Applying Software Engineering Techniques to Parser Design: The Development of a $C^{\#}$ Parser

BRIAN A. MALLOY
Clemson University
JAMES F. POWER
National University of Ireland, Maynooth
and
JOHN T. WALDRON
Trinity College Dublin

In this paper we describe the development of a parser for the $C^{\#}$ programming language. We outline the development process used, detail its application to the development of a $C^{\#}$ parser and present a number of metrics that describe the parser's evolution. This paper presents and reinforces an argument for the application of software engineering techniques in the area of parser design. The development of a parser for the $C^{\#}$ programming language is in itself important to software engineering, since parsers form the basis for tools such as metrics generators, refactoring tools, pretty-printers and reverse engineering tools.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering,Version control*; D.2.7 [**Software Engineering**]: Metrics—*Complexity measures*; D.3.4 [**Programming Languages**]: Processors—*Parsing*

General Terms: Design, Documentation, Measurement

Additional Key Words and Phrases: $C^{\#}$ programming language, grammars, parser design, software engineering.

1. INTRODUCTION

In this paper we describe the development of a parser for the $C^{\#}$ programming language. What is novel to our approach is that we treat this process as a software engineering project, subject to the same techniques and metrication as the design of software for other areas of application. In particular we use the Code Versioning System CVS to checkpoint the development phases, and a rigorous testing framework to guide the progression from one phase to the next. This work is important in that it presents an argument for the application of these techniques in the area of parser design, and describes a framework in which this application can take place. It presents one of the first case studies of parser design, and contrasts sharply with the typical presentation of parsers "as is", with little associated documentation or justification.

The development of a parser for the $C^{\#}$ programming language is in itself important to software engineering, since parsers form the basis for many tools used by software engineers. These tools include metrics generators, class-hierarchy browsers, refactoring tools, pretty-printers and syntax highlighters, cross-referencing tools, and reverse engineering tools. Since the $C^{\#}$ programming language is relatively new, it is opportune to stress the importance of a grammar that is amenable to parsing, if these and other tools are to be built and applied with ease. Certainly, many lessons can be learned from the increasingly complex syntax of the C++ language that, in part, contributes to the difficulty of developing standardised tools for this language [Sutter 2001].

Author Addresses:
B A Malloy, Computer Science Department, Clemson University, Clemson, SC, USA; malloy@cs.clemson.edu
J F Power, Computer Science Department, National University of Ireland, Maynooth, Co. Kildare, Ireland; jpower@cs.may.ie
J T Waldron, Computer Science Department, Trinity College Dublin, Dublin 2, Ireland; john.waldron@cs.tcd.ie

Supported in part by an Enterprise Ireland International Collaboration, Grant, no. IC/2001/061.

Accepted for the Conf. of the South African Inst. of Computer Scientists and Information Technologists
Port Elizabeth, South Africa, September 16-18, 2002

Section 2 presents the background, describing briefly the C$^{\#}$ programming language as well as the area of parser design. Section 3 outlines the development process, while section 4 details its application to the development of a C$^{\#}$ parser. In particular, section 4 presents a number of metrics that describe the parser's evolution. Section 5 concludes the paper.

2. BACKGROUND AND RELATED WORK

In this section we briefly describe the C$^{\#}$ programming language by reviewing its differences with related languages such as Java and C++. We also review the parser design process, and discuss related work in the field of parser design.

2.1 The C$^{\#}$ Programming Language

The C$^{\#}$ programming language [Microsoft Corporation 2001] was developed specifically for programming on Microsoft's .NET platform, where it forms part of a suite of compilers including Visual Basic .NET and Visual C++ .NET. Syntactically, C$^{\#}$ is a descendant of the C programming language, as are its cousins C++ and Java. The language supports a preprocessor, albeit in a much reduced form than the C/C++ version, providing conditional compilation. Semantically, many features of C$^{\#}$ will seem quite familiar to Java programmers: class-based programming, a strong typing system and garbage-collected memory management. However, C$^{\#}$ still has a strong C++ flavour, using namespaces rather than packages, as well as using C++-style keywords ("`const`" rather than "`final`") and naming conventions ("`string`" rather than "`String`").

