Combinator Parsers in C++: From Grammar-Flow Analysis to Design Patterns

Lisa Cosgrave[a], James Power[a], John Waldron[b]

[a]Department of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland
[b]Department of Computer Science, Trinity College Dublin, Dublin 2, Ireland

Abstract

In this paper we describe the design and implementation of a system for representing context-free grammars in C++. The system allows for grammar representation at the object level, providing enhanced modularity and flexibility when compared to traditional generator-based approaches. We also describe the transformation of grammar flow analysis problems into an object-oriented framework using the Visitor pattern, as well as the implementation of a top-down $LL(1)$ parser. As such, this work represents the synthesis of three presently disparate fields in parser design and implementation: combinator parsing, fixpoint-based grammar flow analysis, and object-oriented design.

Keywords: *context-free grammars, parsing, object-oriented design*
Computing Review Categories: *F.4.2, D.3.4, D.1.5*

1 Introduction

Formal language theory, and context-free grammars in particular, play a prominent role in computer science. As well as acting as a theoretical model, context-free grammars are the standard description technique for the syntax of programming languages, and a common starting point for the description of aspects of natural languages. Even in situations where the descriptive power of context-free grammars is insufficient, it is often useful to use them as a foundation for more powerful formalisms.

Viewed as a specification formalism, context-free grammars, perhaps because of their foundational nature, exhibit many features still on the wish list of more modern formalisms. First, they benefit from a reasonably standardised notation - the closely related Extended Backus-Naur Form (EBNF) is now an ISO standard [1]. Second, their semantics are transparent, well-understood, and easily (and often) integrated with other formalisms. Third, they can form the basis of automatic program generation through the use of parser generators such as *yacc* [10].

However, despite these advantages, it is still often difficult to integrate parser-driven applications into modern software practice. Typically, programs generated by *yacc* tend to be large, poorly structured by modern standards, and difficult to modularise, rendering them all but opaque to the user. More modern parser generation tools, such as ANTLR and JavaCC, redress this deficiency somewhat, but there are still many aspects of parser design that can benefit from software engineering approaches.

While it is true that few people will ever write complete compilers for programming languages, there are still many other uses for context-free grammars that justify their continuing study and analysis. Grammars for programming languages may be used as a front-end for program processing tools which perform tasks such as style-checking, metric calculation, maintenance and software re-engineering. Grammars can also be used for small, once-off command languages, as well as an alternative interface for the testing and debugging of GUI-based applications. Indeed, the document type definitions (DTDs) used in the Extensible Markup Language (XML) [3] are closely related to EBNF, making parsers and grammar processors relevant to a new generation of Internet-based applications.

The rest of this paper discusses the design and implementation of a system intended to contribute to the construction of parsers based on context-free grammars, and their integration into the development of object-oriented software. In section 2 we describe the implementation in C++ of a system which allows context-free grammars to be represented at the object-level in a program, overcoming the need for a separate pass during the compilation phase. Then in section 3 Second, we describe the representation of common Grammar Flow Analysis (GFA) algorithms using the *Visitor* pattern. Finally, in section 4 we extend this to the design of an $LL(1)$ parser, and discuss the implementation of a test parser for the programming language Oberon.

2 Grammar Combinators in C++

In this section we overview the representation of a context-free grammar in our system. Particular to this approach is the representation of the grammar at the object-level, the integration with a combinator-style representation for grammatical operators, and the ease with which these operators can be extended.

A context free grammar describes a language over some finite set of terminal symbols. Formally, a grammar consists of a set of *terminal* symbols, a set of *non-terminal* symbols, and a set of *production rules*. Each production rule defines a given non-terminal in terms of the union and concatenation of terminal and non-terminal symbols; we refer to this definition as the *right-part* of the rule. A sequence of terminal symbols from the language corresponding to the grammar is known as a sentence, and the the task of using a grammar to decide if a given sentence belongs to its corresponding language is called parsing. A production rule can be seen as a rewrite rule, where the language defined by the grammar is the set of sentences that can be derived from a distinguished non-terminal, called the start symbol.

