An Interactive Method For Extracting Grammar From Programs

Rahul Jain, Sanjeev Kumar Aggarwal, Pankaj Jalote, Shiladitya Biswas
Department of Computer Science & Engineering
Indian Institute of Technology
Kanpur – 208016. INDIA
(**Contact**: *jalote@iitk.ac.in*, **Fax:** +91-512-590725/590413*)*

Abstract

The grammar of the language in which some given code is written is essential for developing automated tools for maintenance, reengineering, and program analysis. Frequently grammar is available for a language but not for its variants that are implemented by various vendors and in which the given code may be written. In this work we address the problem of obtaining the grammar from source code, which can then be used for generating tools for the programs. We propose an incremental method for obtaining grammar for a particular language variant, from a set of programs written in the language variant and an approximate grammar (presumably of the standard language) with some user interaction. We also present the design of a tool for implementing this approach and our experience in working with grammars of C, C++ and COBOL.

Keywords: Grammar extraction; Legacy systems; Reengineering; Reverse engineering.

1 Introduction

In the software industry today, the effort spent on enhancing and renovating existing software systems constitute a major part of the total development work. Organizations regularly reuse their existing software systems for developing new ones. Frequently, instead of extending an existing system, organizations choose to reengineer it using newer languages, platforms and tools and then work with the reengineered system.

Both enhancement and reengineering require dealing with legacy systems. These legacy systems are generally very large in size, so it is not feasible to analyze them manually. Some kind of tool support is essential to aid the renovation and updation tasks. A good amount of research has been done on automating the reverse-engineering process [3, 8, 9] and several approaches have also been developed. These approaches can be used for recognizing a program's structure, its

components, or for extracting the business logic that drives the application. Tools for quality management like test coverage analyzer, code compliance checker, static analyzer are also useful while adding new features to existing systems, as there is a need to ensure quality of the new code. Most of these approaches and tools require the grammar of the language in which the programs are written.

In general the grammar of the language in which programs are written is essential for developing useful tools which assist in the analysis and maintenance of software. Though grammar is generally available for a standard language, in the business environment one frequently has to work with a software system written in some language variant or dialect for which the grammar is not available. Legacy systems particularly are written in dialects for which usually little or no support is available. The source code for their compilers is also not available, so one cannot hope of getting grammar from the compiler front-end (an approach for obtaining the grammar from compiler source code is described in [2]). Nevertheless, it is highly desirable to design the aforementioned tools for these applications. So, there is a need to devise some approach for obtaining grammar from the given programs. In other words, there are many situations in which programs written in some language are available but the grammar for the language is not. As we have to work with the given language and with given programs, it is desirable to have the grammar for that language.

Some work has been done to address some related problems. An approach for correcting a given grammar, which has errors, is given in [1]. In this approach, the given grammar itself is parsed using a tool. When the parser finds a mistake in the grammar, it is pointed out to the grammar writer who then manually corrects it. An approach for obtaining a reduced and unified grammar adapted for developing reengineering tools from a large grammar and source code is presented in [4]. In [7] a tool called Universal Syntax and Semantics Analyzer (USSA) is described, which generates a grammar from an initial grammar and a given set of rules that can be added and deleted such that the final grammar can parse the given program. An approach for dynamically changing the parser as the grammar evolves is given in [11].

In this paper we describe an approach for extracting grammar from a given set of programs. The extracted grammar should be such that it can parse the given programs successfully. In a sense this problem is the dual of the regular compiler and parsing problem. In a compiler we assume that the grammar is correct and the programs may have errors. In the problem we address here,

we assume that the programs are correct in the sense that they can be parsed and executed, but the grammar we have might have errors with respect to the given programs.

In our approach, we start with an approximate grammar, perhaps of a variant of the language in which the given programs are written. There is an extensible Knowledge Base, which contains various rules and supports the generation of correct grammar. Whenever a language construct is encountered in the program that cannot be parsed with the current grammar, a suitable rule is picked from the Knowledge Base and added to the grammar. If there are multiple rules that can apply, the user selects one. If there is no suitable rule in the Knowledge Base, the user suggests a new rule, which is also added to the Knowledge Base for future use. For writing a new rule, the Knowledge Base can be used to suggest similar rules. We also describe a tool that we have developed for implementing this approach.

