A Systematic Approach to Fuzzy Parsing[1]

RAINER KOPPLER

Department of Computer Graphics and Parallel Processing (GUP Linz)
University of Linz, Altenbergerstraße 69, A-4040 Austria/Europe
email: koppler@gup.uni-linz.ac.at, phone: (+43) 70 2468 9499

SUMMARY

This paper presents a systematic approach to implementing fuzzy parsers. A fuzzy parser is a form of syntax analyzer that performs analysis on selected portions of its input rather than performing a detailed analysis of a complete source text. Fuzzy parsers are often components of software tools and also of program development environments that extract information from source texts. This paper clarifies the term fuzzy parser and introduces an object-oriented framework for implementing reusable and efficient fuzzy parsers. Applications of this framework are described via examples of two software tools. These tools exploit the facilities provided by fuzzy parsers for different purposes and also for different languages.

KEY WORDS Software Tools Syntax Analysis Frameworks Object-Oriented Design C++

INTRODUCTION

A wide range of software tools intended for use with different programming languages have the common property of extracting information from source texts. The information extracted by such tools may be evaluated for various purposes, for example, in order to determine complexity or quality properties. Examples of software tools that extract information from source texts include class browsers, dependency analyzers such as *makedepend*[1], call graph visualizers, cross reference generators and source text beautifiers. Many of these tools use a parser to extract relevant information from their input, for example:

- A class browser for C++ that examines source texts containing class definitions in order to create a tree representation of inheritance relationships;
- A dependency generator that parses and evaluates pre-processor directives, e. g., *ifdef* and *include*, in order to determine dependencies between C modules during compilation;
- A language-specific pretty printer that scans whole source texts and intersperses the token stream with style attributes or restructures the source text by means of a parse tree.

Typically, such tools deal with languages that are “hard” to parse, e. g., Fortran [2], by performing only a restricted form of analysis, i. e., analysis of a subset of the constructs provided by the language. The advantage of this approach is that it removes the need for the software developer to implement a parser for the whole of a language, but it has the disadvantage that it assumes that the language does not contain context-sensitive elements. If the language does contain such elements, then analysis must be based upon complete parse trees. Since parsers for language subsets are typically implemented “from scratch”, such

[1]. appeared in: *Software: Practice & Experience,* Vol. 27, No. 6, pp. 637-649, 1997.

parsers are inherently language-specific and hence lack a reusable structure; for example, they may apply optimizations specific to a particular language or have implementations that make them applicable for particular purposes only. This is especially true of two parsers for the same language that are concerned with different language subsets.

One solution to this lack of reusability is to develop fuzzy parsers that preserve existing properties of parsers developed from scratch, e. g., efficiency, while at the same time providing the flexibility associated with systematically developed parsers, which are insensitive to their application. Parsers that have an application-insensitive or *generic* processing mechanism coupled with a flexible implementation have the potential for reuse since they may be used to parse different components of the same language and also components of different languages. The implementation described in this paper exploits these two interrelated considerations and demonstrates how a systematic approach to development based upon genericity of the processing mechanism together with flexibility of implementation need not preclude efficiency.

The first contribution of this paper is a clarification of the term *fuzzy parser* by means of a semiformal definition. The definition builds upon operational concepts that have been factored out from several software tools and forms the basis for the development of an object-oriented framework for fuzzy parsers. The use of *design patterns* [3] in the design stage should enhance the framework's flexibility. Applications of the framework in the development of two software tools proved profitable from the view of simplicity and efficiency. A critical evaluation of the framework concludes this paper.

BACKGROUND AND TERMINOLOGY

The term *fuzzy parser* has been coined by *Sniff* [4], a commercial C++ program development environment that uses a hand-coded recursive-descent parser for a subset of C++. Sniff can process incomplete software systems containing errors by extracting symbol declarations, i. e., class, member, function and variable declarations, while ignoring method and function bodies. The parser is a fundamental component of the environment since it collects all basic information concerning a C++ software system in a form that can be distributed to all other components of the environment. With increased dissemination of Sniff, the term *fuzzy parser* has become popular especially in the domain of programming environments [5]. Such environments exploit the features of fuzzy parsers rather than conventional whole-language parsers for two fundamental reasons: because conventional parsers may be too time consuming, and because recovery from syntax errors in a conventional parser may cause information loss.