There are some hybrid features also. It is possible to define *unsafe* regions of code, where the language allows direct access to pointers in the old C/C++ style (although not under such a liberal regime). Stack-allocated `struct`s as well as `enum`s will also seem familiar to C++ programmers. The designation of method parameters as either pass-by-value (the default), pass-by-reference or result-only via keywords "`ref`" and "`out`" will not seem unusual to Fortran or Ada programmers.

Perhaps C$^{\#}$'s real strengths lie in those areas where it explicitly differs from both C++ and Java, particularly in its support for object-oriented concepts. The primitive built-in types such as "`int`" are automatically boxed (or "wrapped") into objects as necessary, and converted back as required. This addresses a significant issue in the Java programming language where these types work uneasily with the standard containers in the Java class library. C$^{\#}$'s *delegates* continue in this theme by providing boxed methods, similar to function-pointers in C++ or method objects in Java.

An interesting feature is the provision for the definition of *properties* in classes, which play the role of instance variables with associated accessor and mutator methods (so called "get" and "set" methods). Explicit support for class *attributes* and event-handling brings C$^{\#}$'s support for these features well beyond that of C++ or Java.

2.2 Parsers and Parser Generators

The description or specification of a programming language can be modularised in many ways, but it is typical to distinguish between *static* aspects, such as syntax-, scope- and type-checking, and *dynamic* aspects, such as run-time checks and the actual behaviour of the program. Within the description of the static aspects of a programming language we can distinguish between the language's *syntax*, describing the valid words and sentences in the language, and its (static) *semantics*, concerning issues such as scope and type checking. In practice, the division between these descriptions is typically defined by the descriptive power of *context-free grammars*, the specification formalism most commonly used to describe a language's syntax.

Formally, a context-free grammar (hereafter a *grammar*) is a four-tuple (T, N, S, P), where T is a set of terminal symbols, describing the allowed words in the language, N is a set of non-terminals describing sequences of these words, forming constructs such as declarations, statements, expressions etc. We distinguish a unique non-terminal S, the start symbol, whose corresponding sentences describe whole programs. Finally, P, the set of production rules, is the component most often identified with the grammar, since it describes the relationship between the nonterminal and terminal symbols, defining the syntax of the language.

In practice, the task of describing the characters that make up the words in the language can be done separately from the context-free grammar. A series of regular expressions can be used to describe the set of allowable words, and acts as the basis for the description of a *scanner*, also called a lexical analyser. Scanners can also be used for simple processing tasks such as counting lines, identifying comments, word-highlighting in editors and simple metrics that do not involve structural information.

Typically, a language's syntax is described using either a context-free grammar or *EBNF*, a descriptive formalism with the same power of expression, but employing special notation for optionality and repetition. *Parsing* is the process whereby a given program is matched against the grammar rules to determine (at least) whether or not it is syntactically correct. As part of this process the various parts of the program are identified with

Accepted for the Conf. of the South African Inst. of Computer Scientists and Information Technologists
Port Elizabeth, South Africa, September 16-18, 2002

the corresponding constructs in the grammar, so that program elements such as declarations, statements and expressions can then be identified.

The automatic generation of parsing programs from a context-free grammar is a well-established process, and various algorithms such as LL (ANTLR and JavaCC) and LALR (most notably *yacc* [Johnson 1975]) can be used; the "dragon book" [Aho et al. 1986] is the standard reference here. In this paper we use the *bison* parser generator, a tool similar to *yacc*, released under the GNU public licence. However, not every context-free grammar can be transformed to a deterministic parser. *Ambiguous* grammars contain structures for which multiple parses can be given, and even unambiguous grammars can contain constructs that are not deterministic under a given parsing algorithm.

As well as forming the front-end of a compiler, a parser is also the foundation for many software engineering tools, such as pretty-printing, automatic generation of documentation, coding tools such as class browsers, metrication tools and tools that check coding style. Automatic re-engineering and maintenance tools, as well as tools to support refactoring and reverse-engineering also typically require a parser as a front-end. The amenability of a language's syntax for parser generation is crucial in the development of such tools.

