THE EIFFEL OBJECT-ORIENTED PARSING LIBRARY

Philip Hucklesby

Société des Outils du Logiciel
Centre d'Affaires 3MPP
4 rue René Barthélémy 92120 Montrouge (France).

Bertrand Meyer

Interactive Software Engineering Inc.,
270 Storke Road Suite 7
Goleta, CA 93117 (USA).

Although parsing and compiling techniques are one of the most thoroughly explored areas of computer science, the construction of quality compilers remains a difficult task.

Compilers show all the problems of reusability, extendability, succinctness of code, and maintainability of resulting products which object-oriented programming claims to solve.

The work reported here has resulted in the Eiffel Parsing Library, which is a released part of version 2.2 of the Eiffel system. It is currently being used as one of the central elements in the new architecture of the Eiffel compiler itself.

A **yacc**-like tool that can be used in conjunction with this library is described (called **yoocc**).

Programming in the Microscopic ?

Object-oriented programming is often associated with simulation and graphical applications and prototypes. Its application to lower-level operations like syntactic analysis is less well known.

This bias towards high level frameworks for small or fairly homogeneous applications is largely due to deficiencies in the early designs of object-oriented languages ; Naive implementations of inheritance involved inefficient lookup tables for dynamic binding which rendered extensive use of inheritance at a low level impractical, and the artificial restriction to inheritance from a single parent precluded intricate use of inheritance in high level design.

The availability of multiple inheritance object systems with efficient message passing mechanisms has changed the outlook of object-oriented programmers. Inheritance structures of great complexity can be built with no effect on the efficiency of the message passing, and systems built using multiple inheritance are much more flexible than the "hierarchies" of single inheritance. In practice this flexibility means that inheritance graphs are built in smaller increments, and bottom-up rather than top-down. Each decision about a use of inheritance within a given architecture has less effect on future decisions, so that one can use an inheritance relation locally without worrying about its global effect on the system.

The object-oriented techniques and tools of Eiffel seemed appropriate to address the issues of compiler architecture.

Compiling : the state of the art.

In spite of promising advances such as the PQCC project, the standard techniques used by most compiler builders today are at the level of the **yacc** parser generator. Yacc has been extremely useful to many software developers. It suffers, however, from a number of limitations, of which we were made painfully aware as we applied it to the development of the first Eiffel compiler :

- The LALR (1) restrictions are often unnatural, and may require lengthy debugging of grammars for no apparent purpose.

- The rigidity of these rules is particularly detrimental to language evolution.
- The mixing of syntactic and semantic elements makes it very inconvenient to write several processors for the same language without considerable duplication of effort. In Eiffel, for example, Yacc is used not only by the various passes of the compiling commands **ec** and **es**, but also by documentation and extraction tools such as **short** and **flat**. Each must be updated separately whenever the language or compiling technology is updated.
- As with any preprocessor, and especially one that generates C, debugging can be extremely difficult.
- Yacc is inconvenient for multi-pass compilers. The inherently low-level facilities, very close to C, make it next to impossible to share information between passes on an abstract level (such as AST's which are repeatedly decorated by successive passes).

The Eiffel Parsing Library

Object-oriented programming models the objects of some external reality through classes. This should apply to compiling as well.

Here the main objects of interest are the grammar and, at a finer level of granularity, the **constructs** (terminals and non-terminals of that grammar).

To simplify matters, we adapt the grammar so that every non-terminal appears on the left-hand side of **exactly** one production. Every production is a "choice", an "aggregate" or a "sequence", illustrated respectively by the following (informally described) examples :

```
Instruction =
        Skip | Compound
        | Conditional | Loop
Conditional =
        "if" Expression
        "then" Instruction
        "else" Instruction
        "end"
Compound =
        ("begin", " ;", "end")
        Instruction
```

(A "sequence" construct is characterized by a header, a delimiter, a trailer and a base non-terminal.)

The approach to parsing followed in the library is a direct application of the phrase "syntax-directed compiling". The key idea is to obtain a straightforward one-to-one correspondence between productions of the above form and the corresponding Eiffel classes. For instance the class describing the construct Conditional above is as follows.

```
class CONDITIONAL
inherit
    AGGREGATE
feature

    template : L_LIST [CONSTRUCT] is
    local
        condition : EXPRESSION ;
        then_part : INSTRUCTION ;
        else_part : INSTRUCTION ;
    once
        Result.Create ;
        then_part.Create ;
        else_part.Create ;

        keyword ("if") ;
        branch (condition) ;
        keyword ("then") ;
        branch (then_part) ;
        keyword ("else") ;
        branch (else_part) ;
        keyword ("end")
    end ;

end ; -- class CONDITIONAL
```

As this example suggests, the Eiffel form is a direct translation of the BNF. The most significant part is the sequence of keyword and branch instructions that define the feature *template*. The appropriate ancestor (*AGGREGATE*, *SEQUENCE*, *CHOICE* or *TERMINAL*) must be named in the inheritance clause, and the rest of the text is the minimum required to turn it into a valid Eiffel class.