2 Grammars, Languages and Dialects

Every programming language has rules that describe the syntactic structure of valid programs. These rules are called the grammar of the language. For example, in the programming language C, the rules say that a program is made up of translation units, a translation unit consists of function definitions, and functions are composed of statements and so on. These rules can be considered as a C grammar.

Chomsky has classified four classes of grammars in terms of productions. Of these, context free grammar is the most important in terms of application to programming languages and compiling. Context free grammars are used to describe syntax of programming language constructs. Grammars are capable of describing most, but not all, of the syntax of a programming language. With a little help in describing tokens from lexical analyzer the context free grammar can be used to parse programs or to generate parsers for them.

Though a few constructs found in some programming languages can not be described using context free grammars, they are useful for describing most constructs of a programming language [5]. Mostly the tools that generate parsers automatically, like Yacc [12], make use of context free grammars. For example, one can write the parser for C, C++ or COBOL using context free grammars.

Like in natural languages, different dialects of programming languages have evolved. One of the main reasons for creation of a dialect is the non-availability of some useful constructs in its parent language. That is, the parent language is viewed as lacking some constructs that are probably available in some other language. Hence to enrich the language, vendors may add these constructs in the language, giving rise to a dialect. A dialect will generally process the ANSI/ISO standard language, though it may add few extra features in the language to make it more user-friendly. We define a dialect of a language as a language that has some constructs that the parent language does not have. The overall structure or the grammar of a dialect is not very different from its parent; they differ mostly in the list of tokens, few extra operators and runtime systems. Note that for the purpose of grammar extraction we are concerned only with new constructs, since the use of such a construct in a program can be identified by a parser. Constructs that may be deleted cannot be identified as the lack of use of some constructs in a finite set of programs cannot be taken to mean that the construct has been deleted.

One point that deserves a mention here is that a program itself can be defined and used as its grammar. For a set of programs, $\sum^*$ ($\sum$ is the set of symbols) is a trivial grammar that will parse all the programs. However, this is not an interesting grammar. Languages are normally perceived as hierarchical structures consisting of identifiers, operators, expressions, statements, control flow constructs, functions/procedures etc. $\sum^*$ will fail to capture any of these constructs and therefore, is of no practical use. For program analysis and tool development the language structure must be exposed by the grammar. So we need a grammar, which captures the syntactic structure of the programming language, from which these tools can automatically be generated.

3 Extracting Grammar of a Dialect

We are considering the following problem. The grammar for a language is available, but not for its dialect. Some programs written in the dialect are also available. We want to find, automatically if possible, the grammar of the dialect.

Extraction of grammar from programs is not a trivial problem. It is very similar to the natural language learning problem. There are many issues that make it impossible to automate this process. A given program may conform to a large number of grammars. These grammars differ on the precision with which they capture the syntactic structure of the language. As discussed above, the program itself is one of the grammars to which it conforms. In order to obtain the

correct and precise grammar we need a large number of valid and invalid programs, which is not possible when we only have a finite number of programs written in the language. Also, if we start from the scratch i.e. without any initial grammar, it is very difficult just to find when one construct ends and another starts. This problem becomes more significant when one construct contains another construct, *e.g.* the body of a *for loop* contains an *if statement.* Moreover, there are many classes of languages like imperative, functional, logical etc. and the ways in which programs are written in these languages are very different. So, it makes it all the more difficult to extract the grammar if no information about the language is available.

However, if an approximate grammar for the dialect is available, which may be the grammar of the parent language, the problem becomes simpler. In this case, we need to extract grammar only for the constructs which are new in the dialect. The help of a knowledge base can be useful in extracting the grammar.

The programming languages, as we see them today, have evolved over time. The constructs that are found in one programming language can also be found in other programming languages. For example, statements for iteration, selection, branching etc. are found in many programming languages and have almost the same structure and syntax. So, it is possible to create a Knowledge Base of such common constructs and their syntax rules.

With such a knowledge base, one approach for extracting the grammar is as follows. If the language in which the programs are written is a dialect of some popular language like C or COBOL, then the grammar of the parent language can be taken as the starting grammar. From the approximate grammar a parser is generated. We now try to parse the programs using this parser. As the grammar is not complete, at some point while parsing a construct, the parser will fail. Here, the rule Knowledge Base is searched for a suitable rule for the construct. The rule is added in the grammar and the parser is generated again. The parsing of the program starts again from the first line. This process continues till the program is successfully parsed and the correct grammar is obtained.

