Generation of Software Renovation Factories from Compilers

Alex Sellink and Chris Verhoef
University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
alex@wins.uva.nl, x@wins.uva.nl

Abstract

When a compiler is designed carefully, it is possible to extract its grammar. We reengineer the extracted grammar to one that is geared towards reengineering. From this reengineering grammar we generate an architecture called a software renovation factory. This includes: generic analysis and transformation functionality and a native pattern language using the concrete syntax of the language for which the renovation is necessary. Moreover, we generate the grammar in HTML format so that reengineers can quickly understand the language. We applied our approach successfully to an exceptionally complex and large proprietary language. Our approach enables rapid development of software renovation factories. We believe that our approach can partly solve the lack of Year 2000 tool support for many languages.

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, Software renovation factories, Language description development, Grammar reengineering, Document generation, Computer aided language engineering (CALE).

1 Introduction

In various papers on the Year 2000 problem we can read that outside the mainstream languages the Year 2000 tool support is virtually nonexistent. In [40] Newcomb and Scott argue that the Year 2000 problem is an extraordinarily complex pervasive task and without automated tool support it becomes unmanageable. Moreover, they claim that language independent tool support cannot assess complex situations, and that even the best tools break down on multi-language systems, embedded sub-languages, and embedded data-manipulation languages. Capers Jones, in his recent book on the Year 2000 problem confirms the findings of Newcomb and Scott. In his book we can find that 30 percent of the software is written in hundreds of proprietary or obscure languages for which there are only few trained programmers. Even worse, for some of these languages there is no course or tutorial material [27, loc. cit. p. 86]. Jones mentions the example of CORAL66 and CHILL used for telecommunications that will be very expensive to repair due to the shortage of tools and lack of available programming talent. Indeed, this is a serious problem. In [44] Rekdal reports that there are worldwide 12 to 15 thousand CHILL programmers. Combine this with the fact that telecommunication software is complex, large in size (1 to 10 MLOC of CHILL, or some other switching language), in continuous development, and no single person has full understanding of the whole system [44]. Rekdal estimates that 45 percent of the digital local lines installed have been implemented using CHILL. Jones estimates that 20 percent of the telephone switching systems is implemented using CHILL and that there are a few thousand CHILL programmers so this is the same order of magnitude.

Also the mixed language issue that Newcomb and Scott [40] mention is confirmed by Jones: in [27] it is stated that when multiple programming languages are used for an application many of the Year 2000 search engines come to a halt. About 30 percent of the US software applications contain at least two languages. Twelve languages inside an application is the maximum that Jones has found among his clients. Preliminary data indicate an increase of 5 percent for each extra language of the costs of Year 2000 repair efforts [27, loc. cit. p. 50].

Needless to say that we need rapid development of tools that help in analysis and transformations. The bottleneck for rapid development of such tools is constructing the parser and its connection to existing back-ends. In [40] an automated correction assistant is discussed for which automatically identity transformations are generated. The user can modify them in order to define corrections, changes or translations. In [6] the generation of generic transformations and analyzers from arbitrary context-free grammars is discussed. So, if we have a grammar at our disposal we can generate full tool support that enables automated software renovation. In many papers we experience that construction of grammars is a huge problem. During the sixth workshop on program comprehension it was stated that parsers and their connectivity are a key issues for the Year 2000 problem. In [40] it is stated that organizations should be aware that grammar development is potentially time-consuming. If we combine this with the remark that Jones makes that there is a lack of course material etc, it is imaginable that for solving Y2K problems in the obscure 30 percent it will be difficult to develop tool support. Furthermore, Jones reports Year 2000 search engines support for less than 50 languages and Year 2000 repair engines are available for about 10 languages [27, loc. cit. p. 325].

Motivation If we look at the problems that we briefly sketched above, it is obvious that there is a strong demand for approaches that enable rapid grammar development in general and for reengineering purposes in particular. Since we can already generate software renovation factories from arbitrary context-free grammars [6], we focus in this paper on the huge parser problems.

A possible solution We propose in this paper an approach enabling the instantaneous generation a software renovation factory in the case that we have access to the source code of a compiler under the assumption that the source code of the compiler is carefully designed. We can generate from the source code of the front-end of the compiler a grammar that is geared towards reengineering and from that we can generate generic transformations and analyzers necessary for dealing with many and diverse reengineering tasks as elaborately discussed in [6]. Although we address possible scenarios to deal with the Year 2000 problem in connection with our approach, we stress that we do not generate a Year 2000 repair engine. Generic Year 2000 search engines exist [20], adding a new parser front end is then the bottleneck. Repair engines is more involved. we generate an architecture that enables rapid development of reengineering tools, like Year 2000 (analysis and) repair tools. We propose a solution for the parser problem and its connectivity by reengineering the source code of the parser thus creating a new one geared towards reengineering rather than reusing its output. If the compiler is designed carefully, the process is 100 percent automated, hence the instantaneous generation a software renovation factory. If there is less structure in the compiler source code, instantaneous generation is not possible. Depending on the degree of structure, it then takes more time to extract the grammar. Apart from that, the compiler source code is the best knowledge source for extracting the grammar.

Case study We applied our approach to an exceptionally complex proprietary language used in telecommunications that consists of about 20 (sub)languages, called sectors. Exceptionally, since Jones reports that among his clients 12 languages is the maximum. We call the example language SSL, short for switching system language. In fact, we are dealing with a mixed language containing, high level programming sectors, assembly sectors, finite state machine sectors, marshaling sectors, etc. It was not possible to use the language reference manual of SSL in order to construct or generate the grammar in a short time. Although the manuals were considered correct by the owners an earlier study revealed hundreds of errors. We reported on this approach in more detail in [49]. Apart from that, many sublanguages were not described in the manuals that we obtained.

To give the reader an idea of the usefulness of our approach, we were able to parse a 10 KLOC example program sector in our reengineering environment one hour after we migrated the original compiler sources from our customer. Upon first try the 10 KLOC program sector parsed without any errors. In half a day we generated about 3000 context-free production rules for eleven sublanguages and 3 accompanying lexical modules. As a comparison, in [8] we measured productivity of 300 production rules per month for the construction of COBOL grammars by hand. The code we had at our disposal parsed upon first try (+8 MLOC) except for one file. We inspected the generated HTML version of the grammar and concluded that the file could not be correct. Ericsson reported us later that the file was a compiler test file that was not supposed to parse.