The remainder of this section is concerned with a semiformal definition of a fuzzy parser. A formal definition of a fuzzy parser is difficult to propound because syntax analysis strategies employed in fuzzy parsers, i. e., **what** parts of the input are processed and **how** they are processed, depend on language- and application-specific concerns. Thus, the definition in this section acts as a description of the operational concepts that have been factored out of a number of tools and that have been implemented via the framework introduced in the subsequent section.

Let $G = (V_N, V_T, R, S)$ be a context-free grammar where V_N is the set of nonterminal symbols, V_T is the set of terminal symbols, R is the set of production rules and S is the start nonterminal. Hence $L(G)$ – the language generated by G – is defined as a set of strings s with $s \in V_T^*$ where each s has been derived from S by application of rules of R. In the context of this paper each s corresponds to a *program* of the language $L(G)$.

A fuzzy parser $F(G)$ for a given language $L(G)$ is a syntax analyzer that recognizes some *substrings* of a *subset* of $L(G)$. It can be described by the quadruple (V_N', V_T', R', A). The members of A are called *anchor symbols* or – more concisely, *anchors* – where $A \subseteq V_T'$ and $V_T' \subseteq V_T$. An anchor symbol marks the start of a substring recognized by $F(G)$. Thus each $s \in L(G)$ that contains at least one anchor symbol is partially accepted by the fuzzy parser. For each anchor a there exists a production rule $r_a \in R'$ where the left side consists of one nonterminal $n_a \in V_N'$ called the *anchor nonterminal* and the right side specifies a production that matches the substring of s starting at a. Thus the total of anchor nonterminals in V_N', together with their corresponding production rules in R', specifies a number of "subgrammars" of the original grammar G.

Summing up, the language $L_F(G)$ that is partially accepted by $F(G)$ can be described formally as follows:

$$\mathrm{L}_F(G) = \{\, s \in \mathrm{L}(G) \mid s = \omega_1\, a\, \omega_2 \;\&\; \omega_1 \in V_T^* \;\&\; \omega_2 \in V_T^* \;\&\; a \in A \;\&\; s \text{ has been derived from } S \text{ by application of some } r \in R \,\}$$

Loosely speaking, $F(G)$ recognizes only *parts* of the language $L(G)$ by means of an unstructured set of rules. Compared with whole-language parsers, which utilize each terminal symbol in order to check the correctness of the input program, a fuzzy parser remains idle until its scanner encounters an anchor a in the input. Thereupon the parser tries to recognize n_a by means of r_a. Finally it becomes idle again until its scanner encounters another anchor or reaches the end of the input.

For practical purposes, strict application of r_a in order to recognize a substring starting at a is not obligatory if the analysis routine for r_a is implemented in a recursive-descent fashion. Here the programmer may depart from a stringent implementation of grammar rules and consider individual optimizations, for example, to skip a number of tokens or to apply custom error-handling strategies for parsers that analyze incomplete software systems.

A different notion of fuzzy parsing can be found in the area of linguistics or natural language processing[6]. Here a fuzzy parser corresponds to an algorithm that recognizes a *fuzzy language*. In contrast to ordinary languages, where a string either belongs to a language or not, a fuzzy language L over a given alphabet Σ defines a function $\mu\colon \Sigma^* \rightarrow [0, 1]$ that specifies for each string over Σ the string's "degree of correctness" with respect to L. Fuzzy context-free grammars are defined on top of ordinary grammars by extending the productions of nonterminal symbols with a finite number of "erroneous" productions, i. e., productions with a correctness between 0 and 1. Parsing of fuzzy languages induces problems since an infinite number of grammatical errors can occur while the number of productions for a given nonterminal symbol is limited. Asveld[6] introduced the concept of *fuzzy context-free K-grammars* that allows him to make a finite choice out of an infinity of grammatical errors during each context-free derivation step. Consequently, this enables recognition of fuzzy context-free languages.

While the concept of fuzzy languages and fuzzy grammars may theoretically be applicable to the work presented in this paper, parsers implemented in software tools neither care for the degree of correctness of a given input string nor are interested in which one of a finite set of erroneous productions a given erroneous input string matches. The notion of fuzzy parsing described in this paper is merely informal and regards error handling as an application-specific concern. For example, Sniff must to cope with erroneous source files, so it never reports errors. By way of contrast, a class browser should report errors in order to warn the user of wrong or missing class relationships in the output.