Many language specifications deliberately present the syntax of a language in two styles: one for descriptive purposes, and one that is amenable to a common parsing algorithm such as LALR. Indeed, both the standard reference on the C programming language [Kernighan and Ritchie 1988] and the Java programming language [Gosling et al. 1996] explicitly present a LALR grammar for the language. However, this is not the case for the C++ programming language, as described in Stroustrup [1997] or specified in ISO/IEC [1998], and the efficient construction of parsers and analysers for C++ programs is problematic and still an active area of research [Power and Malloy 2000b], [Willink 2001].

Since the C$^{\#}$ programming language is relatively new, and since it is likely that a considerable body of code will be written using this language over the coming years, we believe that it is vital that a standard parser for C$^{\#}$ be established at this stage, and that changes to the language be considered in the context of their impact on the parsability of the language.

2.3 Related Work

While the area of parser design, as a subset of compiler design is well-established and documented, it is not typically the subject of formalised software engineering techniques. This is somewhat ironic, given that the automatic generation of code from a specification, as typified by parser generators such as *yacc*, could be said to represent something of an ideal in software engineering. However, perhaps because of these early advances, parsers are not typically considered within the modern software engineering context, and the means of derivation of parsers from standards can often be opaque. For example, it would be notably difficult to reconstruct the exact relationship between the C++ parser in the GNU compiler collection *gcc* and the corresponding grammar in the standard, a problem exacerbated by their parallel development over the last decade.

There are a number of other barriers to the deployment of software engineering techniques in parser design. First, parsers used as compiler front-ends are a classic example of "system software", an area not traditionally exposed to software engineering techniques. Second, the traditional user-centered view, as represented by e.g. use cases in the UML, is not readily applicable to such software. Third, most of the programming language source code is *generated* code, and hence standard software metrics would not yield much information about the parser itself. Indeed, since parser design and implementation is a relatively low-level process, we do not deal with many of the more higher-level analysis and design issues in this paper.

The application of object-oriented design principles to parsers and compilers has been investigated in Reiss and Davis [1995] and Holmes [1995]. Formal transformation techniques have been investigated in Lämmel [2001], and automatic test case generation has been attempted, with varying degrees of success [Malloy and Power 2001]. A rough software engineering framework for parser design was proposed in Power and Malloy [2001]. In the remainder of this paper we extend and explicate the parser design process, and demonstrate its use in a case study, the design of a parser for C$^{\#}$.

3. METHODOLOGY

In this section we describe the methodology applied to the design and development of the C$^{\#}$ parser. Figure 1 gives an overview of the components involved in this process.

3.1 Format of the original grammar

The C$^{\#}$ grammar as presented in appendix C of the standard [Microsoft Corporation 2001] is not directly amenable to processing by a parser generator. First, the grammar is effectively a summary of the syntactic constructs described in the preceding chapters and, as such, contains a degree of repetition. Second, the grammar

Accepted for the Conf. of the South African Inst. of Computer Scientists and Information Technologists
Port Elizabeth, South Africa, September 16-18, 2002