Another approach could be to build a parser with an exhaustive set of grammar rules. As the constructs found in many programming languages are similar, this parser would be able to parse all programs belonging to a particular language class. As the parsing proceeds, some rules in this grammar are used for parsing the statements in the program. These rules are marked. After, the

whole program is parsed, the marked rules together form a minimal grammar, which is capable of parsing the given program.

The two approaches discussed above assume that all kinds of constructs and their grammars are known and available. So obtaining the grammar for a particular language variant just requires searching the Knowledge Base for a suitable rule and plugging the rule at a correct place in the approximate grammar.

As it turns out, it is very difficult, if not impossible to come up with an exhaustive list of constructs and their grammar rules. Moreover, locating the correct position in the grammar to insert the rule is also a non-trivial task. Besides, the newly added rule may use some existing grammar symbols. These symbols names may be different from those already present in the grammar. So, before adding the rule in the grammar, its symbols must be replaced by those that are defined in the grammar. Automating this process is also not possible. As a result, a system cannot be expected to recover the grammar automatically from the source program.

Some work has been done in the area of regular grammar learning [10]. The approaches developed are heuristic in nature; they are useful for small problems but do not scale well for large problem sizes. The automatic learning of context free grammar is made more difficult by the fact that several decision problems related to context free grammars are undecidable. Hence, obtaining the grammar from programs should be viewed as a co-operative process involving both the system and the user. The system finds the deficiencies in the grammar and probable solutions. The user assists it in making decisions for choosing a rule from many probable. The user also helps in modifying the rule so that it contains only the symbols that are defined in the grammar.

After a rule is added or modified, the grammar is affected at more than one place. As a result, some of the constructs, which were successfully parsed earlier, may become unparseable. Various kinds of conflicts may arise as the side effect of adding a rule. So it is necessary to do this processing in an iterative manner. After adding a new rule, the parser should be regenerated and the parsing must resume from the beginning of the program. This is the approach we take.

4 An Interactive, Iterative Approach

As discussed above, for extracting grammar from programs, we need an interactive and iterative approach. In this section, we present one such approach. The following figure gives an overview of our approach.

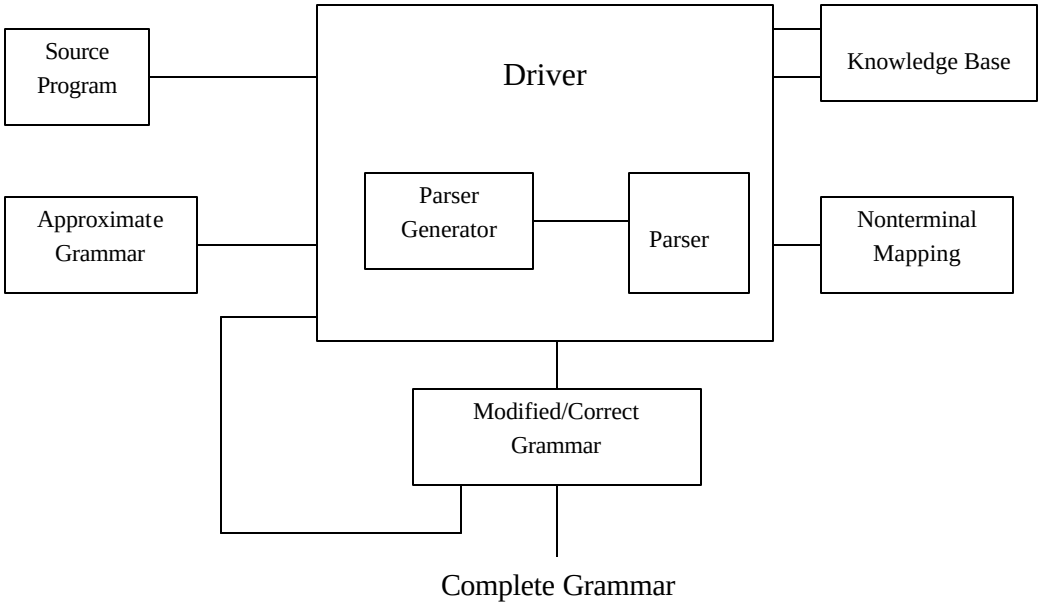

Fig.1 Overview of the Approach

As can be observed from the above figure, we take the source program and an approximate grammar as the input and after a series of modifications to the grammar generate the correct grammar for the program. Below, we discuss the various steps involved in the process.