Frameworks are special instances of *class teams* [3] that embody common characteristics associated with a particular application domain. During *domain analysis* [7], framework designers factor out common aspects of related applications as a collection of abstract designs. These designs are implemented as a set of collaborating classes. Unlike conventional class teams, frameworks consist primarily of abstract classes; as a consequence, reusing frameworks consists primarily of plugging in application-specific components into predefined slots rather than directly utilizing the services of concrete classes. The terms *white-box* and *black-box* reuse characterize this difference respectively.

The main advantage of frameworks lies in the potential for the reuse of both design and code. In conventional approaches to object-oriented software development, each new problem involves determining key abstractions specific to that particular problem. When frameworks provide the basis for object-oriented software development, the framework designer tackles the task of identifying these key abstractions during domain analysis. Thus, a framework's pre-developed design concepts need not be generated by a user of the framework; instead, users must become familiar with respective design concepts and subsequently refine them according to their requirements.

Architecture

The logical building blocks of language analyzers such as a scanner (or lexical analyzer) and a parser are generally well-understood. Consequently, a framework for fuzzy parsers introduces no completely new abstract designs. The only divergences from the common semantics of these building blocks stem from the requirements of partial syntax analysis, and these differences affect the concepts described in the previous section.

The first design step separates the functions of a fuzzy parser into a scanner and a parser and introduces an object-oriented view for these two components. The scanner of a fuzzy parser is an object that is associated with its textual input file. Its main purpose is to read a character stream from the input file and to group characters in the sequence into tokens. Strictly speaking, the scanner's "tokenizing" capability may be restricted to tokens associated with language constructs recognized by the fuzzy parser. Furthermore, the scanner must provide a *string search* facility, which is required in order to detect anchor symbols in the character stream. The scanner also provides access to semantic attributes of token classes such as identifiers, numerical literals and strings.

A *fuzzy parser* is an object that can recognize a subset of a given programming language. Input programs to be analyzed by the parser must be associated with the parser object via scanner objects. The parser uses the scanner to examine the input file for anchor symbols, which represent starting points for the parser's analysis routines, and to obtain tokens that are consumed within the analysis routines. Analysis starts by sending the message *Run* to the parser object. The corresponding method repeats the following two steps until the end of file is reached:

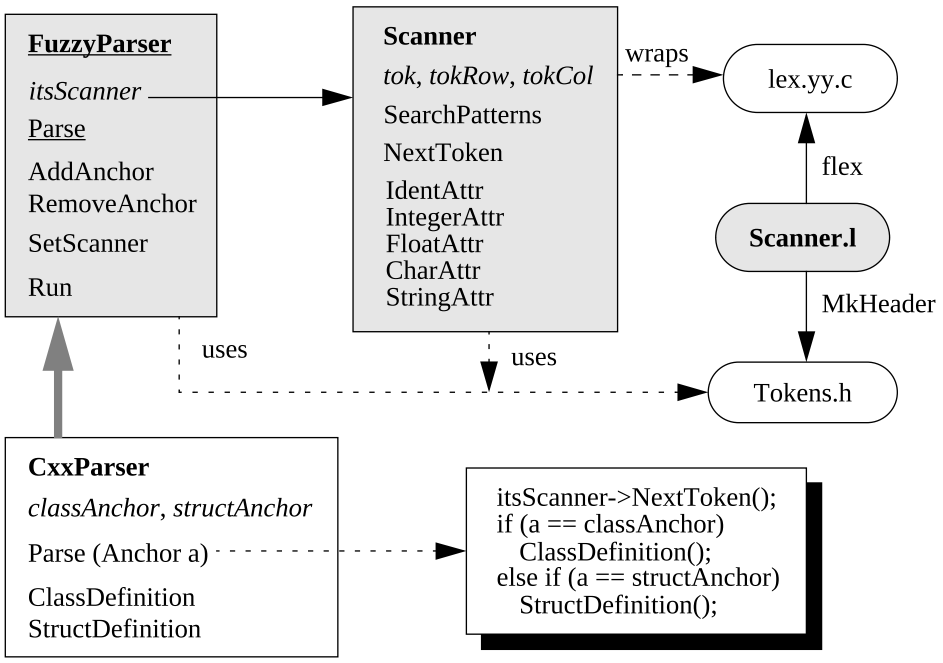
(a) The parser passes control to the scanner by sending it a message to examine the scanner's input file for the next string corresponding to one of the parser's anchors.

(b) If the scanner has found an anchor, the parser determines the corresponding analysis routine and invokes it. For the sake of clarity, analysis routines are considered as methods of the parser object. After invocation the parser consumes a token stream obtained from the scanner; here the first token always represents the anchor.

