grammar description, so complete and correct grammars are the most important artifacts we need to enable tool support. When we find an effective solution for producing grammars quickly for many languages, we have solved the 500LP.

When we find an effective solution for producing grammars quickly for many languages, we have solved the 500LP.

But how do we produce grammars quickly? For years, we and many others have been recapturing an existing language's syntax by hand: we took a huge amount of sources, manuals, books, and a parser generator and started working. But then we realized that this hand work is not necessary. Because we are dealing with existing languages, we just steal and massage the underlying grammars according to our needs.

Grammar stealing covers almost all languages

The following exhaustive case distinction shows that our approach covers virtually all languages. Let's look at the coverage diagram for grammar stealing shown in Figure 2. Because the software we want to convert already exists, it can be compiled or interpreted. We first enter the Compiler Sources diamond. There are two possibilities: the source code is or is not available to you. If it is, you just have to find the part that turns the text into an intermediate form. That part now contains the grammar in some form. You do this by lexically searching the compiler source code for the language's keywords.

Compiler constructors implement a parser in one of three ways: they hard-code it, use a parser generator, or do both (in a complex multilanguage compiler, for in-

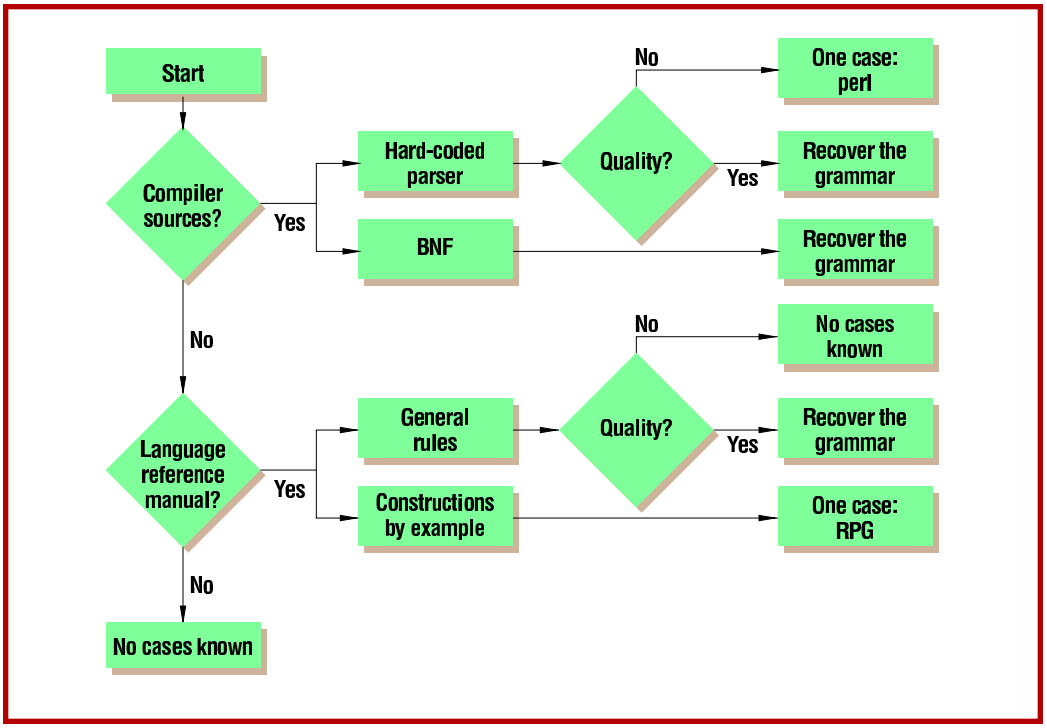

Figure 2. Coverage diagram for grammar stealing.

stance). Figure 2 shows the first two cases—the third is just a combination. If you start with a hard-coded grammar, you must reverse-engineer it from the handwritten code. Fortunately, the comments of such code often include BNF rules (Backus Naur Forms) indicating what the grammar comprises. Moreover, because compiler construction is well-understood (there is a known reference architecture), compilers are often implemented with well-known implementation algorithms, such as a recursive descent algorithm. So, the quality of a hard-coded parser implementation is usually good, in which case you can easily recover the grammar from the code, the comments, or both. Except in one case, the Perl language,[14] the quality of the code we worked with was always sufficient to recover the grammar.