4.1 Parsing the Source Program

Parsing, or syntax analysis as it is sometimes known, is a process in which a string of tokens is examined to determine whether it obeys certain structural conventions explicit in the syntactic definition of the language [6]. We use the shift-reduce approach. From the context free grammar of a language its shift-reduce parser can be generated automatically. Tools like Yacc can be used to generate a LALR (1) parser from a BNF like grammar specification.

The shift-reduce parser works in the following way: The string of tokens is scanned from left to right and the symbols are *shifted* into a stack. Whenever a sequence of tokens from the top of the stack form the right hand side of a production rule in the grammar, it is *reduced* i.e. replaced by the left hand side symbol. If at any stage, no reduction or shift is possible i.e. no grammar rule

matches the sequence of tokens present on the stack, an error is declared. By looking at the stack top, one can determine the place in the grammar where the parsing failed.

The occurrence of an error while parsing signifies that either the program is not correct or the grammar is not complete. In our case, since all the programs are valid, the error represents incompleteness of the grammar. The exact position of the error can be determined by looking at the line being parsed and the parser stack.

An improvement over this approach is to modify the parser to help not only identify the line number of the new construct, but potentially also the structure of the construct. For this, we have modified the Yacc parsing process, particularly when an "error" is encountered. Every time a shift occurs the new terminal symbol is pushed onto an auxiliary stack and every time a reduce occurs, the number of symbols equal to the r.h.s of the corresponding nonterminal, are popped and the new nonterminal is pushed. Thus our stack runs parallely with the Yacc stack. When an error occurs we continue scanning input from the source file (by calling the lexical analyzer) till we get a synchronization symbol (determined currently as semicolon). We keep pushing these symbols onto the stack. From this stack, we constitute the error string as a concatenation of all the symbols starting from the first terminal we encounter from the bottom till the top of stack. This error string can later be used to help determine the grammar rule.

In order to deal with the error, we need to modify the grammar so that it becomes able to parse the statement that produced the error. We maintain a Knowledge Base of grammar rules for this purpose.

4.2 Obtaining Grammar for the New Constructs

The parse error occurs because the grammar is incomplete - it does not contain any rule for the construct used in the source program. We need to modify the grammar by adding a grammar rule for this new construct. For this purpose, we maintain a Knowledge Base of commonly found constructs and their grammar rules. Whenever an error occurs while parsing, the Knowledge Base is searched for a matching rule. If no matching rule is found, the user is asked to write a rule for the construct.

In this approach, the Knowledge Base is the most important resource. It is a repository of grammar rules in BNF notation, which evolves over time with the addition of new rules. A

particular record in the Knowledge Base contains a keyword for searching, category of the statement, the grammar rule for the statement and a list of new symbols the rule defines.

Once the location and cause of error in the program is ascertained, the Knowledge Base is searched for a matching rule. For this search, all the words in the statement can be treated as keywords, or the user can specify the category to which the statement belongs. The search in the Knowledge Base may return more than one matching rules. Some heuristics or help from the user can be employed to select the best possible rule from the list.

One heuristic that we have developed and implemented works as follows. We form a set of symbols present in the error string which is obtained during the modified parsing as discussed above. We search the Knowledge Base and find all the rules which have some keywords common to this set .This is determined by taking the intersection of the two sets. All these rules are made available to the user, who can pick the appropriate rule.

As the Knowledge Base is not exhaustive, it is possible that no matching rule for a particular construct is found in the Knowledge Base. In this case, some help from the user is required. The user is asked to write a rule for the new construct. Once the user writes a rule, it can also be added in the Knowledge Base. So, the next time such a construct is encountered its matching rule can directly be found in the Knowledge Base. While entering the new rule, the user must also enter the definition for any new non-terminals introduced in the rule.

4.3 Modifying the Grammar

After a grammar rule for the new construct is obtained it should now be added to the grammar. Before adding the rule, it must be modified so as to adapt it to the existing grammar. A grammar writer chooses some names to define a particular construct, so the new rule may contain some symbols i.e. terminals and non-terminals that are not defined in the existing grammar. These symbols must be replaced by equivalent symbols that are already defined in the grammar. If no equivalent symbols are defined in the grammar then the user or the Knowledge Base must define them. In the case when the rule is retrieved from the Knowledge Base, the record for the rule also contains a list of the new non-terminals it uses; these non-terminals must also be defined.