Although the correspondance between grammar and code is simple, the task of writing this code for a significant sized grammar is tedious, since each construct definitions resides in a separate source file. The obvious thing to do was to generate the code automatically. A tool

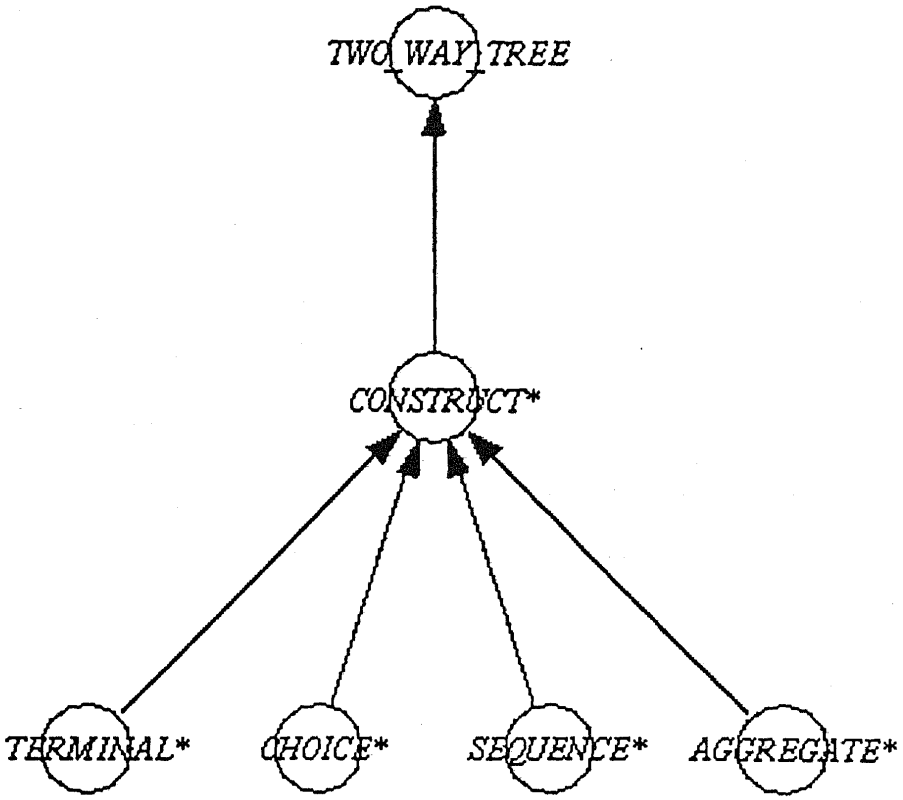

fig.1

called yoocc, ("Yes! an Object-Oriented Compiler Compiler", in homage to yacc) performs this function and is described below.

Abstract Syntax Trees

The basic data structure used in the syntactic library is the *abstract syntax tree* (or *AST*), generated for a given input text by creating a node for every occurrence of a production of the grammar. Thus an "if-then-else-end" statement might be a node in the tree and the text between each adjacent pair of keywords would generate a subtree. Drawing on experience from Cepage, a syntax directed editor, three characteristics were particularly desired:

1. Compactness

The information stored in the nodes of the tree should be the minimum necessary, in particular the abstract syntax, without the concrete syntax. Thus for a node of the tree representing a construct if ... then ... else ... end, the four keywords should not be stored in each node, but only the references to the intervening sub-trees.

2. Completeness

The concrete syntax of each node should be available. It should, for instance, be possible to reconstruct the source code from the AST.

3. Generality

Algorithms for constructing, traversing and updating the tree should be defined in general terms, without reference to the definition of the grammar for each node of the tree.

Main Implementation Techniques

The Eiffel library class *TWO_WAY_TREE* was chosen to implement AST's in the parsing library. Each node of the tree is one of four types of language construct, Either an aggregate, a sequence, a choice or a terminal. The inheritance graph of the library (fig.1) reflects this classification of language constructs.