If the parser is not hard-coded, it is generated (the BNF branch in Figure 2), and some BNF description of it must be in the compiler source code. So, with a simple tool that parses the BNF itself, we can parse the BNF of the language that resides in the compiler in BNF notation, and then extract it.

When the compiler source code is not accessible (we enter the Language Reference Manual diamond in Figure 2), either a reference manual exists or not. If it is available, it could be either a compiler vendor manual or an official language standard. The language is explained either by example, through general rules, or by both approaches. If a manual uses general rules, its quality is generally not good: reference manuals and language standards are full of errors. It is our experience that the myriad errors are repairable. As an aside, we once failed to recover a grammar from the manual of a proprietary language for which the compiler source code was also available (so this case is covered in the upper half of Figure 2). As you can see in the coverage diagram, we have not found low-quality language reference manuals containing general rules for cases where we did not have access to the source code. That is, to be successful, compiler vendors must provide accurate and complete documentation, even though they do not give away their compilers' source code for economic reasons. We discovered that the quality of those manuals is good enough to recover the grammar. This applies not only to compiler-vendor manuals but also to all kinds of de facto and official language standards.

Unusual languages rarely have high-quality manuals: either none exists (for example, if the language is proprietary) or the company has only a few customers. In the proprietary case, a company is using its in-house language and so has access to the source code; in the other case, outsiders can buy the code because its business value is not too high. For instance, when Wang went bankrupt, its

key customers bought the source code for its operating system and compilers to create their own platform and dialect migration tools. This explains why we do not know of low-quality manuals containing general rules. In one case, that of RPG, the manual explains the language through code examples, and general rules are absent. We can examine this case in more detail if we are asked for an RPG renovation project involving a large amount of RPG code. We think we can systematically extract RPG's general rules from the code examples.

In addition, because the manual contains code examples, there is a good chance that the compiler has tested these examples. This means that the manual's formal content could be of a much higher quality than you would expect from such documents.

Finally, we must deal with the case in which we have no access to the compiler sources or a reference manual. 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." He did not come up with actual examples. Recall Wang's bankruptcy: key customers just bought the source code and hence could solve their problems using the upper half of Figure 2. Theoretically, we cannot exclude Jones's case—for instance, responding emotionally, Wang's core developers could have thrown away the sources. You can learn an important lesson from this: Contracts between you and the vendor of a business-critical language should include a solution for source access in case of bankruptcy or terminated support (for example, giving the sealed source code to key customers). Summarizing, our coverage diagram shows that you can recover virtually any grammar, whether you have the compiler sources or not.

But what about semantics?

Some people think you need up-front, in-depth knowledge of a language's semantics to change code. If you recover the BNF, you can generate a syntax analyzer that produces trees, but the trees are not decorated with extra knowledge such as control flow, data flow, type annotation, name resolution, and so on. Some people also think you need a lot of semantical knowledge to analyze and modify existing software, but this is not true. You can try to capture a language's semantical knowledge on three levels:

- for all the compilers of a language (different dialects),
- for one compiler product, or
- on a project-by-project basis.

Because we are trying to facilitate the construction of tools that work on existing software, there is already a compiler. This has implications for dealing with the semantical knowledge. Consider the following Cobol excerpt:

```
PIC A X(5) RIGHT JUSTIFIED
VALUE 'IEEE'.
DISPLAY A.
```

The OS/VS Cobol-compiled code prints the expected result—namely, "`IEEE`"—which is right justified. However, because of a change in the 1974 standard, the same code compiled with a Cobol/370 compiler displays the output "`IEEE `" with a trailing space, which is left justified. This is because the `RIGHT JUSTIFIED` phrase does not affect `VALUE` clauses in the case of the Cobol/370 compiler. There are many more such cases, so trying to deal with the semantics of all compilers in advance is not feasible. Even when you restrict yourself to one compiler, this problem does not go away. Consider this Cobol fragment:

```
01 A PIC 9999999.
MOVE ALL '123' to A.
DISPLAY A.
```

Depending on the compiler flags used to compile this code, the resulting executables display either 3123123 or 1231231. There are hundreds of such problems, so it is also infeasible to capture the semantics in advance for a single compiler. No single semantics is available, and gathering all variants is prohibitively expensive and error prone given the semantical differences between compilers, compiler versions, and even compiler flags used.