Parser reuse In fact, one could argue that we reuse the parser of the compiler. In the literature there are more examples of parser reuse, they report on the reuse of the *output* of existing parsers, whereas we recycle the source code of the parser. As far as we know our approach towards this type of parser reuse is new.

In [21] it is stated that it is difficult, if not impossible, for code analyzers to employ the front-end of compilers. They discuss their experience with building a front-end for C++ from scratch. In [46] it is also noted that reusing existing parsers is very hard. They report on the lessons learned when they tried to adapt a reverse engineering tool to a dialect of the language and an embedded sublanguage. Their claim is to separate parsing and analysis. In [18, 17] this problem was also recognized: a special purpose language (called GENII), designed to convert typical parser output, is used to make converters to a format that is known by an analyzer generator tool called GENOA. It takes 1.600 LOC of GENII code to connect an existing front-end of C++ to GENOA. A serious limitation of this tool is that it is not possible to change code with it. Other limitations are that the grammar is not geared towards reengineering, the one that is simple to explain is the limitation that a parser for a compiler removes comments. This is not too much of a problem for analysis, for which GENOA is designed, but it is for software renovation. In [21] this is the reason that they reimplement the C++ grammar from scratch. So the results of our case study are completely orthogonal to the results that are reported on in the literature.

Related work This paper is our second contribution to crack the huge parser problem. In a previous paper we used similar technology to extract grammars from language reference manuals. Since they contain often many errors, this approach is not always feasible. However, when the documentation is reasonable, it is possible to generate a parser from the documentation, for instance, from language standards. Independently of our work, in [19] a similar approach has been applied with some extra hand work to obtain a C++ grammar from an AT&T document that was on the Internet. In [19] the C++ grammar is used to parse C++ programs in order to perform optimizing source to source transformations using algebraic specifications.

In the realm of natural language processing, the problems with large and complex grammars are also well-known. In [39] we can read that formal grammars for natural languages can become unmanageable when they evolve. In [10] maintenance problems with large and evolving grammars for reengineering purposes are addressed. In [39] it is noted that grammars are usually written by a small group of people. These people are the only ones who are able to understand its structure. If they leave, the grammar cannot be maintained anymore since its size and complexity prevent anyone else from gaining insight in it [39]. Therefore, in [39] tool support is provided for affix grammars [33] over a finite lattice for natural language. It consists of modularity support, analysis support for grammars, random test sentence generation, and a parser generator with trace facilities. The tool support was developed in the natural language front end project aiming at the construction of comprehensive and well-validated grammars and efficient parsers for a number of European languages. So their focus is on development of grammars for natural languages. Our focus is on reengineering of existing grammars for computer languages. Noteworthy perhaps, is that we use parser generator technology (GLR Parsing) for our parsers that also stem from natural language processing [35, 53, 45, 54], see [10] for an elaborate discussion.

Organization of the Paper In Section 2 we address the process of generating a reengineering parser. Then, in Section 3, we briefly discuss the generation of software renovation factories from a grammar. In Section 4 we discuss the obtained results and we put our work in the perspective of the Year 2000 problem. In Section 5 we make some concluding observations.

Acknowledgements We thank Johan Anderson, Leif Ekman, Peter Larsson, and Johanna Persson (Ericsson Reengineering Center) for their stimulating support. We thank Bob Diertens (University of Amsterdam) for carrying out scaling experiments. We thank the reviewers for their useful remarks and recommendations.

2 The Generation Process of the Parser

In this section we discuss the general idea of how we generate a parser for a software renovation factory and what exactly is generated during the process. In fact, the process that we describe in the section could very well be called Computer Aided Language Engineering (CALE). We use computer support to aid us in engineering language definitions. We develop them using CALE tools, we assess existing language definitions using CALE tools, we generate documentation for languages using CALE tools, we reengineer and migrate language definitions using CALE tools. We note that in natural language processing development of grammars using computer support is called lingware engineering [39].

2.1 Summary of CALE Tools

In order to get an idea of CALE tools we will briefly mention a few that we have developed in our CALE factory. All language descriptions are written in a programming language themselves. Most of the times this is a dialect of BNF [2]. Since there are many dialects for BNF it is not a good idea to develop tool support for every single dialect. Even within one compiler it is possible to have more than one BNF dialect (in our case study we about six different dialects). Therefore, one set of useful CALE tools are translators that translate from some BNF dialect to one specific very extended BNF. For our case study we needed about six such translators plus accompanying parsers of the BNF dialects. Once the dialects are translated one-to-one to our dedicated BNF, we have other classes of CALE tools at our disposal. One important class is tools to assess the quality of grammars. We developed simple but very useful quality assessment tools. We have tools that list top sorts (defined but not used sorts), bottom sorts (used but not defined sorts), all the sorts, all the keywords, the double rules, etc. When a grammar has one top-sort and a few bottom sorts then the quality of the grammar is perfect. Specifically, if the bottom sorts are lexical sorts, the chance that this grammar can be reused without human intervention is about 100 percent. In our case study most of the grammars had one top sort and only 3 to 5 bottom sorts that were lexical sorts. In all cases the bottom sorts of the context-free grammars happened to be top sorts in the lexical syntax modules. Therefore, we could construct working parsers without human intervention using our tools. Of course, then the parsers are not yet geared towards reengineering purposes. We use a variety of other tools, e.g., several tools that recognize simulated list constructs. We transform those simulated ones into more natural grammar constructs suited for reengineering. There is also an assembly line that migrates BNF to SDF (SDF stands for Syntax Definition Language, for which sophisticated parser generation support is available [24]. We also developed tools for generating HTML from SDF. Some of the tools mentioned above have been defined and used in [49]. Some others are new.

In the sequel we describe a computer aided language engineering process, where we use the abovementioned tools so that their usage becomes more clear.

2.2 Reverse Engineering the Compiler

The first step in the process is to reverse engineer the compiler source code. If the compiler is designed carefully, it is a simple task to locate the source code of the parser. Parser generator technology is well-known in compiler design (see, e.g., [1]). One of the most well-known parser generators is Yacc [25] in combination with a lexical analyzer called Lex [36]. So tracing the source files that are responsible for the parser is more simple when the reverse engineer knows the general design of compilers. A simple lexical search for keywords of the language reveals those files immediately, or looking for files that end in `.y`. In our case a simple search for a few keywords of the language revealed the entire parser part. Since a proprietary parser generator was used, looking for `.y` files would not have worked. If another technology is used, like recursive descent parsing, then another tool is needed to reverse engineer the grammar. The basic principle stays the same: extract the grammar from the source ocde of the compiler. Only the knowledge is not that explicit in the code.