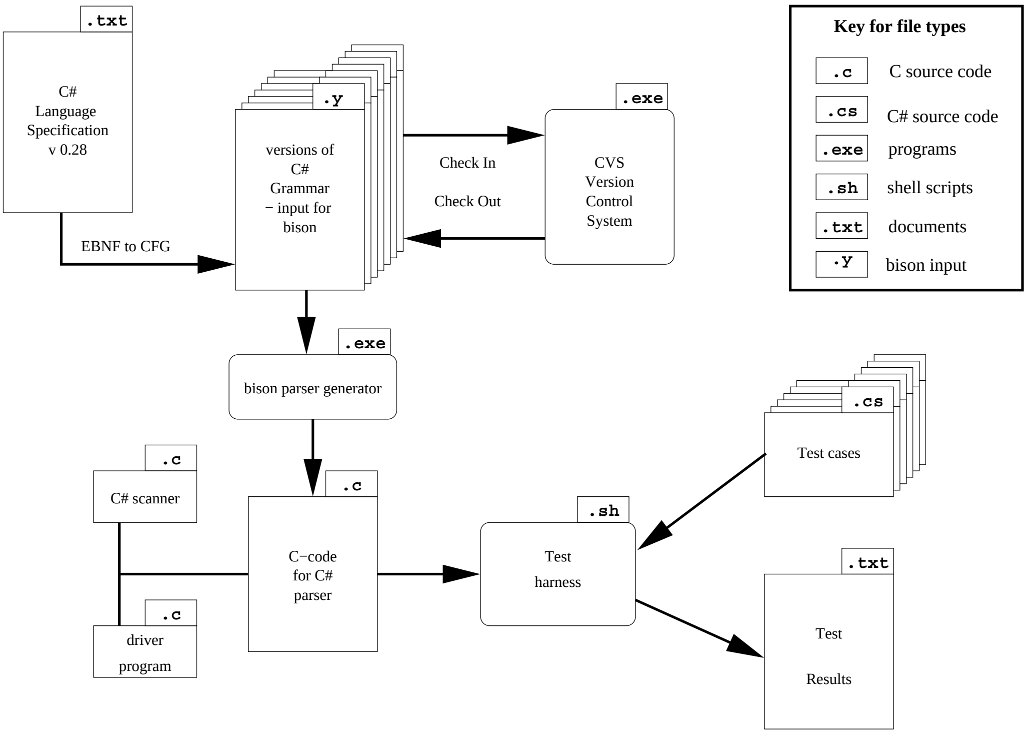

Figure 1. *System overview.* This Figure shows the main components used in the parser design process. The C$^{\#}$ Language Specification was used as the starting point for multiple versions of the C$^{\#}$ parser, controlled using CVS. Each version was tested using a driver and harness against a suite of C$^{\#}$ programs.

contains a number of nonterminals and production rules included for explanatory purposes, and which cause a degree of overlap in the description. Third, the grammar does not directly distinguish between the constructs relevant to scanner generation and those relevant to the parser. Finally, the grammar uses EBNF-style optionality constructs, which must be replaced by equivalent grammar rules for use in a parser generator such as *bison*.

In all of these features the grammar resembles the presentation of the C++ grammar in ISO/IEC [1998]. While the C$^{\#}$ language, and its corresponding documentation, is still relatively new, and while we do not suggest that its syntax even approaches the complexity of the C++ syntax, we believe that it would be reassuring and appropriate if future versions of the C$^{\#}$ language documentation included a parser-compatible grammar, in the style of the C [Kernighan and Ritchie 1988, §A13] and Java [Gosling et al. 1996, Chapter 19] language references.

3.2 Steps in the implementation of the parser

Step I: Assembling the test cases.
Testing has always played an important role in software development, and regular and frequent testing has gained increased prominence due to its emphasis in techniques such as refactoring [Fowler 1999] and extreme programming [Beck 1999], as well as the development of practical testing techniques relevant to modern software design [McGregor and Sykes 2001]. Fortunately, parser design is particularly amenable to the use of test cases, since a parser is essentially a text-processing program, easily tested in a batch-style.

The importance of the C$^{\#}$ programming language is emphasised by the reasonably large body of code already written in the language. For our test cases we chose a range of code, mostly from texts describing programming in the C$^{\#}$ language. Such texts not only tend to provide a large body of code in a single location, but can also be expected to use a good range of the language's features in the process of explaining them. This contrasts with sample applications which might only concentrate on a subset of the language features, either because of their concentration on a particular domain of application, or because of the coding style of the author. Two significant bodies of source code were included, however. These are the Ximian C$^{\#}$ compiler and the DotGNU framework, two freely available preliminary versions of C$^{\#}$ compilers.[1] The texts and programs used in this study are listed in Figure 2.

Step II: Building and testing the scanner.

[1] Microsoft has since release the source code to their own implementation of a C$^{\#}$ compiler under the name "Rotor"

Accepted for the Conf. of the South African Inst. of Computer Scientists and Information Technologists
Port Elizabeth, South Africa, September 16-18, 2002