As shown above, a once function *template* is defined for each construct of the grammar to define both the concrete (via function *keyword*) and the abstract syntax of the construct. The template is simply a list of subconstructs and keywords in the order that they appear in the

construct. During the construction of the tree, new nodes are created by Clone operations from the non-keyword elements of the template. This ensures that although all the tree construction, analysis and traversal algorithms are written in terms of tree nodes of type *CONSTRUCT* (or sometimes *AGGREGATE*, *SEQUENCE*, *CHOICE* and *TERMINAL*), the nodes actually have a dynamic type corresponding to the type of grammatical construction that generated them.

Covering semantics

The discussion so far has made no mention of semantic actions. The parser for a given language is itself a library, and can be specialised by inheritance to build many different tools operating on the same language, or even the same stored ASTs.

A routine *semantics* in class *CONSTRUCT* expresses the general scheme for executing semantic actions in a traversal of the AST's It relies on empty routines *pre_action post_action* which by default do nothing, but may be redefined by any descendant class. *pre_action* is called before recursively calling the semantics routines of each subtree, and *post_action* is executed afterwards. For sequences there is also a *middle_action* which is executed after the semantics routine of each element of the sequence.

Semantic actions can be performed either by using the library routine *semantics* to completely traverse the tree, or for simple operations for which a full traversal is not justified, by using the features of *TWO_WAY_TREE* directly.

Much of the semantics in a typical yacc application simply stores the data found for later use, indeed the cleanest use is probably to make this the sole function of the yacc code. The advantage of this is that all data is read in before any significant manipulations of it are performed, so an operation never depends on data which has not yet been read.

In the Eiffel approach the information is always stored automatically in the relevant nodes of the AST and all other operations are performed afterwards. This is a consequence of the parsing algorithm used. It would not be suitable for retrieving small amounts of information from a large file, but is ideal for multi-pass compilers since syntactic analysis can be done once for all passes.

Yoocc

Yoocc was originally conceived simply as a programmer aid to facilitate generation of the code according to the scheme described above. As usually happens with such tools, it was soon realised that it could usefully do a lot more. The language used will not be described here (it is yet another BNF-like description language). Instead we describe its use of the parsing library in the code that it generates.

Evolution of the grammar

To facilitate regeneration of the syntax definitions as the grammar evolves, yoocc generates two classes for each construct of the grammar. For instance for the construct *conditional* above it would generate class *S_CONDITIONAL* which contains the syntactic definitions and *CONDITIONAL* in which the programmer may add action functions and/or attributes.*CONDITIONAL* inherits from *S_CONDITIONAL* which inherits from *AGGREGATE*. For this discussion *S_CONDITIONAL* will be called the 'S_' (S-underscore) class and *CONDITIONAL* the leaf class. The full text of the S_ class for *CONDITIONAL* is given in fig.2.

For simple modifications of the grammar, such as occur when the language becomes reasonably stable, the semantics already implemented does not need to change. Re-running yoocc with a modified grammar updates the 'S_' classes and creates other classes only if they do not already exist.

Multiple Tool Development

The S_ classes for a given language can be thought of as a library which may be compiled with a given set of leaf classes to make a tool operating on the language. The syntactic analysis is reusable with different semantics.

What is more interesting is to share the constructed AST's between different tools. Since Eiffel objects may be stored and retrieved this is quite possible. The only problem is one of inter-project logistics; The definition of the leaf classes must be agreed upon.

If the developers of the different tools can agree in advance on the contents of the leaf classes then the whole set of S_ and leaf classes may be used as a library.