Context free grammars are typically used as the specification of a *parser*, whose task is to decide whether or not a given sequence of terminal symbols is in the language corresponding to that grammar. The code for the parser may either be written by hand, as is the case with recursive-descent parsing, or generated automatically from the grammar, using some well-known algorithms such as *LL* or *LALR* parsing (see [2] for an overview of parsing techniques). Parsers can be categorised as either *top-down* or *bottom-up*, depending on whether they seek to associate the start symbol with a sentence by forward or backward application of the production rules.

There is usually a trade-off between writing the parser code manually and choosing a parser generator. Writing the code manually facilitates integration into a larger program, provides for full flexibility in grammar manipulation, and makes all the standard modularisation constructs of the implementation language available. However, such code can be difficult to understand and maintain, as the underlying grammatical structure may be obscured. Using a parser generator preserves the grammatical structure, but is not usually as flexible or easily modularised as hand-written code. In addition, an extra code generation pass now becomes part of the programming process.

Combinator parsers for functional languages [7, 5] provide a compromise between the approaches of using a parser generator or writing a parser by hand. By defining the elements of the grammar as entitles within the program we achieve the benefit of full integration with other code; by providing a set of combinators that allow high-level grammar construction we can parallel the transparency of generator-based approaches.

Typical combinator parsers work by overloading operators to represent union and concatenation, and then providing alternative implementations of these for the various grammar-wide operations, such as validity-checking, lookahead calculation and parsing. Our approach is a hybrid of ordinary combinator-based approaches and those which choose to represent the grammar explicitly, such as [8, 9]. We use the combinators as a front-end to the representation of the grammar as an object in our program, and it is this object which is then used as a basis for the remaining grammatical operations.

The representation of grammars in our system is given in figure 1. Each grammar consists of a list of terminals, non-terminals and production rules, where each production rule is an instance of the class RightPart. This class, which basically represents the definition part of a production rule as a regular expression over terminals and non-terminals plays a central role in the remaining design of our program.

Based on these classes it is then straightforward to define the relevant operators in C++. In our implementation we have chosen to overload the operators || and && to represent union and concatenation respectively. A user may then specify a grammar by creating the relevant terminal and non-terminal objects (just specifying their name in each case), and adding the relevant production rules to the grammar using these combinators.

There are two significant advantages of this approach over traditional approaches. First, since the grammar is represented explicitly as an object, it may

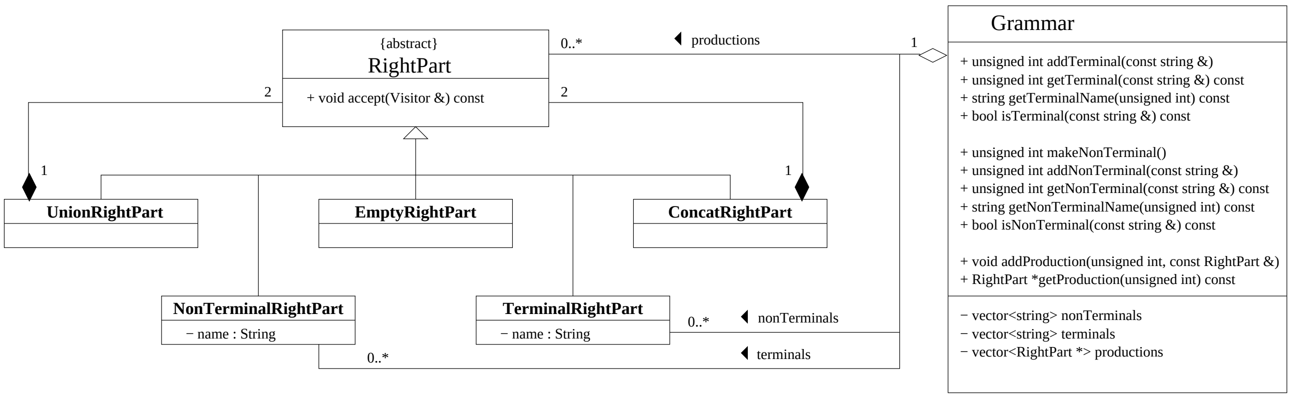
{abstract}
RightPart
+ void accept(Visitor &) const
0..*
productions
1
Grammar
+ unsigned int addTerminal(const string &)
+ unsigned int getTerminal(const string &) const
+ string getTerminalName(unsigned int) const
+ bool isTerminal(const string &) const
+ unsigned int makeNonTerminal()
+ unsigned int addNonTerminal(const string &)
+ unsigned int getNonTerminal(const string &) const
+ string getNonTerminalName(unsigned int) const
+ bool isNonTerminal(const string &) const
+ void addProduction(unsigned int, const RightPart &)
+ RightPart *getProduction(unsigned int) const
− vector<string> nonTerminals
− vector<string> terminals
− vector<RightPart *> productions
2
2
1
1
UnionRightPart
EmptyRightPart
ConcatRightPart
NonTerminalRightPart
− name : String
TerminalRightPart
− name : String
0..*
nonTerminals
terminals
0..*