When the portion of the compiler is detected that contains the grammar, we have to extract what we call the business rules. In this study the business rules are those parts of the compiler that we need to construct a parser geared towards reengineering targets. Usually, such files contain a dialect of BNF [2] for recognition of certain language construct plus code that deals with what to do with the code after recognition. Such additional code is called a semantic action. In a compiler such semantic actions are used for building the parse tree. They can also be used for other calculations, such as building certain reference tables, type checking, partial control-flow analysis, partial data flow analysis and such. We consider the semantic actions not as business rules: for, we want to know on an abstract level what the syntax of the language is, and not what the intermediate tree format used in the compiler for code generation looks like. Note that using the approach taken in [18, 17] the semantic actions cannot be ignored: the format of the parse tree needs to be known so that a conversion to GENOA can be specified with the GENII formalism. For, the output of the parser is reused, not its grammar. Also in other approaches the semantic actions cannot be ignored. Since they are often idiosyncratic for the compiler, reuse becomes then more difficult. This clarifies the problems with parser reuse that are reported on in the literature.

To make the idea of business rules a bit more concrete we provide some source code from a textbook on Lex and Yacc [37]:

```
expression: expression '+' NUMBER { $$ = $1 + $3; }
          | expression '-' NUMBER { $$ = $1 - $3; }
          |                NUMBER { $$ = $1; }
          ;
```

This code implements a simple calculator. As can be seen, the grammar and what to do with it later on are totally intertwined. This prohibits reuse of its output for other applications. To migrate this parser it is not necessary to know the semantic actions—in this examples, the calculations. So the business rule (for our purpose) is:

```
expression: expression '+' NUMBER
          | expression '-' NUMBER
          |                NUMBER
          ;
```

We can extract these rules automatically. This is done as follows. We specify the grammar of the parser generator code and generate a parser to parse the parser generator code that defines the parser. Note that this is an example of self-application. Let us explain this idea in more detail, since this idea is the key to our approach. The code that contains the business rules has itself a grammar. We can express this grammar also in a parser generator language. This generates a parser that parses the code containing the business rules. We can specify the semantic actions to be `LAYOUT` (see below for a definition). Then after parsing the parser generator code, the semantic actions are ignored. We could call this nonstandard parsing. The quality of the code that is available in the compiler is a measure for the time it takes to write such a parser. For high quality code it takes about 15 minutes to write a parser that extracts the business rules. Let us give an example of a simple Yacc grammar in SDF [24] that parses the above example. The specification below is executable and there is no need to specify semantic actions since the support platform we use automatically generates the abstract syntax (this is the ASF+SDF Meta-Environment [31]). Hence, the bare BNF we extract is enough to generate a parser.

```
exports
  sorts
    Yacc-Terminal Yacc-NonTerminal Yacc-Element
    Yacc-Rule Yacc-Program Yacc-Elements
  lexical syntax
    [ \t\n]                          -> LAYOUT
    "/*" Inside-comments* "*/"       -> LAYOUT
    "{" Inside-action* "}"           -> LAYOUT
    ~[*]                             -> Inside-comments
    [*] ~[/]                         -> Inside-comments
    ~[}]                             -> Inside-action
    "'" ~[']* "'"                    -> Yacc-Terminal
    [A-Z]+               -> Yacc-Terminal
    [a-z]+ [a-zA-Z0-9]* -> Yacc-NonTerminal
  context-free syntax
    Yacc-Rule+        -> Yacc-Program
    Yacc-NonTerminal ":" { Yacc-Elements "|" }+ ";"
                      -> Yacc-Rule
    Yacc-Element*     -> Yacc-Elements
    Yacc-Terminal     -> Yacc-Element
    Yacc-NonTerminal  -> Yacc-Element
```

We did not depict the above code in order to explain the Yacc parser in detail. Some specific aspects deserve attention. The `LAYOUT` consists of spaces, tabs and return characters, it consists of comments in C [30] and everything inside actions (which is between curly braces). As soon as we parse a file containing Yacc rules and semantic actions and C comment, all that remains are the business rules. We can also generate a pretty printer automatically from the Yacc grammar using technology described in [11]. So after pretty printing we have the business rules. Of course, this can be done using many other technologies and tricks as well. It is important that the idea to reuse the parser code instead of its output makes the difference between reuse easy instead of impossible.

We note that the definition of `Inside-action` is rather simple: everything but an opening brace. When the quality of the parser generator code is high, then all semantic actions are abbreviated in function calls so that the grammar structure is clearly visible. This implies that the 15 minute implementation suffices. When the quality of parser generator code is low, this implies often that the semantic actions contain a lot of target source code (C in the case of Yacc). If that is the case our parser should be more sophisticated. For instance, the braces must match properly. If the code contains unbalanced braces, it might even be necessary to incorporate large parts of the C grammar. Fortunately, SDF is modular so it is possible to import those parts and use a combined Yacc plus C grammar as the basis for tools to extract the grammar. When you do not have access to modular parsing algorithms, an occasional hack will also help. Jeromy Carrière [13] suggested to preprocess the parser generator code by making the braces context-free distinguishable using a simple lexical perl script. Again, as can be seen, many ways lead to the goal: extracting the bare BNF. In our case the approach is systematic, repeatable, and reliable. Summarizing, high quality compiler source code leads to instantaneous BNF extraction, if the quality is less, we have to put more effort in the reverse engineering task. Either way, the BNF can be extracted relatively easily if we compare it to making a grammar by trail and error using source code and reference manuals only.

Of course, one could argue, write a robust Yacc parser so that every Yacc file can be parsed and thus, the business rules are immediately at your disposal. True, but there are many variations on Lex and Yacc. The number of BNF dialects is impressive. It is our experience that for every parser in a compiler it is necessary to write at least one BNF parser. In our case study we had more than six BNF dialects in one compiler. In the next section we explain what we do with the obtained business rules.

2.3 Migrating the Parser to SDF

In order to migrate the parsers we translate the obtained business rules into SDF. We do this so that we can use the grammar in our implementation platform that will form the Software Renovation Factory in the end. As said earlier, there are a myriad of BNF dialects. So it is not a wise idea to specify these migrations over and over again for each new compiler. Therefore, we defined a domain specific language called EBNF (for Extended BNF) for which we made the translation and a number of assessment tools. When we migrate a new compiler we simply write a one-to-one translation of the dialect BNF to our EBNF. So for Yacc there is a tool called Yacc2EBNF. The construction of those tools is simple. We discuss the top and bottom sort detectors in more detail, since they help in further reverse engineering of the compiler. They make sure that we do not miss grammars.