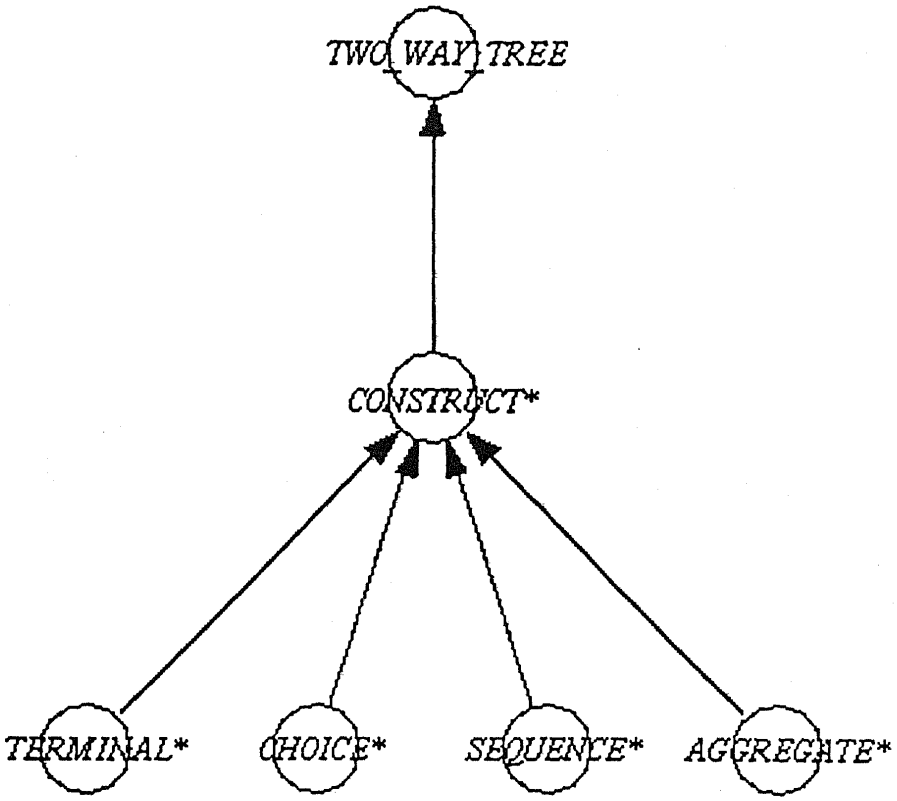

fig.1

called yoocc, ("Yes! an Object-Oriented Compiler Compiler", in homage to yacc) performs this function and is described below.

Abstract Syntax Trees

The basic data structure used in the syntactic library is the *abstract syntax tree* (or *AST*), generated for a given input text by creating a node for every occurrence of a production of the grammar. Thus an "if-then-else-end" statement might be a node in the tree and the text between each adjacent pair of keywords would generate a subtree. Drawing on experience from Cepage, a syntax directed editor, three characteristics were particularly desired:

1. Compactness

The information stored in the nodes of the tree should be the minimum necessary, in particular the abstract syntax, without the concrete syntax. Thus for a node of the tree representing a construct if ... then ... else ... end, the four keywords should not be stored in each node, but only the references to the intervening sub-trees.

2. Completeness

The concrete syntax of each node should be available. It should, for instance, be possible to reconstruct the source code from the AST.

3. Generality

Algorithms for constructing, traversing and updating the tree should be defined in general terms, without reference to the definition of the grammar for each node of the tree.

Main Implementation Techniques

The Eiffel library class *TWO_WAY_TREE* was chosen to implement AST's in the parsing library. Each node of the tree is one of four types of language construct, Either an aggregate, a sequence, a choice or a terminal. The inheritance graph of the library (fig.1) reflects this classification of language constructs.

As shown above, a once function *template* is defined for each construct of the grammar to define both the concrete (via function *keyword*) and the abstract syntax of the construct. The template is simply a list of subconstructs and keywords in the order that they appear in the

construct. During the construction of the tree, new nodes are created by Clone operations from the non-keyword elements of the template. This ensures that although all the tree construction, analysis and traversal algorithms are written in terms of tree nodes of type *CONSTRUCT* (or sometimes *AGGREGATE*, *SEQUENCE*, *CHOICE* and *TERMINAL*), the nodes actually have a dynamic type corresponding to the type of grammatical construction that generated them.

Covering semantics

The discussion so far has made no mention of semantic actions. The parser for a given language is itself a library, and can be specialised by inheritance to build many different tools operating on the same language, or even the same stored ASTs.

A routine *semantics* in class *CONSTRUCT* expresses the general scheme for executing semantic actions in a traversal of the AST's It relies on empty routines *pre_action post_action* which by default do nothing, but may be redefined by any descendant class. *pre_action* is called before recursively calling the semantics routines of each subtree, and *post_action* is executed afterwards. For sequences there is also a *middle_action* which is executed after the semantics routine of each element of the sequence.

Semantic actions can be performed either by using the library routine *semantics* to completely traverse the tree, or for simple operations for which a full traversal is not justified, by using the features of *TWO_WAY_TREE* directly.

Much of the semantics in a typical yacc application simply stores the data found for later use, indeed the cleanest use is probably to make this the sole function of the yacc code. The advantage of this is that all data is read in before any significant manipulations of it are performed, so an operation never depends on data which has not yet been read.