The result of the first design step is an object-oriented decomposition of fuzzy parsers. The next step separates functions common to all fuzzy parsers from those that are specific to a given application.

The abstract class *FuzzyParser* implements common functions such as anchor management and dispatching of anchors to analysis routines. In order to add application-specific behavior such as syntax rules a concrete class must be derived from class *FuzzyParser*.

Concerning the two steps of method *Run*, the services provided by the abstract class subsume the first step entirely. Since the second step contains application-specific behavior, a generic parser must implement an abstract *mechanism*, i. e., **what** to do, instead of a concrete *policy*, i. e., **how** to do it. The separation of mechanism from policy is an important technique in framework design. Concerning the fuzzy parser framework, policies correspond to application-specific anchors, analysis routines and associations between these components. Implementation of policies is delegated to concrete classes. A method named *Parse*, which receives an anchor and acts as a dispatcher, covers the association between an anchor and an analysis routine. This method is defined as abstract within class *FuzzyParser*. The mechanism consists of calling method *Parse* from method *Run* whenever an anchor has been detected. Figure 1 sketches an example implementation of *Parse* within a concrete parser class named *CxxParser*. The following sections give details about this class.



```
itsScanner->NextToken();
if (a == classAnchor)
   ClassDefinition();
else if (a == structAnchor)
   StructDefinition();
```

Fig 1: Basic framework and extensions

In addition to the dispatching mechanism, derived classes inherit a few utilities from class *FuzzyParser*, including error reporting tools and an operation that skips all tokens between two specified tokens, for example, between `{` and `}` or `BEGIN` and `END`.

Figure 1 sketches the complete framework together with an example of its reuse by means of a *class diagram*[3]. Gray rectangles represent framework classes, white rectangles represent derived classes. Class names are printed bold, instance variables in italic. Method names are printed in plain style unless they name abstract methods. In such a case the class is also abstract and both the class name and the method name are underlined. A gray arrow connects a derived class with its base class. An instance variable of a class type points to its type by means of a plain arrow. A dotted arrow connects a method name with a shaded rectangle containing the method implementation. The remaining labeled arrows show semantic relationships. Boxes with rounded corners denote automatically generated source files, which the subsequent section describes in detail.

Customization

Figure 1 shows the core of the framework, which consists of the classes *Scanner* and *FuzzyParser*. On top of this core programmers build extensions for concrete applications as listed in the introduction. This requires separating *language-specific* concerns from *application-specific* concerns.

The scanner component features the most important infrastructure for a given fuzzy parser. It encapsulates all language-specific aspects by defining a token set and providing access to semantic attributes of tokens. In order to allow easy adaption of the scanner component to individual languages, we chose a pragmatic solution. Class *Scanner* serves as a wrapper of the scanner implementation, which is either coded manually or provided by a scanner generator such as *flex*, an extension of *lex* [8]. Figure 1 shows an example of a *flex*-aided scanner implementation. The file `Scanner.l` contains a scanner specification for *flex* and is part of the framework because it represents the basis of the scanner implementation. The implementation file `lex.yy.c` is generated from the file `Scanner.l` and embedded into class *Scanner*'s implementation by means of a well-defined internal interface. Furthermore, an auxiliary tool named *MkHeader*, which Figure 1 shows as well, extracts the token set from the scanner specification and generates a token data type that is used by both class *Scanner* and class *FuzzyParser*. In order to support fuzzy parsers for a broad range of languages, the current scanner library of the framework provides scanner specifications for C, C++, Pascal and Ada.