Figure 1: *Overview of the representation of context-free grammars.* This UML diagram shows the representation of the essential components of a context-free grammar in our system. As can be seen from the public methods, the index of terminals, non-terminals and right-parts in the vectors stored with the `Grammar` objects are used by other classes to reference them.

be created and changed at run-time, providing us with the potential of fully dynamic or adaptable grammars, in the sense of [4, 12]. Indeed, the explicit representation of the grammar as an object in the program makes it independent of the front-end used for its specification; at present we have facilities allowing for the grammar to be specified using combinators, or using a *yacc*-like syntax.

The second advantage is that the user is not limited to the basic context-free combinators, but may define others for specific purposes. The implementation described already has more flexibility of expression than traditional parser-generators such as *yacc* in that it allows arbitrary combinations of union and concatenation in the right-part of a rule. One obvious, and easily implemented, extension is to define operations that allow for optionality, as well as zero or more repetitions, giving us the flexibility of full EBNF.

Figure 2 shows how some of these operations are implemented. It is relatively straightforward to extend the grammar combinators beyond concatenation and union by providing a translation of new combinators back into the basic structure. Indeed, since the programming language is effectively a meta-language with regard to the grammar, we achieve a functionality similar to that provided by two-level grammars. The `listOf` method is essentially a *meta-rule* for the generation of ordinary context-free grammar rules.

As an example of the style of specification, consider the following EBNF right-part definition for simple expressions (here the square brackets denote optionality, and the curly braces denote repetition).

```
SimpleExpr ::=
    ["+" | "-"] Term {AddOp Term}
```

Given suitable definitions for the terminals and non-terminals involved, we can represent this directly in C++ as a `RightPart` object by writing:

```
RightPart *SimpleExpr =
  optional(PLUS || MINUS) && listOf(Term,AddOp);
```

3 Grammar Flow Analysis as a Visitor

In this section we briefly describe the theoretical foundations of grammar flow analysis (GFA) problems, which are used to derive information about a grammar's properties, and as a basis for parsing. A full discussion of such problems, along with their theoretical foundations, can be found in [13]. We show how the representation of context-free grammars discussed in section 2, combined with the use of the *Visitor* design pattern, facilitates the implementation of grammar flow analysis problems in an object-oriented setting.

A context-free grammar effectively defines a set of mutually recursive equations over the non-terminals of that grammar. Any information that can be extracted from the grammar, including the information required for parsing purposes, must be extracted by processing these equations. As with any set of mutually-recursive equations, issues such as well-foundedness and termination are crucial to ensuring the correctness of this process.

```
RightPart *optional(const RightPart &rp)
{  // An optional rp
    int index = theGrammar.makeNonTerminal();
    NonTerminalRightPart *opt = new NonTerminalRightPart(index);
    theGrammar.addProduction(index, rp || *(new EmptyRightPart()));
    return opt;
}

RightPart *closure(const RightPart &rp)
{  // Zero or more occurrances of rp
    int index = theGrammar.makeNonTerminal();
    NonTerminalRightPart *close = new NonTerminalRightPart(index);
    theGrammar.addProduction(index, rp && *close || *(new EmptyRightPart()));
      return close;
}

RightPart *listOf(const RightPart &el, const RightPart &sep)
{  // A list of one or more el, separated by sep
    int index = theGrammar.makeNonTerminal();
    NonTerminalRightPart *seq = new NonTerminalRightPart(index);
    theGrammar.addProduction(index, el && (sep && *seq || *(new EmptyRightPart())));
    return seq;
}
```