Top and Bottom Sorts We developed two simple tools that list the top sorts and the bottom sorts. A top sort is a sort that is defined but no other sort uses it in its definition. They are commonly named `Program`, or after the language that they represent. If a mistake is present in the grammar definition we obtain more top sorts. The bottom sorts are sorts that are used but not defined. The ideal situation is that there is one top sort and there are no bottom sorts. For, then a grammar is completely defined. If the number of bottom sorts is high there is something missing. Often bottom sorts are the lexical definitions. For instance, Yacc only defines the context-free grammar, but not the lexical terms. The simple calculator example contains a sort `NUMBER`, which is a bottom sort: it is used but not defined. This sort is defined using Lex. So we also have a Lex parser and a translation that turns lexical code into SDF. The detection of bottom sorts reveals sort names that we can use to `grep` the source code in order to find the lexical grammar files. Ideally, the top sorts of the lexical grammars are the bottom sorts of the context-free grammar. Since SDF is modular we can import the lexical SDF into the context-free one and we have a complete SDF definition of the language once we migrated both lexical and context-free BNF to SDF.

At this point we are able to test the results and parse code in the support platform we use (the ASF+SDF Meta-Environment [31]). In our case study we used +8 MLOC of SSL code to test the parser. We first tested all sector grammars in isolation. We extracted from the SSL code the sectors and tested whether those sectors parsed. To give an indication, we could parse about one MLOC of the most important and large sector in an hour without a single error upon first try. We used this test suite in order to perform regression tests on consecutive modifications to the grammar. We discuss those changes in the next section.

2.4 Reengineering the SDF Code

At this point in the process we have a parser in a new environment. However, it is geared towards a compiler but we need a parser that is convenient for reengineering. In this section we discuss the issues that play a role in retargeting the parser source code.

First of all, we should provide some intuition as to what a reengineering grammar is. In our situation we focus on a grammar with the following properties:

- comments are part of the language
- scaffolding to include extra information
- list matching for pattern recognition
- native patterns
- modularization for dealing with dialects
- additional syntax for reengineering tools

We briefly discuss the above properties. For a start, a compiler does not use comments since they are no-ops. However, for a reengineer, it is not possible to just throw away comments. This means that we need to deal with them in a way similar to the executable code: we treat them as part of the language. Furthermore, we also wish to incorporate certain code ourselves during sophisticated reengineering tasks. Therefore, we adapt the grammar so that this extra code can be inserted and later on parsed so that other tools can use the information themselves. In fact, we scaffold the code like is common during development [51]. Furthermore, we need to recognize certain code patterns. Therefore, it is convenient when we can speak of, e.g., zero or more statements. We also need variables in order to denote certain pieces of the code, this we do as native as possible [50]. An example is that we can say `Statement-1*` which stands for zero or more arbitrary statements matched by the variable `Statement-1*`. Then we wish to modularize the grammars. This in order to deal with dialects, extensions, and other modifications. If the grammar is not modular, this enables a lot of maintenance problems [10]. Finally, we extend the grammar with syntax (and semantics) so that construction of reengineering tools becomes easy. As can be seen, none of the above issues is available in the source code of the parser of a compiler.

Next we will explain the steps necessary to turn a grammar into a reengineering grammar. We use between 20 to 30 separate tools to arrive in this situation. We will only discuss some main steps and tools. The first step is that we start to restructure the BNF that was extracted from the compiler source code. Such code is not always optimal. In fact, in the case of compilers, the grammar was mostly first available in a natural form, like a standard, or a manual, then the grammar is squeezed into Yacc-like formalisms, something we call informally yaccified. What we basically do, is deyaccify the code. A compiler grammar often contains so-called dummy sorts. A standard way to force an action in Yacc is to use such dummy sorts. They are in the grammar on certain places only to force special actions, the dummy sorts only ring a bell that something special should be done next. Dummy sorts do not produce any syntax. We do not rely on LALR(1) parsing, so we do not need such dummy sorts. Note that the semantic actions are generated automatically by the ASF+SDF Meta-Environment so we do not have to deal with this level at all. We can safely remove them since we are not interested in the semantic actions. Another phenomenon that occurs frequently is that certain rules are chains: `A -> B` is a chain if there are no other productions for `B`. This implies that we can eliminate `A` in favor of `B` (or vice versa). Also the removal of dummy sorts can create chains. If we remove dummies and chains, we can create double production rules: rules that become the same after the first steps. So we remove those. These—and more—steps are performed automatically, using our CALE-Factory. Now we have massaged the BNF code in such a way that we can swap syntax in a one-to-one fashion: we turn the BNF code into SDF code. However, this SDF code is far from optimal for reengineering purposes. So we perform another step: we restructure the SDF code. We mention the introduction of lists in the grammar. We give an example that we took from our case study. We abstracted from the real sort names. In the original code we found:

```
A B -> A
A C -> A
B   -> A
```

As can be seen this is a list that starts with a `B` and then is followed by zero or more `B`s or `C`s. We turn this into:

```
B -> B-C
C -> B-C
B B-C* -> A
```

Here we first define the sort `B-C` meaning a `B` or a `C`. Then we have the production rule stating that first we want a `B` and then zero or more `B-C`s expressed by the asterisk. As can be seen, we recognized a certain pattern that *simulates* a list construct in Yacc, and we turned it into a *real* list construct. Also other list constructs are recognized, including lists with and without separators, lists with zero or more occurrences or with one or more occurrences, etc.