Source of Test Cases	**No. of files**	**No. of non-blank lines**
C$^\#$ Primer: A Practical Approach by Stanley B. Lippman, ISBN: 0201729555	142	14,670
C$^\#$ Primer Plus by Klaus Michelsen, ISBN: 0672321521	177	8,918
C$^\#$ Unleashed by Joseph Mayo, ISBN: 067232122X	131	8,659
Teach Yourself C$^\#$ Web Programming in 21 Days by Phil Syme and Peter G. Aitken, ISBN: 0672322358	24	985
Sams Teach Yourself C$^\#$ in 21 Days by Bradley Jones, ISBN: 0672320711	264	12,336
C$^\#$ and the .NET Framework by Robert Powell, Richard Weeks, ISBN: 067232153X	161	11,890
C$^\#$: A Beginners Guide by Herbert Schildt, ISBN: 0072133295	26	1,810
A Programmer's Introduction to C$^\#$ by Eric Gunnerson, ISBN: 1893115623	277	11,236
C$^\#$: How to Program by Deitel, Deitel, Listfield, Nieto, Yaeger and Zlatkina, ISBN: 0130622214	504	54,586
MCS: The Ximian C$^\#$ compiler (31-Jan-02), http://www.go-mono.com/	1,411	161,395
DotGNU Portable.NET version 0.1.2, http://www.dotgnu.net/	469	59,614
Total	**3,586**	**346,099**

Figure 2. *Source of the sample code for the test cases.* The C$^\#$ programs associated with these textbooks and programs were used in the compilation of test cases for the parser.

A correctly-functioning scanner is a *sine qua non* of any parser and, since this can be developed and tested separately, it forms the first step of the development. The grammar rules relating to terminal composition were identified and translated into appropriate regular expressions. The GNU scanner generator *flex* was used to generate the scanner, and a harness program was written to lexically analyse a C$^\#$ program. Since the lexical constructs in C$^\#$ are similar to those of other language such as C, C++ and Java, this stage of the development was relatively straightforward. Only a few, easily corrected errors were detected at this point.

Step III: Converting the grammar to a parser.
The next stage involved converting the EBNF specification in the standard [Microsoft Corporation 2001, appendix C] into a format suitable for the GNU *bison* parser generator. The optionality in rules was rewritten using separate productions, and a number of duplicate rules were eliminated at this point. The *bison* parser generator is flexible in its input format, allowing duplicate rules and a loose syntax. It was relatively easy to write a tool to check *bison* grammars for strict conformance to context-free grammar syntax, and this proved useful in tidying up the grammar.

Step IV: The development process.
Once the grammar had been translated into *bison* format, a driver program framework was then written, along with some error-processing routines in the parser itself. Since the goal was just to produce a minimal working parser the error-handling was fairly basic - on encountering unrecognised syntax, the parser just reported an error, along with the line number and next token, and stopped processing.

Testing
A test harness was written that ran the parser over the test cases, recording failures in a log file. Some UNIX shell scripts were used for this purpose - a scripting language such as *perl* or *Python* could have been used here with equal ease. The ease of implementation of the test harness related directly to the batch-processing nature of the parser: the test methodology corresponded to specification-based (or "black-box") testing, rather than more elaborate white-box or unit tests. Such a testing framework was quite adequate for the C$^\#$ grammar, although an investigation of the possible benefits of unit tests in relation to the development and maintenance of a larger grammar would be interesting.

Often many checks are inserted into a grammar that cause a degree of complexity to be added, and prove to be the source of many syntactic ambiguities. In such cases it is common practice to "widen" the grammar, to accept some extra, incorrect, programs, and postpone the identification of such errors to a later phase. An example of this is the use of modifiers such as `public`, `virtual`, `abstract` etc. in various parts of the program. Even though only a specific subset of these modifiers may be used with a given construct (such as a delegate, instance variable, method, constructor etc.), it proves simpler to ignore this distinction at parse-time.

This has an important implication for test-cases, since the parser cannot now be expected to reject all incorrect programs. Thus, the tests we used were all *positive* tests - i.e. programs that were expected to pass the parsing phase. In practice this is a reasonable distinction to make, since it will not hamper the development of many

Version	02	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
s/r	40	40	58	40	39	37	6	6	6	3	1	1	1	1	1	1
r/r	617	579	149	147	147	31	6	6	2	1	1	1	1	1	1	1
in. states	58	57	42	38	38	31	8	8	6	4	2	2	2	2	2	2
test fails	2826	2819	2816	1136	926	1613	1591	1591	1541	1264	1261	1261	1261	85	59	55

Figure 3. *Quality measures during the parser's development.* This table shows the number of shift-reduce and reduce-reduce conflicts, the number of inadequate states, and the number of test-case failures for each version.

tools designed to process correct programs. This corresponded reasonably well with the test cases used, relatively few of which were examples of incorrect programs.