Figure 2: *Extensions to the usual context-free notation*. This figure shows the C++ code for three functions, which add some extra constructs that can be used in forming the right-parts of production rules. The first two functions add the standard EBNF operations of optionality and closure (zero or more occurrences of a right-part). The last function provides a convenient mechanism for describing lists of one or more elements separated by some given element.

A GFA algorithm typically seeks to collect information about the non-terminals from the grammar. As with parsing techniques, GFA algorithms can be characterised as either top-down or bottom-up, depending on whether the information about a non-terminal is a function of the production rules where it is used, or of the production rule which defines it. Examples of GFA problems are:

- $REACHABLE$: can the non-terminal be used in *any* derivation starting from the start symbol
- $PRODUCTIVE$: is there any derivation starting from this non-terminal which produces a sentence
- $FIRST_k$: what are the first k symbols of all sentences derivable from a non-terminal
- $FOLLOW_k$: what are the first k symbols that may follow any part of a sentence derived from this non-terminal

Ideally, all non-terminals in a complete context-free grammar will be both productive and reachable, and so these algorithms are typically used to validate the structure of a grammar. The $FIRST$ and $FOLLOW$ algorithms form the basis of both top-down and bottom-up predictive parsing.

To implement a GFA problem we must define a *transfer function*, which specifies how the information is moved either down or up through the grammar, in accordance with the production rules. Since the production rules are mutually recursive, a naive approach to defining a transfer function may lead to an infinite loop. It can be shown straightforwardly that if the information to be collected is from a finite domain (as is the case with each of the examples above), and if the transfer function can only monotonically increase the information associated with each non-terminal, then an iterative procedure for collecting this information must necessarily reach a fixpoint and is thus guaranteed to terminate.

In terms of our implementation it is clear that the transfer function must have a close relationship with the structure of the grammar, in particular the structure of the right-parts of the production rules.