One simple approach for adapting the rule that we have implemented is to provide a mapping between nonterminals used in the Knowledge Base and to their equivalent nonterminals in the

grammar used. Note that for this approach, a new mapping has to be provided each time a new grammar is used. Also note that this mapping has to be created by the user and cannot be automated. We believe such mappings are easy to identify.

After the correct and adapted rule for the new construct has been obtained, it must be inserted in the grammar at an appropriate position. This position can be ascertained by looking at the parser stack. Some help from the user may also be needed for it. The rule is added in the grammar and appropriate changes are also made in the lexical analyzer. If some new keywords are introduced then the lexical analyzer must be able to recognize them, so definitions for new tokens must also be given.

After the addition of a new rule in the grammar, it may become of the form

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \ \ldots$$

This kind of grammar may give rise to various conflicts while generating the parser. To avoid this problem we left factor the grammar. After left factoring the above grammar is transformed into

$$A \rightarrow \alpha A_1$$
$$A_1 \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \ \ldots$$

This grammar is correct and is free from conflicts.

As discussed before, the new rule added in the grammar may also define some new symbols. The user must also define these symbols. So, after adding the rule a check is performed to identify and list all the grammar symbols that are used but are not defined. The definition for these symbols must then be retrieved from the Knowledge Base, and if not found they must be entered by the user in the mapping that is speicified.

After performing the above operations and making the required changes in the grammar, the parser can be regenerated. Now the parsing of the program restarts from the beginning of the program using the new parser. If the changes made to the grammar are not correct or another new construct is found in the program, the parser will again declare an error and the whole process

will be repeated. This process continues till the regenerated parser successfully parses the program. At this point it is assumed that a correct grammar for the language has been obtained.

4.4 Redundant Rules and Symbols

The grammar obtained from the procedure we described above is to be used for reengineering purposes. So, we must make sure that it is free from ambiguities and is easy to use. As a result of repeated modifications, some redundant rules may creep in the grammar. Also, the grammar may contain some symbols that are defined but never used by any other rule. It is desirable to remove these redundant rules and symbols to make the grammar short and precise. A simple analysis of the grammar would expose all the symbols that are defined but not used. These can be removed from the grammar.

Note that we can only analyze the grammar for its internal structure. We cannot delete rules that are not used by a set of programs being parsed, as these programs may simply not have used these rules. In other words, we use the programs to find new rules, but cannot use them to identify any redundant rules.

As we parse the given programs one by one (and each program in a linear fashion) and add rules every time an error string is encountered, it is possible that the set of rules added after parsing a set of programs may not be minimal. The set of rules added can also depend on the order in which programs are parsed. The only way to eliminate such redundancies is to analyze the grammar to make it more compact. For this grammar analysis and compaction techniques may be applied [6].

5 Implementation and Experience

We have developed a tool called *GramEx*, which implements the approach discussed above. Here we present a brief overview of the design and working of the tool. We also give an example from COBOL for illustration. We have chosen COBOL since most legacy systems are written in COBOL and it has many dialects.

The system has three main components -

1. The Parser
2. The Rule Knowledge Base, and
3. The Driver (User Interface)

5.1 The Parser

We use a LALR (1) parser to parse the source programs. We use the tools Lex and Yacc to generate the parser from the grammar specifications. So, the starting grammar is nothing but approximate Lex and Yacc specifications for the language. The modifications to the grammar are made in both the specification files, and the correct grammar obtained as the result is also in the form of Lex and Yacc specifications. It is easy to get the grammar rules from these specifications, as the syntax is very similar to BNF.

5.2 The Rule Knowledge Base

The Knowledge Base is a plain text file containing record for one grammar rule in each line. Each record contains the fields - search keyword(s) for the rule, category of the statement and the grammar rule itself. For example, a record for the PERFORM statement in COBOL would look like.

Keyword(s)	Category	Rule
perform, thru	Control	perform_st: **PERFORM** proc_name **THRU** proc_name perform_until_phrase

This rule can be retrieved either by using the keywords *perform* and *thru* or the category *Control*. Similarly , if the records for the *iterative* and *selection* statements in the C language added to the Knowledge Base then Knowledge Base will contain the rules of the form shown in Figure 2.