The good news is that you only need specific ad hoc elements of the semantics on a per-project basis. We call this *demand-driven* semantics. For instance, the `NEXT SENTENCE`

Some people think you need up-front, in-depth knowledge of a language's semantics to change code.

We recovered the PLEX grammar in two weeks, including tool construction, parser generation, and testing, at a cost of US$25,000.

```
IF X=1 THEN
  IF Y=1 THEN
    NEXT SENTENCE
  END-IF
  DISPLAY 'Nested IF passed'
END-IF.
DISPLAY 'SENTENCE passed'.
```

(a)

```
IF X=1 THEN
  IF Y=1 THEN
    CONTINUE
  END-IF
  DISPLAY 'Nested IF passed'
END-IF.
DISPLAY 'SENTENCE passed'.
```

(b)

Figure 3. A code segment (a), transformed inappropriately into the code segment in (b). The change in line 3 results in wrong output.

phrase in Cobol directs control to the statement after the next separation period (denoted with a dot). So, depending on where people put a dot, the code jumps directly behind the dot. Omitting a dot can lead to different behavior. One of our customers wanted tools to get rid of this potentially hazardous implicit jump instruction. Luckily, it turned out that for this project, we could replace the implicit jump instruction `NEXT SENTENCE` with the innocent no-op `CONTINUE`. So, after our semantical investigation, we knew we could use a simple transformation tool to make this change. However, in another project, this transformation might break down—for example, if the `NEXT SENTENCE` phrase is used in problematic code patterns. The transformation of the code in Figure 3a into the code in Figure 3b has changed the program's meaning: you cannot turn `NEXT SENTENCE` into `CONTINUE` in the context of Figure 3. Specifically, assuming both X and Y are equal to 1, the code in Figure 3a prints "`SENTENCE passed`" while the code in Figure 3b prints first "`Nested IF passed`" and then "`SENTENCE passed`". As you can see, you must be utterly aware of the semantics to find out whether it is necessary to implement any of it. In most cases we have seen, *implementing* this type of intricate semantical issue was not necessary—but knowing about the potential problems was necessary, if only to check whether they were present.

To give you an idea how far you can go with demand-driven semantics, consider this: we have developed relatively dumb tools for some Cobol systems that can wipe out complex `GO TO` logic.[15] You do need to know the semantics for many different tasks, but it is not necessary in advance to encode the compiler semantics in a parse tree or otherwise. So, a tool developer can construct (mostly) syntactic tools taking semantical knowledge into account on a per-project basis.

Grammar stealing in practice

We—and others from industry and academia—have applied grammar stealing successfully to a number of languages, including Java, PL/I, Ericsson PLEX, C++, Ada 95, VS Cobol II, AT&T SDL, Swift messages, and more. Here, we focus on PLEX (*P*rogramming *L*anguage for *Ex*changes), a proprietary, nontrivial, real-time embedded-system language (for which the compiler source code was accessible to us), and, at the other end of the gamut, VS Cobol II, a well-known business language (for which no compiler code was available). Both languages are used in business-critical systems: the AXE 10 public branch exchange uses PLEX, and numerous IBM mainframe systems run VS Cobol II. These two languages represent the two main branches in Figure 2.

Our approach uses a unique combination of powerful techniques:

- automated grammar extraction,
- sophisticated parsing,
- automated testing, and
- automated grammar transformation.

If one of these ingredients is missing, the synergy is gone. Extraction by hand is error prone, and basic parsing technology limits you to work with grammars in severely limited formats. With powerful parsing technology, you can work with arbitrary context-free grammars and test them regardless of their format. Without automated testing, you cannot find many errors quickly. Without tool support to transform grammar specifications, analyses are inaccurate and corrections are inconsistent; without transformations, you cannot repeat what you have done or change initial decisions easily. So, to steal grammars, you need to know about grammars, powerful parsing techniques, how to set up testing, and automated transformations.

```
<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)
        ;
```

Figure 4. Raw compiler source code for the PLEX language.