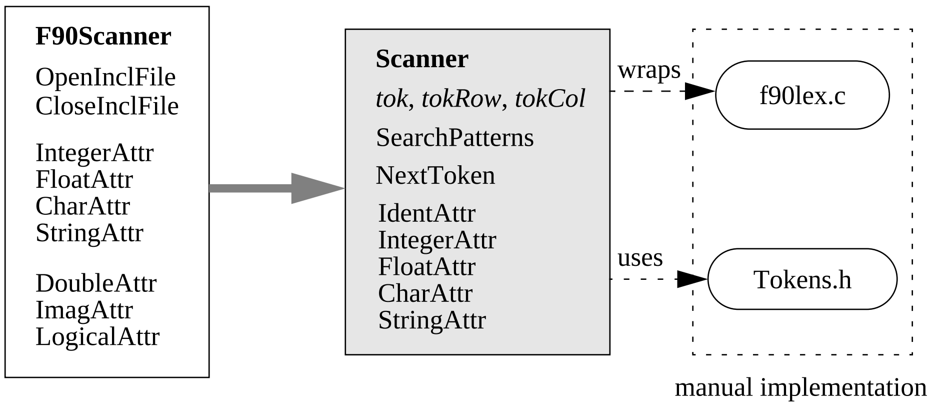


Fig 2: Adaption of the scanner component to Fortran 90

Whenever automatic implementation of a scanner proves impossible, the programmer must provide a manual implementation. The code can be embedded into the framework without problems as long as it satisfies some interface conventions. For example, a hand-coded Fortran 90 scanner was integrated into the framework by adapting its external interface to the framework's needs. Besides the scanner *implementation*, the programmer might have to extend the scanner *interface* due to individual lexical properties of the given language. Fortran requires additional access methods for token attributes as shown in Figure 2, because the language considers boolean constants and complex literals as tokens. Class *F90Scanner*, a descendant of class *Scanner*, encapsulates the extensions.

The implementation of the parser component consists of application-independent and application-specific tasks. The former include derivation of a parser class from class *FuzzyParser*, definition of anchor symbols, e. g., within the constructor, and implementation of the dispatcher method *Parse*. The extensions that Figure 1 shows implement a fuzzy parser for C++ that parses class and structure definitions. The implementation of the analysis routines *ClassDefinition* and *StructDefinition* depends on the parser's purpose and thus is an application-specific issue.

Design Concepts

This section reveals some important design patterns that the fuzzy parser framework embodies. A design pattern describes the solution to a particular object-oriented design problem or issue. It characterizes the key aspects of a common design structure that make it useful for reusing it in future framework designs. The pattern captures participating classes and their instances, defines roles, collaborations and distribution of responsibilities (see References 3 and 9 for classifications and descriptions of design patterns). The few patterns given here are useful examples as they frequently occur in framework design.

The classes *Scanner* and *FuzzyParser* together form a *framework*, a special instance of a *team*. The abstract class *FuzzyParser* features a key component since it allows the user to plug in custom behavior by means of a concrete class. This action transforms the framework into a team.

The key component of the framework, class *FuzzyParser*, is a typical example of a *behavior class* or *Strategy*, a pattern for implementing encapsulated algorithms. The algorithm is invoked by calling the method *Run*. The behavior class *FuzzyParser* provides a white-box interface for derived classes by means of the method *Parse*. Programmers overwrite this method and so customize the parser's behavior.

The scanner component is loosely coupled with the parser component in that class *FuzzyParser* utilizes services provided by class *Scanner* rather than by a derived scanner class. In other words, it uses the *abstract protocol* provided by class *Scanner*. Whenever a modified or extended scanner, for example, *F90Scanner*, is attached to the parser, the latter has full access to the extended services by means of dynamic binding while keeping flexibility by utilizing exclusively the abstract protocol. So class *Scanner* forms the vital basis for reusable scanners and thus features the root of a family of *parameter classes*, which are essential for configuration of parser classes with language-specific scanning capabilities.

RESULTS

This section illustrates applications of fuzzy parsers by means of two software tools that utilize the framework introduced in the previous section. The tools concern different programming languages but serve similar purposes: they parse certain language constructs, evaluate their semantics and construct a graphical representation.

The first tool is *ctg*, which stands for *class tree generator*. It reads a number of C++ source files, extracts class definitions, and generates a tree representation that illustrates inheritance relationships between the extracted classes. Optionally, the tool also considers *struct* definitions or definitions inside of private class sections [15]. Thus the parser of *ctg* uses the strings `class`, `struct`, `private`, `protected` and `public` as anchors, where the last four are added only on demand. In contrast to a full-fledged C++ parser, the fuzzy parser implemented in *ctg* employs the following parsing strategy, where it is assumed that none of the additional options are chosen:

(a) Search for the string `class` in the source text.

(b) Parse the class header: parse the class name; if the base class is specified, then parse the base class name.

(c) Take semantic action: if the base class is specified, then attach the new class to the base class; else attach the new class to the implicit root class.

(d) Skip the class body: ignore everything between `{` and `}`.

(e) If the end of file has not been reached, go to step (a).