Rule No.	**Keyword**	**Category**	**Rule**
1.	if	selection	selection_statement : IF '(' expression ')' statement
2.	else	selection	selection_statement : IF '(' expression ')' statement ELSE statement
3.	switch	selection	selection_statement : SWITCH '(' expression ')' statement
4.	while	iteration	iteration_statement : WHILE '(' expression_opt ')' statement
5.	do	iteration	iteration_statement : DO statement WHILE '(' expression ')' ';'
6.	repeat	iteration	iteration_statement : REPEAT '(' statement ')'

			UNTIL expression
7.	for	iteration	iteration_statement : FOR '(' expression_opt ';' expression_opt ';' expression_opt ')' statement
8.	for	iteration	iteration_statement : FOR '(' declaration expression_opt ';'expression_opt ';' expression_opt ')' statement

Fig 2. Knowledge Base with selection and iteration rules added.

The current approach of using a flat file for Knowledge Base should suffice for most situations as we expect that the approximate grammar to be almost complete and the Knowledge Base will need to contain very few rules. In our experience the Knowledge Base has of the order of about 20 rules. If , however, Knowledge Base increases to thousands, then it would be better to maintain it as a database. For that we will have to suitably modify the interfacing of the system with the Knowledge Base.

5.3 The Driver

The Driver that controls the application provides a graphical interface for user interaction. It accepts the grammar specifications and source programs from the user and generates the parser. It then feeds the source program to the parser. The searching of the Knowledge Base for a matching rule in case of an error and modification of the grammar and Knowledge Base are all taken care of by the driver itself. Help from the user is taken when a matching rule is not found in the Knowledge Base or when some unprecedented error occurs. It maintains the list of all terminals and non-terminals in the grammar and also of newly added rules so that any changes can be undone. After the successful parsing of one program, it parses another program starting with the grammar obtained from the previous program. In this way the grammar for a set of programs is obtained.

The tool has GUI interface and is written in Java. Some of the features the tool provides are :

1. At the start the tool asks the user for the source program, the Lex and Yacc specifications of the grammar, and the mapping of the nonterminals.
2. On encountering an "error", it shows the source code with the error portion highlighted. It also gives all the matching rules and allows the user to select any of these.

3. If no matching rules exist in the Knowledge Base then the user is provided with a GUI to create a new rule.
4. A separate window is provided for browsing the Knowledge Base.
5. A command to allow user to undo changes made to the grammar. That is, if after some rule has been added , if the user does not find the modification satisfactory, he can revert back to the grammar before this addition was made.
6. Once one program has been successfully parsed (with the new grammar) , the user can specify the next file to be parsed.

5.4 An Example

We take an example source program and trace the major events that take place while extracting the grammar from it. The example program is written in a hypothetical dialect of COBOL. The input program is shown in figure 3.

```
1.  IDENTIFICATION DIVISION.
2.  PROGRAM-ID. GRAMEX-TEST.
3.  ENVIRONMENT DIVISION.
4.  DATA DIVISION.
5.  WORKING-STORAGE SECTION.
6.    01      VAR1   PIC 9(4).
7.    01      VAR2   PIC 9(4).

8.  PROCEDURE DIVISION.
9.  PARA-ONE.
10.   MOVE 10 TO VAR1.
11.   PERFORM PARA-TWO 10 TIMES SETTING VAR1 TO 0.
12. PARA-TWO.
13.   ADD 1 TO VAR1.
14.   MULTIPLY VAR1 TO VAR1 GIVING VAR2.
15.   DISPLAY VAR1, VAR2.
16.   REPEAT PARA-THREE UNTIL VAR2 > 1.
17. PARA-THREE.
18.   DISPLAY VAR2.
19.   SUBTRACT 10 FROM VAR2.
```

Fig 3: Example program in COBOL

The COBOL-74 grammar is used as the starting grammar. There are two constructs used in the program which are not there in this grammar, namely the PERFORM construct (line 11) and the

REPEAT construct (line 16). Once we get the grammar for these constructs, our grammar would be correct and complete as far as the given program is concerned.