The tools we built automatically recovered most of the 3,000+ production rules in an afternoon.

Stealing from compiler source code

Ericsson uses the extremely complex proprietary language PLEX to program public telephone switches. PLEX consists of about 20 sublanguages, called *sectors*, including high-level programming sectors, assembly sectors, finite-state-machine sectors, marshaling sectors, and others. We applied our grammar-stealing approach to PLEX as follows:[16]

1. Reverse-engineer the PLEX compiler (63 Mbytes of source code) on site to look for grammar-related files. We learned that there were BNF files and a hard-coded parser.
2. Find the majority of the grammars in some BNF dialect.
3. Find a hand-written proprietary assembly parser with BNF in the comments.
4. Write six BNF parsers (one for each BNF dialect used).
5. Extract the plain BNF from the compiler sources and convert it to another syntax definition formalism (SDF) for technical reasons.
6. Find the files containing the lexical analyzer and convert the lexical definitions to SDF.
7. Combine all the converted grammars into one overall grammar.
8. Generate an overall parser with a sophisticated parser generator.
9. Parse the code.

We recovered the PLEX grammar in two weeks, including tool construction, parser generation, and testing with 8-MLOC PLEX code, at a cost of US$25,000. Ericsson told us that a cutting-edge reengineering company had estimated this task earlier at a few million dollars. When we contacted this company, they told us that US$25,000 was *nothing* for such a grammar.

To illustrate the limited complexity of the work, consider the fragment of raw compiler source code in Figure 4. A PLEX program consists of a header, a list of statements, the phrase `END PROGRAM`, and a closing semicolon. The other code in the figure deals with semantic actions relevant to the compiler. Our tools converted this to a common BNF while removing the idiosyncratic semantic actions:

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

Then our tools converted this into SDF, which was subsequently fed to a sophisticated parser generator accepting arbitrary context-free grammars. The output was

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

The tools we built automatically recovered most of the 3,000+ production rules in an afternoon. Then we tested each sector grammar separately. We used a duplicate detector to weed out production rules that were used in more than one sector grammar, so that we could construct an overall grammar able to parse complete PLEX programs. One assembly sector parser was hard-coded (see Figure 2), so we had to recover its grammar through reverse engineering. The comments accompanying the code contained reasonably good BNF, so we had no problem with this task. With all the sector grammars combined, we generated a parser to test it with an 8-MLOC PLEX test suite. The only files that did not parse were compiler test files that were not supposed to parse—the rest passed the test. In addition, we generated a Web-enabled version of the BNF description as a basis for a complete and correct manual.