The next step is to adapt the grammar so that comments are included in the language as well. Since the same holds for scaffolding we treat them in one step. We explain this by giving an example. Suppose we have a grammar fragment:

```
lexical syntax
  [a-z]+  -> C
context-free syntax
  "b"     -> B
  B "a" C -> D
```

Since at every location both comments and scaffolding can occur, we could translate this into:

```
lexical syntax
  [a-z]+  -> C
context-free syntax
  X "b"         -> B
  X B X "a" X C -> D
```

In this way we have incorporated the sort `X` short for extension of the grammar. However, this cannot be done since this grammar is ambiguous. Suppose that the comments are as in C. The term `/*foo*/ b a c` has two valid parse trees. One parse tree is made by using the last context-free rule. There we choose the first `X` to be `/*foo*/`, and then the `b` is recognized by the first context-free rule, by choosing the `X` empty. etc. The other valid parse tree is to take the last context-free rule and choose `X` empty, then to recognize the `/*foo*/ b` we use the first context-free rule, and so forth. As can be seen, it is very dangerous to modify grammars in a naive way. We solve this by connecting each `X` to the next concrete token. As is done below:

```
lexical syntax
  X [a-z]+ -> C
context-free syntax
  X "b"      -> B
  B X "a" C -> D
```

This introduces another problem. Every token is either a context-free keyword, e.g., `b`, or a lexically defined regular expression, e.g., `[a-z]+`. Since the lexer chops the strings into tokens, the `X` in lexical definitions becomes a single token. This is an unwanted situation, for, we wish to detect `X` context-free in order to recognize it in tools. Therefore, the correct solution is:

```
lexical syntax
  [a-z]+ -> C-lex
context-free syntax
  X C-lex   -> C
  X "b"     -> B
  B X "a" C -> D
```

Now we are almost done. We have to take into account that at the end of a program also comments are allowed. As can be seen from the above grammar, this is not allowed at the moment. We transform the rule that produces the top-sort in a special way. This is the resulting grammar:

```
lexical syntax
  [a-z]+ -> C-lex
context-free syntax
  X C-lex     -> C
  X "b"       -> B
  B X "a" C X -> D
```

The resulting grammar is not ambiguous and we can manipulate comment and scaffolding context-free on *every* location. Moreover, this step is fully automated. For implementation details we refer to [51]. As can be seen, it is far from trivial to incorporate comments and scaffolding to a grammar. We have automated the above algorithm so addition of scaffolding is a matter of pressing a button. In the case study we of course performed extensive regression testing.

Some readers may worry about the issue of scaling up with respect to performance of the generated parser of the heavily reengineered grammar. Namely, on virtually every location in the grammar the scaffolding sort has been added. It is not possible to ignore comments and other intermediate results in program code when we have to reengineer it. On the other hand, software renovation deals with large systems in general. First and foremost, we do not care about performance, since automated renovation should be compared to renovation by hand. Moreover, the progressions in hardware development solve a number of performance issues just by waiting. Apart from that, any tool with very bad performance is faster and more accurate than renovation by hand. However, for the sake of the (in our opinion irrelevant) argument, we have done a few measures. We took a 108.771 LOC COBOL/SQL file and parsed the code. This took 37 CPU minutes. The maximal usage of internal memory was 568 Mb (on a machine with 576 Mb internal memory we were also using swap space). We store the files in a binary form of ATF (Annotated term Format [5]) called BAF (Binary ATF) that uses term sharing heavily [4, 29]. This leads to a file of 1.7 Mb. The file of 108.771 LOC is 4.7 Mb, so the parsed version is 2.8 times smaller than the text file. Indeed, many empty nodes are produced by the scaffolding but they are shared in the binary ATF version. We refer the reader to more information on scaffolding in a paper focusing on that issue entirely [51].

Since parser generators are often somewhat limited[1] in expressive power, it is a wise idea to be prepared for unusual situations. We scanned the parser source code comments for words like `NOTE`, or `should`, and such. We detected two irregularities due to the nature of the used parser generator algorithm. Fortunately, everything was documented very well, so we could modify the two BNF rules to what they should be, since our parser generation algorithm has no problems with those rules. We asked the compiler specialists whether there were other irregularities that we missed. There is no indication that this was the case. Also the +8 MLOC test provided an indication that there were no irregularities.

In some cases it is useful to modularize the grammar. This means that we put certain parts of the grammar in separate files. We do this since modular grammars enable the possibility of dealing with different dialects of a language. In our case study, the compiler contains more than one grammar and depending on the sector it chooses a subparser. In our case we cannot use subparser technology. This is due to the fact that some changes affect more than one sector. We also cannot simply unite both grammars since in some sectors production rules occur more than once. This causes true ambiguities. We have to remove rules that occur more than once. We have a tool that lists double productions. The duplicates are good candidates for modules: they can then be imported by the parts containing them in the original situation. In this way we

[1] This includes the parser generators that are used in reengineering, such as Refine's Dialect [43], ANTLR [42], and many more parser generators. For an elaborate discussion on the problems and limitations of current parser generator technology in reengineering we encourage the reader to consult [10].

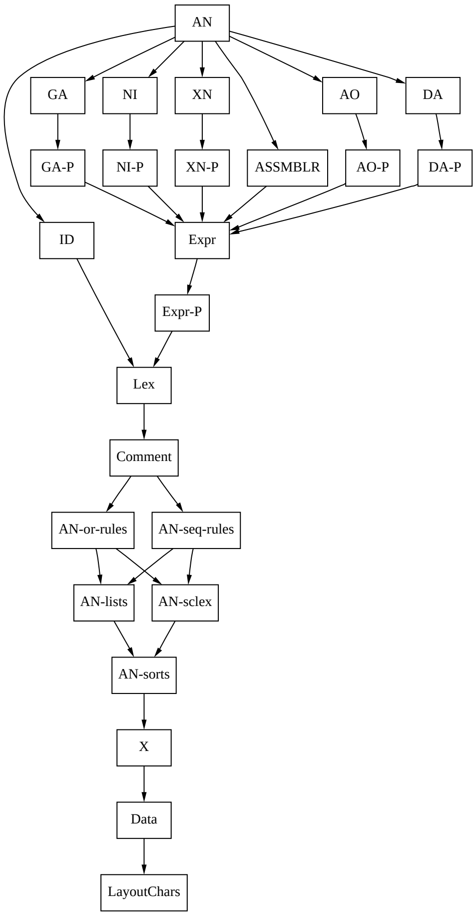

Figure 1: Import graph of SSL for large public switches

also modularized the SSL grammar. To give the reader an idea, we depict the import-graph of part of the SSL grammar that is suited for certain large public switches in Figure 1. A box is an SDF module, an arrow from box `AN` to `ID` means that `ID` is imported by `AN` and so forth. The information of Figure 1 is automatically generated using a tool for SDF [38]. The visualization of this information is being done using `dot` [22, 41]. The number of production rules that is represented by Figure 1 is about 2000. The upper part of the picture has been extracted from the compiler and reengineered. Below the `Comment` module we automatically generated all the syntax necessary to become a reengineering grammar.