Version Tracking

It was fundamental to our approach that we sought to keep a record of all changes made to the grammar, as we transformed it from the format given in the standard to one that could be parsed without ambiguity. To this end the code versioning system CVS [Fogel and Bar 2001] was used. This provided for the development of the parser to be broken into separate, recorded phases (or "releases" in CVS terminology). Alterations in moving from one phase to the next can be recorded, thus providing a trail of all the modifications made to the grammar. This is particularly important in grammar development, where a given set of rules may be transformed many times during the development.

It is worth noting at this point that since the transformations are being applied to a formal instrument, a context-free grammar, it would be possible to formally record such transformations, and to formally verify that they preserve correctness, in that they describe a superset of the original language; such an approach is outlined in Lämmel [2001]. Our approach sought to apply a less rigorous model, but we note that our approach can be seen as complimentary to the formal approach, since it provides for the documentation of the transformations, which could be subsequently verified.

In the next section we provide a detailed description of this process of transforming the initial version of the parser corresponding to the C$^{\#}$ standard, to one that minimises the ambiguity and conflicts.

4. CASE STUDY

In this section we describe the details of the construction of the C$^{\#}$ parser, and present some metrics measuring its evolution.

The task of developing a parser for a predictive algorithm such as LALR is one of conflict elimination. An LALR parser works by matching the input with the right-hand sides of grammar rules. Conflicts arise when the input matches two different right-hand sides (a reduce-reduce conflict), or when the input appears to match one full right-hand side and part of another (a shift-reduce conflict). While conflicts do not necessarily mean an incorrect parser, it is desirable to eliminate as many as possible, and to fully predict the behaviour of the remainder.

Thus, developing a parser is quite similar to refactoring a program - the constructs must be rearranged to eliminate duplication and overlaps, while preserving the overall behaviour of the program. Figure 3 shows this process in action. There were 16 major versions of the parser, numbered in Figure 3 from version 02 to version 19 (the omitted versions reflect only minor modifications). As can be seen, the original grammar of version 02 contained 40 shift-reduce and 617 reduce-reduce conflicts.

Such a high number of conflicts is not unusual for a grammar not explicitly designed for a particular parsing algorithm. It should be noted that not all of these conflicts represent different problems with the grammar - many are multiple reflections of the same problems. Perhaps a better measure of the number of grammatical problems that need to be dealt with is the number of *inadequate states* - those states in the parsing machine that contain one or more conflicts. The number of these states is also given in Figure 3 and, as can be seen, reflects a much less dramatic, and more realistic, picture of the development.

The changes from versions 02 to 05 reflect some minor alterations relating to the over-specification of identifiers, as namespace or type names. It is notable that C$^{\#}$ differs from C++ and (to a lesser extent) from C in that it does not require context-sensitive identifiers, which would distinguish between identifiers used to represent types and those used to represent ordinary variable names. This feature, reflecting C$^{\#}$'s Java ancestry, is a welcome development, since it means that parser and tool developers do not need to construct a symbol table in order to parse the language.

Versions 06 through 09 also reflect C$^{\#}$'s similarity with Java, since they correspond to the elimination of ambiguities almost exactly as described in Gosling et al. [1996, Chapter 19]. These ambiguities include the over-specification of modifiers and types (including the return type of methods and the types of fields). The

Accepted for the Conf. of the South African Inst. of Computer Scientists and Information Technologists
Port Elizabeth, South Africa, September 16-18, 2002

Version	02	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
$\|T\|$	138	138	138	138	138	138	138	138	138	142	142	142	142	142	142	142
$\|N\|$	293	291	256	256	254	243	233	233	232	230	232	232	234	240	241	241
$\|P\|$	414	412	314	316	313	304	298	298	299	288	289	290	293	293	294	295
$\overline{R}$	4.1	4.1	4.2	4.2	4.2	4.3	4.4	4.4	4.4	4.4	4.4	4.4	4.4	4.4	4.4	4.4

Figure 4. *Grammar metrics charting the parser's development.* This table shows the number of terminals, non-terminals, rules and the average RHS length for each version.

changes between versions 09 and 10 are specific to the C$^{\#}$ grammar - overlaps in the definition of catch clauses, attribute arguments and expression lists.