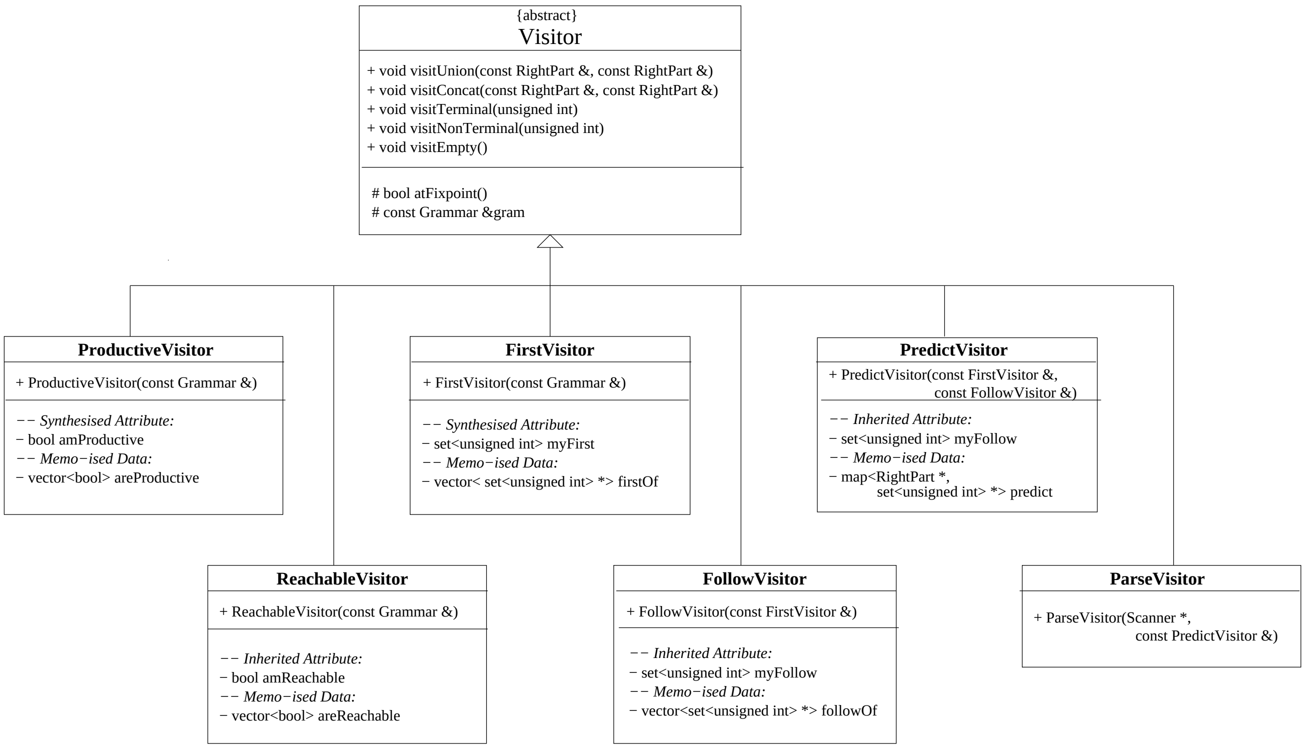
{abstract}
Visitor
+ void visitUnion(const RightPart &, const RightPart &)
+ void visitConcat(const RightPart &, const RightPart &)
+ void visitTerminal(unsigned int)
+ void visitNonTerminal(unsigned int)
+ void visitEmpty()
# bool atFixpoint()
# const Grammar &gram
ProductiveVisitor
+ ProductiveVisitor(const Grammar &)
−− Synthesised Attribute:
− bool amProductive
−− Memo−ised Data:
− vector<bool> areProductive
FirstVisitor
+ FirstVisitor(const Grammar &)
−− Synthesised Attribute:
− set<unsigned int> myFirst
−− Memo−ised Data:
− vector< set<unsigned int> *> firstOf
PredictVisitor
+ PredictVisitor(const FirstVisitor &, const FollowVisitor &)
−− Inherited Attribute:
− set<unsigned int> myFollow
−− Memo−ised Data:
− map<RightPart *, set<unsigned int> *> predict
ReachableVisitor
+ ReachableVisitor(const Grammar &)
−− Inherited Attribute:
− bool amReachable
−− Memo−ised Data:
− vector<bool> areReachable
FollowVisitor
+ FollowVisitor(const FirstVisitor &)
−− Inherited Attribute:
− set<unsigned int> myFollow
−− Memo−ised Data:
− vector<set<unsigned int> *> followOf
ParseVisitor
+ ParseVisitor(Scanner *, const PredictVisitor &)

Figure 3: *The Visitor class hierarchy*. This UML diagram depicts the `Visitor` class hierarchy. Each subclass overrides the various visitor methods to implement its main functionality; the `iterate` method drives the computation until a fixpoint is reached. The current state, as well as the end result of each GFA operation is represented in the attributes of the corresponding visitor object.

A procedural approach might define a single transfer function for each GFA problem, but in an object-oriented implementation the functionality is clearly more naturally distributed using methods in each subclass of RightPart. This however presents two immediate problems. First, the functionality of each GFA algorithm is distributed around the class, and can be difficult to maintain as a unit. Second, each new GFA algorithm requires an update or extension to he RightPart class, in order to add methods to each subclass of RightPart for its implementation.