We connected all the sector grammars using a super grammar `AN` that we found in the compiler as well. In that grammar we detected a few sectors that we had not found yet. We found all but two. They were one of the 4 embedded assembly grammars `ASSMBLR`, and a grammar of a no-op sector `ID`. There was no grammar in the compiler, for it is a no-op sector. We made that one by hand. The assembly grammar was a recursive descent grammar. We generated the grammar from the comments containing a form of BNF. Here we see another difference with a compiler grammar: the `ID` sector is important for version and configuration management, and is used by such tools. Is does not add to the executable. However, when reengineering such code, these `ID` sectors also change, and should not be removed.

Of course, after each step in this grammar reengineering process we did a regression test using our test suite. In this way irregularities in our process could readily be found. It took about a week to make the tools, to make the test suite, to perform the tasks, to test the various version of the grammars, and to do all the hand work. Total costs of the project, including acquiring hardware, were about 25.000 US$. We heard from Ericsson that Cordell Green estimated the construction of this total grammar on a few million US$. We have to add that this was a ball park estimate. Moreover, we think that he based his estimate on hand work. Cordell Green told us that 25.000 US$ is nothing for such a grammar.

Now the grammar is reengineered and it is ready to be used to generate a software renovation factory. We discuss this briefly in the next section.

3 Generation of a Software Renovation Factory

In this section we briefly discuss the generation of a software renovation factory from a reengineering grammar. Moreover we explain what functionality is generated. Since some of the issues below have been discussed extensively in other papers, we keep those explanations short. Some issues are new though. For a short introduction to the process of generating an entire software renovation factory we refer to [14, 52].

generating a parser Usually, the grammar that resides in a compiler is implemented using a parser generator. The form of such grammars is not always natural for humans, since the parser generation algorithms put restrictions on the form of the grammar. We removed such idiosyncrasies. However, this implies that our reengineering grammars are usually not within the class of mainstream parser generator technology. Since we use sophisticated parsing technology (GLR) this is not a problem. See [35, 53, 45, 54] for general information on GLR parsing, and see [10] for the connection of GLR parsing and reengineering. So we can safely generate a parser from this reengineering grammar. We refer to [10] for an overview on parser technology in combination with reengineering and comparisons with our approach.

generating HTML for the grammar Since large grammars contain so many constructs and production rules, it is important to have access to the grammar so that rapid understanding of the language is enabled. We generated for all the sectors an HTML version. The generation process and complete documentation of the HTML files took about half a day. If you click on a sort name you will be send to its use, if you click on its use you will be send to where it is defined, if you click on the definition you are back in the declaration section, etc. The HTML files gave us within a day an enormous insight in the SSL language. As a side remark, this technology will be used not only for reengineering targets but probably also for producing correct manuals. For, it is possible to generate from the compiler source code the core of a language reference manual, without a single error in the formal part of the manual. Telelogic is a company that makes SDT (SDL Design Tool). This is a tool that can generate code from a design in SDL. At Telelogic also a design tool is developed that generates SSL code from a proprietary version of SDL. Our HTML version of SSL has been shipped to Telelogic.

generation of a native pattern language In order to make analysis or transformation tools we need a means to recognize patterns in source code. Since it is likely that the software renovation factory is going to be used by the (possible few, see [27]) programmers that know the language, it is crucial for productivity that they do not have to learn another obscure pattern recognition language [26]. We refer to [50] for a full treatment of the generation of native pattern languages and their use in the real-world applications. Note that we hide all implementation issues for people who develop components for renovation. An elaborate treatment of the organizational part of our approach is discussed in [52].

generation of generic transformations and analyzers From the reengineering grammar we generate so-called generic transformations and analyzers that enable rapid development of components that support automated analysis and change. We refer to [6] for an elaborate treatment of the generation process.

software renovation factory When we have all the above ingredients, we can load the reengineering grammar, the native patterns, the generic transformations and analyzers into our support platform, the ASF+SDF Meta-Environment [31]. This is a generic interactive programming environment. We recall that SDF stands for syntax definition formalism and is used to describe arbitrary syntax (see [24] for a reference manual). ASF stands for algebraic specification formalism, and is used to describe semantics (see [3] for a textbook on this subject). When all specifications loaded, we have an architecture that can be called a software renovation factory. The software renovation factory has two modes. An interactive mode that is used for development and testing of components and assembly lines. When the reengineering tasks have been implemented and tested we can start modifying code. We use the batch mode of the software renovation factory to perform system wide automatic changes. Since our particular implementation platform is based on the idea that as much as possible is generated from the context-free syntax, we get a lot of extra functionality for free. We mention structured editors, incremental GLR parsers, and generation of pretty printers. Further, we have a rewrite engine at our disposal to implement the tool specifications.

example factories An example of a software renovation factory is our CALE factory. It is the infrastructure that we use to develop, assess, reengineer, and migrate language definitions. We reused the SDF grammar that is available in the ASF+SDF Meta-Environment and we implemented a special BNF grammar by hand. Another example of a software renovation factory is our COBOL factory. This factory is not generated from the compiler. The COBOL/CICS/SQL grammar is made by hand. Another example is, of course, an SSL factory that we generated from the reengineered grammars that we extracted from the source code of the SSL compiler. We plan to use our approach for generating a software renovation factory for Philips Electronics N.V. to facilitate architecture transformations on complex systems [34].

example applications In order to get an idea of the applicability of such renovation factories, we give some references to other papers where the factory technology is applied. We refer to [9] where an assembly line is presented that performs control flow normalization of COBOL/CICS legacy systems. We refer to [50] where an assembly line is presented that transforms a faulty leap year calculation in COBOL to a correct one. We refer to [48] where an assembly line implementing several maintenance transformations is presented in order to restructure COBOL/SQL systems. We refer to [15] where cluster analysis technology is implemented to detect classes in legacy COBOL/CICS code. We refer to [16] where a type inference method is implemented for COBOL with embedded CICS. We refer to [47] where sophisticated restructuring on a COBOL/CICS system is performed. Finally, in [12] a COBOL software renovation factory is used to migrate COBOL85 back to COBOL74.