Figure 3 splits up *ctg*'s implementation into three major parts and illustrates how much code each part takes. In connection with *ctg*, the framework is part of a single-rooted foundation class library, which takes about one half of the whole implementation. The remaining half consists of the fuzzy parser framework and portions specific to *ctg*, including the C++ parser and the object interface to the external preprocessor. The C++ parser together with the symbol table and the tree generator takes about the half of the application-specific part, which corresponds to 15% of the overall application. This number shows that adding minor extensions to the framework suffices to produce a fuzzy parser for simple purposes.

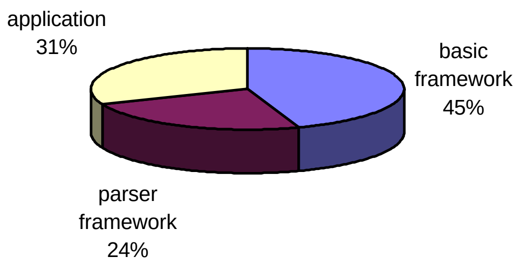


Fig 3: Parts of the application *ctg* and their ratios

Besides ease of reuse run-time efficiency is an important issue for fuzzy parsers. This shows a comparison between *ctg* and two other tools that contain fuzzy parsers for C++. Both of these tools have been optimized manually for some C++ constructs and thus cannot be reused for other languages, not even in part.

The information extractor provided by Sniff, the programming environment described in the Background and Terminology Section, parses class definitions, variable definitions and function and method headers. The main goal of the hand-optimized implementation is minimizing parsing time. Thus the parser resolves preprocessor directives itself rather than using the C++ preprocessor. This makes the parser faster, but may cause confusion with non-syntactic macros [4] or macro names used as class names; thus recent versions of Sniff allow optional use of the preprocessor. The size of Sniff's parser makes up 1.500 lines of C++ code.

The application framework ET++ [3] offers a simple fuzzy parser for C++ in the class *CodeAnalyzer*. This class serves mainly as a basis for C++ pretty printers. Derived classes are provided with a white-box interface consisting of methods for formatting comments, class definitions, function declarations and definitions. Whenever the base implementation encounters such a construct, it invokes the corresponding method. Similar to Sniff, *CodeAnalyzer* resolves preprocessor directives itself. The implementation takes 315 lines of code, but it depends on several other ET++ classes. Including all interdependences, the overall size makes up about 10.000 lines.

The test suite used for measurement of *ctg*, Sniff and *CodeAnalyzer* consists of all source files of the ET++ main directory; table 1 shows its properties. The size of the preprocessor output must be considered because *ctg* processes files independently of each other by passing each file on the preprocessor and parsing the preprocessor's output. The timings were determined on an unloaded SPARCstation IPX, which runs at 40 MHz. In the experiment with Sniff, we took the time to create a project with the given files manually; the other tools were augmented with calls to timer routines.

Table I. Properties of the test suite used for measurements

input	code size (MB)	lines of code	no. of files	no. of classes
C++ sources	1.1	53 775	280	303
preprocessor output	13	562 191	134	7 875

Table 2 shows a direct comparison of all three tools. Clearly it is not fair to compare the timings directly since the amount of data processed by *ctg* is about twelve times larger. Furthermore, *CodeAnalyzer* does not perform any semantic evaluation. In order to decouple measurements from the size of the input, a metric named *parsing rate* was established; it is given as processed kilobytes per second and measures the parser throughput. Naturally it must consider a parser's complexity, which is roughly equal for all three tools. With respect to this metric, *ctg* performs better than its competitors. In another experiment that showed *ctg*'s advantage over Sniff, the latter parsed the preprocessor output. This task was assessed at about 10 minutes, but aborted due to an unexpected exit of Sniff after 7 minutes and 40 seconds. Memory exhaustion caused Sniff's failure obviously.

Table II. Direct comparison of *ctg* with Sniff and *CodeAnalyzer*

tool	parsing time (secs.)	parsing rate (KB / sec.)	notes
ctg	126.6	105.9	preprocessor output, semantic evaluation
Sniff	11.5	98.2	C++ sources, semantic evaluation
CodeAnalyzer	24.8	45.6	C++ sources, no semantic evaluation

Table 3 shows a fair comparison between *ctg* and *CodeAnalyzer*. Both tools processed the preprocessor output, and semantic evaluation in *ctg* was disabled. Although *CodeAnalyzer* parses slightly more constructs than *ctg*, the decisive reason for its bad performance affects how it processes input. While *ctg* operates on tokens, *CodeAnalyzer* processes character by character by using ET++'s text iterators. Here *ctg*'s pragmatic solution by means of the *flex*-aided scanner implementation proves advantageous.