The processing starts by taking the grammar specification and generating the parser for the grammar. The generated parser then parses the given program. On line 11, the parsing fails. At this point, the rule Knowledge Base is searched for a matching rule using the keywords given in the line containing the error. As a result all the rules for keyword PERFORM are listed. If there are multiple rules, the system asks the user to choose the most appropriate rule from the list. Suppose the exact rule for the construct is not there in the Knowledge Base, but some similar rules exist. These will be displayed. From these rules suppose the closest rule is the following:

perform_st : PERFORM proc_name int_var TIMES

As this rule is not capable of parsing the statement on line 11, the user modifies it so that it gives the correct syntax of the statement causing the error.

perform_st : PERFORM proc_name int_var TIMES SETTING var TO int_var

This rule has to be added to the grammar. Before adding the rule, if needed, the symbols names in the rule should be changed to equivalent names defined in the existing grammar. The lexical definitions for the new keywords that do not exist in the grammar will also have to be added in the lexical specifications for the grammar.

The driver searches for the appropriate position in the grammar where this rule should be added. In this case, the grammar is searched for a rule for the *perform_stmt*, on finding such a rule, the RHS of the new rule is added to the RHS of the existing rule. The grammar now looks like

```
perform_stmt :  . . .      /* existing grammar rules */
              | . . .      /*  for perform */
              | PERFORM proc_name int_var TIMES SETTING var TO int_var
```

The parser is regenerated now with this new grammar and the source program is parsed again from the first line. This time the PERFORM statement is successfully parsed. The parsing fails for the REPEAT statement on line 16 and the process described above is repeated. If no rule is

found in the database, the user can search for similar constructs using suitable constructs and then pick a rule and suitably modify it. For example, suppose there is no rule with keyword REPEAT in the Knowledge Base. By looking at the statement it is easy to observe that the REPEAT construct is very similar to the PERFORM construct, so the user can search the Knowledge Base for the keyword PERFORM. Suppose one of the rules found by searching is:

perform_st : PERFORM proc_name UNTIL condition

Appropriate changes are made to this rule, so that it becomes:

clause : REPEAT proc_name UNTIL condition

This rule is capable of parsing the REPEAT statement and is added in the grammar as well as the Knowledge Base. The lexical definition for the new keyword REPEAT is added in the lexical specifications and the parser is regenerated. The source program is parsed again from the beginning, this time the parsing is successful. The extracted grammar can be taken as the correct grammar for the input program.

5.4 Experience

We have applied our approach for extracting grammar from a set of COBOL-85 programs. We took 65 COBOL-85 source programs of varying sizes. A partial COBOL-74 grammar was chosen as the starting grammar. There were fourteen constructs in the programs that were new to the COBOL-74 grammar. Using our tool, it took us about five hours to obtain the correct grammar for the programs. This time includes the time spent in generating the parser, parsing the programs, and modifying the grammar, all of which were done automatically.

We have also used our approach for programs written in C and C++. We took the complete grammar from [13] and deleted rules for some constructs. Then we gave programs using the deleted construct and used our system to identify and add the necessary rules to the grammar to make it complete.

For example , with C, we created an initial Knowledge Base with 8 entries for statements *like if, for, while* , etc.This Knowledge Base is shown earlier in Fig. 2. We deleted the rules relating to iteration(a total of 4 rules) .We then gave the program shown in Fig.4 as input.

```
1.int get_next_tok(char e[MAXOP])
2. {
3.          int c,i=0;
4.          while ((c = postr[ind++] ) == ' ')
5.                              ;
6.          if (c == '\0') return ENDEXPR;
7.          ind--;
8.          if(!isdigit(c) && c != '.' && !isspace(c) ) {
9.                    e[i++] = c = postr[ind++];
10.                   e[i-1] = '\0';ind--;
11.                   if (i>2) return UNARYOP;
12.                   else return BINARYOP;
13.         }
14.         if (isdigit(c))
15.                   for(;isdigit(e[i++] = c = postr[ind++]);)
16.                             ;
17.         if (c == '.')
18.                   while(isdigit(e[i++] = c = postr[ind++])
19.                             ;
20.         e[i-1] = '\0';ind--;
21.         return OPERAND;
22. }
```

Fig. 3. Sample C program

When the first while loop (line 4) is handled by the parser (as no rule exists in the grammar) , our modified parsing technique and heuristic for selection of rules from Knowledge Base is applied. This selection throws up two rules – one for *while* and one for *do-while* (rules 4 and 5 in Fig 2. The user selected the *while* statement , and the rule was added to the grammar after replacing its nonterminals with that of the grammar[13]. Only one rule needs to be added to the grammar. Then the parsing is restarted by the user. This time the parser successfully parses the *while* loops and halts at the first for loop (line 15). The same process is repeated and the rule for *for* is added. Next time when the parser is restarted the whole program is successfully parsed. This whole process of parsing, selecting the rules, and reparsing took less than 15 minutes to complete.