4 Discussion

In [55] interoperability of program understanding tools is discussed. We can read that each program understanding system tends to have its own parser for a given language, with slightly different output formats and language coverage [55]. One of the problems with interoperability is that as soon as the components are not rigorously defined and delimited, sharing components is out of the question. In [55] a common model is proposed that should be adopted by everyone so that interoperability is possible. The problem with parser reuse is exactly what they indicate as “slightly different output formats”. In fact, since mainstream parser generators mix the description of the pure syntax with semantic actions, the component *parser* is not clearly defined. Everyone has a different output format (since they all need different semantic actions). If the bare syntax description is separated from the semantic actions describing the output, it is possible to share the language description and dump any format that is best for some tool. This is exactly what we do. The platform that we use generates automatically the abstract syntax from the syntax description. So, there are no semantic actions necessary to obtain a working parser. All desired operations on the syntax are separated from the grammar description. Any form of analysis, even type checking, for instance, is located in a separate component. For more details on our viewpoints on component-based software engineering we refer to [32].

We recall that in [46] it is mentioned that reusing a parser is a real problem. They strongly suggest that parsing and analysis should be separated. The basic impediment to extending some program understanding tool called CMS2REV is the tight coupling of language syntax and semantic processing. CMS2REV does not maintain a separation of parsing and analysis functions. The parser actions of the CMS2REV tool directly populate idiosyncratic data structures used to produced the desired reports [46]. Our approach is even more rigorous than the ones proposed in [46] or recently [55]. We separate even tree building from the syntax description. Since this is done automatically under the surface we can effortlessly reuse parsers by extracting just the syntax rules and migrate them to the SDF formalism supported by the ASF+SDF Meta-Environment. This approach towards components enables effortless parser reuse.

We think that it is also possible to enable parser reuse among other tools along similar lines as discussed in this paper. First extract the business rules from the compiler front-end. Then migrate them to the desired input format of the parser generator that is used for generating a parser for the program understanding tool. Add a skeleton for the idiosyncratic semantic actions to it and complete the idiosyncratic part by hand. Then a parser can be generated that is geared towards program understanding or another target. Another idea would be to dump the parse tree in a language independent format, as proposed by [46], [5], and [55] so that other tools can use this format.

4.1 How much work is it?

In order to go from the compiler source code to its corresponding software renovation factory, we need to reverse engineer the source code of the compiler. Normally it is difficult to reverse engineer systems software, in particular, when the high level design is unknown. In the case of compilers this is almost never the case. Compiler design has been lectured everywhere and the reference architecture of compilers is well-known and described in many textbooks. In our case the source code of the compiler was about 63 Mb. It took us less than 5 minutes to find the part where the front-end code resided. Then we found almost all grammar files. Then we needed to implement about 6 small grammars: one for lexical definitions and five for various dialects for context-free definitions. These take 15 minutes each. One of the 5 assembly sectors was implemented as a recursive descent parser. Also the recursive decent parser was carefully designed. Concrete, this implied that the BNF was added in comments. We could use the comments to generate a new grammar. If the recursive decent parser had been implemented without comments (which is less careful) than we could have used a C renovation factory (generated from the source code of the C compiler), to write a few simple transformations that take the recursive descent code and transform it into BNF. This would have prevented *instantaneous* generation of renovation factories, but it would not at all have prevented it. It would just take more time: generating a C factory and making the appropriate transformation for extracting the BNF from say recursive descent parser source code.

We note that if lexical definitions are not defined in a natural fashion, it may be the case that this translation can better be done by hand. Since lexical definitions are often very small this is not a problem, although the repetitiveness of the generation process is lost.

When we can parse all the raw source code containing the grammar information, we map the code to a domain specific language: in our case an extended form of BNF. These transformations are straightforward. The difficult part of the transformation process is from this EBNF to SDF. And, of course, from SDF to a renovation factory. In our case study, transforming from raw source code to EBNF is about 20 minutes for the entire grammar. The part to move to SDF is taking about half a day. The generation of a software renovation factory takes about 50 hours calculation time.

4.2 Parser Reuse and the Year 2000

As noted in the introduction, Jones estimates that 30 percent of the Year 2000 problems are in the obscure languages. Our experience and intuition tells us that the more obscure a language is, the better the source code of the compiler can be made available for supporting in solving the Year 2000 problem. We have a few reasons to believe this. First, the owners the compiler will be glad that they can deliver a Y2K solution for their limited set of customers. If the owners of the compiler are also the owners of the code that needs Y2K repair, then there should be no problem at all. For very common languages such as COBOL there is a lot of tool support.

In our opinion, a possible scenario could be to use a combination of tools that reuse parsers in order to attack the problem. For instance, the GENOA/GENII combination could be used to reuse the output of the parser so that an impact analysis can be done. Since GENOA is independent of the front-end, it means that a generic Year

2000 search engine should be possible. In the mean time a process that we described in this paper could be started to implement a software renovation factory. Another possibility is to gear a CALE factory towards adding parsers to commercial systems such as COSMOS [20]. We note that according to Gartner [23, 28] COSMOS is leading technology providing Year 2000 search and repair engines. Since there is a separation between parsing and the Y2K search engines in COSMOS, it is in principle possible to generate a skeleton parser generator file from a compiler front-end that needs to be completed by hand, by COSMOS developers. We estimate that this approach can also be applied to the Revolution 2000 Maintenance Environment [40].

We note that is not necessary to use the implementation platform that we used. In fact, we think that this approach can also be implemented using, for instance, Reasoning's Software Refinery. Since the implementation of our CALE Factory did not take much time we think that other sophisticated programming environment development tools can also be used to generate automated Year 2000 search and repair support by reusing the source code of existing parsers rather than their output.

5 Conclusions

We have developed a powerful approach that enables the rapid development of software renovation factories from the source code of a compiler. If the quality of the source code of the compiler is high and the compiler is not as complex as the case study we reported on, it should be possible to generate such a factory with a very short lead time. The example compiler that we discussed contains so many grammars that the development speed of other reengineering grammars will take less time so that a software renovation factory can be generated. We believe that the generation of the about 3000 production rules in half a day and the additional problems that we dealt with in about a week's time indicate that our approach is quite effective, and that the generation of software renovation factories from the source of the compiler is a promising technology. Since legacy systems will be around forever, it might be an idea to incorporate in new compilers not only the usual code generation but also support for the generation of a software renovation factory.

References

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

[2] 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.

[3] 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.

[4] M.G.J. van den Brand, H.A. Jong, P. Klint, and P.A. Olivier. Efficient annotated terms, 1999. Submitted to Software: Practice and Experience. Available at ftp://ftp.wins.uva.nl/pub/programming-research/software/aterm/.

[5] 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. Available at http://adam.wins.uva.nl/~x/sofsem/sofsem.html.

[6] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. Available at http://adam.wins.uva.nl/~x/scp/scp.html. An extended abstract with the same title appeared earlier: [7].