In the Eiffel approach the information is always stored automatically in the relevant nodes of the AST and all other operations are performed afterwards. This is a consequence of the parsing algorithm used. It would not be suitable for retrieving small amounts of information from a large file, but is ideal for multi-pass compilers since syntactic analysis can be done once for all passes.

Yoocc

Yoocc was originally conceived simply as a programmer aid to facilitate generation of the code according to the scheme described above. As usually happens with such tools, it was soon realised that it could usefully do a lot more. The language used will not be described here (it is yet another BNF-like description language). Instead we describe its use of the parsing library in the code that it generates.

Evolution of the grammar

To facilitate regeneration of the syntax definitions as the grammar evolves, yoocc generates two classes for each construct of the grammar. For instance for the construct *conditional* above it would generate class *S_CONDITIONAL* which contains the syntactic definitions and *CONDITIONAL* in which the programmer may add action functions and/or attributes.*CONDITIONAL* inherits from *S_CONDITIONAL* which inherits from *AGGREGATE*. For this discussion *S_CONDITIONAL* will be called the 'S_' (S-underscore) class and *CONDITIONAL* the leaf class. The full text of the S_ class for *CONDITIONAL* is given in fig.2.

For simple modifications of the grammar, such as occur when the language becomes reasonably stable, the semantics already implemented does not need to change. Re-running yoocc with a modified grammar updates the 'S_' classes and creates other classes only if they do not already exist.

Multiple Tool Development

The S_ classes for a given language can be thought of as a library which may be compiled with a given set of leaf classes to make a tool operating on the language. The syntactic analysis is reusable with different semantics.

What is more interesting is to share the constructed AST's between different tools. Since Eiffel objects may be stored and retrieved this is quite possible. The only problem is one of inter-project logistics; The definition of the leaf classes must be agreed upon.

If the developers of the different tools can agree in advance on the contents of the leaf classes then the whole set of S_ and leaf classes may be used as a library.

In practice this is too restrictive so the scenario illustrated in fig.3 is used. For the development of two tools 1 and 2, the developer of tool 1 works with an empty class *_2CONDITIONAL* and vice versa and the classes from each developer are compiled together to produce the final system.

Command line options on yoocc allow this structure of classes to be generated to any number of levels.

Exported Interface of the AST

Also generated by yoocc are interface functions in each 'S_' class returning each subtree with the correct static type. This is one of the rare uses of the "reverse assignment attempt" (represented by the operator '?='). If B is a descendent of A, The assignment B := A is not normally allowed, since class B may have features which class A does not. The assignment in the other direction is allowed, so that entities of type A may at runtime have dynamic type B. If the dynamic type of an entity is known statically to be of a particular dynamic type then the reverse assignment attempt may be used. If the dynamic type is not correct, the assignment fails and the target entity becomes a void reference. Testing whether the reference is void for targets of different types gives a way of testing the dynamic type of an object.

Use of the reverse assignment attempt is at best a suspicious event, and the operator was added to the Eiffel language with some reluctance. If it is used to perform different operations depending on the dynamic type of the object, one must ask why dynamic binding is not used, and if the dynamic type is known statically, one must ask why it is not the same as the static type. Either situation may indicate a weakness of the program architecture.

In this instance, the tree nodes are necessarily declared of type *CONSTRUCT* to implement the tree in a general form, but as remarked above, their dynamic types correspond to their grammatical function since they are Cloned from the template list. For someone writing an application to process the language, the interface to the AST should be in terms of the actual types of the nodes so that any features defined at the level of *CONDITIONAL* may be used although they would not be exported by *CONSTRUCT*.

This use of the reverse assignment attempt may be thought of as a space saving device. An equivalent implementation would be to use attributes instead of functions, but then there would be two pointers in a tree node to each of its subtrees.

Parsing Techniques

The mechanisms defined in the general-purpose classes define a recursive left-descent algorithm with backtracking. For an LL(k) grammar, the amount of backtracking can be reduced by calling the procedure *commit*, which discards hopeless subtrees in a manner similar to Prolog's cut mechanism. The position of the *commit* also determines the error messages given if syntax errors occur.

The same overall scheme could undoubtedly be used with other compiling techniques; top-down parsing seemed appropriate for the purposes of the parsing library because it allowed a direct and intuitive correspondence between the code written by a client programmer and the internal algorithm of the parser. By compiling the library with the debug option, debugging print statements provide a trace of the successfully and unsuccessfully parsed constructs.

Incremental compilation

One of the aims of the current Eiffel development is to progress toward a finer grained incremental compilation system. The current compiler already detects whether the interface of a feature or class has changed, enabling it to recompile less of the system if changes are localised.