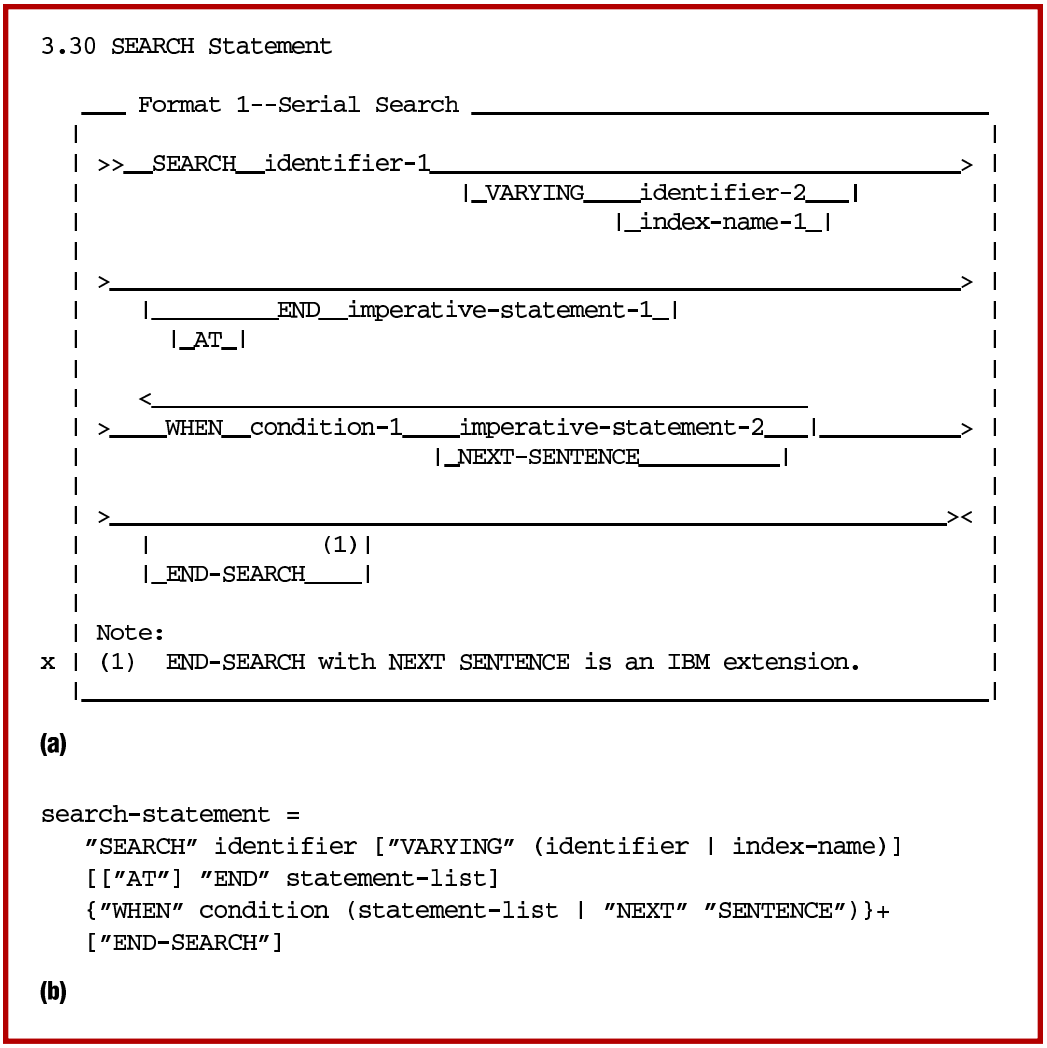

Figure 5. (a) The original syntax diagram for the Search statement; (b) the same diagram after conversion to BNF and correction.

Stealing from reference manuals

Some of our colleagues felt a little fooled by the PLEX result: "You are not really constructing a parser; you only converted an existing one. We can do that, too. Now try it without the compiler." Indeed, at first sight, not having this valuable knowledge source available seemed to make the work more difficult. After all, an earlier effort to recover the PLEX grammar from various online manuals had failed: they were not good enough for reconstructing the language.[17] Later, we discovered that the manuals lacked over half of the language definition, so that the recovery process had to be incomplete by definition. We also found that our failure was due not to our tools but to the nature of proprietary manuals: if the language's audience is limited, major omissions can go unnoticed for a long time. When there is a large customer base, the language vendor has to deliver better quality.

In another two-week effort,[5] we recovered the VS Cobol II grammar from IBM's manual *VS COBOL II Reference Summary*, version 1.2. (For the fully recovered VS Cobol II grammar, see www.cs.vu.nl/grammars/vs-cobol-ii.) Again, the process was straightforward:

1. Retrieve the online VS Cobol II manual from www.ibm.com.
2. Extract its syntax diagrams.
3. Write a parser for the syntax diagrams.
4. Extract the BNF from the diagrams.
5. Add 17 lexical rules by hand.
6. Correct the BNF using grammar transformations.
7. Generate an error-detection parser.
8. Incrementally parse 2 million lines of VS Cobol II code.
9. Reiterate steps 6 through 8 until all errors vanish.
10. Convert the BNF to SDF.
11. Generate a production parser.

12. Incrementally parse VS Cobol II code to detect ambiguities.
13. Resolve ambiguities using grammar transformations.
14. Reiterate steps 11 through 13 until you find no more ambiguities.