In our approach we take a different approach, employing the *Visitor* pattern of [6]. This pattern is commonly used where there is a structure whose representation is fixed, along with a set of operations to be performed on this structure which may be changed or extended. Each of the operations to be performed, in our case each GFA algorithm, is represented by a subclass of a Visitor class, which has methods that deal with the specific cases, one for each possible subclass of RightPart. The fixed RightPart class itself needs only to have a single method `accept`, parameterised by a Visitor object, which then dispatches control and any relevant accompanying information to the received Visitor. Using the *Visitor* pattern has an additional advantage, in that any information relevant to the state of the GFA algorithm can be encapsulated in the corresponding Visitor subclass, rather than being held in the RightPart class itself, or elsewhere.

In figure 3 we show the class hierarchy for some of the GFA algorithms that we have implemented using this *Visitor* pattern. Each instance of Visitor is responsible for maintaining its own state information. Most typically this information is a map from the set of non-terminals to the information gathered by the GFA algorithm. Each Visitor subclass has its own `iterate` method which repeatedly visits each non-terminal and its corresponding definition, thus performing the transfer function. This process is repeated until the information gathered does not change, in which case a fixpoint has been reached.

Figure 4 gives the C^{++} code for the implementation of a Visitor subclass that implements the *PRODUCTIVE* GFA problem. Augmenting this class with the usual << operator to allow printing means that the code needed to call this visitor (and

```
void ProductiveVisitor::visitUnion(const RightPart &rp1, const RightPart &rp2)
{  // Implement as non-strict "or"
   rp1.accept(*this);
   if (!amProductive)
     rp2.accept(*this);
}

void ProductiveVisitor::visitConcat(const RightPart &rp1, const RightPart &rp2)
{  // Implement as non-strict "and"
   rp1.accept(*this);
   if (amProductive)
     rp2.accept(*this);
}

void ProductiveVisitor::visitTerminal(unsigned int)
{  // All terminals are productive
   amProductive = true;
}

void ProductiveVisitor::visitNonTerminal(unsigned int index)
{  // Get value from previous iteration
   amProductive = areProductive[index];
}

void ProductiveVisitor::visitEmpty()
{ // Empty string is productive
   amProductive = true;
}
```

Figure 4: *Methods from the class* `ProductiveVisitor`*.* This figure contains the C++ code showing the main actions of a `ProductiveVisitor` object. In this case the synthesised attribute is represented by the `amProductive` field, which is consulted and set as the visitor processed each `RightPart` object.

thus perform the GFA algorithm) is a straightforward process:

```
// First create the visitor:
ProductiveVisitor productive(theGrammar);
// Then start the iteration:
productive.iterate();
// Finally, print the results:
cout << productive;
```

4 LL(1) Parsing

One of the most obvious uses of the results from GFA algorithms, aside from validating grammar consistency and correctness, is as a basis for parsing. In this section we describe our design, implementation and testing of an $LL(1)$ parser, and its integration with the object-oriented framework described in previous sections.

The central issue in parsing a sentence to see if it belongs to the language described by a context-free grammar is the method used in making choices. For a top-down parser, a simple approach would be to use a depth-first search, trying the first alternative and, if it fails, trying the second. This is readily implemented using the Visitor patterns described in the previous section, giving a functionality similar to Definite Clause Grammars (DCGs) in Prolog.

However, where speed or accurate syntax error reporting is needed, it is usual to make a choice between alternatives based on some predictive strategy, such as the use of one or more symbols of lookahead. For $LL(1)$ parsing, we choose between two alternatives by forming a $PREDICT$ set for each alternative, formed by taking the $FIRST$ set, and, if this contains the empty string, adding in the relevant $FOLLOW$ set.

To this end we have implemented three more visitors: a PredictVisitor that calculates a $PREDICT$ set for each choice-point, a ConflictVisitor to report on any situations where this does not fully disambiguate the code, and a ParseVisitor that performs $LL(1)$ parsing. The characteristic method in each visitor is the `visitUnion` method, since this is where the decisions must be made between alternatives. By constructing and iterating a PredictVisitor and then supplying this for use in a ParseVisitor, we achieve an effect similar to "memoising" in functional languages.