One approach to incremental compilation would be to couple the compiler tightly to an intelligent editor. While this has other advantages, like the possibility of syntax directed editing, it is undesirable in that users will always have certain tasks for which they prefer to use their favourite straightforward text editor.

It is hoped that the parsing library will provide a framework where incremental compilation can be implemented in a language independent fashion, without the need for any particular editor. The idea is simple: functions will be added to the parsing library to perform a special type of tree traversal where tokens are

```
class S_CONDITIONAL
inherit
        AGGREGATE
feature

        construct_name: STRING is
        once
                Result := CONDITIONAL;
        end;

        template: LINKED_LIST [CONSTRUCT] is
        local
                condition: EXPRESSION;
                then_part: INSTRUCTION;
                else_part: INSTRUCTION;
        once
                Result.Create;
                then_part.Create;
                else_part.Create;

                keyword ("if");
                branch  (condition);
                keyword ("then");
                branch  (then_part);
                keyword ("else");
                branch  (else_part);
                keyword ("end")
        end;

        condition: EXPRESSION is
        do
                child_go (2);
                Result ?= item;
        end;

        then_part: INSTRUCTION is
        do
                child_go (4);
                Result ?= item;
        end;

        else_part: INSTRUCTION is
        do
                child_go (6);
                Result ?= item;
        end;

end;  -- class S_CONDITIONAL
```

fig.2 (below)

Inheritance used to partition the AST between developers of different tools.

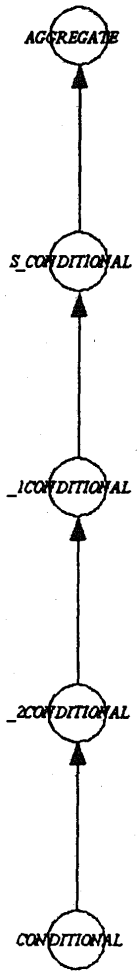

fig.3 (left)

A sample of the output of yoocc for the if-then-else-end construct showing the 'template' function and the interface functions using a reverse assignment attempt to return the data with the correct static type.

read from a modified text and compared with the concrete syntax information from each node. As soon as a difference is detected, the current subtree is regenerated and the traversal continues. Thus although a lexical analysis of the whole input file must be performed, only the modified parts of the AST are regenerated.

Lexical analysis

The lexical analysis library contains classes for defining lexical tokens in a regular expression language, building finite automata, and converting them to deterministic automata. The state table for a given lexical analyser may be built and stored once, and then be retrieved by several different tools, so that the so on.

In keeping with the principles of reusable programming, predefined elements cover the most common types of tokens (integers, identifiers, strings etc.).

Assessment

The set of tools resulting from the work described above seems to fulfil the promises of versatility and reusability.

Although it provides an almost ideal flagship example of the entire Eiffel technology, this project is more than just an academic example. It was conceived as the cornerstone of future Eiffel compiling technology and seems to live up to the expectations.

Acknowledgements

The parsing libraries were implemented chiefly by Bernard Nieto, Jean-Francois Macary, Philip Hucklesby and Philippe Stephan. This paper owes much to lively discussions among the dynamic teams of engineers at Interactive Software Engineering and Societe des Outils du Logiciel (special thanks in this respect are due to Philippe Elinck).

References

1. Interactive Software Engineering Inc., "Eiffel The Language," Technical Report TR-EI-17/RM, October 1989 (version 2.2, Oct 1989).
2. Interactive Software Engineering Inc., "Eiffel The Libraries," Technical Report TR-EI-7/LI, October 1989 (version 2.1, Oct 1989).
3. K. John Gough , *Syntax Analysis and Software Tools*, Addison Wesley, 1988.
4. Bertrand Meyer, "Reusability: the Case for Object-Oriented Design," *IEEE Software*, vol. 4, no. 2 , pp. 50-64, March 1987.
5. Bertrand Meyer, "Eiffel: A Language and Environment for Software Engineering," *The Journal of Systems and Software*, 1988.
6. Bertrand Meyer, "Genericity, static type checking, and inheritance," *The Journal of Pascal, Ada and Modula-2*, 1988. (Original version in OOPSLA 86 proceedings, SIGPLAN Notices, Sept. 1986, pp. 391-405.)
7. Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.
8. Bertrand Meyer, "From Structured Programming to Object-Oriented Design: The Road to Eiffel," *Structured Programming*, vol. 10, no. 1, pp. 19-39, 1989.