[7] 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 Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997. Available at http://adam.wins.uva.nl/~x/trans/trans.html.

[8] 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. Available at http://adam.wins.uva.nl/~x/coboldef/coboldef.html.

[9] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy systems. In P. Nesi and F. Lehner, editors, *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering*, pages 11–19, 1998. Available at http://adam.wins.uva.nl/~x/cfn/cfn.html.

[10] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117, 1998. Available at http://adam.wins.uva.nl/~x/ref/ref.html.

[11] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[12] J. Brunekreef and B. Diertens. Towards a user-controlled software renovation factory. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 83–90. IEEE Computer Society, 1999.

[13] J.S. Carrière. SEI, Personal Communication, September 1998.

[14] A. van Deursen, P. Klint, and C. Verhoef. Research issues in the renovation of legacy systems. In J.-P. Finance, editor, *Proceedings 2nd Conference on Fundamental Approaches to Software Engineering*, volume 1577 of *LNCS*, pages 1–21, Amsterdam, 1999. Springer-Verlag. Available at http://adam.wins.uva.nl/~x/etaps/etaps99.html.

[15] A. van Deursen and T. Kuipers. Finding classes in legacy code using cluster analysis. In S. Demeyer and H. Gall, editors, *Proceedings of the ESEC/FSE'97 Workshop on Object-Oriented Reengineering*, Report TUV-1841-97-10. Technical University of Vienna, 1997.

[16] A. van Deursen and L. Moonen. Type inference for COBOL systems. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 220–230. IEEE Computer Society, 1998.

[17] P. Devanbu. Genoa - a customizable, front-end retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology, (to appear)*, 1999.

[18] P.T. Devanbu. GENOA - a language and front-end independent source code analyzer generator. In *Proceedings of the 14th International Conference on Software Engineering*, pages 307–319. IEEE, 1992.

[19] T.B. Dinesh, M. Haveraaen, and J. Heering. A domain specific programming style for numerical software, 1998. Work in progress.

[20] Emendo Software Group, The Netherlands. *Emendo Y2K White paper*, 1998. Available at http://www.emendo.com/.

[21] Y. Fuqing, M. Hong, Y. Wanghong, W. Qiong, and Y. Guo. Experiences in building C++ front end. Technical report, Department of Computer Science and Technology, Peking University, 1998.

[22] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19:214–230, 1993.

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

[24] 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.

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

[26] C. Jones. *Programming Productivity.* McGraw-Hill, 1986.

[27] Capers Jones. *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences.* Addison-Wesley, 1998.

[28] N. Jones. Year 2000 market overview. Technical report, Gartner Group, Stamford, CT, USA, 1998.

[29] H.A. Jong and P.A. Olivier. *ATerm Library User Manual.* University of Amsterdam, 1999. Available at ftp://ftp.wins.uva.nl/pub/programming-research/software/aterm/.

[30] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

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

[32] P. Klint and C. Verhoef. Evolutionary software engineering: A component-based approach. In R.N. Horspool, editor, *IFIP WG 2.4 Working Conference: Systems Implementation 2000: Languages, Methods and Tools*, pages 1–18. Chapman & Hall, 1998. Available at: http://adam.wins.uva.nl/~x/evol-se/evol-se.html.

[33] C.H.A. Koster. Affix grammars for programming languages. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 358–373. Springer-Verlag, 1991.

[34] R. Krikhaar, A. Postma, M.P.A. Sellink, M. Stroucken, and C. Verhoef. A two-phase process for software architecture improvement. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, 1999. Elsewhere in this volume. Available at http://adam.wins.uva.nl/~x/sai/sai.html.

[35] 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.

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

[37] J.R. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly & Associates, Inc., 2nd edition, 1992.

[38] Leon Moonen. Graph-tools: tools for import graphs, 1997. Available at http://adam.wins.uva.nl/~leon/graph-tools/.

[39] M.J. Nederhof, C.H.A. Koster, C. Dekkers, and A. van Zwol. The grammar workbench: A first step towards lingware engineering. In W. ter Stal, A. Nijholt, and H.J. op den Akker, editors, *Proceedings of the second Twente Workshop on Language Technology – Linguistic Engineering: Tools and Products*, volume 92-29 of *Memoranda Informatica*, pages 103–115. University of Twente, 1992.

[40] P.H. Newcomb and M. Scott. Requirements for advanced year 2000 maintenance tools. *IEEE Computer*, 30(3):52–57, 1997.

[41] S. C. North and E. Koutsofios. Applications of graph visualization. *Graphics Interface'94*, pages 235–245, 1994.

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

[43] Reasoning Systems, Palo Alto, California. *DIALECT user's guide*, 1992.

[44] K. Rekdal. CHILL – the international standard language for telecommunications programming. *Teletronikk*, 89(2/3):5–10, 1993.

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

[46] 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.

[47] M.P.A. Sellink, H.M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 72–82, 1999. Available at http://adam.wins.uva.nl/~x/cics/cics.html.

[48] M.P.A. Sellink and C. Verhoef. An architecture for automated software maintenance. Technical Report P9807, University of Amsterdam, Programming Research Group, 1998. Available at: http://adam.wins.uva.nl/~x/asm/asm.html.

[49] M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. In *Proceedings of the 13th International Automated Software Engineering Conference*, pages 314–317. IEEE Computer Society, 1998.

[50] M.P.A. Sellink and C. Verhoef. Native patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society, 1998. Available at http://adam.wins.uva.nl/~x/npl/npl.html.

[51] M.P.A. Sellink and C. Verhoef. Scaffolding for software renovation. Technical Report P9903, University of Amsterdam, Programming Research Group, 1999. Available via http://adam.wins.uva.nl/~x/scaf/scaf.html.

[52] M.P.A. Sellink and C. Verhoef. Towards automated modification of legacy assets. In N. Callaos, editor, *Proceedings of the Joint third World Multiconference on Systemics, Cybernetics and Informatics and the fifth International Conference on Information Systems Analysis and Synthesis (SCI/ISAS'99)*. International Institute of Informatics and Systemics, 1999. To appear. Available at http://adam.wins.uva.nl/~x/aml/aml.html.

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

[54] E. 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.

[55] S.G. Woods, L. O'Brian T. Lin, K. Gallagher, and A. Quilici. An architecture for interoperable program understanding tools. In S. Tilley and G. Visaggio, editors, *Proceedings of the sixth International Workshop on Program Comprehension*, pages 54–63, 1998.