To test the system, we prepared a grammar for the programming language Oberon-2 [16], which was chosen as it represented a non-trivial "real-world" programming language, without the inherent context-sensitivity or ambiguities of languages such as C or C++. Apart from left-recursion elimination, which is essential to all left-to-right top-down parsers, very little grammar manipulation was needed, since our system allows for the use of all the usual EBNF constructs. As test cases we used the sample code available from four texts, [11, 14, 15, 16], comprising some 23,000 lines of Oberon-2 code in total.

The testing phase threw up relatively few errors in the parsing code. Since the system had been tested and debugged for the easier GFA problems of $PRODUCTIVE$ and $REACHABLE$, most of the issues relating to the implementation of the *Visitor* pattern had been resolved. Indeed, the incremental nature of the parser development, from simple GFA algorithms, through $FIRST$, $FOLLOW$ and then $PREDICT$ had the effect of isolating many of the bugs at an earlier stage, and greatly facilitated the implementation of the ParseVisitor.

At present, tests of the system show it parsing 23,000 lines of Oberon-2 code in just under 9 seconds[1]. This compares favourably to the equivalent task performed in 0.6 seconds using a parser generated by GNU Bison version 1.28, a particularly efficient $LALR(1)$ parser generator. Profiling suggests that a significant amount of the parser's time is spent in the `visitUnion` method, checking the lookahead token against the relevant $PREDICT$ set. However, the design of our code concentrated on correctness and ease of implementation, rather than optimality. In particular, our use of containers such as `vector` and `map` from the C++ Standard Template Library could be replaced in many instances by larger, but faster, lookup tables, as is common in parser generators.

[1]The tests were carried out on a 300 MHz Pentium II based PC, with 64 MB of memory

5 Concluding Remarks

In this paper we have described a technique for grammar representation and manipulation, as well as parser design, based on a fusion of formal grammar flow analysis techniques and object-oriented design.

The system we have developed and implemented in C++ is still at the prototyping stage and, in particular, would benefit from thorough optimisation. We envisage that systems of this nature will be of use for the construction of parsers used for auxiliary purposes in larger, object-oriented programs. As mentioned in section 1, this has applications in a number of domains including in particular the testing of GUI-based applications, and the processing of DTD-based XML documents.

We hope to continue to develop and extend the system. Specifically, incorporating attribute manipulation, as well as allowing the lookahead limit to be extended beyond one symbol, are immediate goals.

References

[1] ISO/IEC 14977:1996. *Information technology – Syntactic metalanguage – Extended BNF*. International Standards Organisation, 1996.

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

[3] T. Bray, J. Paoli, and C.M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. W3C XML Working Group, 1998.

[4] H. Christiansen. A survey of adaptable grammars. *ACM SIGPLAN Notices*, 25(11):35–44, 1990.

[5] J. Fokker. Functional parsers. In Johan Jeuring and Erik Meijer, editors, *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, pages 1–23, Baastad, Sweden, May 1995.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[8] J. Jeuring and D. Swierstra. Bottom-up grammar analysis - a functional formulation. In Donald Sannella, editor, *5th European Symposium on Programming (ESOP '94)*, volume 788 of *Lecture Notes in Computer Science*, pages 317–332, Edinburgh, UK, April 1994.

[9] J. Jeuring and D. Swierstra. Constructing functional programs for grammar analysis problems. In *Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, pages 259–269, La Jolla,, California, USA, June 1995.

[10] J. Levine, T. Mason, and D. Brown. *Lex and Yacc*. O'Reilly, 1992.

[11] H. Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer Verlag, 1995.

[12] J.N. Shutt. Recursive adaptable grammars. Master's thesis, Worcester Polytechnic Institute, June 1998.

[13] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.

[14] N. Wirth. *Compiler Construction*. Addison-Wesley, 1996.

[15] N. Wirth and J. Gutknecht. *Project Oberon - The Design of an Operating System and Compiler*. Addison-Wesley, 1992.

[16] N. Wirth and M. Reiser. *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley, 1992.