6 Conclusion

For maintenance of large legacy software systems, some kind of support in the form of automated tools is necessary. The most important resource for building such tools is the grammar of the language in which the legacy software is written, which is frequently not available. Hence, there is a need for extracting grammar from programs. Ideally one would like the grammar extractor to

be fully automatic. Unfortunately, this is not possible. In this paper, we have presented an approach for obtaining grammar from programs using an approximate grammar and some assistance from the user.

Starting with an approximate grammar, we obtain the correct grammar in an iterative manner. The key idea behind this approach is that the constructs found in programming languages are limited in number and one can create a Knowledge Base of the commonly found constructs and their grammar. After the creation of such a Knowledge Base, one needs to insert a matching grammar rule from the Knowledge Base in the approximate grammar for the unparseable constructs found in the programs. Help from the user is taken when the construct has no matching rule in the Knowledge Base. The Knowledge Base is dynamic and grows with time as the user adds new rules. As the Knowledge Base becomes larger, the process tends to be more and more automatic and the need for user interaction lessens.

The paper also presented a brief overview of a tool called GramEx, which we have developed to implement our approach. The preliminary results that we have obtained by working on C, C++, and COBOL grammars are highly encouraging.

The current approach works well when single missing constructs are not nested. For multiple missing constructs, which are nested the help provided by the current system will be minimal and the user will probably have to provide most of the inputs. We are examining ways of handling nested constructs and an initial approach has been proposed to handle nested constructs in [14].

This work is a part of a larger research project aimed at automatically generating tools to help in maintenance and reengineering of legacy systems. Currently we are developing approaches for automatic generation of test coverage analyzer, code analysis tools etc. from the extracted grammar.

References

1. **Alex Sellink, Chris Verhoef.** *"Development, Assessment, and Reengineering of Language Descriptions,"* Proceedings of the 13th IEEE International Conference on Automated Software Engineering, Oct. 1998. Available at http://adam.wins.uva.nl/~x/cale/cale.html

2. **Alex Sellink, Chris Verhoef.** *"Generation of Software Renovation Factories from Compilers,"* Proceedings of the International Conference on Software Maintenance, Aug. 1999. Available at http://adam.wins.uva.nl/~x/com/com.html
3. **Alex Sellink, Chris Verhoef.** *"Towards Automatic Modification of Legacy Assets,"* Available at http://adam.wins.uva.nl/~x/aml/aml.html.
4. **Alex Sellink, Chris Verhoef, Mark van den Brand.** *"Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes, "* Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications. Available at http://adam.wins.uva.nl/~x/coboldef/coboldef.html
5. **Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman.** *"Compilers. Principles, Techniques and Tools,"* Addison-Wesley, 1988.
6. **Alfred V. Aho, Jeffrey D. Ullman.** *"The Theory of Parsing, Translation and Compiling,"* Prentice Hall, Inc. 1972.
7. **Boris Burshteyn.** *"USSA-Universal Syntax and Semantics Analyzer,"* ACM SIGPLAN Notices, Vol 2, No 1, Jan 1992. pp 42-60.
8. **Charles Rich, Linda M. Willis.** *"Recognizing a Program's Design: A Graph-Parsing Approach,"* IEEE Software, Jan 1990.
9. **David N. Chin, Alex Quilici.** *"DECODE: A Co-operative Program Understanding Environment,"* Software Maintenance: Research and Practice, Vol 8, 3-33 (1996).
10. **Rajesh Parekh, Vasant Honavar.** *"Grammar Inference, Automata Induction, and Language Acquisition,"* Available at http://www.cs.iastate.edu/~honavar/Papers/gi.ps.
11. **S. Cabasino, P.S. Paolucci, G.M. Todesco.** *"Dynamic Parsers and Evolving Grammars,"* ACM SIGPLAN Notices, Vol 27, No 11, Nov 1992. pp 39-48.
12. **S.C. Johnson.** *"Yacc – Yet Another Compiler Compiler, "* Technical Report 32, Bell Laboratories, 1975. Unix Programmers Manual Vol. 2.
13. **Jim Roskind** *C and C++ Yacc files.* Available at: *ftp://iecc.com/pub/file*
14. **S.Biswas**, "Extracting CFG from programs", MTech thesis, Dept of Computer Science and Engg., IIT Kanpur, 2003.