Versions 11 and 12 moved the use of identifiers up through the definition of expressions to eliminate a conflict between qualified names and member access, both of which use a '.' as a separator. Again, this reflects a problem found and disambiguated in the Java grammar. Version 13 reflects a problem specific to C$^{\#}$ - the use of rank specifiers to specify the number of dimensions of a rectangular array. A "ragged" 2-D array of integers is declared to have type "`int [][]`" (as in Java), but C$^{\#}$ uses "`int [,]`" to describe a rectangular array. This creates problems for a parser using just a single token of lookahead (the "`[`" in this case). We chose to solve this by tokenising the entire rank specifier in the scanner. This is a (very mild) example of the approach of increasing the power of the scanner in preference to rearrangement of the grammar.

Versions 14 through 16 were attempts to disambiguate the conflict between a parenthesised expression and a cast expression. These become difficult to separate when the type/expression involved is a simple identifier e.g. "`(x)`". This is a problem shared with Java, and is amenable to the same solution of deferring the choice until more information becomes available. However, the problem is exacerbated in C$^{\#}$ as this language allows for the use of explicit pointers, and thus an extra level of ambiguity is involved between the pointer operator and multiplication. For example, a line beginning "`(x *`" might be the start of a declaration of a pointer to type `x`, or the start of an expression involving multiplication by variable `x`.

Versions 18 and 19 reflect minor changes to the parser as mistakes reflected by the test cases were fixed. We have already noted the similarity between the parser development process and refactoring. However, test cases are crucial in refactoring to test that the software's functionality has not changed, whereas the parser's development is necessarily driven by conflict elimination. Indeed the parser, while exhibiting fewer conflicts from version to version, actually exhibits an increase in the number of failed test cases between versions 08 and 09, and a relatively small decrease until version 17. These changes are really secondary to the development of the parser - the change from version 08 to 09 reflected the uncovering of a deeper problem in the grammar which was not solved until much later. Indeed, apart from causing an anxious re-examination of the parser for version 09, the test cases did not begin to play a significant role in the development until after version 17, where they could be dealt with on an individual basis.

Finally, versions 17 through 19 deal with another problem specific to C$^{\#}$ - the use of context-sensitive keywords. C$^{\#}$ has a small set of these words used for attribute targets (e.g. `assembly`), the accessor types `get` and `set`, as well as event accessor specifiers `add` and `remove`. Unusually for a modern programming language, these words are not defined as keywords in general, but act like keywords in the relevant context. While not difficult to deal with at the moment, the use of such context-sensitive keywords seems a retrogressive step in language design. Indeed a number of the test cases were already using these words as identifiers, which may provide maintenance headaches should the C$^{\#}$ language change in future versions.

As can be seen from Figure 3, the final version of the parser still has two conflicts and fails some 55 test cases. One of these conflicts is the harmless "dangling-else" conflict. The other conflict relates to an ambiguity between types and expressions which we believe to be benign, but we will need to investigate this further. The 55 failed test cases have been examined by hand and we are happy that our parser correctly rejects these as erroneous programs.

Another view of the parser development process is given by the metrics shown in Figure 4. Here we measure $|T|$, the number of terminals in the grammar, $|N|$ the number of non-terminals in the grammar, and $|P|$ the number of production rules. Also measured is $\overline{R}$, the average number of symbols on the right-hand sides of the grammar rules. A more comprehensive treatment of the role of metrics in parser development can be found in Power and Malloy [2000a].

As can be seen from Figure 4, the development process is largely one of consolidation of the production rules, with a gradual decrease in the number of non-terminals, a sharper decrease in the number of production rules, and a corresponding increase in the average size of the grammar rules (as reflected by $\overline{R}$). The slight increase in the number of non-terminals and rules at the end reflects the introduction of special productions to deal with the context-sensitive keywords.