So, apart from some cycles to correct errors and remove ambiguities, the process is the same as in the earlier case, where we had access to the compiler source. An error-detection parser detects errors in the grammar from which it is generated. In this case, we used an inefficient top-down parser with infinite lookahead. It accepts practically all context-free grammars and does not bother with ambiguities at all. We use this kind of parser to test the grammar, not to produce parse trees. Because we only used compilable code, all the errors that this parser detects raise potential grammar problems. In this way, we found all the omissions, given our Cobol testbed. When all our test code passed the top-down parser, we converted the grammar to SDF, generated a parser that detects ambiguities, and corrected them. This project also took two weeks of effort, including tool construction and testing. We did this for free, so that we could freely publish the grammar on the Internet as a gift for Cobol's 40th birthday.[18]

To get an idea of the limited complexity of this technique, consider the syntax diagram shown in Figure 5a, taken from the manual. After conversion to BNF and correction, the diagram looks like the one in Figure 5b.

We used grammar transformations to remove the dash between `NEXT` and `SENTENCE` and to replace the two occurrences of `imperative-statement` with `statement-list`. The diagram was overly restrictive, allowing only one statement. However, in the manual's informal text we learned, "A series of imperative statements can be specified whenever an imperative statement is allowed." Our error-detection parser found these errors: first, the tool parsed code when it found `NEXT SENTENCE`, that is, without a dash. After inspecting the manual and grammar, we wrote a grammar transformation repairing this error. The error-detection parser also found that, according to the compiler, more than one statement was correct whereas the manual insisted on exactly one statement. We repaired this error with a grammar transformation.

Next, in a separate phase, we removed ambiguities. For example, the following fragment of a syntax diagram is present in the Cobol `CALL` statement:

```
_____identifier_____________________
  |__ADDRESS__OF__identifier__|
  |__file-name_________________|
```

This stack of three alternatives can lead to an ambiguity. Namely, both `identifier` and `file-name` eventually reduce to the same lexical category. So, when we parsed a `CALL` statement without an occurrence of `ADDRESS OF`, the parser reported an ambiguity because the other alternatives were both valid. Without using type information, we cannot separate `identifier` from `file-name`. This is the ambiguous extracted BNF fragment:

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

With a grammar transformation, we eliminated the `file-name` alternative, resulting in

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

The adapted grammar accepts the same language as before, but an ambiguity is gone. Note that this approach is much simpler than tweaking the parser and scanner to deal with types of names. In this way, we recovered the entire VS Cobol II grammar and tested it with all our Cobol code from earlier software renovation projects and code from colleagues who were curious about the project's outcome. For the final test, we used about two million lines of pure 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.

The more mainstream a language is, the more likely that you will have direct access to a reasonably good, debugged language reference.

Apart from PLEX and Cobol, we have recovered several other grammars, as have others. From our efforts in solving the 500LP, we learned two interesting lessons. First, the more uncom-

mon a language is, the more likely that you will have direct access to the compiler's source code, an excellent starting place for grammar recovery. Second, the more mainstream a language is, the more likely that you will have direct access to a reasonably good, debugged language reference, also an excellent source for grammar recovery.

Acknowledgments

Thanks to Terry Bollinger, Prem Devanbu, Capers Jones, Tom McCabe, Harry Sneed, Ed Yourdon, and the reviewers for their substantial contributions.

About the Authors

Ralf Lämmel is a lecturer at the Free University of Amsterdam and is affiliated with the Dutch Center for Mathematics and Computer Science (CWI). His research interests include program transformation and programming languages. As a freelancer and consultant, he has designed, implemented, and deployed developer tools, migration tools, and software development application generators based on Cobol and relational databases. He received his PhD in computer science from the University of Rostock, Germany. Contact him at the Free Univ. of Amsterdam, De Boelelaan 1081-A, 1081 HV Amsterdam, Netherlands; ralf@cs.vu.nl; www.cs.vu.nl/~ralf.