Table III. Comparsion between *ctg* and *CodeAnalyzer*

tool	parsing time	parsing rate
ctg	114.2	116.9
CodeAnalyzer	376.2	34.5

Another example of a software tool that provides an experimental platform for the fuzzy parser framework is *GDDT* [11] (Graphical Data Distribution Tool), a visualization tool for distributed data structures. GDDT parses source texts written in the data-parallel programming language Vienna Fortran [12], extracts processor specifications and array declarations with distribution annotations, and produces a range of diagrams for interactive investigation of mapping relationships, load distribution and inter-processor communication. Here the parser uses type identifiers, program unit keywords and the `PROCESSORS` keyword as anchors. Syntax analysis is based on a hand-coded scanner implementation, which is encapsulated in a class named *VFScanner*. This class works similar to class *F90Scanner*, an example class described in the Framework Section.

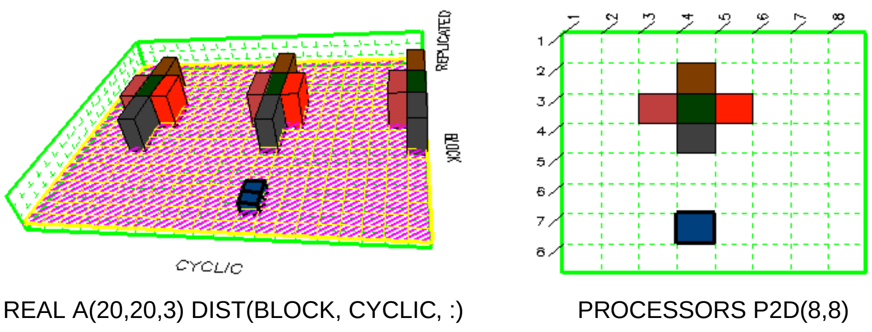


Fig 4: Visualization of mapping relationships with GDDT

Figure 4 shows the investigation of a three-dimensional array's distribution to a two-dimensional, user-defined abstraction of a multicomputer with 64 processors. The user may select processors on the right in order to determine the number and locations of data elements assigned to them. Investigations in the opposite direction are also possible.

RELATED WORK

Existing language processing tools, such as those tools mentioned earlier, build on more or less systematic parser implementations. Besides systems that use hand-coded, non-reusable fuzzy parsers, there are tools that build on systematic but mainly full-fledged parser implementations. The following gives a survey of alternative approaches that allow systematic construction of language processing tools and which address the same problems as the fuzzy parser framework described in this paper.

A traditional approach is the use of parser generators such as *yacc* [13]. Theoretically yacc could be used for the generation of fuzzy parsers by restricting input specifications to the required production rules and by appending appropriate semantic actions. This solution is acceptable unless the parser is expected to operate primarily on erroneous or incomplete software systems; otherwise application-specific parsing and error handling strategies are required in order to ensure robustness. This causes problems with yacc because generated parsers are illegible and difficult to modify.

Another approach to systematic generation of language processors is the *A** language [14]. A* is an extension of *Awk* [15] that provides facilities for processing high-level language programs. A major goal of A* is support of rapid prototyping.

Analysis tools based on A* are created in two steps. First a language-specific parser component is automatically generated from a user-supplied yacc specification by replacing semantic actions with code for building a standardized parse tree. The combination of the parser with the A* interpreter yields a tool that builds parse trees for programs written in the given language and interprets tree traversal specifications written in A*. The final tool is built in a second step by supplying the analysis component with an application-specific traversal specification. A* provides flexible pattern-matching facilities in order to allow the user to process only selected parts of the parse tree.

As soon as the analysis component has been built for a given language, A* allows development of a variety of tools. However, both creation and use of the analysis component confronts the user with several tasks. First the user must provide a scanner for the modified yacc specification. Furthermore, since parse trees are based on the concrete syntax, the tool developer must be familiar with the yacc specification in order to write traversal specifications. Finally, the concrete syntax representation often makes writing specifications a tedious task.

The previously mentioned limitation concerning the use of yacc-generated parsers as fuzzy parsers also applies to A*. Furthermore, the object-oriented solution surpasses the above approaches in reusability. Code reuse is not restricted to general services provided by the framework but also extends to specific parsers. On the one hand, the capabilities of a parser can easily be extended by means of implementation inheritance without any changes to existing code. On the other hand, a number of black-box parser objects may be configured with a common scanner and easily combined to a more sophisticated fuzzy parser, even at run time. Both above approaches complicate reuse of existing analysis capabilities due to their static nature.