5. CONCLUSIONS

In this paper we have presented the design of a parser for the C$^{\#}$ programming language in the context of a software engineering project. We have demonstrated that some of the standard low-level software engineering techniques such as code versioning and metrication can provide a useful track of the evolution of a parser, as well as enhancing confidence in its correctness.

We believe that it is of vital importance that simple, reliable parsers exist for commonly-used programming languages, since parsers form the basis for many software engineering tools. While such parsers form part of the specification of languages like C and Java, the task of writing a parser for C++ is considerably more complex. We hope that this work will contribute toward the development of a standard parser for C$^{\#}$, ensuring that the development of tools for this language is not unnecessarily hindered.

Two particular areas merit further investigation. First, the transformations applied to a grammar can be formalised and thus verified in the manner of Lämmel [2001]. As with all formal methods, the challenge here is in integrating the formal approach with a more pragmatic programming approach in a realistic manner. Second, refactoring tools, such as exist for object-oriented software engineering languages [Fowler 1999] could be very useful in transforming a grammar into a parser. Much of the work involved in parser design involves assessing the impact of local modifications on the grammar as a whole - this process could be considerably eased by a good grammar-specific editor.

Trademarks

Visual Basic, *Visual C++*, *Visual C$^{\#}$* and/or other Microsoft products referenced herein are either registered trademarks or trademarks of Microsoft Corporation in the U.S. and/or other countries. *Java* and all *Java*-based marks are registered trademarks of Sun Microsystems Inc. in the United States and other countries.

REFERENCES

Aho, A., Sethi, R., and Ullman, J. 1986. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, USA.

Beck, K. 1999. *Extreme Programming Explained: Embrace Change.* Addison Wesley, Reading, Massachusetts, USA.

Fogel, K. and Bar, M. 2001. *Open Source Development with CVS.* Coriolis Group, Scottsdale, AZ, USA.

Fowler, M. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Reading, Massachusetts, USA.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification.* Addison-Wesley, Reading, Massachusetts, USA.

Holmes, J. 1995. *Object-Oriented Compiler Construction.* Prentice-Hall, Englewood Cliffs, New Jersey, USA.

ISO/IEC. 1998. *International Standard: Programming Languages - C++.* Number 14882:1998(E) in ASC X3. American National Standards Institute, New York, NY, USA.

Johnson, S. C. 1975. YACC – yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, USA.

Kernighan, B. and Ritchie, D. 1988. *The C Programming Language*, second ed. Prentice-Hall, Englewood Cliffs, New Jersey, USA.

Lämmel, R. 2001. Grammar Adaptation. In *Formal Methods Europe.* LNCS, vol. 2021. Springer-Verlag, Berlin, Germany, 550–570.

Malloy, B. and Power, J. 2001. An interpretation of Purdom's algorithm for automatic generation of test cases. In *International Conference on Computer and Information Science.* International Association for Computer and Information Science, Orlando, Florida, USA.

McGregor, J. D. and Sykes, D. A. 2001. *A Practical Guide To Testing Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, USA.

Microsoft Corporation. 2001. C# language specification. Version 0.28.

Power, J. and Malloy, B. 2000a. Metric-based analysis of context-free grammars. In *8th International Workshop on Program Comprehension.* IEEE Computer Society, Limerick, Ireland.

Power, J. and Malloy, B. 2000b. Symbol table construction and name lookup in ISO C++. In *Technology of Object-Oriented Languages and Systems.* IEEE Computer Society, Sydney, Australia.

Power, J. and Malloy, B. 2001. Exploiting metrics to facilitate grammar transformation into LALR format. In *16th ACM Symposium on Applied Computing.* ACM, Las Vegas, USA.

Reiss, S. and Davis, T. 1995. Experiences writing object-oriented compiler front ends. Tech. rep., Brown University. January.

Stroustrup, B. 1997. *The C++ Programming Language*, Third ed. Addison-Wesley, Reading, Massachusetts, USA.

Sutter, H. 2001. C++ conformance roundup. *C/C++ User's Journal 19,* 4 (April), 3–17.

Willink, E. D. 2001. Meta-compilation for C++. Ph.D. thesis, Computer Science Research Group, University of Surrey, U.K.

Accepted for the Conf. of the South African Inst. of Computer Scientists and Information Technologists
Port Elizabeth, South Africa, September 16-18, 2002