Chris Verhoef is a computer science professor at the Free University of Amsterdam and principal external scientific advisor of the Deutsche Bank AG, New York. He is also affiliated with Carnegie Mellon University's Software Engineering Institute and has consulted for hardware companies, telecommunications companies, financial enterprises, software renovation companies, and large service providers. He is an elected Executive Board member and vice chair of conferences of the IEEE Computer Society Technical Council on Software Engineering and a distinguished speaker of the IEEE Computer Society. Contact him at the Free Univ. of Amsterdam, Dept. of Mathematics and Computer Science, De Boelelaan 1081-A, 1081 HV Amsterdam, Netherlands; x@cs.vu.nl; www.cs.vu.nl/~x.

References

1. C. Jones, *Estimating Software Costs*, McGraw-Hill, New York, 1998.
2. C. Jones, *The Year 2000 Software Problem: Quantifying the Costs and Assessing the Consequences*, Addison-Wesley, Reading, Mass., 1998.
3. G.M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.
4. P. de Jager, "You've Got To Be Kidding!" www.year2000.com/archive/NFkidding.html (current 20 Sept. 2001).
5. R. Lämmel and C. Verhoef, "Semi-automatic Grammar Recovery," *Software: Practice and Experience*, vol. 31, no. 15, Dec. 2001, pp. 1395–1438; www.cs.vu.nl/~x/ge/ge.pdf (current 20 Sept. 2001).
6. P.T. Devanbu, "GENOA—A Customizable, Front-End Retargetable Source Code Analysis Framework," *ACM Trans. Software Eng. and Methodology*, vol. 8, no. 2, Apr. 1999, pp. 177–212.
7. B. Hall, "Year 2000 Tools and Services," *Symp./ITxpo 96, The IT Revolution Continues: Managing Diversity in the 21st Century*, Gartner Group, Stamford, Conn., 1996.
8. N. Jones, *Year 2000 Market Overview*, tech. report, Gartner Group, Stamford, Conn., 1998.
9. M.G.J. van den Brand and E. Visser, "Generation of Formatters for Context-Free Languages," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 1, Jan. 1996, pp. 1–41.
10. 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*, vol. 36, nos. 2–3, Mar. 2000, pp. 209–266; www.cs.vu.nl/~x/scp/scp.html (current 20 Sept. 2001).
11. M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef, "Current Parsing Techniques in Software Renovation Considered Harmful," *Proc. 6th Int'l Workshop Program Comprehension*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 108–117; www.cs.vu.nl/~x/ref/ref.html (current 20 Sept. 2001).
12. D. Blasband, "Parsing in a Hostile World," *Proc. 8th Working Conf. Reverse Eng.*, IEEE CS Press, Los Alamitos, Calif., 2001, pp. 291–300.
13. J. Brunekreef and B. Diertens, "Towards a User-Controlled Software Renovation Factory," *Proc. 3rd European Conf. Maintenance and Reengineering*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 83–90.
14. L. Wall, T. Christiansen, and R.L. Schwartz, *Programming Perl*, 2nd ed., O'Reilly & Associates, Cambridge, Mass., 1996.
15. M.P.A. Sellink, H.M. Sneed, and C. Verhoef, "Restructuring of Cobol/CICS Legacy Systems," to be published in *Science of Computer Programming*; www.cs.vu.nl/~x/res/res.html (current 20 Sept. 2001).
16. M.P.A. Sellink and C. Verhoef, "Generation of Software Renovation Factories from Compilers," *Proc. Int'l Conf. Software Maintenance*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 245–255; www.cs.vu.nl/~x/com/com.html (current 20 Sept. 2001).
17. M.P.A. Sellink and C. Verhoef, "Development, Assessment, and Reengineering of Language Descriptions," *Proc. 4th European Conf. Software Maintenance and Reengineering*, IEEE CS Press, Los Alamitos, Calif., 2000, pp. 151–160; www.cs.vu.nl/~x/cale/cale.html (current 20 Sept. 2001).
18. R. Lämmel and C. Verhoef, *VS COBOL II Grammar Version 1.0.3*, 1999; www.cs.vu.nl/grammars/vs-cobol-ii (current 20 Sept. 2001).