CONCLUSION

This paper presented a systematic approach to the implementation of fuzzy parsers and described its implementation by means of an object-oriented framework. The main purpose of the framework is to enable easy and fast production of fuzzy parsers for software tools and programming environments. Few but powerful components should help the developer to get acquainted with the framework's concepts quickly and allow maximum reuse of existing work. Furthermore parsers based on the framework should show efficiency comparable to parsers that have been coded from scratch.

The framework described here proves that these goals are not inaccessible but can be reached by performing an accurate domain analysis and successfully combining application of design patterns with pragmatic solutions. Domain analysis is crucial to obtaining broad applicability, design patterns contribute to flexibility and reusability, and pragmatic considerations help to utilize existing efficient solutions.

Although the framework's benefits take effect in many cases of reuse, situations may occur that preclude easy application. In cases of special-purpose languages, scanner implementations may be not available but must be coded manually and adapted to the framework's interface. For such cases, a well-documented, generic `Scanner.l` file is provided, which must be adapted to the required properties of the language. In the worst case, automatic scanner generation is not feasible. Nevertheless developers should carry out a manual implementation and integrate it into the framework as they may benefit from it in future applications. With each scanner that enriches the framework's scanner library the probability of such a case decreases while the value of the framework increases. Currently the library provides scanners for C, C++, Fortran, Pascal and Ada and so covers many common applications.

Future work will primarily address technical issues. Currently the internal scanner interface does not allow opening several files simultaneously, which prevents use within shared libraries. This nuisance can be redressed by encapsulating global scanner data in classes as done by scanners generated by *flex++*. Additionally, an auxiliary tool (like *MkHeader*), which generates a parser skeleton from a class name and a set of pairs denoting anchor symbols and corresponding parsing routines, can automate the development process further. Finally, classes for language-independent symbol handling shall help to extend the framework's reusability to the semantic level.

ACKNOWLEDGEMENTS

The author thanks Peter Asveld and David Ladd for providing material about their work. Furthermore, he gratefully acknowledges the anonymous referees for their helpful suggestions on how to improve this paper.

REFERENCES

1. S. Talbott and A. Oram, *Managing Projects with make*, O'Reilly & Associates, 1991
2. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1986.
3. E. Gamma, *Objektorientierte Software-Entwicklung am Beispiel von ET++*, Springer-Verlag, 1992.
4. W. Bischofberger, 'Sniff - A Pragmatic Approach to a C++ Programming Environment', *Proc. of the USENIX C++ conference*, Portland, Oregon, 1992.
5. J. Sametinger and S. Schiffer, 'Design and Implementation Aspects of an Experimental C++ Programming Environment', *Software – Practice and Experience*, **25** (2), 111-128 (1995).
6. P. Asveld: 'A Fuzzy Approach to Erroneous Inputs in Context-Free Language Recognition', *Proc. of the 4th International Workshop on Parsing Technologies*, Prague, Czech Republic, 1995.
7. G. Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991.
8. M. E. Lesk, 'LEX - A Lexical Analyzer Generator', *CSTR 39*, Bell Laboratories, Murray Hill, NJ, 1975.
9. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Object-Oriented Software Architecture*, Addison-Wesley, 1995.

10. M. Ellis and B. Stroustup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1991.
11. R. Koppler, S. Grabner and J. Volkert: ‘Visualization of Distributed Data Structures for HPF-like Languages’, *Scientific Programming – special issue on implementations of High-Performance-Fortran*, 1995. (in press)
12. H. Zima, P. Brezany, B. Chapman, P. Mehrotra and A. Schwald, ‘Vienna Fortran – a Language Specification’, *Report ACPC-TR92-4*, Austrian Center for Parallel Computation, University of Vienna, 1992.
13. S. C. Johnson: ‘YACC - Yet Another Compiler Compiler’, *CSTR 32*, Bell Laboratories, Murray Hill, NJ, 1975.
14. D. A. Ladd and J. C. Ramming: ‘A*: a Language for Implementing Language Processors’, *Proc. of the 1994 International Conference on Computer Languages*, IEEE Computer Society Press, 1994.
15. A. V. Aho, B. W. Kerninghan and P. J. Weinberger: *The AWK Programming Language*, Addison-Wesley, 1988.